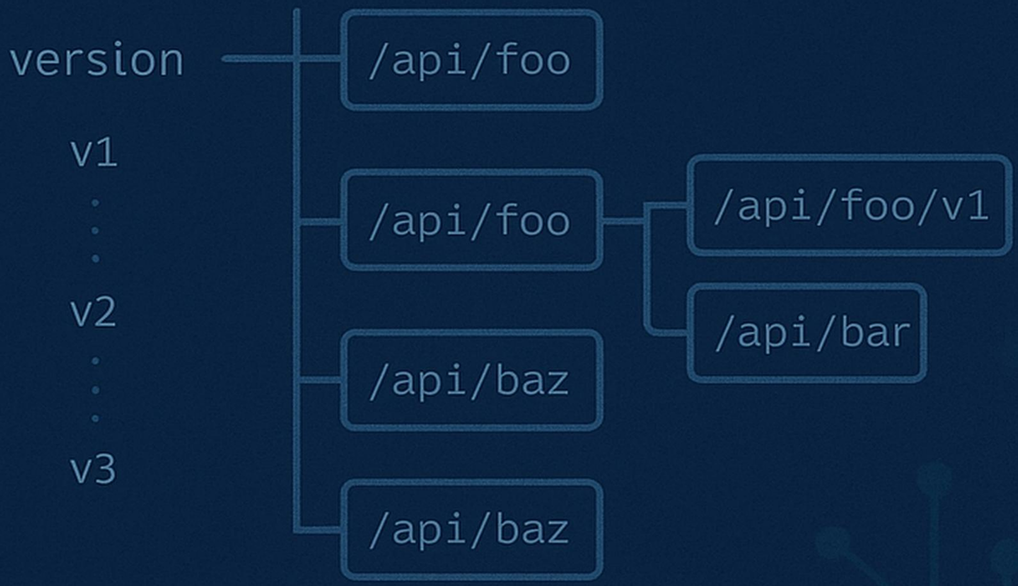
WEB API DESIGN AND EVOLUTION

Souhaila Serbout



PhD Dissertation

Università della Svizzera italiana

Supervisor: Prof. Dr. Cesare Pautasso

Web API Design and Evolution

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Souhaila Serbout

under the supervision of
Prof. Dr. Cesare Pautasso

February 2025

Dissertation Committee

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Souhaila Serbout
Lugano, 19 February 2025

To my loved ones

Science will not give you part of itself until you give it all of yourself

Avicenna

Abstract

Web APIs (Application Programming Interfaces) are a cornerstone of modern software development, enabling interoperability and integration across diverse systems. Despite their importance, there is a lack of large-scale empirical studies on the design, evolution, and versioning of APIs. This research addresses this gap by leveraging a comprehensive dataset of OpenAPI Specifications (OAS) from public repositories such as GitHub and SwaggerHub. The thesis provides empirical results describing the Web API landscape, focusing on structural patterns, design smells, changes, and versioning practices.

Through mining commit histories and metadata, the study identifies trends in API evolution and assesses their adherence to versioning principles. Clustering techniques and natural language processing are employed to detect common structural patterns and understand the semantic context of API elements. The research adopts a language-agnostic, automated approach to analyze API design and evolution at scale. The goal is to answer questions on Web API design and evolution without being tied to a specific programming ecosystem.

We empirically identified four recurring structural patterns in Web API designs to provide access to enumerable, dependable, and mutable collections, which serve as modular and reusable building blocks. Additionally, the analysis uncovers design smells that hinder usability, maintainability, and security, providing actionable insights to improve API quality.

Web API structures are inherently tied to the operations they provide for handling resources derived from specific data models. As such, studying APIs requires a dual focus on their structural design and the underlying data models to ensure coherence. We examined the relationship between API structures and their data models, highlighting frequent misalignments with design principles such as logical structuring and consistent naming conventions.

Regarding API evolution, the study categorizes over 200 types of changes and reveals that breaking changes occur 2.44 times more often than non-breaking ones. While many breaking changes appear minor, their cumulative impact on client applications can be significant.

When it comes to Web API versioning, we detected a high level of diversity in the adopted versioning schemes. In addition, APIs that claimed to use Semantic Versioning often failed to adhere to its basic rules.

The findings reveal that while Web APIs have been part of the software landscape since more than 20 years, they still do not meet the theoretical design principles when it comes to their practical implementation and management. This is explained by the lack of straightforward tools that can guide developers in adhering to best practices, identifying design flaws, and ensuring consistent versioning and evolution management throughout the API lifecycle.

In light of this empirical evidence, we propose supporting both API developers and users through language-agnostic visualization tools that can be integrated into development environments (IDE-based) or easily accessible for users who only want to learn about the API (web-based). The proposed tools are initial research prototypes built based on our findings. OAS2tree enhances API feature navigation and integrates functionality to identify and flag potential design flaws. APIcture offers a picture of API histories, providing an intuitive way to track their evolution. Although effective in their initial form, these tools have the potential to be further refined through user feedback and studies involving the target audience, ultimately improving their features, usability, and impact.

Acknowledgements

This thesis marks the culmination of a journey filled with challenges, growth, and discovery. It would not have been possible without many individuals' guidance, support, and encouragement, to whom I am profoundly grateful.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Dr. Cesare Pautasso, for his unwavering support throughout this long and challenging journey. His guidance, insightful feedback, and ability to navigate decision-making processes have been invaluable. Prof. Pautasso has played a pivotal role in shaping this thesis, and his mentorship has been an essential factor in bringing it to completion.

I am also immensely grateful to the internal committee members, Prof. Dr. Gabriele Bavota and Prof. Dr. Michele Lanza, for their thoughtful and constructive initial feedback, which helped refine the direction of my work. Additionally, I would like to extend my heartfelt thanks to Prof. Dr. Jordi Cabot and Prof. Dr. Uwe Zdun for serving on my committee and for dedicating their time to reviewing this thesis.

To my family, your love, patience, and constant encouragement have been my anchor throughout this journey. I will forever be grateful.

I also would like to express my heartfelt gratitude to Hassan Atwi, who has been more than a colleague during this journey; he has been a true friend. Thank you, Hassan, for your camaraderie throughout this time. I wish you the very best of luck with your own PhD thesis.

I extend thanks to all the colleagues with whom I has had the pleasure of collaborating during the past few years. I am equally thankful to the old colleagues, including Alejandro, Masiar, Andrea, Ana, and Diego, whose company brought joy and balance to this journey, and to the new colleagues, including Diana, Edoardo, Akshatha, and Shifat, who have recently embarked on their own journeys. I wish you the very best of luck in your endeavors.

To my friends who are also navigating their PhD journeys, including Nouran and Fatima, your hard work and dedication inspire me. I am confident that your perseverance will lead to well-deserved success.

I also want to extend my gratitude to everyone I have crossed paths with throughout this journey. Whether through collaborations, discussions, or even brief encounters, each interaction has left a mark on this chapter of my life. Listing all the names would require several pages, but please know that your presence and support, however big or small, have been deeply appreciated. Thank you for being part of this incredible journey.

This thesis stands as a testament to my efforts and the collective support, guidance, and encouragement of all those who have accompanied me on this path. Thank you.

Contents

Figures

Tables

Part I

Contextualization

Chapter 1

Introduction

1.1 Contextualization

The Growing Complexity and Ubiquity of Web APIs In the contemporary digital landscape, Web Application Programming Interfaces (APIs) have become fundamental to the fabric of modern software development [83]. They serve as the backbone for a myriad of services, ranging from web applications to mobile apps and cloud-based solutions. Rapid expansion and evolution of web technologies have led to a corresponding increase in the complexity and diversity of web APIs [191]. This complexity presents significant challenges in terms of the design, development, and maintenance of these APIs [52]. As such, there is a pressing need for systematic approaches to analyze, understand, and improve the design of Web APIs [53].

From a business perspective, Web APIs allowed new business models to emerge, where providers can sell functionalities or data accesses through endpoints. For example, a large number of brokerage service providers (such as Swissquote) and financial data providers (such as Bloomberg) sell daily access to data through endpoint on financial products to millions of consumers around the globe. Other service providers such as OpenAI and Claude benefit from selling access to large language models by calling an endpoint and sending a data object that includes a prompt, where users pay a monthly subscription or depending on the number of API calls made [27, 186].

From a software architecture perspective, Web APIs play a pivotal role in enabling modularity, scalability, and interoperability within complex software systems. In a microservices architecture, Web APIs are foundational, as they facilitate communication between independently deployable services that encapsulate specific business functions [111, 165, 90]. By using RESTful or gRPC-based APIs, microservices can interact without direct dependencies, promoting loose coupling and enhancing system flexibility and resilience [131, 165]. This decoupling allows for independent scaling, deployment, and maintenance of each service, which aligns with DevOps practices and agile development methodologies, making the architecture highly adaptable to changing requirements [47, 130].

Similarly, in event-driven architectures [103], Web APIs serve as critical conduits for event publication and subscription, where components communicate asynchronously by producing and consuming events [85]. APIs in this context support real-time data streaming and event propagation, enabling reactive systems that can handle high-throughput, low-latency requirements. This architectural style is particularly effective for systems with fluctuating workloads and real-time data needs, such as IoT platforms and e-commerce applications [13]. Furthermore, the role of Web APIs in both microservices and event-driven architectures underscores their importance in achieving distributed system scalability, fault tolerance, and service autonomy, aligning with modern architectural principles of resilience and flexibility [152].

In addition to their roles in microservices and event-driven architectures, Web APIs gave birth to *API-centric architecture*, which emphasizes APIs as the primary structural component and interaction layer within a system.

API-centric architecture adopts an *API-first* approach, prioritizing API design at the outset to ensure consistency, usability, and well-defined contracts for both internal and external consumers. Unlike architectures where APIs are layered on existing functionalities, API-driven architecture integrates APIs as the core framework through which business logic and services are exposed [129].

API-first design is fundamentally supported by Interface Description Languages (IDLs) [1], which provide structured, standardized templates for defining API specifications early in the development lifecycle. By enabling clear, formal descriptions of API structures, data types, and protocols, IDLs facilitate collaboration among developers, stakeholders, and other teams, ensuring that API requirements are well-defined from the start. The history of IDLs dates back to the 1980s when they were first used in distributed systems to outline communication protocols, with DCE RPC or CORBA's Interface Definition Language (IDL) pioneering interoperability between objects in different programming languages in 1991.

As web services evolved, IDLs adapted to meet the needs of RESTful and gRPC-based systems, becoming essential to API-first development practices [138]. Today, several IDLs are widely used in API-driven architectures. The OpenAPI Specification (originally Swagger), introduced in 2011, offers a comprehensive, language-agnostic framework for defining RESTful APIs, supporting automatic code generation, testing, and documentation [118]. RAML (RESTful API Modeling Language) [3], launched in 2013, emphasizes readability and modularity, promoting reusability in collaborative environments. API Blueprint, introduced in 2013 as well, focuses on simplicity and human readability, making it particularly effective for early design and documentation. For gRPC APIs, Protocol Buffers (protobuf) [2], developed by Google and released in 2008, serves as an efficient IDL that defines structured data and communication protocols in a compact binary format, optimizing for high performance.

Among these description languages, the OpenAPI Specification (OAS) has emerged as a leading standard for describing the interface of HTTP-based APIs. Its widespread adoption offers a unique opportunity to mine a large number of descriptions of real-world APIs.

1.2 Thesis Goal

The overall goal of this research is to understand the design of the current web API landscape, identify patterns and recurrent practices, and pinpoint areas where API design and evolution practices meet established guidelines. By analyzing real-world API implementations, this research seeks to provide a foundation of empirical evidence that supports the need for API design and management tools focused not only on continuous integration and deployment, as with existing API management tools [3][4][5], but also on the critical aspects of design and evolution. In addition to identifying these gaps, this research aims to propose prototypes of tooling examples that can be built around Web APIs to help maintain alignment with expectations and best practices. These prototypes would serve as practical solutions, illustrating how API management[6] can extend beyond basic deployment needs to support robust design, high structural quality, and sustainable evolution.

The empirical analysis and tooling proposals of this thesis focus on three aspects of Web APIs:

- **Web API Design**: This involves defining the specific functionalities and data exposure required to meet functional requirements. It requires careful consideration of the API's usability and maintainability. Our goal is to understand how developers implement specific functionalities and whether their implementation choices accurately cover functional requirements.

[1]Interface Description Language. `https://www.omg.org/spec/IDL`
[2]Protocol Buffers. `https://protobuf.dev/`
[3]Kong API Management. https://konghq.com/
[4]Apigee API Management. `https://cloud.google.com/apigee/docs/api-platform/get-started/what-apigee`
[5]AWS API Management. `https://aws.amazon.com/api-gateway/`
[6]API lifecycle management. `https://cloud.google.com/endpoints/docs/openapi/lifecycle-management`

- **Web API Change**: This involves introducing additions to extend the range of functionalities, modifying existing ones, or removing deprecated features. Such changes aim to adapt to new requirements. Our goal is to understand the commonly introduced changes in Web APIs and the order in which they are typically applied to build a knowledge base for expected evolutionary behaviors.

- **Web API Versioning**: Changing an API requires proper versioning to manage its evolution effectively and inform clients about the available options. A consistent versioning strategy fosters the usability and reliability of the API. Our goal is to examine versioning practices in Web APIs, including whether known versioning schemes such as semantic versioning are accurately followed.

1.3 Research Questions

As APIs serve as the primary interface between systems, their design and management directly affect system interoperability, security, and long-term maintainability. Although there has been extensive research on API design principles [124, 100, 23, 184, 29] and individual case studies or evolution [154, 90, 93, 106, 125], comprehensive, large-scale analysis in diverse real-world API ecosystems remains limited.

By examining real-world API designs and usage patterns, we can uncover how APIs evolve, where they commonly diverge from best practices, and the factors that hinder their usability and maintainability. An empirical approach provides a foundation for identifying recurring design flaws and inefficiencies, enabling us to establish data-driven best practices[89]. This insight allows us to advocate for API management tools that prioritize deployment and integration and continuous design validation and improvement, ensuring that APIs remain effective, user-centered, and adaptable in the face of changing demands and technological advancements.

Having an overview of the Web API design space and how instances of these spaces evolve helps to build a foundation for the prediction of the proneness to changes in the Web API [82].

Concretely, this thesis strives to fill part of the existing research gap by answering the following questions.

1. **What identifiable structural patterns within Web APIs can serve as modular and reusable building blocks to accommodate diverse application needs?**

   The goal of this question is to explore how identifying reusable structural patterns, a concept widely used in software design to promote modularity and consistency, can be applied specifically to Web APIs. In software development, patterns serve as proven solutions to recurring design challenges, enabling developers to create scalable and maintainable systems. Similarly, for Web APIs, identifying structural patterns can provide modular building blocks that streamline API design, reduce inconsistencies, and address the diverse needs of applications.

   Chapter 6, describes the approach we followed to identify and create structural building blocks we refer to as "API fragments" by analyzing real-world API structures.

2. **How are Web APIs and their data models interconnected, and to what extent is this relationship considered in designing APIs that align with expected design principles, enhance client developers' experience, and ensure seamless integration [75]?**

   Adhering to established design principles is crucial for creating systems that are intuitive, efficient, and developer-friendly. This is particularly true for Web APIs, where design choices directly impact the experience of client developers and the ease of integration into their applications. By examining how often API designs align with these principles, such as consistent naming conventions, resource exposure, and logical structuring, it is possible to quantify their usability and know to what extent it is required to invest in adequate support design tools.

Chapter 4 presents an analysis of the relationships between API features, focusing on how their structure aligns with the data models of resources they handle. The study evaluates design principles identified by Jin et al.[75] and Lauret[89] as essential for creating APIs that developers find intuitive and user-friendly. The analysis investigates the extent to which real-world APIs adhere to these principles or whether they remain theoretical ideals without practical implementation.

3. **What types of changes are introduced in the evolution of Web APIs and what is their impact on API clients?**

   In the evolution of APIs, changes are inevitable as they adapt to new requirements, technologies, and user demands [97, 96]. Understanding the most common types of changes, such as modifications to endpoints, updates to data models [21], and adjustments to authentication mechanisms, is crucial for assessing their impact on usability and client integration. These changes often reflect efforts to introduce new functionality, improve performance, or enhance security. However, their implementation can create challenges for client developers, particularly when changes are breaking or poorly communicated. By identifying patterns in these changes, it becomes possible to evaluate how to evolve APIs effectively while maintaining backward compatibility and minimizing disruptions.

   Chapter 7 presents a detailed list of fine-grained changes observed in the histories of real-world APIs. This list highlights the modifications that are most likely to occur during the evolution of Web APIs and emphasizes the need to account for these changes when designing APIs. Chapter 8, suggests a visualization that can be used to summarize the changes happening during a specific time frame of the API lifetime in a compact and interactive manner.

4. **What versioning practices are commonly applied in Web APIs, and how do they impact usability, reliability, and client awareness?**

   Versioning practices are essential for managing the evolution of Web APIs and addressing the inevitable changes introduced over time [106, 89]. Clear versioning practices enhance usability by clearly signaling the scope of changes, improve reliability by maintaining predictable behavior across versions, and foster client awareness by communicating when breaking changes requires action [170, 45].

   Chapter 9 presents analysis results showing the commonly used versioning schemes for Web APIs. We extracted from real-world APIs a list of adopted version identifier formats and verified the extent to which practices such as path-based and header-based versioning are implemented in real-world APIs.

5. **To what extent do real-world APIs follow established versioning schemes, such as semantic versioning, and what variations exist?**

   While many APIs claim to use semantic versioning [41], variations often emerge in practice, such as inconsistent handling of breaking changes or adopting custom versioning formats. These deviations can impact the clarity and predictability of API updates, leading to integration challenges for client developers. As for the adherence to API design principles exploring these variations in adherence to versioning scheme helps highlight gaps between theoretical guidelines and practical real-world implementations [116, 101].

   Chapter 10 focuses on Semantic Versioning, identified in Chapter 9 as the most widely adopted format for version identifiers. The analysis investigates the extent to which APIs accurately convey the type of changes introduced in a release through their version identifiers. It quantifies the proportion of APIs that consistently adhere to Semantic Versioning principles across all releases and examines the proportion of individual releases that maintain consistency with semantic versioning rules.

1.4 Research Approach

This research adopts a mining software repositories approach, inspired by methods outlined in works such as Mens et al. [105] and Kagdi [78], to investigate the design, evolution, and versioning practices of Web APIs. Using a large dataset of real-world API repositories, we analyze API specifications and repository metadata to uncover structural patterns, common changes, and versioning practices. The MSR approach allows us to derive insights from historical data and code artifacts, ensuring that the findings are firmly grounded in practice rather than theory.

To ensure wide applicability, this research uses language-agnostic analysis methods, similar to the approaches recommended by Effendi et al. [50] for large-scale analysis of multi-language systems. This enables the identification of reusable patterns and best practices in APIs across diverse programming environments.

1.5 Dissertation Outline

This thesis walks the reader through empirical results drawn from a longitudinal study of thousands of Web APIs, utilizing different analysis approaches, including static analysis and patterns mining. These results show how existing Web APIs are designed and evolve in practice. Our empirical analysis has led to a set of contributions consisting of empirical shreds of evidence, tools, and artifacts.

The analysis and their insights as details in eight chapters grouped into 4 parts :

- **Part 1 - Web API Structural and Datamodel analysis**

  This part of the thesis spans three chapters, addressing the questions about the design and data models of Web APIs.

  The initial chapters present the *Web API Tree*, an intuitive visualization designed to represent the structure of web APIs. This visualization, supported by a tool we call OAS2Tree[7], showcases the branching relationships between paths and their associated operations, parameters, and responses through a chosen graphical notation.

  Based on the *Web API Tree* representation, the second chapter in this part identifies structural patterns within web APIs that can serve as modular, reusable building blocks for diverse application needs. By exploring these patterns, the research highlights their potential to streamline API design, reduce inconsistencies, and promote modularity and maintainability.

  The second chapter in this part investigates the interconnection between Web APIs and their data models, emphasizing how this relationship influences adherence to design principles, the client developers' experience, and seamless integration. The analysis delves into established principles like consistent naming conventions and logical structuring, examining their application in real-world scenarios and assessing whether they remain theoretical ideals or are practically implemented in existing APIs.

- **Part 2 - Web API changes**

  This part focuses on the types of changes Web APIs undergo during their evolution and how these changes impact clients [20]. It examines updates to endpoints, data models, and authentication mechanisms, identifying patterns in these modifications. Chapter 7 provides a detailed list of changes observed in real-world APIs.

  It also introduces visualizations to summarize and communicate these changes effectively. The visualization can be automatically generated due to a CLI tool we implemented, which me name APIcture[8].

[7]Demo Video Link: https://youtu.be/E48c9Rwntz8
[8]APIcture trailer video: https://souhaila-serbout.me/readmore/serbout2023interactively/post

Chapter 8 proposes compact and interactive methods to display API changes over time, helping developers understand their impact and track API evolution.

- **Part 3 - Web API versioning analysis**

  This part analyzes versioning practices in Web APIs and their impact on usability, reliability, and client awareness. It examines how versioning methods, like path-based and header-based approaches, are implemented in real-world APIs. Chapter 9 lists common versioning schemes and evaluates their effectiveness in signaling changes and maintaining predictable API behavior.

  It also explores the adherence to semantic versioning and variations in its application. Chapter 10 evaluates how accurately APIs use version identifiers to reflect changes, identifying gaps between theoretical standards and practical implementations. The findings highlight inconsistencies and suggest ways to improve versioning practices.

- **Part 4 - Conclusions**

  This part summarizes the key contributions of the research by answering the research question raised at the start of this chapter and presents a section on research publications derived from the thesis, showcasing how the findings have been shared with the broader academic communities.

Chapter 2

State of the Art

This chapter presents a review of the state-of-the-art studies on Web APIs in terms of design, analysis, evolution, testing, and documentation. We begin by highlighting research contributions that adopted different approaches to studying Web API characteristics, starting from different types of resources. The sections further explore the research work on understanding the dynamic nature of Web API evolution, examining the impact of API changes on client applications and the strategies for mitigating the challenges associated with frequent updates.

2.1 Background on Web APIs

2.1.1 Web APIs history

Web APIs have evolved significantly since their inception, becoming a cornerstone of modern software development (Figure 2.1). The origins of APIs can be traced back to the early days of software engineering when the need to enable communication between different software systems first emerged [55]. The advent of the World Wide Web in the 1990s introduced the concept of web services, which laid the foundation for the first generation of APIs. These early web services were primarily built using protocols such as SOAP (Simple Object Access Protocol) and XML-RPC, which emphasized structured communication between systems. However, the complexity and overhead of these protocols led to the development of more lightweight alternatives, culminating in the rise of REST (Representational State Transfer), a paradigm introduced by Fielding in his doctoral dissertation [55]. REST provided a simplified model for building APIs by leveraging HTTP protocols, focusing on statelessness, scalability, and resource-oriented architecture, which gained widespread adoption during the early 2000s.

The growth of RESTful APIs coincided with the rise of major technology companies, such as Google, Twitter, and Facebook, which began exposing their core functionalities as public APIs to foster innovation and integration within

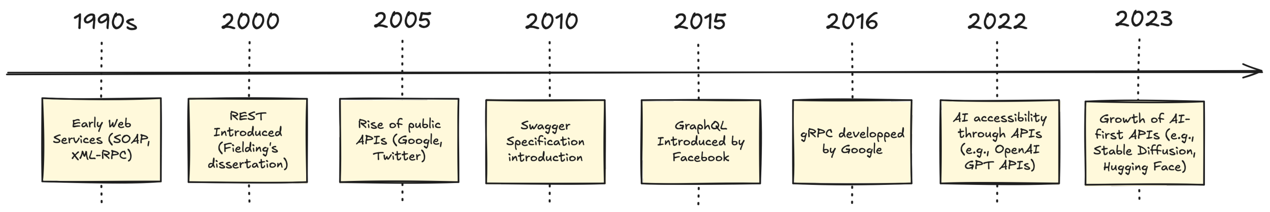

1990s
2000
2005
2010
2015
2016
2022
2023
Early Web Services (SOAP, XML-RPC)
REST Introduced (Fielding's dissertation)
Rise of public APIs (Google, Twitter)
Swagger Specification introduction
GraphQL Introduced by Facebook
gRPC developped by Google
AI accessibility through APIs (e.g., OpenAI GPT APIs)
Growth of AI-first APIs (e.g., Stable Diffusion, Hugging Face)

Figure 2.1. Web APIs history timeline

their ecosystems [83, 117]. This period also marked the emergence of API management platforms and marketplaces, enabling developers to easily discover, integrate, and monetize APIs [104]. In recent years, the proliferation of microservices architectures and cloud computing has further accelerated the adoption of APIs, as they serve as the primary mechanism for inter-service communication. The introduction of the OpenAPI Specification (formerly Swagger) in 2010 standardized the documentation and design of APIs, making them more accessible and easier to adopt [118].

In the last decade, the integration of Web APIs with artificial intelligence (AI) has opened new frontiers for innovation. In 2020, APIs like OpenAI's GPT-3 enabled developers to access advanced AI models through simple API calls, democratizing AI capabilities such as natural language understanding, text generation, and conversational AI [163]. AI-first APIs have since proliferated, with platforms like Hugging Face providing access to pre-trained NLP models and tools, and Stable Diffusion enabling developers to integrate generative AI capabilities with minimal overhead [162]. This shift has fundamentally reshaped industries, as AI-driven APIs allow businesses to scale their AI capabilities without needing deep technical expertise. APIs have become the backbone of AI services, powering subscription-based business models and facilitating the rapid deployment of AI solutions in sectors such as healthcare, finance, and education.

From the early days of SOAP to the modern era of REST and GraphQL Web APIs have evolved to meet the demands of modern applications [67, 36, 69]. Today, APIs are indispensable for connecting systems, enabling interoperability, and driving digital transformation in a world increasingly reliant on AI-powered technologies.

2.1.2 Web API design and modeling: Interface Description Languages

In [38], Davis suggests a set of principles for software design, among which reuse plays a prominent role: software design should not reinvent the wheel, but can instead benefit from combinations of existing proven design patterns. Following this recommendation, in the context of web API design [89], in our research, we focus on supporting and investigating the modeling phase of web API creation. The first goal of our analysis is to assess the most frequent design decisions made by developers to create their REST APIs [110]. In other words, we want to empirically detect API design patterns [189] based on how current real-world APIs are designed. We want to achieve a fully automated, systematic, and programming language-independent approach for capturing design decisions made by developers in conceiving their Web APIs, depending on the API's domain. For that, we planned to exploit API documentation writing in machine-readable Interface Description Languages (IDLs) [61].

Before the appearance of adequate description languages, REST APIs were modeled using informal notation or natural language, informal models, or general-purpose modeling languages. This lack of machine-readable description formats at the beginning of the 21st century urged the birth of some domain-specific languages for describing web APIs over different aspects, such as RAML [3], WADL [62], WSDL [32], I/O Docs [4], and OpenAPI [118] (Table 2.1), which gained more importance in the five last years, by being selected as a standard language for APIs description. Moreover, thousands of these description documents can be easily found by crawling software repositories shared in collaborative platforms, such as GitHub. In December 2020, we started to crawl open-source software repositories looking for OpenAPI descriptions, assessing the feasibility of a large-scale analysis on API descriptions, having the purpose of detecting recurring design decisions made by Web API developers. By classifying the gathered descriptions by the year of the last commit, we noticed a clear increment in the number of OAS actively updated each year (Figure 6.3), which is promising for further empirical research on API analytics. Previous research work has exploited the granularity descriptions of the functional characteristics of Web APIs, written in OAS, for purposes other than analytics, such as automatically generating Web frontends [84], and generating test cases for REST APIs [48]. On the other hand, other works tried to extract formal API descriptions [**?**] written in RAML or OAS from informal API descriptions web pages. Another work that intended to infer structured API descriptions is [160], where the author combined the use of both API descriptions written manually by service providers and the usage data of the API.

Having modeled the representation of Web API induces the feasibility of studies using Model-Driven Engineering (MDE) techniques. MDE is a software development paradigm that advocates the use of models as active elements in the development cycle [136, 46]. Such models can be created with General Purpose Modelling Languages (e.g., UML) or using a Domain-Specific Language (DSL), such as the ones in Table 2.1 in the case of Web APIs.

At the same time, AI and ML have shown their potential to enhance software engineering approaches in many areas [11, 14, 181, 150], but their applications for addressing tasks in the modeling domain are still relatively recent. For instance, in [112], neural networks are used to classify meta-models into application domains. In particular, clustering techniques have shown their usefulness in organizing collections of models [16, 17], and graph kernels have been proposed as a means to characterize similar models [33].

Most ML applications in MDE tackle tasks related to classifying models in a supervised manner [112] or in an unsupervised manner [15, 16, 17] and only very few ones are designed to assist the modeler in modeling tasks. In the case of Web API design, such modeling support tools are non-existent.

IDL	Appearance Year	Last Update Year	Available Versions	Relative Popularity
WSDL	September 2000	June 2007	1.1, 2.0	1.7M
WADL	August 2009	July 2018	2009	371k
RAML	September 2013	May 2016	0.8, 1.0	6.6M
API Blueprint	April 2013	June 2015	1A	15.7K
OpenAPI (Swagger)	July 2011	February 2021	2.0, 3.0, 3.1	7.6M + 10.4M

Table 2.1. Overview of Web API Description Languages with Relative Popularity

Table 2.1 summarizes key Web API Description Languages (IDLs), including their appearance years, last update years, available versions, and relative popularity based on the number of GitHub files referencing their names.

The OpenAPI Specification dominates in popularity, with `7.6M` files referencing "Swagger" and `10.4M` referencing "OpenAPI" This reflects its widespread adoption and the transition from Swagger to OpenAPI as the preferred standard for API documentation.

RAML follows with `6.6M` references, indicating its niche adoption for reusable API definitions. **WSDL**, with `1.7M` files, remains significant in legacy SOAP-based systems. In contrast, **WADL** and **API Blueprint** show limited adoption, with `371k` and `15.7k` files, respectively, reflecting their declining relevance.

2.2 Web APIs Analysis

2.2.1 Analyzing Web API functional features through documentation

The existence of several approaches to document Web APIs in human and machine-readable formats leads to diverse approaches for their exploitation to produce comprehensive descriptive insights. In 2010, Maleshkova et al. adopted in [99] a manual examination approach to study 222 Web API descriptions from the ProgrammableWeb directory. This study focused on technical aspects such as general API information, input parameters, output formats, and invocation details. It revealed that 61% of the APIs were not used in any mashups, and only 12% achieved high popularity, like the Flickr API with 506 mashups. The data used in this study contains mainly information about API mashups and resources such as SDKs, libraries, and external documentation. It contains also documentation of the API operations. However, these latter are not structured in a machine-readable way, which makes it difficult to be exploited for API structure and data model analysis. Haupt et al. performed analysis on API structures in [63, 64] starting from machine readable API descriptions. Unlike [99], the authors of [63, 64] built a systematic approach based on Swagger and RAML. Their analysis involves transforming the API's Swagger model into a defined canonical metamodel that depicts the elements of the resources handled by an API and the methods it provides for each resource. The metamodel also describes the relationships between the resources (navigation or creation), which are not describable using the Swagger and RAML IDLs, but the authors' approach infers these relationships based on the hierarchical relationships between the paths. However, this work considers Swagger and OAS as distinct IDLs, although they can be seen as different versions of the same language. The study presented in [63, 64] is based on metrics related to the API structure, such as the size (number of resources), number of read-only resources (around 24.5% of all APIs have a share of read-only resources between 90% and 100%), number of POST and DELETE operations, number of root resources, number of links, and the number of components and their size. The authors also defined metrics to measure the user-perceived API complexity, describing how a user judges the complexity of the API without objectively measuring it. They performed an experiment with nine API developers, applying the Analytic Hierarchy Process (AHP)[135] by carrying out a pairwise comparison of ten APIs out of the 286 selected ones. Participants indicated how much one API is more complex than the other using a scale (as defined by Saaty[135]) ranging from 1 (equally complex) to 9 (extremely more complex).

While [63, 64] and [99] picked their APIs from documentation platforms, the authors of [110] chose to pick APIs from popular hosting websites by examining the Alexa.com top 4000 most popular sites to identify 500 websites claiming to provide a REST web service API. This study's main goal was to assess these APIs' compliance with REST best practices, finding that only 0.8% of the selected services strictly comply with all REST design constraints[55]. Similar to [99], the authors performed a manual analysis of Web APIs, starting from their documentation, against 26 technical features, mostly collected from the literature (e.g., HTTP verb support, design schema used in URIs, versioning use).

Unlike [64, 99] and [110], who focused more on metrics extraction and providing static results about the sample under study, the authors of [120] performed an in-depth analysis to detect five REST design patterns and eight anti-patterns. To do so, they defined and implemented detection heuristics. Other studies like [166, 175, 174, 31, 59] have also analyzed Web APIs on different levels, such as usability, performance, and quality. However, none of them performed systematic and quantitative studies to discover and abstract common structural design decisions adopted by Web API creators.

While most of the main existing studies on Web API structure analysis have focused on API documentation rather than their actual implementations, none have applied their approaches to a large sample of API documentation, utilizing repository mining techniques to collect machine-readable specifications.

2.2.2 Analyzing Web API features changes over time

Web API changes analysis and classification

A key motivation for our focus on analyzing Web API evolution is the scarcity of studies that have explored and identified the types of changes and evolution patterns in Web APIs. An API change can have a specific impact, potentially breaking backward compatibility or maintaining it. Although some studies have examined the impact of Web API changes and their alignment with versioning strategies, there is a lack of large-scale research that comprehensively explores these changes and how they are reflected in versioning practices.

In a recent study performed by Koci et al. in [82], the authors explored the types of changes in Web APIs by analyzing the DHIS2 API. However, they have only compared two consecutive versions of one API controller to identify changes and examine four key artifacts: release notes, API documentation, issue trackers, and versioning systems. Their approach involves categorizing changes by their types and causes, and assessing the impact on API consumers through usage logs. They identified 38 changes, including 19 new parameters, 10 new endpoints, 5 removed endpoints, 2 new authorities, 1 changed authority, and 1 supported request method. Over 75% of these changes were non-breaking, which explains why consumers often delay upgrading. The study found less than 50% of changes were documented, with only 17 out of 38 reflected in at least one artifact, and only 5% documented in all artifacts. Usage logs showed that only 10 out of 38 changes were adopted by consumers. The main limitation of this work lies in its exclusive focus on an internally implemented API, which is largely due to the difficulty of accessing the code or the usage logs for multiple real-world APIs. Additionally, the approach used in this study lacks reusability, even if other API codes were accessible, as it relies on manual code analysis rather than a systematic method.

In 2015, Sohan et al. [154] have also focused on how API changes are communicated as the APIs evolved. The authors analyzed nine web APIs (Facebook, Twitter, WordPress, Salesforce, Google Calendar, Stripe, GitHub, Google Maps, OpenStreetMap), through 18 randomly sampled change logs of these nine APIs, and found 114 moved, 31 renamed, and 247 behavior changes across releases (e.g., Facebook's 11, Stripe's 30, Google Maps' 102). Most releases broke backward compatibility, highlighting the need for semantic versioning. The study stresses the importance of separate releases for bug fixes and new features and calls for automated, version-aware documentation tools. Communication of changes often relied on unstructured text, suggesting a need for more structured methods for logging and tracking API changes. The authors looked mostly at the official API websites to see how the changes are communicated and how new releases are announced to the community. Nowadays, many providers such as Meta, OpenAI, and GitHub use tools such as Discord to push notifications and share change logs.

Focusing on the impact of the introduced changes, in 2013 Li et al. [93] examined client programs of five popular web APIs: Google Calendar, Google Gadgets, Amazon MWS, Twitter, and Sina Weibo. The study analyzed a total of 256 API changes across these APIs, finding that more than half of the old API elements became incompatible with the new versions. Specifically, the study identified 16 change patterns, 12 of which caused compile time errors and 4 of which caused runtime errors. The authors discovered that web APIs change more frequently at the wrapper library level than at the HTTP level. They identified six new challenges unique to web API migration, such as transformations between JSON and XML and handling authorization protocol changes.

The impact of API changes on developers

The changes introduced in an API have a direct impact on their existing clients but also on the developers who consume the APIs for their clients. Starting from this perspective, researchers have also investigated the challenges faced by developers due to the evolution of Web APIs through interviews rather than only analyzing the client software [52, 53, 171, 70]. Through semi-structured interviews with six developers and analysis of major Web API providers (Twitter, Google Maps, Facebook, Netflix), Espinha et al.[52] uncover significant distress caused by frequent and often unannounced changes, forcing developers to adapt their code continuously. Their research highlights the impact of API changes on client software and identifies best practices for API evolution, emphasizing the need for better communication and stability from API providers. Extending their work in [53], Espinha et al. provided a deeper qualitative analysis, including the server-side and client-side evolution of open-source APIs (VirtualBox and XBMC), and underscore the maintenance burden imposed by frequent API changes.

API Changes and their consistency with versioning

Other studies have examined the correlation between software changes and versioning in package manager tools [128, 182, 116]. Ochoa et al.[116] replicated Raemaekers et al.[128] and found 83.4% of package upgrades comply with semantic versioning, but 20.1% of non-major releases introduce backward incompatible changes. Non-major breaking releases decreased from 67.7% in 2005 to 16.0% in 2018. Zhang et al.[182] studied 180 real-world examples to understand breaking issues in compatible version upgrades, identifying probabilities for changes (e.g., additional parameters) causing incompatibility. Li et al.[95] analyzed GoLang's ecosystem, using GoSVI to detect breaking changes in 124K libraries and 532K client programs, finding 86.3% compliance with semantic versioning but 28.6% of non-major upgrades introducing breaking changes. They noted that 33.3% of client programs could be affected.

identified, including 1132 breaking changes, of which 424 required manual intervention. The study involved collecting and examining version histories, change logs, and migration practices from various companies to provide a comprehensive understanding of the impact of API evolution. This study highlights the need for systematic approaches to mitigate the impact of API changes on client applications and provides insights into effective strategies for managing API evolution [90].

Lercher et al. [90] explores the evolution strategies and challenges of microservice APIs. Conducted through 17 semi-structured interviews with developers, architects, and managers across 11 companies, the study used grounded theory for analysis. Key findings include six evolution strategies such as maintaining backward compatibility, versioning, collaboration with consumer teams, designing flexible APIs, and extensive regression testing. For instance, all participants (17/17) emphasized backward compatibility and versioning, with many (13/17) maintaining multiple API versions simultaneously. The study identified six primary challenges, including impact analysis difficulties and consumer reliance on outdated API versions. Specifically, 14 out of 17 participants found manual change impact analysis error-prone, and 12 out of 17 reported issues with consumers relying on old versions, leading to maintenance overhead. The study highlighted two major problems: tight organizational coupling and consumer lock-in, suggesting mitigation through automated change impact analysis and improved communication.

2.2.5 API changes across language ecosystems

Instead of targeting web APIs, studies on the impact of API changes have focused on specific ecosystems [87], such as Pharo. Hora et al. [70] analyzed six years of evolution data, including 3,588 client systems and 2,874 contributors, to understand the propagation of API changes. They mined **344 API changes** and focused on 118 significant changes, assessing the reactions of client systems. Spanning versions 1.0, 1.4, 2.0, and 3.0 of Pharo Core, they excluded deprecations, resulting in 59 method suggestions and 59 method replacements. They found that 53% of API changes caused reactions in at least one client system, involving 178 distinct client systems and 134 developers. Method replacements were followed more consistently than method suggestions. Earlier API changes had a longer adaptation time, with a median of 284 days compared to 18 days for later changes.

Cossette and Walker [35] analyzed the evolution and migration of five Java-based APIs: Apache Struts, log4j, jDOM, DBCP, and SLF4J. They manually investigated **697 binary incompatible changes** (BIs) across these libraries to determine adaptive changes for migrating client code. They found no single change recommendation technique identified the correct replacement for more than one or two changes, with an average success rate of 20% per technique. Hybrid techniques that combine multiple methods may be more effective. Most API changes were not documented in release notes or source code comments, highlighting the need for better documentation.

In Summary

Most studies on Web APIs are limited by small datasets and manual methods, which reduces scalability and generalizability. Automated large-scale analyses of API changes, their impact on usability, and alignment with best practices remain underexplored. Critical gaps include understanding the link between API design and domain requirements, the effects of changes on developers and clients, and real-world adherence to semantic versioning. This research addresses these gaps by employing automated, scalable and language-agnostic methods to analyze large datasets of web APIs.

One of our key objectives is to address the existing gap in understanding how Web APIs evolve and how this evolution aligns with the type of release. Similar to the analyses conducted by Raemaekers et al. [128], Ochoa et al. [116], and Zhang et al. [182] for software packages, we aim to assess the compliance of APIs with versioning schemes such as semantic versioning.

2.2.3 Mitigating and facing web API evolution challenges

Addressing the challenges of web API evolution has become increasingly important due to the potential for unexpected impacts. Schmiedmayer et al. proposed an approach aimed at reducing the impact of breaking changes in web service APIs, with a particular focus on mobile applications. They present an automated process that utilizes type-independent evolution patterns to generate stable client libraries, thereby significantly reducing the manual effort required for client-specific migrations. This approach was validated using 13 web service version increments, demonstrating its effectiveness in adapting to API changes more seamlessly and improving overall software stability [137].

To prevent potential breakages when updating to a new version of a third-party API, it is essential to communicate changes clearly, allowing developers to make informed decisions about the impact of the update [107, 74]. A common practice for maintaining backward compatibility is to keep older versions operational while notifying clients of their eventual deprecation. Lübke et al.[98] outlines this practice, along with seven other patterns, to effectively manage API evolution, tackling challenges like compatibility maintenance and the introduction of new features. Key patterns include Semantic Versioning (a three-number versioning scheme), Two in Production (supporting two API versions simultaneously), and Limited Lifetime Guarantee (providing a fixed support duration). The study emphasizes the importance of strategic versioning and API descriptions, advocating for the use of both minimal and elaborate descriptions to balance clarity and ease of maintenance. To examine the practical implementation of such practices, Yasmin et al. [179] focused on the deprecation mechanism [26], often adopted with Two in Production and Limited Lifetime Guarantee pattern defined in [98]. They develop the RADA framework to analyze deprecated API elements and their impact on client operations. Their empirical study on 1 368 RESTful APIs reveals several issues in current deprecation practices, such as only 33% of APIs following the deprecated-removed protocol and insufficient deprecation-related information. These findings underscore the need for better deprecation practices to mitigate the negative impact on client applications and provide insights for future improvements [179]

Refactoring is a key driver of API changes, offering a chance to improve the API's structure and performance. However, it also presents a risk, as these changes can inadvertently introduce bugs if not managed carefully [19]. To address specific refactoring-related challenges, Stocker and Zimmermann[157] propose a structured method for managing API changes through an Interface Refactoring Catalog (IRC). This catalog includes 22 refactorings designed to manage changes in API endpoints and operations effectively. Eleven of these patterns leverage existing API design patterns, while the remaining focus is on renaming API elements and managing architectural changes.

2.2.4 APIs evolution in microservices-based architectures

APIs play a key role in microservices-based architectures [91, 123] and the operation of enterprise software systems [76], which are often based on heterogeneous IT infrastructures. This type of architecture allows the different parts of the systems to evolve with different speeds [108]. At the same time, the API connecting them should also be updated to ensure continued compatibility between these software components. In this direction, Knoche and Hasselbring [79] presents an approach for interface evolution that is easy to use for developers, and also addresses typical challenges of heterogeneous enterprise software, especially legacy system integration.

To understand the challenges faced, Lercher et al.[90] conducted an empirical study on microservice API evolution, focusing on maintaining backward compatibility and managing frequent changes. They analyzed real-world scenarios involving 3896 API changes across several microservices, revealing that 86.1% of changes were correctly

Chapter 3

A Large Dataset of OpenAPI Specifications

This thesis is grounded in data-driven analysis to uncover insights about Web API design and evolution. This chapter outlines the methodology employed to collect a large dataset of OpenAPI specifications from multiple sources and examines its implications on the types of Web APIs represented by the collected specifications. Additionally, this chapter introduces a set of metrics designed to quantify various aspects of Web API structures, data models, security configurations, and the natural language descriptions of API elements.

3.1 Web APIs Specifications Dataset Utility

The dataset provides a source for benchmarks for API designers to assess quantitative aspects of their API in relationship to a large collection of API descriptions mined from open source repositories (e.g., as shown in this threshold derivation method [23]). API analytics researchers can also use the dataset to quantitatively observe the state of the practice in API design, study to which extent some known API design patterns (or smells) are adopted in practice [191], select and extract smaller samples based on quantitative and domain-specific criteria for further study, identify new indicators based on the given raw metrics that can help detect outliers or cluster similar API designs, making it possible to assess and manage the quality of entire API landscapes [104]. By including artifacts obtained from different sources, the dataset includes APIs at different stages of their development lifecycle: from early API sketches found in GitHub repositories to mature APIs deployed in production by major service providers. Thanks to the corresponding interactive exploration tool, researchers to search and filter our comprehensive API analytics dataset for empirical studies and pattern mining. We aim to regularly update this public dataset with new features.

To demonstrate the dataset's utility, this thesis includes a descriptive analysis of key API metrics, comparing artifacts from various sources. We address questions about the variability of the metrics over time and across datasets. For example, we observe the usage of specific HTTP methods and API security features as well as measure the readability of natural language documentation.

The rest of this thesis exploits portions of this datasets to perform in-depth analysis on Web API design And evolution.

3.2 Data Collection Approaches

To collect OpenAPI specifications a few potential data sources can be targeted:

- **Documentation Pages**: The OpenAPI specification is often hosted directly on the documentation pages of API providers, allowing users to access detailed information about the API.

- **GitHub**: Many API providers use GitHub to share and maintain their OpenAPI specifications, offering version control and collaboration features. When API providers do not offer machine-readable documentation (e.g., Google), independent developers often create and maintain GitHub repositories to host OpenAPI documentation for popular APIs.

- **APIs.guru**: A community-driven initiative that aggregates and standardizes OpenAPI definitions for widely used APIs. APIs.guru bridges the gap between incomplete or non-existent documentation and the need for standardized machine-readable formats, enabling developers to integrate APIs more efficiently.

- **SwaggerHub**: A platform that provides a centralized hub for designing, documenting, and collaborating on OpenAPI specifications. SwaggerHub offers tools for creating and maintaining machine-readable documentation, enhancing collaboration for teams managing API ecosystems.

Targeting non-uniform data sources is not straightforward. We discarded scanning the web for API documentation pages because of the high complexity and resource requirements involved in identifying and parsing machine-readable specifications from unstructured documentation. Additionally, such an approach often yields incomplete or outdated results, as many API providers do not adhere to consistent formats or standards for publishing their documentation. Instead, in addition to APIs.guru, we focused on repositories and platforms that explicitly host or aggregate OpenAPI specifications, such as GitHub and SwaggerHub, ensuring a more reliable and efficient collection of valid and standardized API definitions.

3.2.1 APIs.guru

To collect the specifications from APIs.guru, we could fetch them using the commands:

```
$ wget https://api.apis.guru/v2/list.json ; cat list.json | jq -r '.[]["versions"][]["swaggerUrl"]' >
    ↪ urls
$ wget -i urls
```

The commands automate the process of downloading OpenAPI specifications listed in the `APIs.guru` directory. The first command uses `wget` to fetch a JSON file (`list.json`) containing a comprehensive list of APIs and their metadata from the `APIs.guru` repository. It then pipes the contents of this file through `jq`, a command-line JSON processor, to extract all Swagger (OpenAPI 2.0) URLs from the `swaggerUrl` fields under the `versions` object of each API entry, saving these URLs into a file named `urls`. The second command utilizes `wget` with the `-i` flag to read the URLs listed in the `urls` file and download all the OpenAPI specification files they point to. We then process these obtained files and store them alongside their metadata (provenance URL and processing date) in a MongoDB collection.

The APIs in this collection have been manually classified into 18 categories where the most common category is related APIs 3.1. This can be explained by the fact that most of the APIs in this collection are Azure, Google, and AWS APIs 3.2.

3.2.2 SwaggerHub

While SwaggerHub does not offer an explicit API to retrieve the API specification, we exploited the API behind the frontend (`https://api.swaggerhub.com/apis/swagger-hub/registry-api/1.0.66`) to retrieve the maximum possible number of specifications using different combinations of the provided filters to build queries that will allow us to fetch new specifications. Because SwaggerHub limits the maximum number of results to be retrieved by a single query to 10 000.

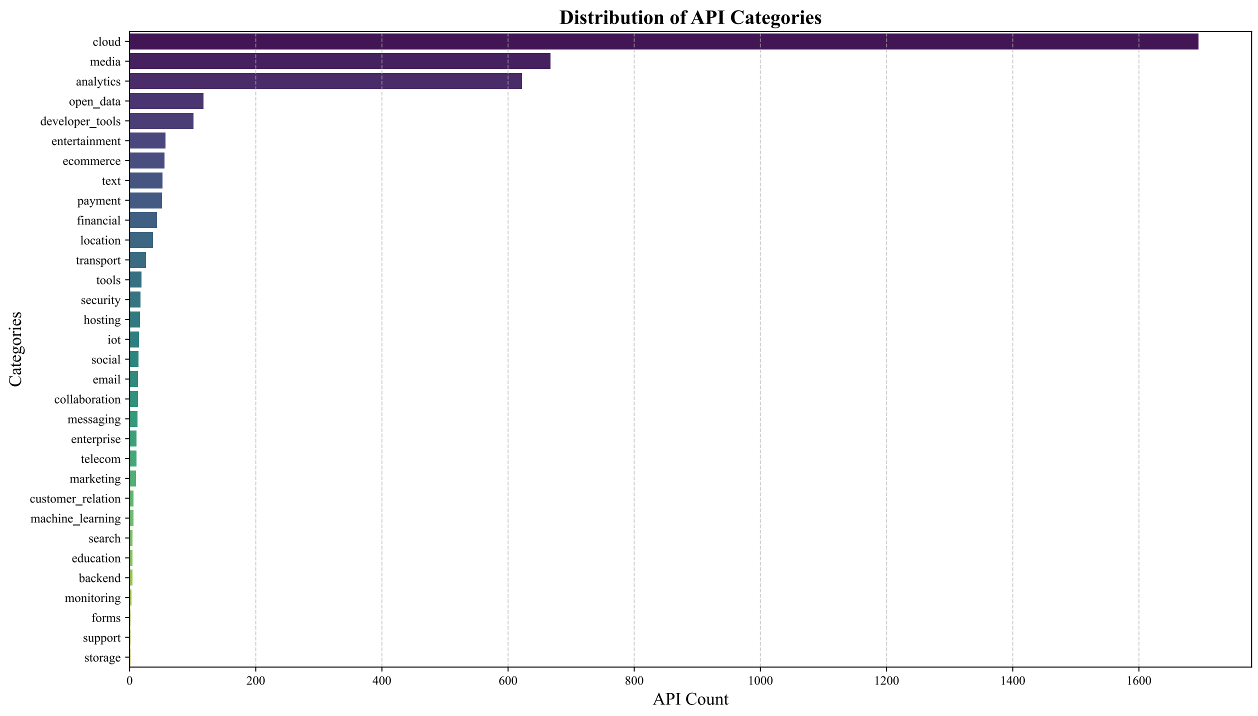


Figure 3.1. Categories of the APIs in the APIsGuru sources collection

For instance, the query below allows retrieving more than 250k results containing URLs pointing to OpenAPI specifications. However, the maximum number of pages that can be read is 99. Queries exceeding this limit return a `500` response code.

```
https://app.swaggerhub.com/apiproxy/specs?specType=API&limit=100&sort=BEST_MATCH&order=ASC
    ↪ &query=inventory&page=0
```

In the query below, I set the number of pages to 100:

```
https://app.swaggerhub.com/apiproxy/specs?specType=API&limit=100&sort=BEST_MATCH&order=ASC
    ↪ &query=inventory&page=100
```

This results in the following error message when attempting to access any page beyond 99:

```
There was an error processing your request. It has been logged (ID: c00d620e1a75f747).
```

Thus, to read all the results, we use different combinations of filter values. The responses from these queries allow for collecting a set of URLs pointing to OpenAPI specifications. These URLs are then placed in a queue for a consumer to process, which fetches the specifications while ensuring that the rate limit is not exceeded. Notably, SwaggerHub does not explicitly describe its rate limit in its documentation. To expedite the download and processing of the specifications, a scheduler ensures that calls from different instances remain under the rate limit [1].

[1]Running the SwaggerHub crawler for a long period is not beneficial and is just energy-consuming since discovering new specifications is faster when the number of undiscovered documents is large. Thus, to save bandwidth we run the crawler only for a few days until it stops making new discoveries.

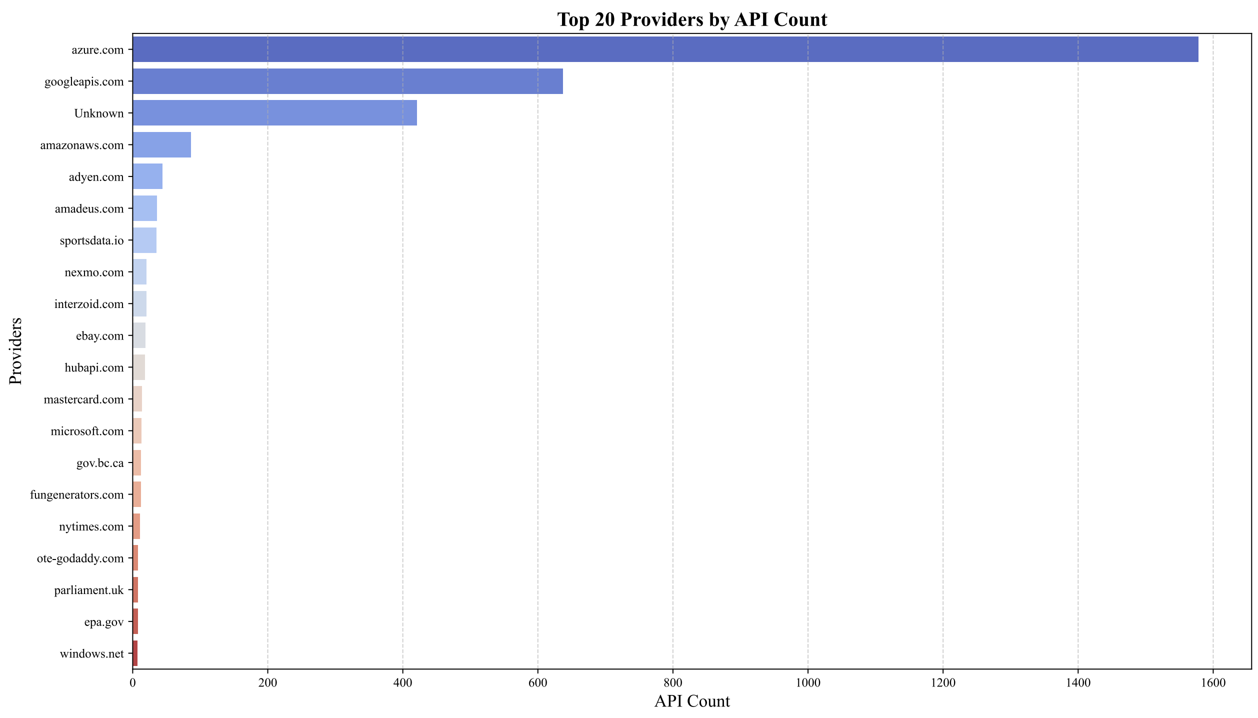


Figure 3.2. Top 20 web API providers in the APIsGuru sourced collection

We also have another mode in which the URL producer calls the **GET** `/api/{owner}` endpoint instead of **GET** `/specs`. When we collect a newly discovered owner, we use it to call all the specifications collected by this owner. A watcher on the collection instantiates a URL producer that calls the **GET** `/api/{owner}` with the newly discovered owner. The watcher is aware of the consumer's owners because they have persisted in a `consumer_owners` collection. These producers are not run in parallel.

In Figure 3.3 we show that the monthly distribution of the created OpenAPI specification is SwaggerHub, classified by the specification version identifier used in the files to refer to the version of the used OpenAPI specification language.

Listing 3.1. Example of MongoDB document corresponding a entry in the database for WILCO API

```
{
 ␣id: ObjectId("65febec31375dedb07465fb9"),
 ␣API␣reference: "https://api.swaggerhub.com/apis/flightwatching/wilco-api",
 ␣name: "WILCO␣API",
 ␣description: "This␣API␣allows␣you␣to␣pull␣and␣push␣data␣with␣your␣WILCO␣deployment\n[https://github.
     ↪ com/flightwatching/wilco-api](https://github.com/flightwatching/wilco-api)␣or␣on␣\n[www.
     ↪ flightwatching.com](www.flightwatching.com).",
 ␣created␣at: "2017-06-07T12:08:30Z",
 ␣last␣modified: "2024-03-21T13:53:27Z",
 ␣created␣by: "",
 ␣API␣url: "https://api.swaggerhub.com/apis/flightwatching/wilco-api/3.0.0",
 ␣version: "3.0.0",
 ␣OPENAPI␣version: "2.0",
  api: {
    swagger: "2.0",
    info: {
      x-errors: [],
      description: "This␣API␣allows␣you␣to␣pull␣and␣push␣data␣with␣your␣WILCO␣deployment\n[https://github
          ↪ .com/flightwatching/wilco-api](https://github.com/flightwatching/wilco-api)␣or␣on␣\n[www.
          ↪ flightwatching.com](www.flightwatching.com).\n",
      version: "3.0.0",
      title: "WILCO␣API",
      termsOfService: "http://www.flightwatching.com/terms/",
      contact: {
        email: "contact@flightwatching.com"
      },
      license: {
        name: "Apache␣2.0",
        url: "http://www.apache.org/licenses/LICENSE-2.0.html"
      }
    },
    ....
  },
  isValid: false
}
```

The structure of the stored document in the collection includes the following fields:

- **_id**: A unique identifier automatically generated for each document in the MongoDB collection. This field ensures each API entry is uniquely identifiable for efficient retrieval and management.
- **_API_reference**: A URL pointing to the SwaggerHub reference for the API. This field provides a direct link to the public API documentation or definition hosted on SwaggerHub.
- **_name**: The name of the API, such as "WILCO API". This field serves as a human-readable identifier for the API.
- **_description**: A textual description of the API, detailing its functionality and purpose. This may include links to related resources or repositories for additional context.
- **_created_at**: The timestamp indicating when the API was first created or added to the database. This helps track the API's creation date for historical or analytical purposes.
- **_last_modified**: The timestamp of the most recent modification to the API entry. This field helps monitor updates to the API's information or specification.
- **_created_by**: The identifier of the entity or user who created the API entry. This field may be left empty if the creator is unknown or unspecified.
- **_API_url**: The URL pointing to the specific version of the API's definition. This field facilitates direct access to the API's OpenAPI specification or other documentation formats.
- **_version**: The version number of the API (e.g., "3.0.0"). This helps distinguish between different versions of the API for versioning and compatibility purposes.

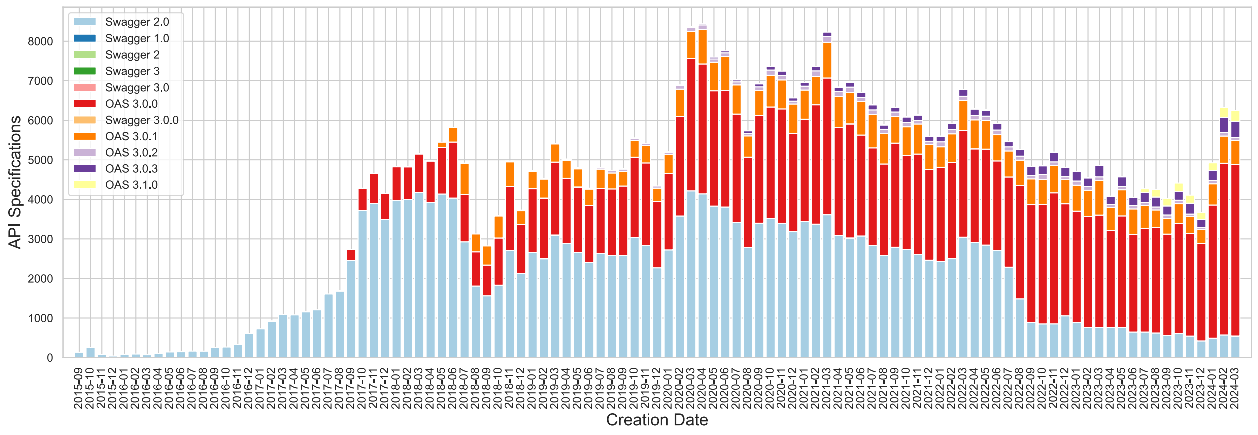


Figure 3.3. Monthly distribution by creation date of valid OpenAPI/Swagger specifications in SwaggerHub dataset

- **_OPENAPI_version**: The version of the OpenAPI (or Swagger) specification used by the API. For instance, "2.0" indicates that the API follows the OpenAPI Specification version 2.0.
- **api**: A nested object containing the parsed OpenAPI specification.
- **isValid**: A boolean flag indicating whether the API's OpenAPI specification is valid. This helps filter APIs that meet specification standards.

3.2.3 GitHub

Unlike SwaggerHub, GitHub explicitly offers a dedicated Search API which allows fetching data through queries sent to the `GET /search` endpoints. This API enforced a rate limit of 5 000 calls per authenticated user. However, identifying OpenAPI files among the 321 million results that GitHub sends back when searching for YAML or JSON files is like looking for a needle in a haystack, as these files are often mixed with countless other configurations, data structures, and unrelated documents. This immense volume of results makes it challenging to isolate OpenAPI specifications efficiently, requiring precise filtering and advanced heuristics to pinpoint the relevant files.

We choose a heuristic to search by file content. However, GitHub only allows access to the first 10 pages, with a maximum of 100 results per page. Thus, we decided to introduce another pagination mechanism where we query with specific ranges of file sizes in bytes. This ensures that we retrieve only a manageable number of results fitting the accessible limit per query.

Listing 3.2. OpenAPI search queries

```
// Define the maximum and minimum file size ranges in bytes
const FILE_SIZE_MIN = 1000; // Minimum file size in bytes
const FILE_SIZE_MAX = 104857600; // Maximum file size in bytes

// Define the queries used for searching
const QUERIES = [
  "language:yaml␣language:json␣info␣title␣paths␣openapi",
  "language:yaml␣language:json␣info␣title␣paths␣swagger",
];

// Example usage: Iterating over file size ranges and constructing queries
for (let size = FILE_SIZE_MIN; size <= FILE_SIZE_MAX; size += 1000) {
  QUERIES.forEach(query => {
    console.log(`${query} size:${size}`);
  });
}
```

The fields: info, title, paths, and openapi or swagger, are required to be present in a valid specification. Thus we use them as a heuristic for the content. 104 857 600 bytes is the maximum file size allowed by GitHub. Once this maximum size is reached, the process restarts from the defined `FILE_SIZE_MIN`. This approach ensures the discovery of newly pushed files. Separate scripts are employed to handle various tasks: one script fetches the history of newly discovered files, while another keeps the history of previously fetched files up to date.

In an ideal scenario, a single query could potentially retrieve up to 1000 OpenAPI Specification (OAS) files, calculated as 10 pages × 100 results per page. However, this is a highly optimistic scenario, because we lower the range of sizes queried files to make sure no files will be missed and not being processed. Additionally, the results from the search API do not include the file contents, which require separate API calls to fetch. Retrieving repository metadata also requires additional calls. Specifically:

- To retrieve 1 000 files and their metadata, 3 000 calls are needed:

  1000 (files) × 3 (calls per file: search, content, metadata)

- To retrieve the commits metadata (without the file content at each commit), another 1 000 calls are required.

☛ Fetching 1 000 potential OpenAPI files content and their metadata (excluding commit contents) consumes a total of 4 000 API calls. This nearly exhausts the maximum rate limit of API calls allowed per hour, making it a resource-intensive process.

Our dataset of OpenAPI specifications sourced from GitHub comprises more than one million artifacts. It is the largest OpenAPI collection that exists in the literature.

Listing 3.3. Example of MongoDB document corresponding a commit of the specifications of a HubSpot API

```
{
  _id: ObjectId("6740f3701b7812026479a289"),
  api_spec_id: 7040,
  sha: "463b8cbc69682c92f60d87ad3324e4992a658ee9",
  raw_url: "https://raw.githubusercontent.com/HubSpot/HubSpot-public-api-spec-collection/68238
      ↪ e49d4635438ef136edc39ac744ef15c5f73/PublicApiSpecs/CRM/Associations/Rollouts/130902/v4/
      ↪ associations.json",
  file_url: "https://www.github.com/HubSpot/HubSpot-public-api-spec-collection/blob/main/
      ↪ PublicApiSpecs/CRM/Associations/Rollouts/130902/v4/associations.json",
  api: {
    openapi: "3.0.1",
    info: Object,
    servers: Array(1),
    tags: Array(3),
    paths: Object,
    components: Object,
    x-hubspot-available-client-libraries: Array(4),
    x-hubspot-product-tier-requirements: Object,
    x-hubspot-documentation-banner: "NONE"
  },
  commit_date: "2024-05-15T08:02:42Z",
  processed_at: "2024-11-22T21:11:12.887+00:00",
  isValid: true,
  api_title: "CRM␣Associations",
  api_version: "v4",
  commits: 1450,
  repoDocId: ObjectId("673f38ba83fd560065c9cd53"),
  commitMetadata: {
    author: Object,
    committer: Object,
    message: "Updating␣spec␣file␣=␣PublicApiSpecs/CRM/Associations/Rollouts/130902/v...",
    tree: {
      url: "https://api.github.com/repos/HubSpot/HubSpot-public-api-spec-collection/...",
      comment_count: 0
    },
    verification: Object
  }
}
```

The structure of the stored document in the collection includes the following fields:

- **_id**: A unique identifier automatically generated for each document in the MongoDB collection. This field ensures each specification is uniquely identifiable for efficient retrieval and management.
- **api_spec_id**: A custom ID assigned to each API. This allows tracking all the commits belonging the the same API.
- **sha**: The SHA hash of the commit that corresponds to the specific version of the OpenAPI specification. This ensures version control and allows linking the specification to its exact state in the GitHub repository.
- **raw_url**: The direct URL to retrieve the raw content of the OpenAPI file. This field facilitates automated processing and downloading of the file for validation or analysis.
- **file_url**: The URL to the GitHub interface for the OpenAPI file. This provides a human-readable reference, allowing users to view or edit the file directly on GitHub.
- **api**: A nested object containing the parsed OpenAPI specification.
- **commit_date**: The timestamp of the commit in which the OpenAPI file was updated. This is crucial for tracking changes over time and understanding the evolution of the specification.
- **processed_at**: The timestamp when the specification was processed and stored in the collection. This helps monitor the freshness of the stored data and identify when updates were last made.
- **is_valid**: A boolean flag indicating whether the OpenAPI specification passed validation. This helps quickly filter valid specifications for further analysis.
- **api_title** and **api_version**: Metadata fields describing the title and version of the API as extracted from the corresponding **api.info** fields.

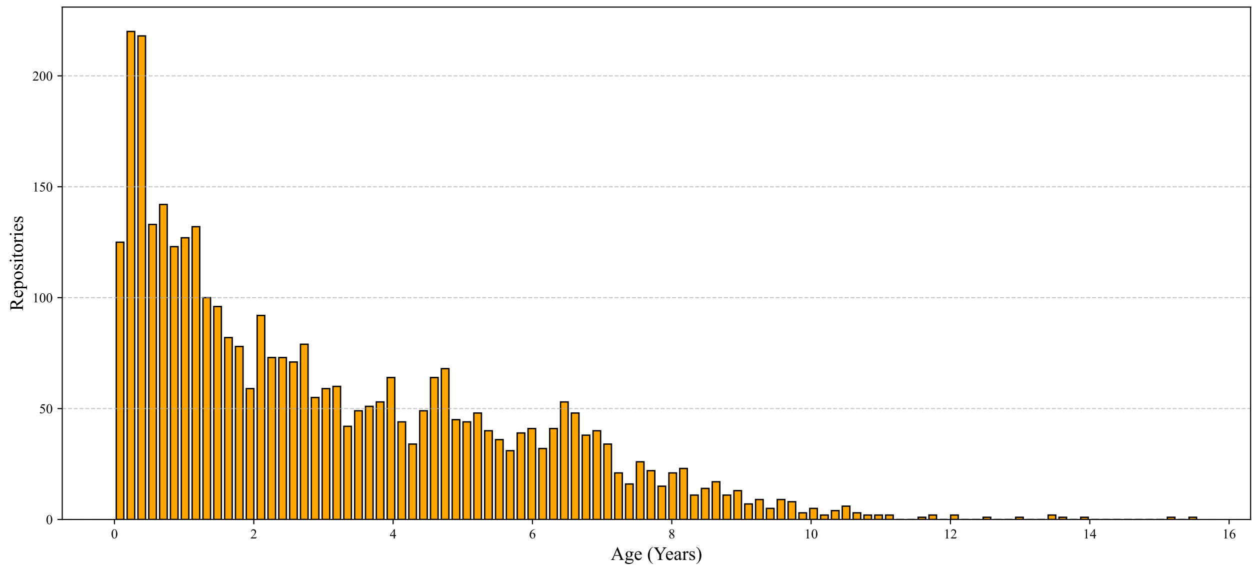


Figure 3.4. Distribution of ages of GitHub repositories to which the OpenAPI specification of the dataset belong. A repository age is computed by subtracting from the 31st of December 2024 the repository creation date

- **commits**: The total number of commits associated with the OpenAPI file.
- **repoDocId**: A reference to another document in the `repos_metadata` collection in database containing metadata about the repository. This allows linking the API specification to its broader project repository context.
- **commitMetadata**: A nested object containing details about the commit, including the `message` describing the change, the `author` and `committer`, and the `tree` object that links to the file hierarchy in the repository. This field provides traceability and context for each version of the specification.

Listing 3.3 shows an example of an entry in the GitHub commit collection.

GitHub Dataset Overview

The collection includes commits of API specifications from repositories with up to 16 years of history. Most repositories are between 1 and 2 years old (Figures 3.4|3.6) and have fewer than 10 stars, although some are starred more than 10 000 times (Figure 3.5). The top 20 most starred repositories are listed in Table 3.1. They include widely know ones such are Kubernetes, Apache and meta-llama.

While the majority of the repositories contain only one API, Figure 3.7 shows that there is a number of repositories that contain specifications of more than one API.

Figures 3.8 and 3.9 shows that a repository's age does not correlate with the number of commits modifying the OpenAPI specification. Similarly, as seen in Figures 3.10 and 3.11, a high number of stars does not necessarily imply frequent API changes.

Figure 3.12 illustrates the monthly distribution of commits in the collection, covering from 2015 to the end of 2024. The noticeable drop towards the end of the timeline is attributed to the methodology used for collecting the specifications. Our approach focused on completing the historical data for the repositories already identified,

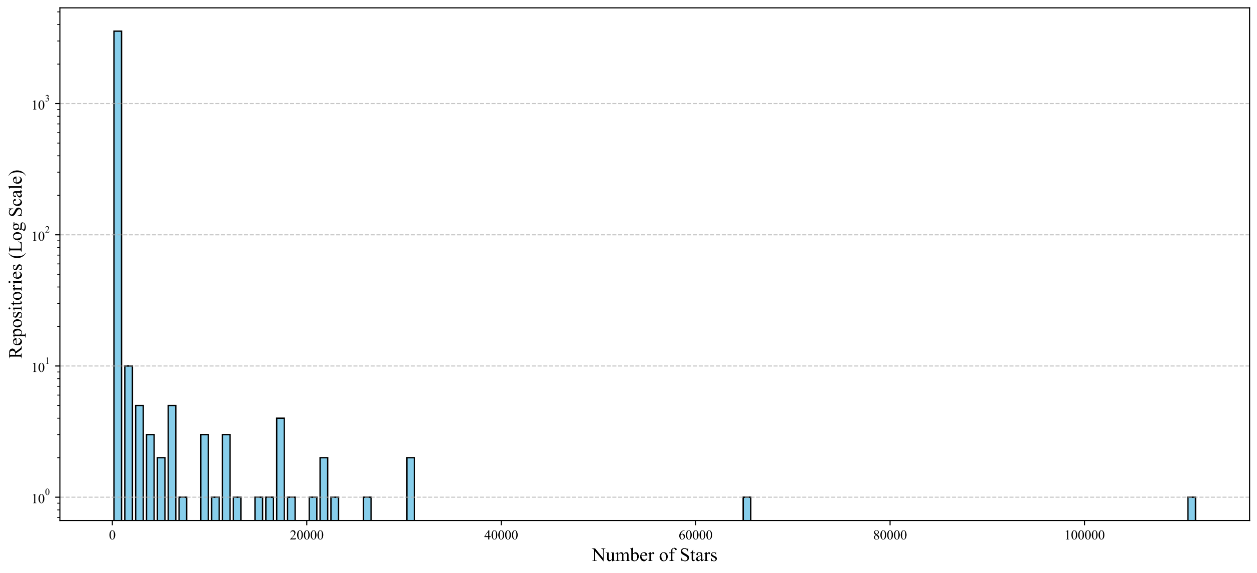


Figure 3.5. Distributions of stars in Github repositories to which the OpenAPI specification of the dataset belong

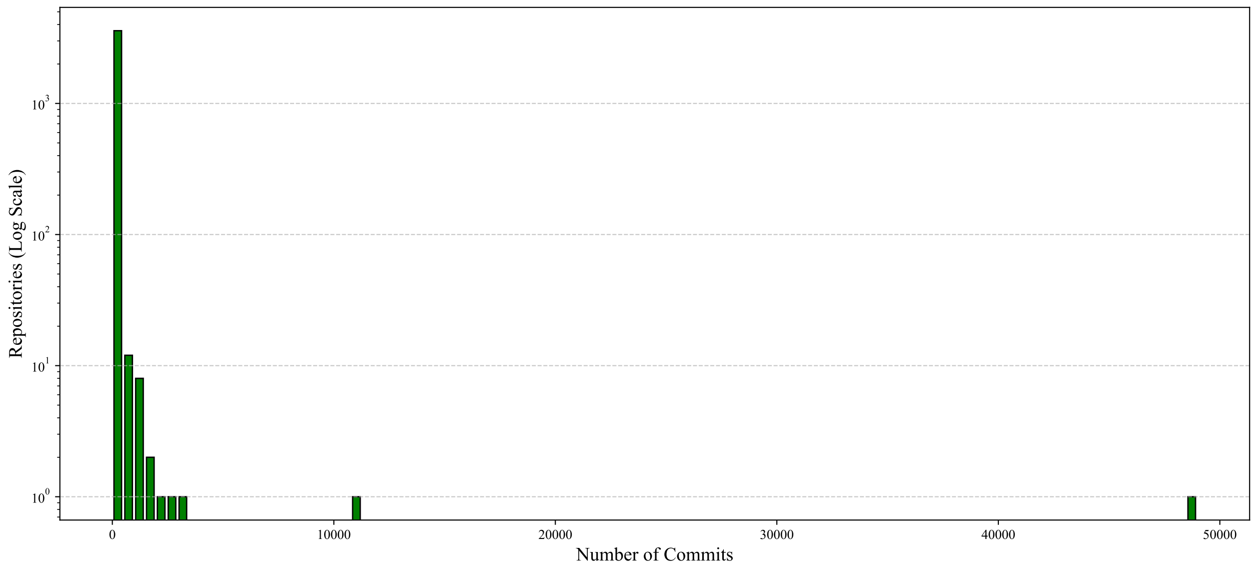


Figure 3.6. Distributions of number of commits from each Github repository to which the OpenAPI specification of the dataset belong.

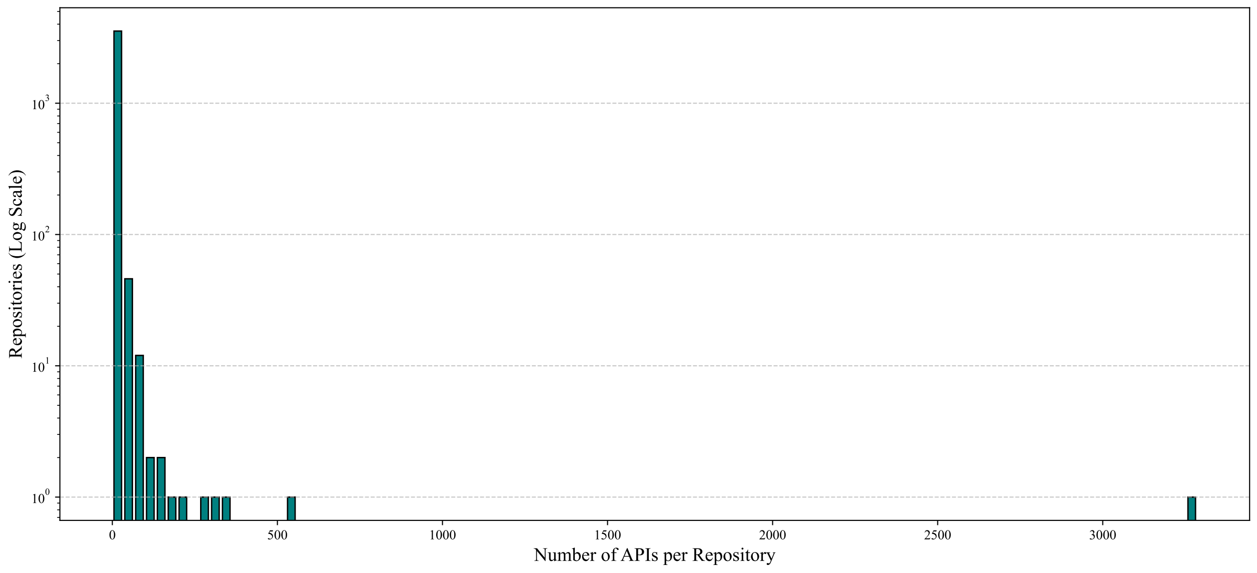


Figure 3.7. Distributions of the number of API per GitHub repository

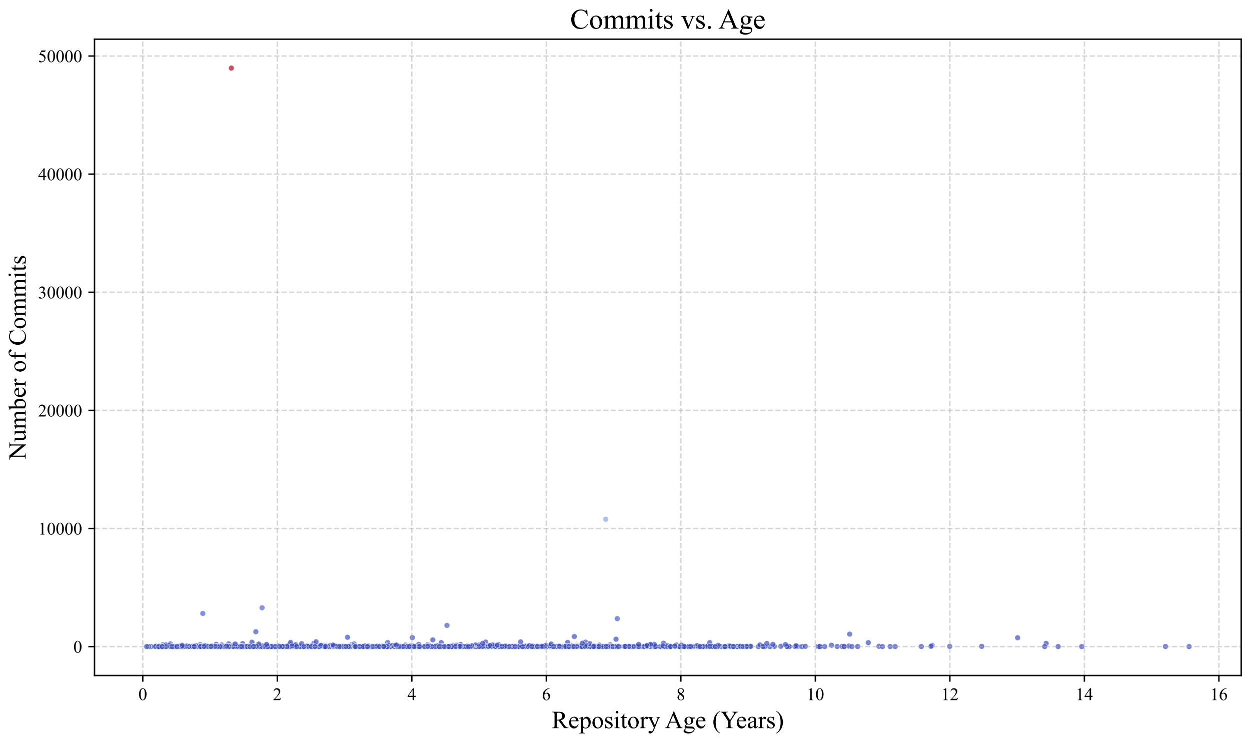


Figure 3.8. Scatter plot showing the correlation between a repository age and the number of commits changing the OpenAPI specification it contains

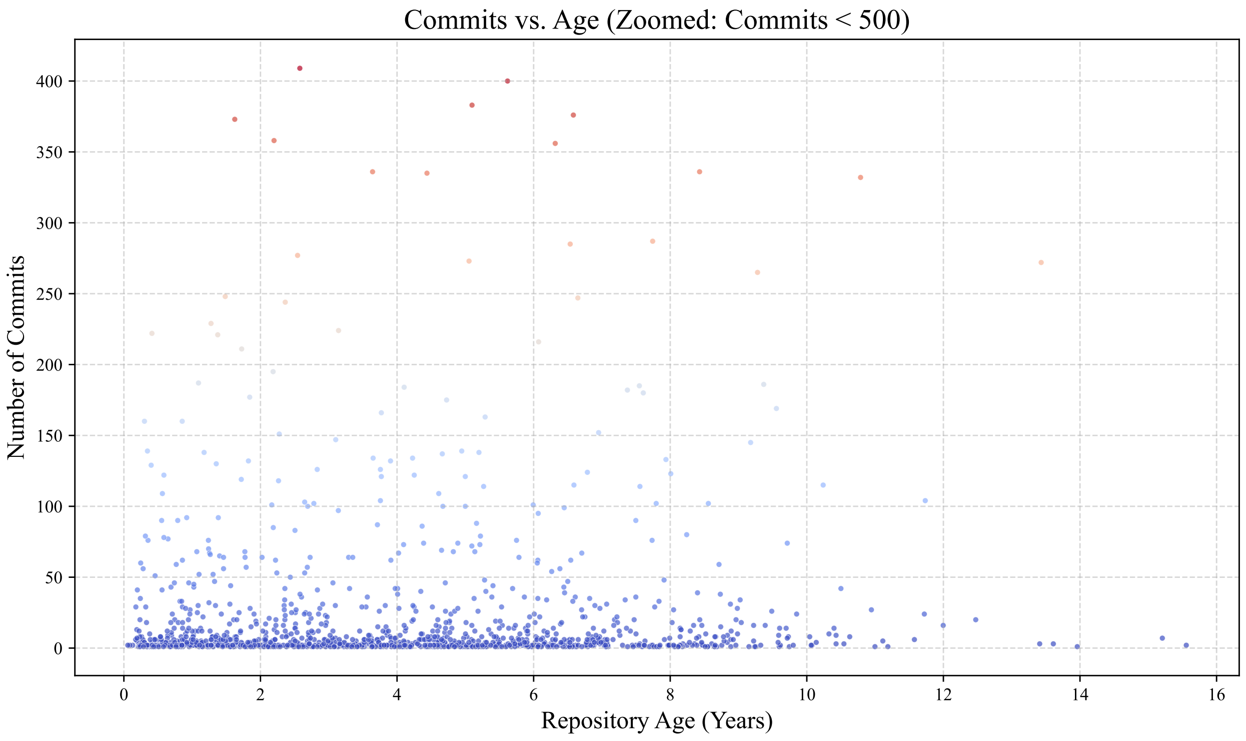


Figure 3.9. Scatter plot showing the correlation between a repository age and the number of commits changing the OpenAPI specification it contains (Zoomed-in: commits<500)

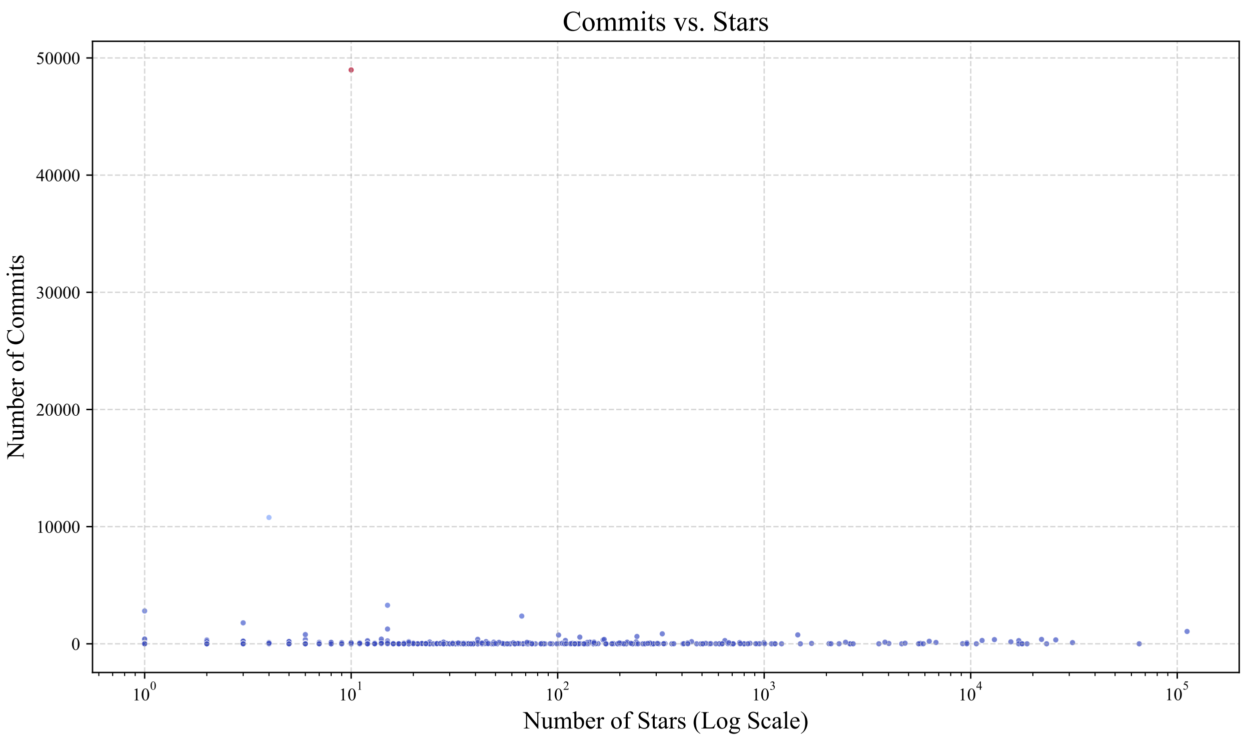


Figure 3.10. Scatter plot showing the correlation between a repository popularity (number of stars) and the number of commits changing the OpenAPI specification it contains

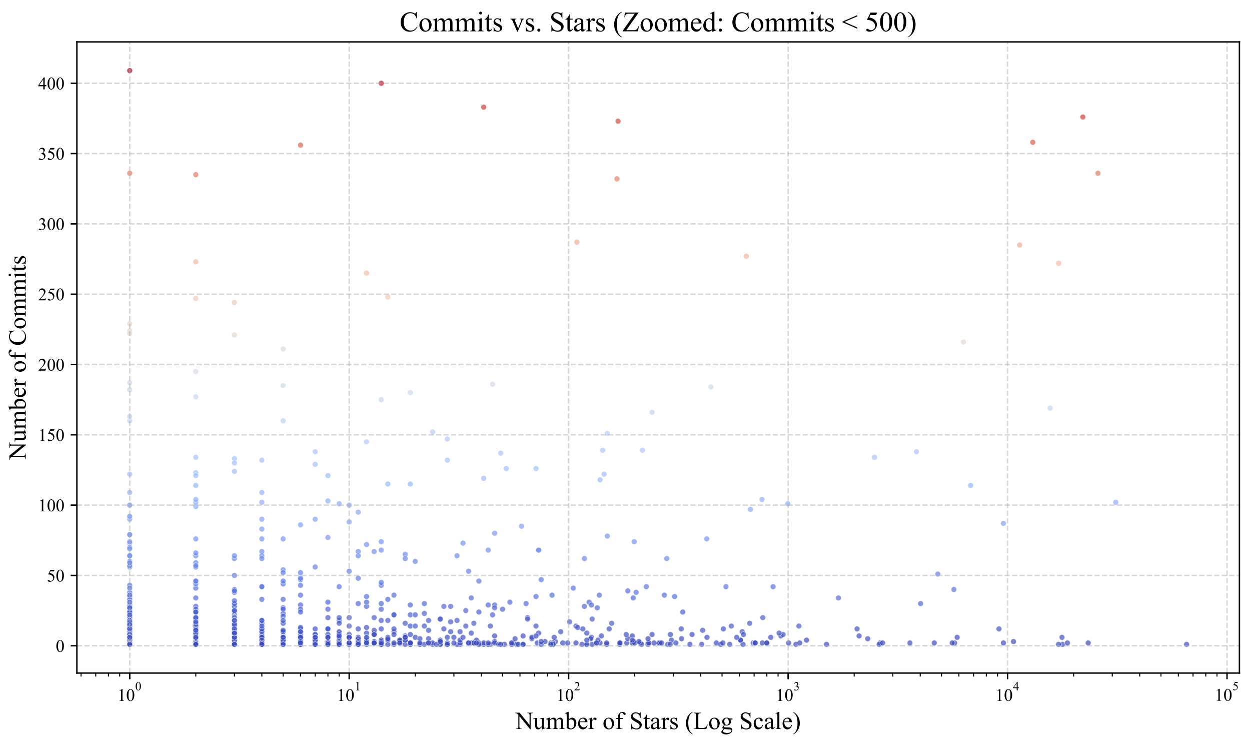


Figure 3.11. Scatter plot showing the correlation between repository popularity (number of stars) and the number of commits changing the OpenAPI specification it contains (Zoomed-in: commits<500)

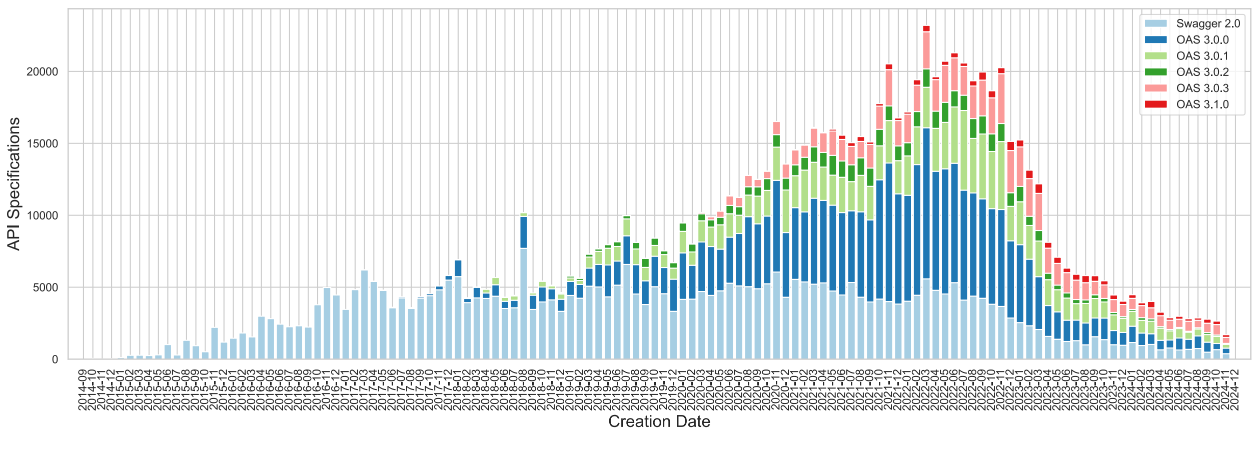


Figure 3.12. Monthly Distribution of Commits of Valid OpenAPI/Swagger Specifications in GitHub Dataset

Table 3.1. Top 20 Most Starred Repositories in which OpenAPI specification in our collection are found

Owner	Repository	Age (Years)	APIs	Commits	Stars
kubernetes	kubernetes	10.51	41	1055	111588
portainer	portainer	8.56	1	102	31113
AdguardTeam	AdGuardHome	8.43	4	336	25809
OpenAPITools	openapi-generator	6.58	58	376	22029
swagger-api	swagger-codegen	13.43	30	272	17089
ory	hydra	9.56	1	169	15644
apache	incubator-answer	2.20	3	358	13025
ory	kratos	6.54	1	285	11351
gravitl	netmaker	3.71	2	87	9576
kubernetes	client-go	8.30	4	12	9136
chaos-mesh	chaos-mesh	5.27	2	114	6789
meshery	meshery	6.07	2	216	6297
xinliangnote	go-gin-api	5.30	2	40	5695
meta-llama	llama-stack	0.46	3	51	4810
xlang-ai	OpenAgents	1.34	30	30	4014
redpanda-data	console	5.20	3	138	3845
frain-dev	convoy	3.65	1	134	2477
open-ani	animeko	2.36	2	12	2061
ArtalkJS	Artalk	6.19	2	34	1696
actiontech	sqle	4.01	2	763	1454

rather than continuously searching for newly created specifications on GitHub. This prioritization comes from the fact that we use the number of commits as a maturity metric to do further filtering. And because of our interest in analyzing Web API evolutions.

In addition to GitHub API, BigQuery [5, 54] allows accessing GitHub archive [6] data using its SQL-like query language which makes it flexible and powerful but has limitations. The data is not real-time, typically lagging by 24–48 hours, and is limited to public repositories, excluding private data. The dataset retains only two years of event history, with simplified payloads and file contents restricted to files under 1 MB. Complex queries can be expensive due to BigQuery's pricing model, and navigating the nested schema requires advanced SQL knowledge. While it is excellent for large-scale aggregate analyses of public metadata, it lacks the granularity and real-time capabilities provided by GitHub's REST and GraphQL APIs, making it unsuitable for live monitoring or highly detailed repository insights.

We received a 5 000$ credits from Google Cloud Research to use for querying data from BigQuery for research purposes, in order to see whether it will allow discovering other files which were not reached using our queries send through the GitHub API.

Listing 3.4. Query used to discover OpenAPI specification in the GitHub archive accessible through BigQuery

```
SELECT f.id, f.repo_name, f.path, f.ref, c.content, c.size
FROM `bigquery-public-data.github_repos.files` AS f
JOIN `bigquery-public-data.github_repos.contents` AS c
   ON f.id = c.id
WHERE ((f.path LIKE '%.json'
  OR f.path LIKE '%.yml'
```

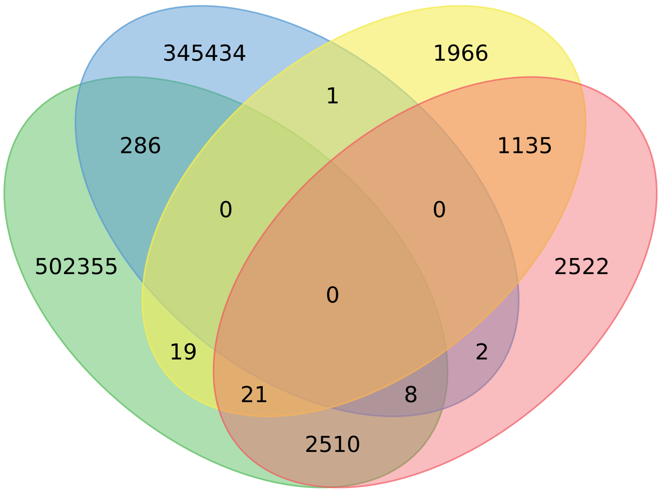


Figure 3.13. Number of overlapping API specifications
🟩 GitHub | 🟦 SwaggerHub | 🟨 APIs.Guru | 🟥 BigQuery

```
        OR f.path LIKE '%.yaml'
    ) AND
    (c.content LIKE '%swagger%' OR c.content LIKE '%openapi%')
    AND (c.content LIKE '%paths%'))
         AND c.content IS NOT NULL
         AND c.content != '';
```

Figure 3.13, shows that indeed searching in BigQuery allowed us to reach other files that were not discovered using GitHub API

3.3 OpenAPI Dataset Metrics

Building upon the static analysis metrics proposed by Haupt et al., Bogner et al. in [63, 24] to study the size and complexity of API structures, the metrics in our dataset share the goal of providing a quantitative assessment of the size of API structures and data models, but also of selected quality attributes of API specifications, including their complexity, readability, versioning and security [30].

Metrics are computed starting from bundled OpenAPI specifications by running custom analytics code. Some metrics can be computed directly by running database queries, while others require parsing and processing the

OpenAPI specifications with custom analytics scripts. The scripts we built to compute the metric and classifications are all available in: `https://github.com/souhailaS/APISTIC-public`.

3.3.1 API structure metrics

By web API structure we refer to the part of the API that includes the endpoint paths, the operation methods, their request parameters, and responses. The API structure metrics evaluate the size of the operational features of the API, providing insights into its functional scope and diversity.
• **Paths**: The number of paths in the API. This metric indicates the breadth of the API's functionality, with each path representing the address of a different communication endpoint, resource or service provided by the API.
• **Operations**: The total count of operations available in the API. This reflects the API's operational capabilities, encompassing all possible actions that can be performed through it.
• **Used Methods**: The number of distinct HTTP methods (GET, POST, PUT, DELETE, etc) used across the API operations. It signifies the diversity in the API's interaction modes.
• **Parametric Operations**: The number of operations that use path or query parameters. This metric helps in understanding the complexity and customization potential of the API operations.
• **Distinct Parameters**: The count of unique parameter names used across the API. It representing the variety of parameters that the API can accept, reflecting its versatility.
• **Used Parameters**: The total number of times parameters are used in the API. This indicates how frequently the API relies on parameterization for its operations.

3.3.2 API data model metrics

The data model refers to the data structures that are exchanged within the messages sent and received by the API, which are also expected to be described in the OpenAPI specification, typically using JSON schemas. This API datamodel of metrics delves into the structure and usage of data models within the API, highlighting the size and complexity of its data representation.
• **Defined Schemas**: To gauge the size of the API data model we count the number of schemas defined in the API description.
• **Distinct Used Schemas**: The total number of distinct schemas that are actually mentioned in API request or response messages. This reflects the overlap between the theoretically provided API data model and the API data model clients can use in practice.
• **Properties**: The total count of properties within those schemas represents the granularity and detail of the data models used.
• **Used Properties**: The number of properties that are explicitly used as part of API request or response messages. It indicates to which extent the data model of the API is usable by API clients.
• **Distinct Used Properties**: The unique property names count indicates the diversity of data attributes the API handles.

3.3.3 API natural language descriptions metrics

These metrics focus on the quality and thoroughness of the API's natural language documentation augmenting its machine-readable, structured description [39]. APIs with extensive natural language descriptions can be considered as high-priority candidates to use as inputs for machine learning models for clustering or classification tasks.

The OpenAPI specifications, which are machine-readable, can contain natural language descriptions in multiple languages. Out of many possible metrics to assess the readability of natural language [34, 153, 10], to quantitatively compare the expressiveness of these descriptions across various languages, we employed two widely recognized, language-independent indices. The Automated Readability Index (ARI) was chosen for its simplicity, easy computation, suitability for a range of texts, and language-agnostic nature. In contrast, the Coleman-Liau Index (CLI) is preferred for technical texts due to its emphasis on letter count, and it is notable for its simplicity, making it easily understandable.

• **Coleman-Liau Index**: A readability index computed based on readability formula designed to gauge the understandability of a text based on its characters per word and sentences per 100 words [34].

CLI is particularly suited for technical documents like API documentation, as it focuses on characters and sentences rather than syllables, which are more challenging to accurately assess in technical language [183].

For each API, we compute the following two metrics:

- **mccphw**: Mean Character Count per Hundred Words.
- **mscphw**: Mean Sentence Count per Hundred Words.

The average mean sentence count per hundred words (*Average mscphw*) is calculated as the total mean sentence count per hundred words for each path divided by the total number of paths. Mathematically, it can be represented as:

$$avg\_mscphw = \frac{\sum(\text{mscphw for each path})}{\text{Total number of paths}} \tag{3.1}$$

Similarly, the average mean character count per hundred words (*Average mccphw*) is computed as the mean character count per hundred words for each path divided by the total number of paths:

$$avg\_mccphw = \frac{\sum(\text{mccphw for each path})}{\text{Total number of paths}} \tag{3.2}$$

The Coleman-Liau index is then calculated using the formula:

$$\text{Index} = (0.0588 \times avg\_mccphw) - (0.296 \times avg\_mscphw) - 15.8 \tag{3.3}$$

• **Automated Readability Index**: A readability index that estimates the understandability of a text based on its character, word, and sentence counts [153]. This index provides an estimate of the US grade level needed to comprehend the text. It is formulated as follows:

$$\text{ARI} = 4.71 \times \frac{\text{characters}}{\text{words}} + 0.5 \times \frac{\text{words}}{\text{sentences}} - 21.43 \tag{3.4}$$

Where the characters, words, and sentences are the average counts per API endpoint.

• **Endpoints Description Coverage**: This is a percentage value indicating the proportion of API endpoints that include a non-empty description. It measures whether and to which extent all API endpoints have been completely documented.

3.3.4 API security metrics

This section assesses the security protocols and strategies implemented in the API, reflecting its overall security posture. OpenAPI provides explicit support for client authentication and authorization schemes such as API keys, the OAuth2 protocol, or HTTP basic authentication. Developers can customize how such schemes are mapped to the HTTP protocol request and response payloads and indicate in which endpoint they are employed.

• **Security Schemes**: The number of security schemes of each type listed in the API security component.

• **Secured Endpoints**: The number of API endpoints which explicitly employ specific security schemes.

Dataset	Max	Min	Average	Median	StDev
GitHub	657	0	15.82	7	24.36
SwaggerHub	2882	0	12.05	3	42.22
APIs.Guru	537	0	19.33	8	44.28
BigQuery	356	0	13.17	4	28.90
Combined	2882	0	14.94	7	29.52

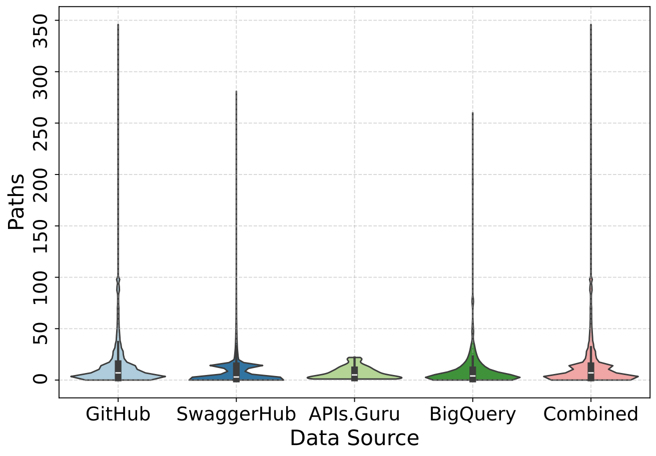


Figure 3.14. Comparative Analysis of Path Number Distributions Across All Sources

3.4 OpenAPI Dataset Exploration

The pre-computation of metrics enables a filtering-based exploration of the dataset. We named this dataset with computed metrics APIstic, which is publicly accessible at: `http://openapi.inf.usi.ch/`. The dataset is accessible through an API that supports a query language closely resembling MongoDB's syntax, allowing users to filter specifications based on combinations of metric values.

The **APIstic dataset** contains OpenAPI specifications gathered from various sources, such as `APIs.guru`, along with pre-computed metrics. These metrics enable users to explore the dataset through filters, providing a structured way to retrieve relevant API specifications. For example, users can filter specifications by specifying a source, such as `"source": "APIs.guru"`, or by applying conditions on metrics, such as `"Endpoints Description Coverage": "$gte": 80`.

To make this exploration more user-friendly, we provide access to the dataset through an API that supports a query language closely modeled after MongoDB's syntax. This query language allows users to define conditions and logical combinations of filters to retrieve data.

Query language overview

The query language supports a variety of operators, such as:

- **$eq**: Matches values equal to the specified value, e.g., `"Title": "$eq": "Pet Store API"`.
- **$gt** and **$gte**: Match values greater than or greater than or equal to a specified value, e.g., `"Operations": "$gte": 50`.

- **$lt** and **$lte**: Match values less than or less than or equal to a specified value.
- **$in** and **$nin**: Match values in or not in a specified list, respectively.
- **$regex**: Allows searching using regular expressions, e.g., `"Name": "$regex": "API.*"`.
- Logical operators like **$and** and **$or** combine multiple conditions.

Example queries

- **Filter by Source**: Retrieve specifications from `APIs.guru`:

```
{"source": {"$eq": "APIs.guru"}}
```

- **Combine Conditions with AND**: Retrieve APIs with category `API Management` and version `1.0.0`:

```
{"$and": [{"Title": {"$regex": "banking"}}, {"Version": {"$eq": "1.0.0"}}]}
```

- **Filter by Metric**: Retrieve APIs with at least 80% of natural language description coverage:

```
{"Endpoints Description Coverage": {"$gte": 80}}
```

3.5 Web APIs Metric-based Exploration Through APIstic

3.5.1 API size: structure and data model

Are all the APIs across the datasets of the same structure size?

Figure 3.14 showcases the path count distributions for the GitHub, SwaggerHub, BigQuery, and APIs.Guru datasets through violin plots. GitHub's data indicates a moderate skew with an average of 15.82 paths and a median of 7, pointing to some APIs with many more paths than others. SwaggerHub's distribution is more skewed, with an average path count of 12.05, a median of only 3, and a range extending to 2882, signaling that a few APIs possess a large number of paths. BigQuery's trend is akin to SwaggerHub, with an average of 13.17 and a median of 4, again showing a few APIs with elevated path counts. APIs.Guru differs, with a higher average path count of 19.33 and a median of 8, indicating a broader distribution of path counts among its APIs.

How does the size of data model vary across sources?

As shown in Figure 3.15, for GitHub collection, the maximum number of schemas is 325, with an average of approximately 7.85, indicating a moderate concentration of APIs with a relatively small number of distinct schemas. SwaggerHub, however, shows a strikingly high maximum of 1479 distinct schemas, but its average is the lowest at around 0.35, suggesting that while most APIs have very few schemas, a few outliers are exceptionally having a complex data model. BigQuery's data displays a maximum of 286 schemas and an average of 10.31, aligning closely with GitHub's distribution. APIs.guru shows a higher diversity with a maximum of 360 schemas and an average of about 12.92, indicating a slightly wider range of complexity for data models in its APIs.

Dataset	Max	Min	Average	Median	StDev
GitHub	325	0	7.85	2	16.88
SwaggerHub	1149	0	0.35	0	2.96
APIs.Guru	360	0	12.92	4	28.64
BigQuery	286	0	10.54	4	26.26
Combined	1149	0	6.37	0	15.63

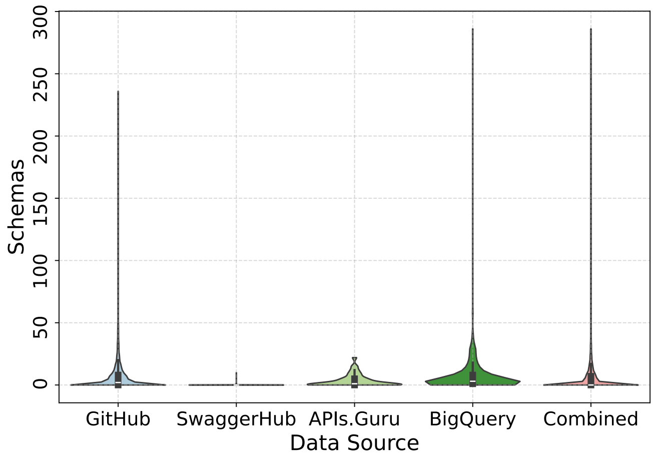


Figure 3.15. Comparative Analysis of Distinct Schema Number Distributions Across All Datasets

Do fresher APIs have larger structure?

Analyzing the API paths in GitHub and Swagger datasets yearly, we can see from Figures 3.16 and 3.17 that the APIs exhibit a clear upward trend, indicative of an increasing expansion of APIs structures between 2014 and 2017. The intensity of this growth has decreased in the next years. However, the SwaggerHub dataset shows a more pronounced variability and a higher presence of significant outliers, especially in the later years. But overall the structure size distributions did not exhibit major change over the years.

Do fresher APIs have larger data models?

As shown in Figure 3.18, GitHub dataset reveals a consistent annual growth in the average number of distinct schemas within APIs. This show that the specifications obtained from GitHub can be subject to farther Web API data model evolution analysis.

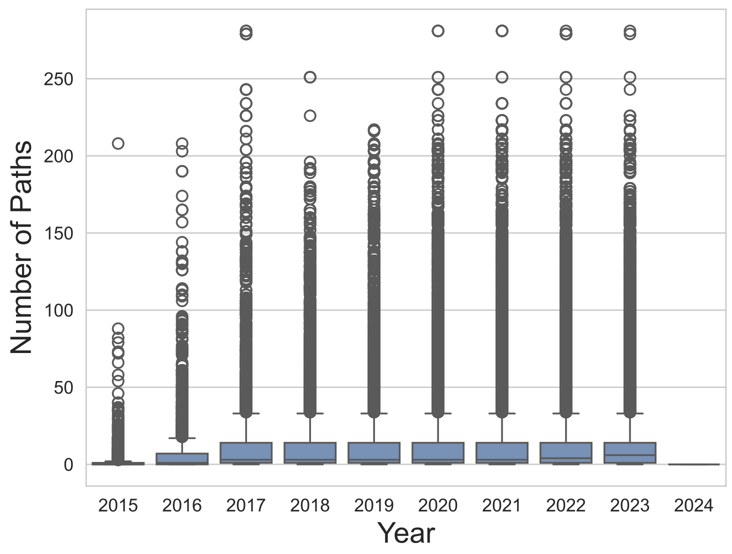


Figure 3.16. Distribution Number of Paths over the years in SwaggerHub Dataset

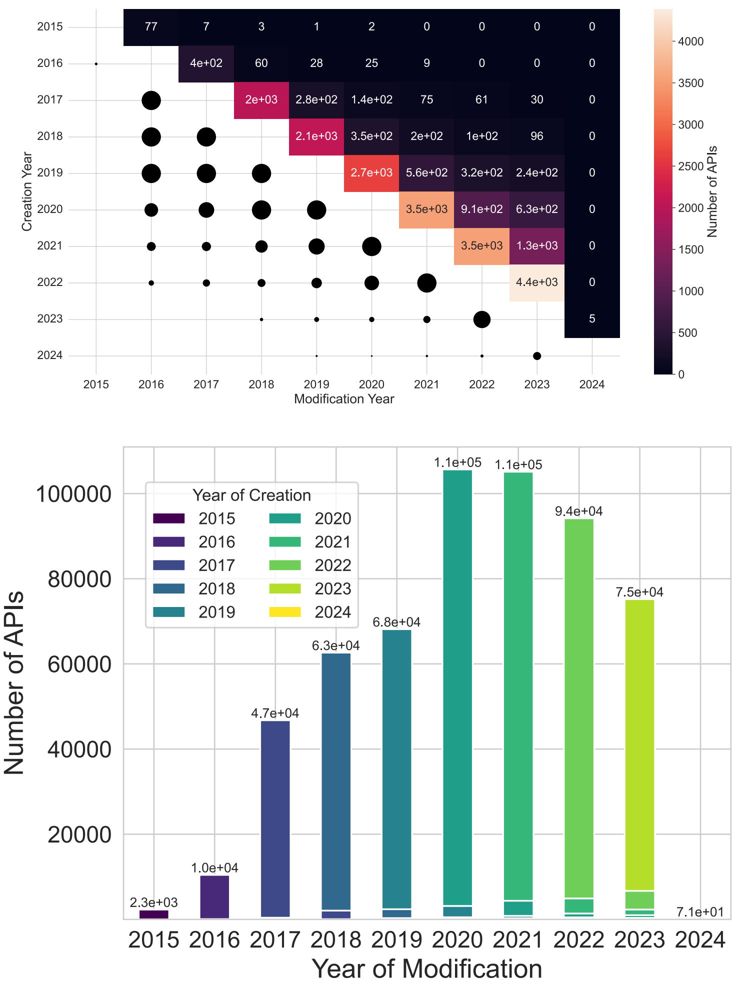


Figure 3.19. Number of Created and Modified APIs each year in the SwaggerHub Dataset

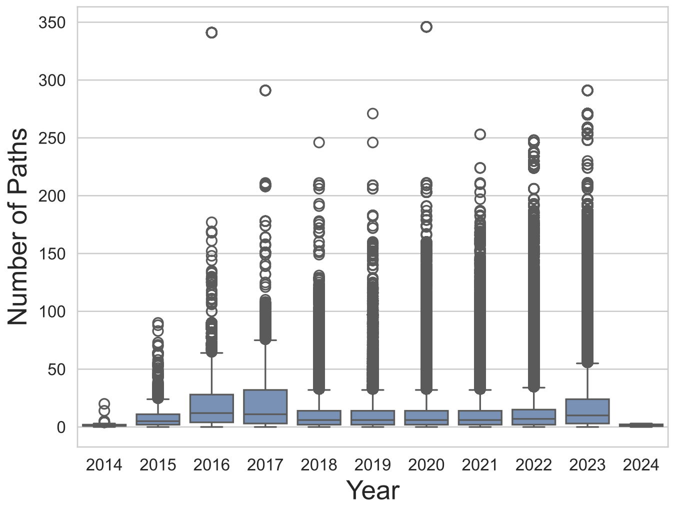


Figure 3.17. Distribution Number of Paths over the years in GitHub Dataset

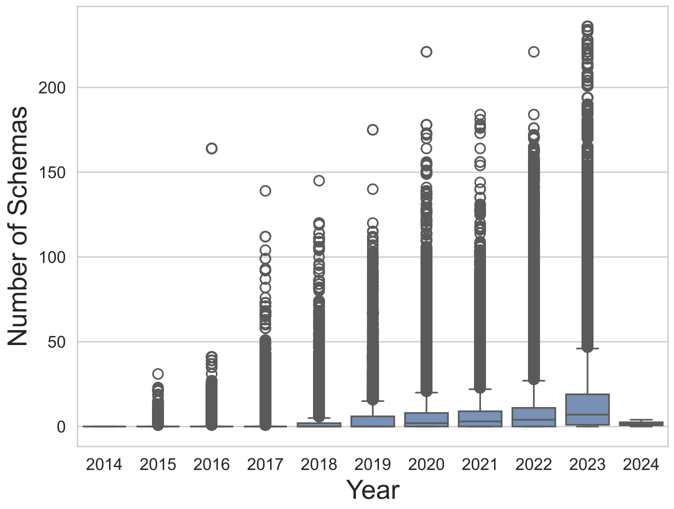


Figure 3.18. Distribution Number of Distinct Schemas over the years in GitHub

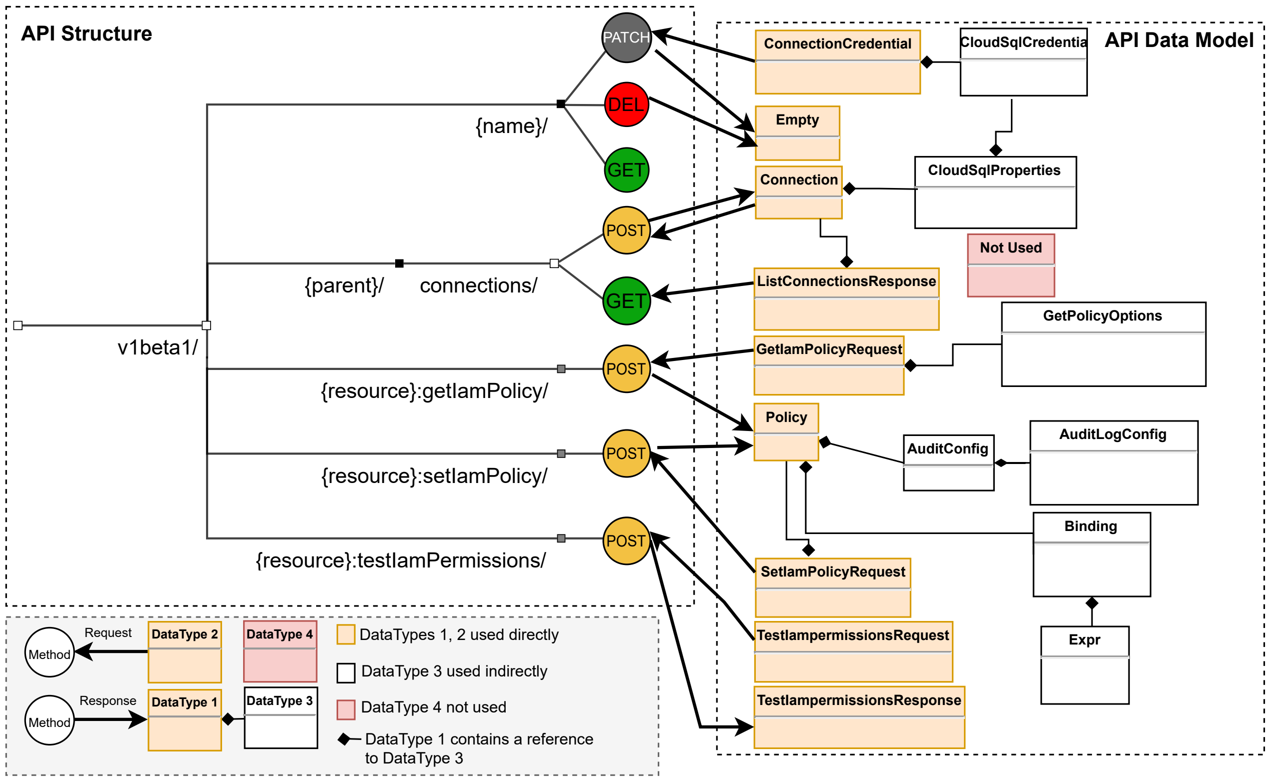


Figure 3.20. Example of BigQuery API Tree model including its structure and data model

3.5.2 Web API structure and data model correlations

In Figure 3.20 we show an example visualization of the API structure and data model, where the arrows show the relationships between the operations and the corresponding data model entities.

By examining the correlations between pairs of the structure and datamodel metrics, we can uncover relationships such as how paths and operations scale together or identify independent API elements.

The correlation heatmap matrix in Figure 3.21 provides insights into the relationships between various structure size metrics. Strong positive correlations are observed between Paths and Operations ($r = 0.96$) and between Paths and Used Methods ($r = 0.96$), indicating that as the number of paths in the structure increases, the number of operations and the variety of HTTP methods used also increase proportionally. Similarly, Parameterized Operations and Used Parameters ($r = 0.77$) exhibit a notable correlation, suggesting that operations requiring parameters scale with the overall parameter usage. Furthermore, **GET** operations are strongly correlated with both Paths ($r = 0.89$) and Operations ($r = 0.92$), highlighting its dominant presence in API functionalities compared to other HTTP methods.

On the other hand, some metrics show weaker correlations. For instance, Put operations moderately correlate with Paths ($r = 0.56$), and Distinct Parameters display a moderate relationship with Paths ($r = 0.62$), indicating less dependency compared to the stronger correlations. Certain features, such as using the Webhooks and Options method, exhibit negligible correlations with most other metrics, implying that their usage is largely independent of structural changes or complexity.

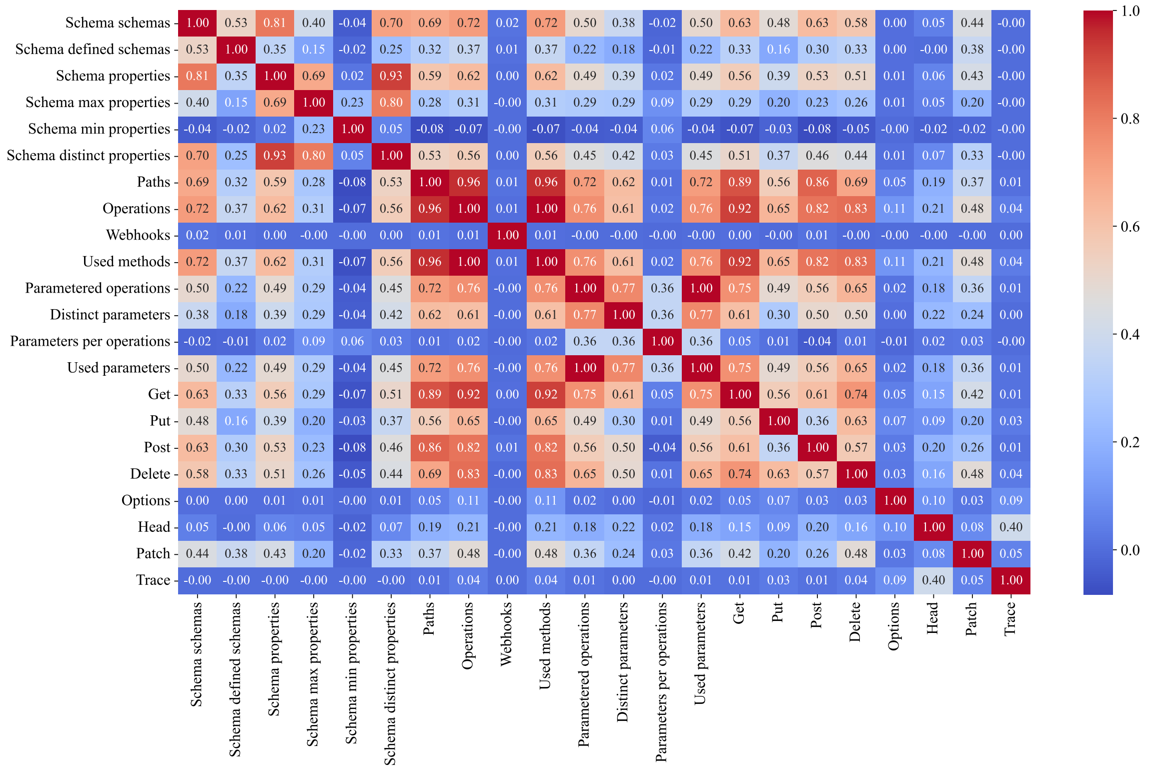


Figure 3.21. Heatmap showing pairwise Pearson correlation coefficients between structure size metrics. Red indicates a strong positive correlation, while blue represents a weaker or negative correlation. The heatmap highlights key relationships, such as strong correlations between Paths, Operations, and Used Methods, as well as weaker correlations involving Webhooks and optional HTTP methods like Options and Trace

3.5.3 HTTP methods usage

Does the proportional usage of different HTTP methods change over time?

Figures 3.23 and 3.22 show trends in API method usage over nine years, detailing the changing prevalence of methods like GET, POST, PUT. Each year is represented by a bar divided into sections for each method's count, with total operations annotated on top. Analysis of SwaggerHub and GitHub API operations reveals consistent proportions of HTTP method adoption across years. GET (read-only) remains most common, followed by POST (remote procedure calls). PUT and DELETE are used less frequently and do not show growth in recent years. Other methods like PATCH, HEAD, OPTIONS, and TRACE are rare.

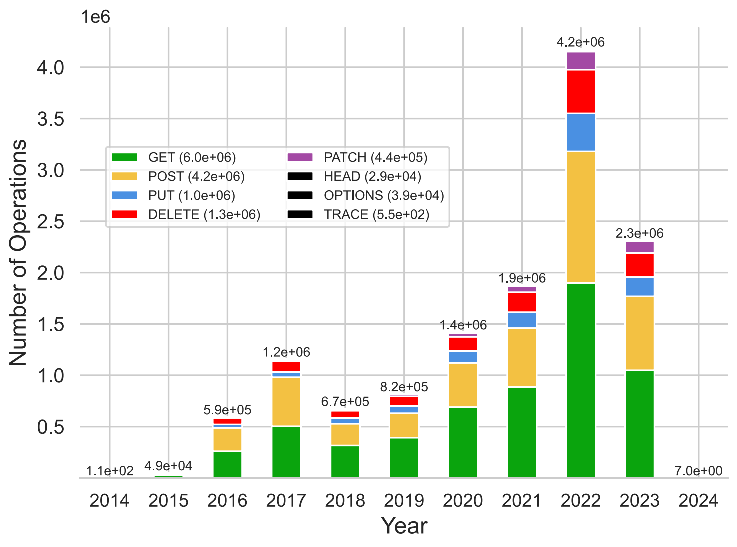


Figure 3.22. Yearly Trends in API Method Usage in GitHub Dataset

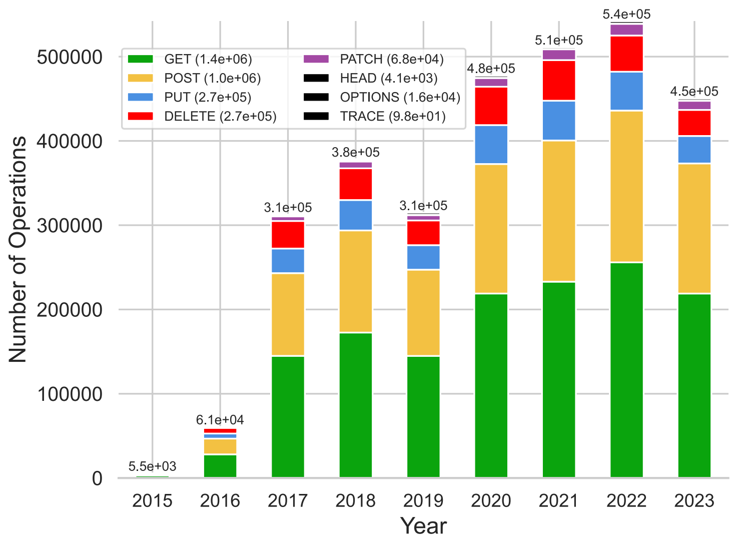


Figure 3.23. Yearly Trends in API Method Usage in SwaggerHub Dataset

3.5.4 API maintenance lifecycle

Were the API specifications just created and then abandoned?

Figure 3.19 illustrates the lifecycle of APIs in SwaggerHub descriptions, focusing on whether APIs created in a specific year underwent modifications in subsequent years. It is important to note that initial creations are also counted as modifications. For instance, in 2023, Swagger Hub recorded 63,271 new API specifications. Additionally,

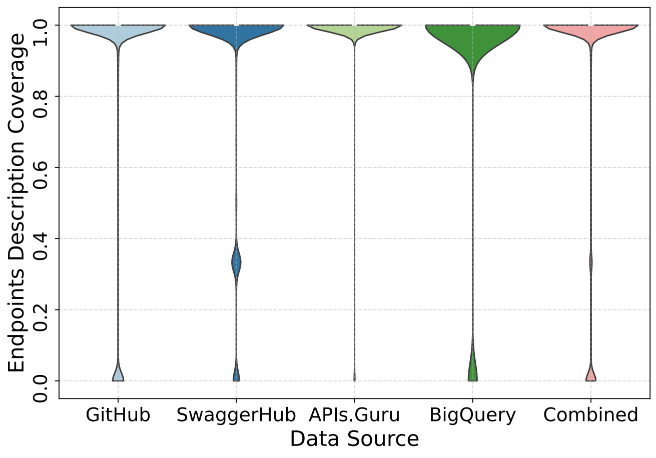


Figure 3.24. Comparative Analysis of Endpoint Description Coverage Distributions Across All Sources

there were 4,124 modifications to APIs originally created in 2022, 1,293 modifications to those from 2021, 620 from 2020, 242 from 2019, 96 from 2018, and 31 from 2017. In particular, APIs created in 2016 and 2015 have not been modified to the present day.

3.5.5 Readability of natural language documentation

How often are endpoints described in the APIs in each dataset?

In Figure 3.24, all of the datasets show a concentration of values at the upper end, suggesting a significant number of endpoints with descriptions. SwaggerHub dataset displays a broader spread, indicating greater variability and a higher likelihood of described endpoints in and API.

How readable are the natural language operations descriptions found in OpenAPI specifications?

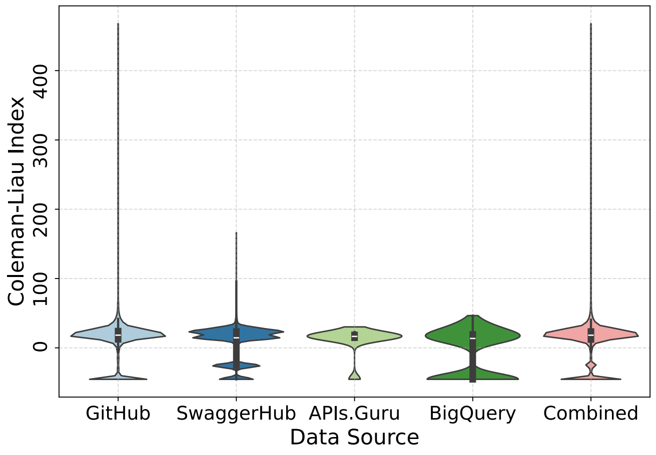


Figure 3.25. Comparative Analysis of Coleman-Liau Index Distributions Across Sources

For each data source, the distribution of the Coleman–Liau index (CLI) and Automated Readability Index (ARI) is illustrated in Figures 3.25 and 3.26. These figures reveal relatively similar distributions for both indices, except for a few outliers in GitHub with exceptionally high values. This indicates that the complexity of the text descriptions is consistent across datasets.

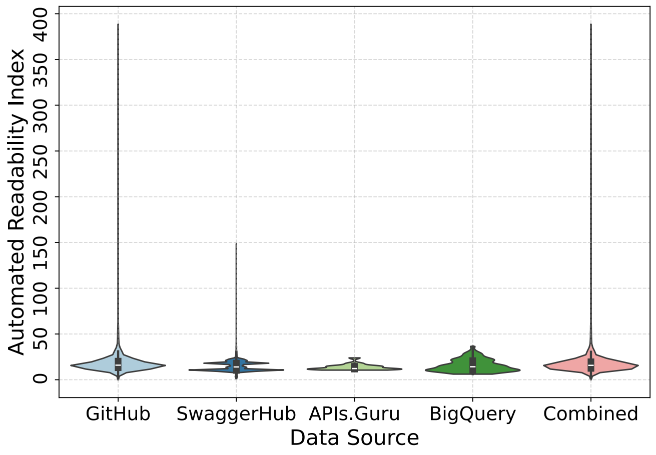


Figure 3.26. Comparative Analysis of the Automated Readability Index Distributions Across All Sources

For instance, comparing two examples of from our dataset with distant index values, Elastic Email API, exhibits a high ARI of 22.78, suggesting a text complexity that demands an advanced understanding, likely targeting specialists or users with considerable expertise in the field. In contrast, the Cinema WebApp API, presents a significantly lower

ARI of 9.49. This score typically characterizes APIs used for educational purposes or as examples, indicating a more accessible and user-friendly documentation, suitable for a broader audience, including beginners or students.

3.5.6 Types of security schemes in APIs

API security documentation is not required for a valid OpenAPI specification. The metrics we have established are not suitable for assessing the frequency of security mechanism adoption in API endpoints. Instead, they serve as a criterion to identify APIs with documented security features. Subsequent analyses can then be conducted on this filtered group of APIs.

Dataset	Max	Min	Average	Median	Std Dev
GitHub	1	0	0.47	0	0.49
SwaggerHub	1	0	0.03	0	0.15
APIs.Guru	1	0	0.69	1	0.46
BigQuery	1	0	0.51	1	0.49
Combined	1	0	0.30	0	0.45

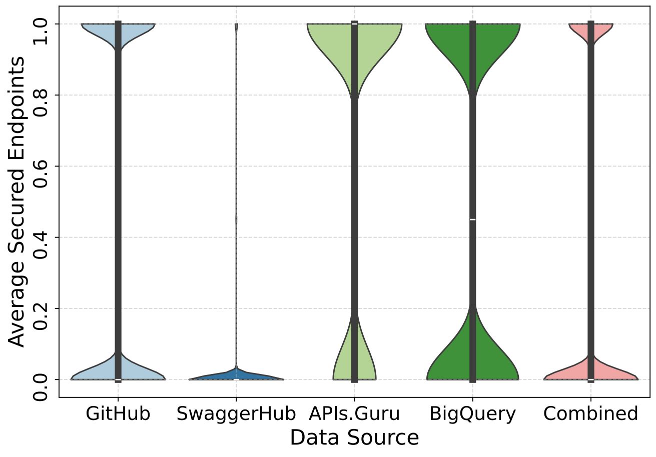


Figure 3.27. Comparative Analysis of Average Secured Endpoints Distributions Across All Sources

How diverse are the security schemes types used in APIs?

Table 3.2 reflects diverse security practices in APIs across GitHub, SwaggerHub, BigQuery, and APIs.guru. The usage of API keys is the most prevalent, followed by the OAuth2 protocol and the basic HTTP authentication scheme (described as 'http' in Swagger 2.0 and as 'basic' in the more recent OpenAPI 3.0). OpenIDConnect is present only in a small number of APIs.

In Table 3.2, we include the security schemes used in at least three of the four sources.

What is the average number of secured endpoints in APIs across datasets?

GitHub and APIs.Guru datasets show a higher average coverage of endpoints (Fig. 3.27), with GitHub being more consistent. SwaggerHub has notably lower coverage, while BigQuery dataset suggests a moderate coverage. Excluding SwaggerHub, we clearly observe a bi-modal distribution of artifacts where either there is a high level of

Table 3.2. Number of APIs making use of different OpenAPI Security Schemes across different datasets

Type	GitHub	SwaggerHub	BigQuery	APIs.guru
apiKey	261358	6660	1770	528
oauth2	140227	5642	2651	2743
http	142906	2686	138	123
basic	37825	566	124	17
openIdConnect	4099	58	1	0
Secured APIs	26%	3%	74%	96%

coverage or no endpoints make use of security features. While almost all artifacts found in APIs.guru make use of security features, only very few of the API descriptions sourced from

3.6 API Specification Dataset Usage in this Thesis

This dataset served as foundation of this research for studying the characteristics of Web API structures and data models, as well as the evolution of Web APIs over time and their versioning practices. Since the dataset was incrementally mined and steadily grew from the very first day of collection, the studies conducted leveraging this dataset relied on different snapshots captured at various points in time. Each study focused on specific aspects of the data, selecting relevant subsets based on the particular objectives of the analysis. The most recent snapshot of the GitHub sourced specifications is used in Chapter 7, since we are interested in capturing the Web API evolution patterns and analyzing them. The findings presented in that chapter haven't been published.

Work	Snapshot Size	Updated at	Source
EuroPLoP [148]	Latest commit of 6 619 Specifications	January 2021	GitHub
ICSA [139]	Latest commit of 42 194 Specifications	January 2022	GitHub
ICWE [142]	186 259 Specifications of API commits	November 2022	GitHub
JWE [144]	603 293 Specifications of APIs	July 2023	GitHub, SwaggerHub, APIsGuru
ICWE [145]	915 885 Specifications of API commits	December 2023	GitHub
MSR [143]	1 275 568 Specifications of APIs	October 2023	GitHub, SwaggerHub, APIsGuru
Chap. 3	1 894 505 Specifications of API commits	December 2024	GitHub, SwaggerHub, APIsGuru
Chap. 7	1 372 550 Specifications of API commits	December 2024	GitHub

Table 3.3. Dataset snapshots used in Web API analysis Studies

While we targeted different sources to discover API descriptions, GitHub remains the richest source since it also provides contextual metadata including the provider, the project, the specification' history and contributors.

In Summary

In this chapter we have described how we have collected a dataset of API specifications which serves as a resource to extract samples for large-scale empirical studies in a field where most existing studies have been performed considering up to a few thousand artifacts.

The current version of the dataset, presented in this chapter, comprises measurements and provenance metadata of a substantial number of 1 894 505 API descriptions sourced from GitHub, SwaggerHub, BigQuery, and APIs.guru, providing a comprehensive source to analyze current API practices and trends in API design, documentation, security strategies, and data modeling. Particularly noteworthy is the portion of the dataset sourced from GitHub, which includes historical data of APIs, providing insights into the evolution of these artifacts and their metrics over time.

Part II

Web API structure and data model analysis

Chapter 4

Web APIs Structure and Data Models Characteristics Analysis

The complexity of the structure and data models of Web APIs play a critical role in determining their usability, maintainability, and scalability. Measuring the API complexity involves analyzing factors such as the depth of hierarchical models, the relationships between entities, and the overall data model schema design. Understanding the correlation between structural complexity and data model intricacy could help to estimate maintainability and evolution costs. Further refining the metrics defined in Chapter 3, this chapter explores these characteristics of 255 309 APIs, providing insights into how the complexity of API functionality can correlate with the complexity of the corresponding data model.

4.1 Design Principles and their Impact on API Structures and data models

In addition to functional requirements, conventional web API design decisions have a direct impact on how the API structure and datamodels are designed. Their design must balance functionality, usability, and robustness [109]. One major challenge is to ensure that APIs are intuitive and easy to learn for developers [75], which can be achieved by following consistent design principles and providing comprehensive, clear documentation [100, 94, 25, 159]. Learnability is crucial to reduce the onboarding time for new developers and to minimize the probability of errors or misuses [23]. Poorly designed APIs can cause misunderstandings about how the API should be used, resulting in incorrect implementations and increased support costs [66].

In this analysis, we examine the adoption of seven design principles highlighted by both Lauret[89] and Jin et al.[75], focusing on their application to API paths, operations, and data models.

4.1.1 Path design principles

Resource hierarchies should be reflected in paths.

The API paths design determines the endpoints with which the clients need to interact using specific HTTP methods to use the API. The structure of the paths is expected to correspond to natural relationships within the data model. For example:

- A parent-child relationship, such as Users and Orders, is commonly represented as `/users/{userId}/orders`.

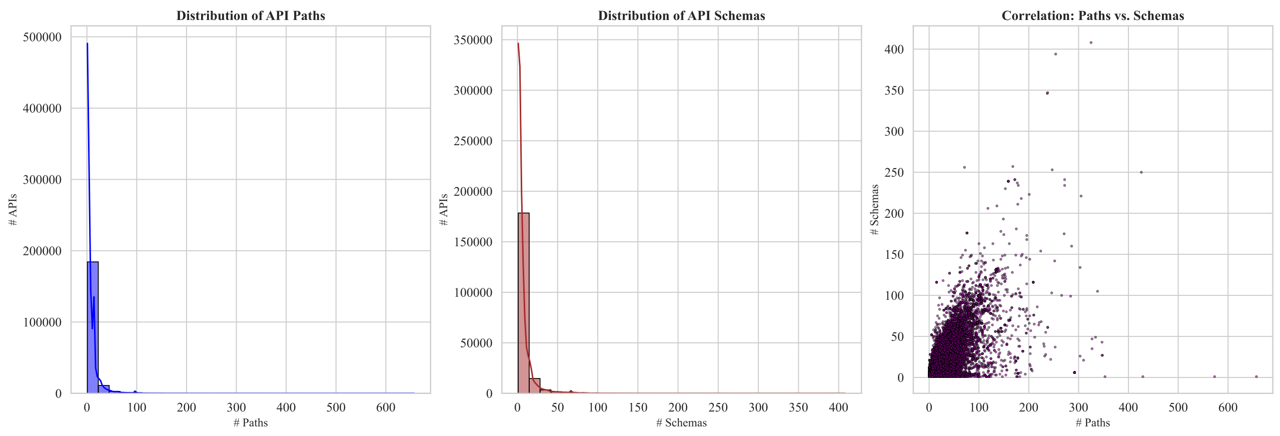


Figure 4.1. Distribution of API Paths (left), API Schemas (middle), and the correlation between Paths and Schemas (right) – excluding APIs with no paths or no schemas.

- Resources at the same level in the hierarchy are typically represented without excessive nesting. Structures such as `/company/division/users/{userId}/orders/{orderId}` are considered less clear unless all segments provide a meaningful context. In such cases, separate endpoints can be introduced to reflect different resource relationships. For instance, `/users/{userId}/orders/{orderId}` may be used to represent user-specific orders, while `/company/division` could serve to represent organizational structures independently. This approach improves readability and reduces the complexity of understanding resource relationships.

The scatter plot in Figure 4.1 shows the correlation between the number of paths and the schemas, indicating that as the number of paths increases, the number of schemas tends to increase as well. However, the variation becomes more pronounced for APIs with a higher number of paths, suggesting diverse design practices in large APIs.

Paths are generally structured to represent logical relationships.

Relationships in the data model, such as a one-to-many association, are typically mirrored in the path structure. For example:

- A one-to-many relationship between Users and Orders is often represented as `/users/{userId}/orders`.

where `user` and `userId` are so semantically tied.

To verify the state of the art of the implementation of this principle we check how many out of the 3711 APIs contain paths that are designed taking into account representing logical relationships. Each API path is decomposed into its constituent segments, referred to as *labels*, while dynamic parameters enclosed in curly brackets (e.g., `{groupId}`) are extracted and normalized by removing the brackets. This ensures placeholders do not interfere with the analysis. From the cleaned segments, consecutive pairs of labels are generated to evaluate their relationships. For example:

- The path `/group/{groupId}/members` produces three labels: `group`, `groupId`, and `members`. The following pairs are formed:
  - (`group`, `groupId`)
  - (`groupId`, `members`)

Table 4.1. Logical Path Structure Analysis

Metric	**Count**
Total Analyzed APIs (at leat 50 commits)	3711
APIs Using Logical Relationships	1229
APIs Without Logical Relationships	2482

- Similarly, the path `/requests/{requestId}/request` produces:
  - (`requests`, `requestId`)
  - (`requestId`, `request`)

For each pair of subsequent labels, we compute the *semantic similarity* using the **GloVe pre-trained word embeddings** (`glove-wiki-gigaword-100`). These embeddings provide vector representations of words based on large-scale text corpora. The similarity score is calculated as the cosine similarity between the vector representations of two words, where a score of 1 indicates identical meaning and 0 indicates no semantic relationship.

For example:

- `group` and `groupId` yield a similarity score of 0.65, reflecting their conceptual connection.
- `groupId` and `members` produce a score of 0.71, indicating a logical relationship where `members` are part of a `group`.
- `logos` and `blob` yield a score of 0.14, suggesting no meaningful semantic relationship.
- `reports` and `metrics` produce a score of 0.07, further demonstrating dissimilarity.

A path is classified as *logically structured* if at least one pair of its subsequent labels has a similarity score greater than or equal to 0.5. This threshold ensures that relationships considered logical are semantically meaningful.

For example:

- The path `/group/{groupId}/members` is classified as logically structured because:
  - (`group`, `groupId`) $\rightarrow$ 0.65
  - (`groupId`, `members`) $\rightarrow$ 0.71

  Both pairs exceed the threshold of 0.5.
- In contrast, the path `/logos/urls/blob` is not logically structured because:
  - (`logos`, `urls`) $\rightarrow$ 0.12
  - (`logoId`, `blob`) $\rightarrow$ 0.14

  Neither pair meets the threshold.

Using this approach, we created a dataset consisting of 144 596 pairs of subsequent labels extracted from API paths. Each pair is associated with its computed similarity score, which were later to determine whether a path in logically structured or not.

We conducted the analysis on APIs with more than 50 commits. Out of the analyzed APIs, 1,233 were found to include logical paths, with a maximum of 53 logical paths in a single API, a minimum of 1, and an average of

3.79 logical paths per API, accompanied by a standard deviation of 5.07. The lengths of logical paths ranged from a minimum of 2 segments to a maximum of 9 segments. In contrast, 2,478 APIs did not include any logical paths.

An example of an API that takes into account the logical relations between its resources in the API design, is the the OpenAI API [1] represented in Figure 4.2. The API is well organized with clear separation of resources. Each resource focuses on a specific function. For example, the `messages` resource manages conversations with methods like `POST` to send a message and `GET` to retrieve messages. The `files` resource handles file operations with methods like `POST` to upload files, `GET` to retrieve file details, and `DELETE` to delete files. The `models` resource provides access to AI models with methods like `GET` to list models or get model details. Each resource has its own methods and parameters, keeping the API clean and easy to use. The depth of paths matches the logical relationships between resources. For example, `/models` lists models, while `/files/{file_id}` operates on a specific file. This structure makes paths intuitive and matches how resources relate to each other. Each resource focuses on a specific function. For example, the `messages` resource manages conversations with methods like `POST` to send a message and `GET` to retrieve messages. The `files` resource handles file operations with methods like `POST` to upload files, `GET` to retrieve file details, and `DELETE` to delete files. The `models` resource provides access to AI models with methods like `GET` to list models or get model details. Each resource has its own methods and parameters, keeping the API clean and easy to use. The depth of paths matches the logical relationships between resources. For example, `/files/{file_id}/metadata` retrieves metadata for a specific file, while `/models/{model_id}/versions/{version_id}` fetches details about a specific version of a model. These deeper paths show hierarchical relationships, where operations depend on specific identifiers. This structure makes paths intuitive and matches how resources relate to each other.

Plural nouns are recommended for collections, and singular nouns are suited for individual resources.

This convention aligns the path structure with the resource representations provided in the API responses. For example:

- `/users` is generally used to represent a collection of users. The response can either return:
  - A plain array of user objects (4.1).
  - An object that includes metadata (e.g., `total_users`) alongside the array of user objects (4.2).
- `/users/{id}` is employed to represent a specific user. The response should return a single user object. See the OpenAPI schema in 4.3.

Listing 4.1. OpenAPI schema for a collection of users as an array.

```yaml
paths:
 /users:
  get:
   summary: Retrieve a list of users
   responses:
    '200':
     description: A JSON array of user objects
     content:
      application/json:
       schema:
        type: array
        items:
         type: object
         properties:
          id:
           type: string
           description: Unique identifier for the user
          name:
```

[1] https://github.com/openai/openai-openapi/blob/master/openapi.yaml

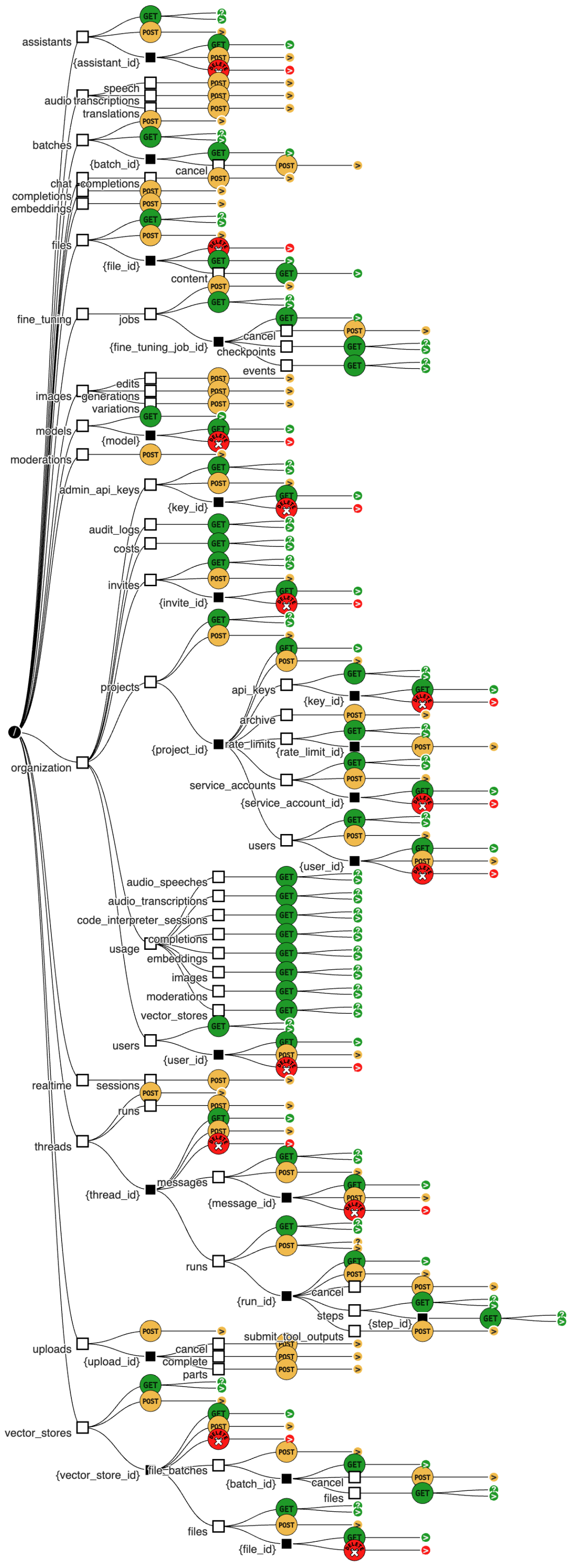


Figure 4.2. Tree visualization of OpenAI API using OAS2Tree [1]

```
              type: string
              description: Name of the user
```

Listing 4.2. OpenAPI schema for a collection of users with metadata.

```
paths:
 /users:
  get:
   summary: Retrieve a list of users with metadata
   responses:
    '200':
     description: A JSON object containing metadata and a list of users
     content:
      application/json:
       schema:
        type: object
        properties:
         total_users:
          type: number
          description: Total number of users available
         users:
          type: array
          items:
           type: object
           properties:
            id:
             type: string
             description: Unique identifier for the user
            name:
             type: string
             description: Name of the user
```

Listing 4.3. OpenAPI schema for an individual user resource.

```
paths:
 /users/{id}:
  get:
   summary: Retrieve a single user by ID
   parameters:
    - name: id
     in: path
     required: true
     schema:
      type: string
   responses:
    '200':
     description: A JSON object representing the user
     content:
      application/json:
       schema:
        type: object
        properties:
         id:
          type: string
          description: Unique identifier for the user
         name:
          type: string
          description: Name of the user
```

As illustrated in Listing 4.1, paths such as `/users` are used to represent collections, with the response typically structured as an array of resource objects. Alternatively, as shown in 4.2, the response can include metadata—such as `total_users`—alongside the resource array, providing additional context and improving API usability. These metadata-enriched responses are particularly beneficial in scenarios requiring pagination or aggregation of data. In contrast, singular nouns, as exemplified in 4.3, are reserved for paths like `/users/{id}` that operate on individual resources. This distinction not only aligns the API design with developers' expectations but also enhances the semantic expressiveness of the interface.

Table 4.2. Usage of plurals in API paths and array type responses

Label	Response	#APIs	#Endpoints	Min Endp.	Max Endp.	Avg. Endp.	Stdev Endp.	Median Endp.
Plural	No Array in Response	101,282	989,240	1	709	9.77	19.77	4.00
Plural	Array in Properties	70,445	394,464	1	513	5.60	13.07	2.00
Plural	Array Response	477	925	1	204	1.94	9.39	1.00
Non-Plural	No Array in Response	174,360	3,225,181	1	1,474	18.50	44.79	7.00
Non-Plural	Array in Properties	86,796	712,007	1	650	8.20	18.66	3.00
Non-Plural	Array Response	696	1,959	1	521	2.81	19.86	1.00

To determine whether the last segment label is a plural word, we use the spaCy library, a robust NLP tool that provides part-of-speech tagging to identify plural forms. Specifically, we check if the word's part-of-speech tag is one of the plural noun tags (`NNS` for plural common nouns or `NNPS` for plural proper nouns), as shown in Listing 4.4.

Listing 4.4. Checking for plural words using spaCy

```python
import spacy

# Load spaCy model
nlp = spacy.load("en_core_web_sm")

def is_plural(word):
    """
    Use spaCy to check if a word is plural.
    """
    doc = nlp(word)
    for token in doc:
        if token.tag_ in ["NNS", "NNPS"]: # Plural common/proper nouns
            return True
    return False
```

In Figure 4.2, we analyze whether endpoints with a path segment ending with a plural label return an array of results in their response. This can occur either through the response being of type "Array in Response" or through one of the properties of the response object being of array type, referred to as "Array in Properties." When the category is labeled "No Array in Response", it indicates that neither the response itself nor any of its properties is of array type. In contrast, when categorized as "Array in Properties", the response is of an object type where at least one of its properties is an array. We found that nearly a million endpoints use plural labels, while more than 3 million endpoints do not. The results show that when a response contains an array, it does not need to be paired with an endpoint that ends in a plural label. In fact, it is twice as common for responses with arrays to come from endpoints that do not use plural labels. Endpoints with plural labels are about four times less common than those without plural labels. Another observation is that arrays in responses are more likely to be wrapped inside an object (as shown in Listing 4.2) rather than being sent directly as standalone arrays (as shown in Listing 4.1).

☛ Endpoints with plural labels are less common, appearing in only a quarter of cases compared to non-plural labels. When responses contain arrays, they are twice as likely to come from non-plural endpoints. In addition, the arrays in the responses are more commonly encapsulated within objects than sent as standalone arrays.

Figure 4.3 shows the distribution of the proportion of endpoints that fall into a specific category among the total endpoints for an API. The results show that there exist indeed cases where all the endpoints of an API are ending

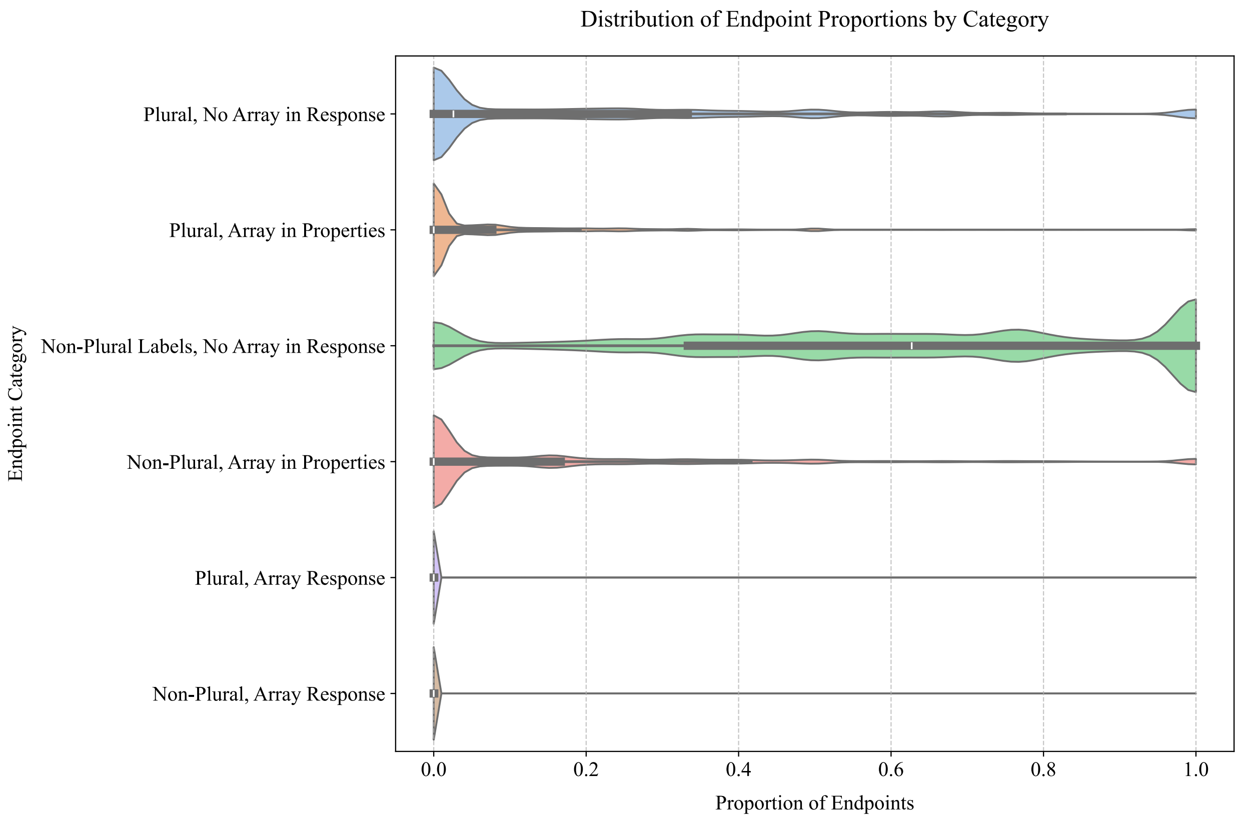


Figure 4.3. Distribution of proportion of endpoints belonging to each category

with plural labels or sending arrays in response. However, most of the cases of API having the majority or endpoints not sending arrays and not ending with plural are widely dominant.

Figure 4.4 illustrates the usage of plural labels in API endpoints and the presence of arrays in response data, categorized by HTTP methods. The upper graph highlights the absolute count of endpoints, while the lower graph presents their proportional distribution.

For the GET and POST methods, there is significant variability, with a notable presence of non-plural labels and arrays in responses. This suggests that these methods are often used to retrieve collections of resources (**`GET`**) or create multiple resources (**`POST`**). In contrast, methods such as **`DELETE`**, **`PATCH`** and PUT show a smaller proportion of plural labels or arrays in responses, reflecting their typical usage for single-resource operations. Lesser-used methods such as **`OPTIONS`** **`HEAD`** and **`OPTIONS`** redominantly align with non-plural labels and do not involve arrays, consistent with their more specific and low-level operational purpose.

4.1.2 Operation design principles

HTTP methods are mapped to CRUD operations based on the data model.

Operations are usually aligned with standard HTTP methods as follows:

- **`GET`**: Retrieval of resources is typically represented, such as `/products/{id}` for fetching a specific product.

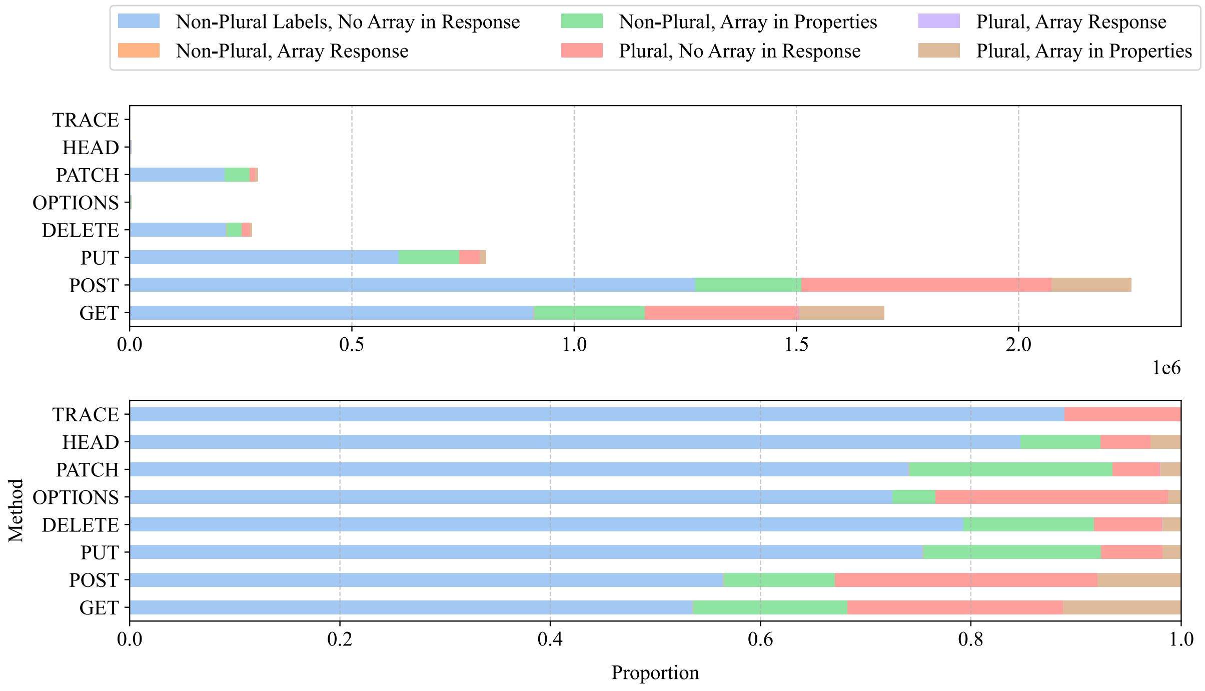


Figure 4.4. Usage of plural labels in endpoints and array in response data by type of HTTP method

- **POST**: Creation of new resources is generally indicated, such as `/users` for adding a new user.
- **PUT**: Replacement of an entire resource is commonly expressed, such as `/users/{id}` for updating all fields of a user.
- **PATCH** Partial updates to a resource are typically represented, such as modifying specific fields of a user at `/users/{id}`.
- **DELETE**: Deletion of resources is generally handled, such as `/orders/{id}` for removing an order.

Although the correlation can give an idea about the co-usage of the HTTP method, in Figure 4.5 we studied the usage of HTTP methods in our collection. We clustered the APIs depending on the methods combinations they employ, distinguishing between four categories. The top plot in Figure 4.5 shows the distribution of APIs across different classifications, highlighting that APIs that are combining more than four methods (REST) are the most common, with 70,061 instances. This reflects the widespread adoption of REST principles due to their flexibility and ability to support various interaction patterns. Read-only APIs, which use only the **GET** method for data retrieval, are the second most common, indicating their simplicity and efficiency for read-heavy applications. CRUD APIs, which strictly implement **GET**, **POST**, **PUT**, and **DELETE** for complete resource management, follow closely. Read/Write APIs, using only **GET** and **POST**, also represent a significant portion, suggesting their utility in scenarios requiring basic read and write operations. Finally, RPC APIs, relying exclusively on **POST**, are the least common, likely due to their more specialized use cases.

The bottom plot of Figure 4.5 examines the distribution of HTTP methods within each classification. CRUD APIs show a balanced use of **GET**, **POST**, **PUT**, and **DELETE**, supporting the full lifecycle of resource operations. This result is slightly different than what we found in our study performed in 2021 [139] on a smaller set of API specifications (31 118), where the proportion of **GET** methods was found to be slightly higher than the other methods. Read-only APIs rely exclusively on **GET**, while Read/Write APIs primarily use a mix of **GET** and **POST** methods. RPC APIs predictably use only **POST**, reflecting their specific focus on remote procedure calls. REST APIs demonstrate the broadest method distribution, with significant use of **GET**, **POST**, and **DELETE**, as well as some usage of less common methods like **PATCH** showcasing their adaptability to diverse use cases.

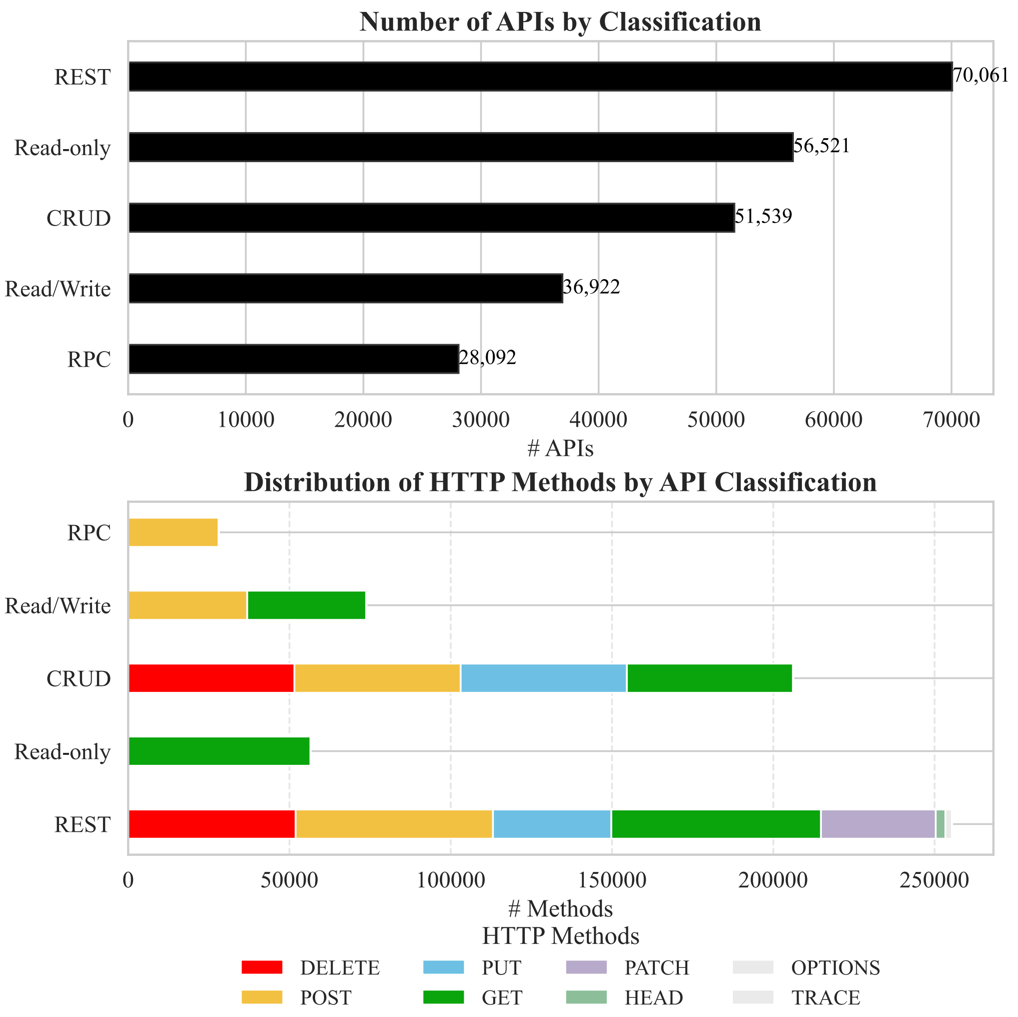


Figure 4.5. Classification of APIs by HTTP method usage. The top plot shows the number of APIs for each cluster, while the bottom plot illustrates the distribution of HTTP methods used within each classification

API change their HTTP method usage style as they become more (or less) RESTful

The heatmap in Figure 4.6 shows how long it takes, on average, for APIs to change their use of HTTP methods. APIs classified as CRUD (using exactly **GET**, **POST**, **DELETE**, and **PUT**) transition quickly to REST (using a wider mix of methods) or vice versa, averaging around 8–14 days. Changes in Read/Write (using only **GET** and **POST**) take a little longer, about 14 to 25 days, whether using CRUD, REST, or Read-only APIs. The slowest transitions occur

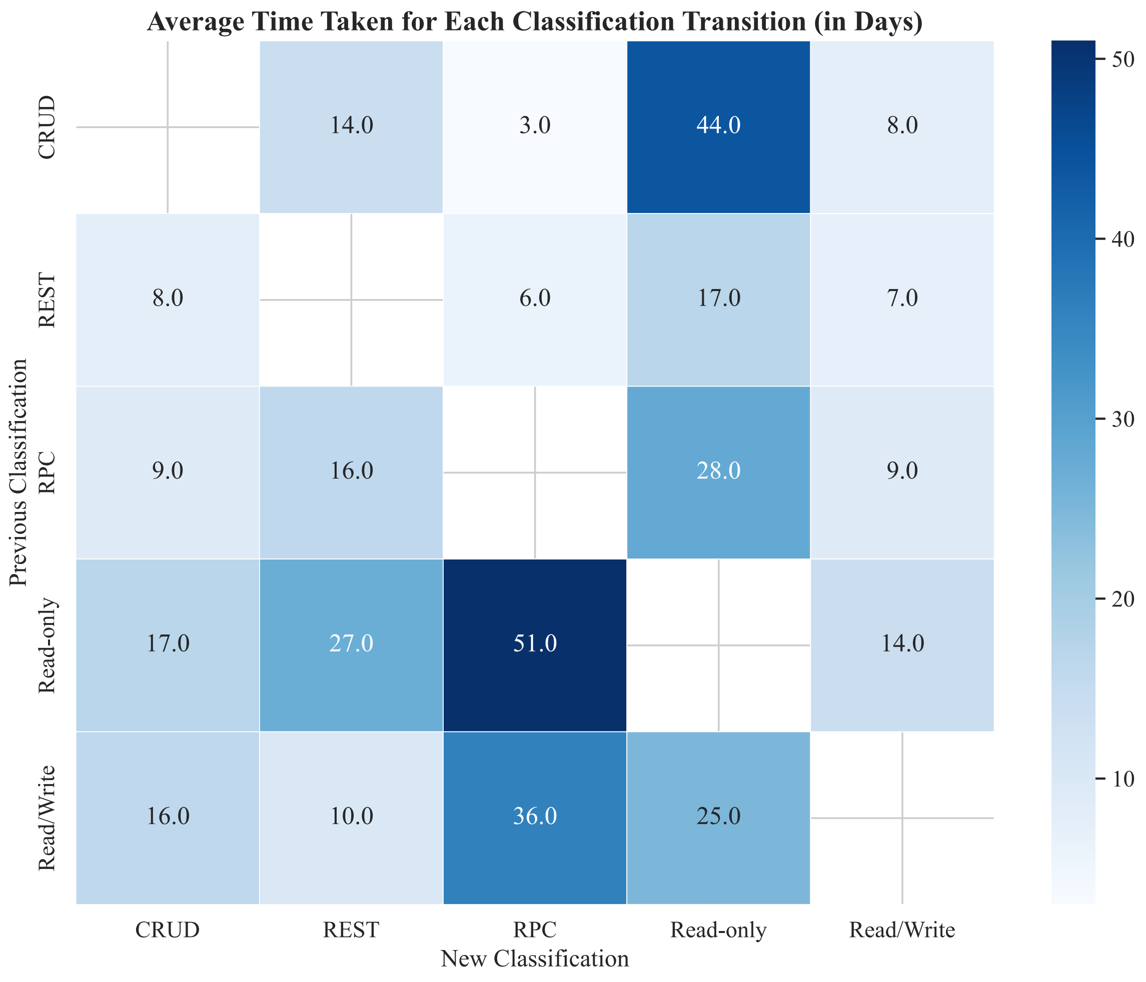


Figure 4.6. Transition chain analysis for APIs with more than 50 commits

when APIs move from Read-only (using only **GET**) to RPC (using only **POST**), averaging 51 days, suggesting these shifts are more complex and less common.

☛ We can conclude from the results that simpler changes, like expanding or limiting existing methods, happen faster, while more significant changes in functionality, like switching from read-only to write-only operations, take more time.

Figure 4.7 shows that using a combination of methods (classified as REST) consistently dominates the distribution across all years, indicating its widespread adoption as the preferred API style for delivering functionality. However, the proportion of CRUD APIs, initially significant, shows a slight decline over the years, suggesting a move away from strictly standardized operations toward more flexible or specialized API behaviors.

Read/Write APIs, which combine basic operations like GET and POST, have grown in relative proportion, reflecting an increasing need for APIs that support both data retrieval and updates without full CRUD operations.

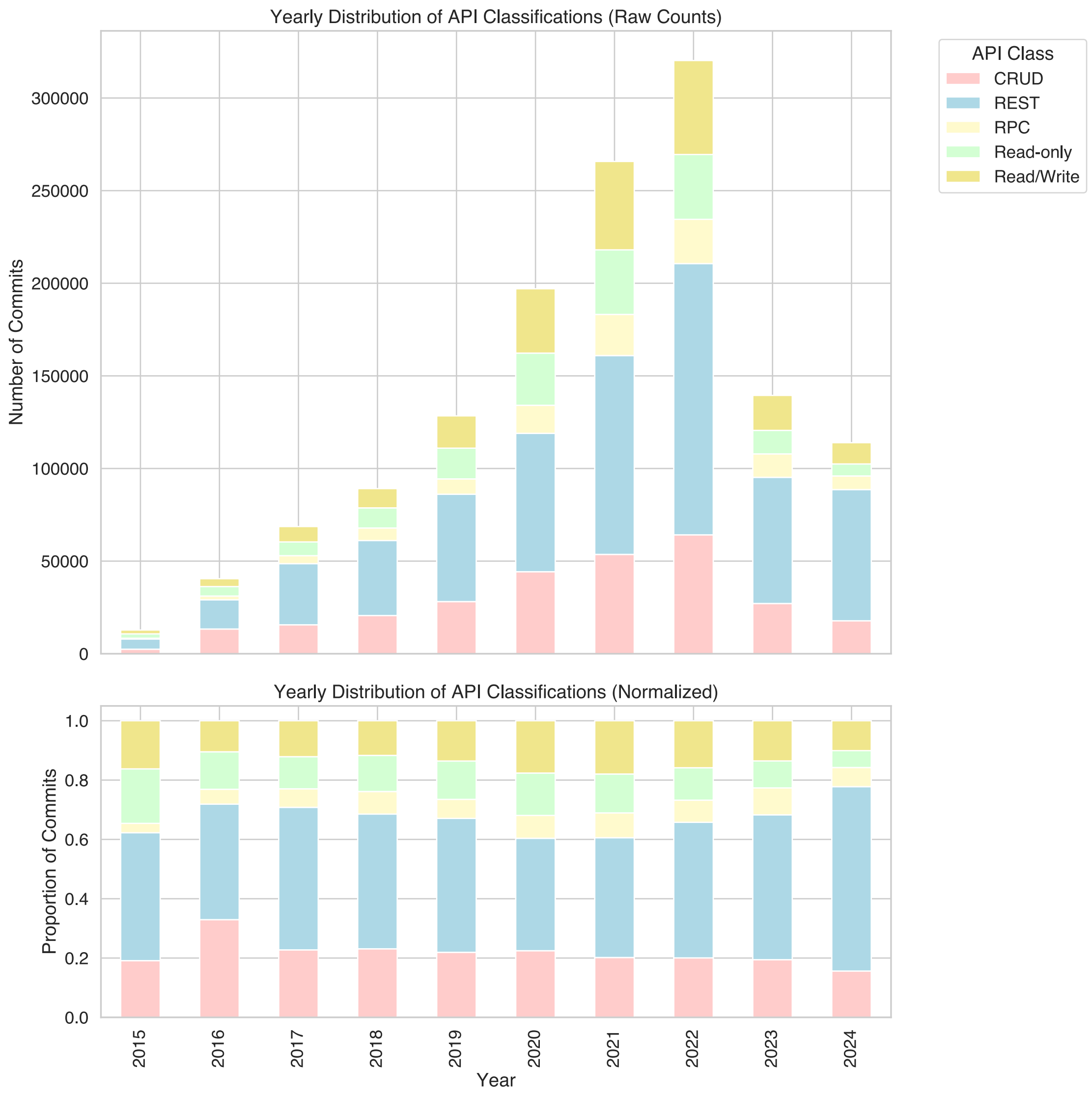


Figure 4.7. Yearly Distribution of Commits Classifications (2015–2024): The upper plot displays the raw counts of APIs classified as CRUD, REST, RPC, Read-only, and Read/Write for each year. The lower plot shows the normalized proportions of these classifications, highlighting trends in the relative distribution of API functionalities over time.

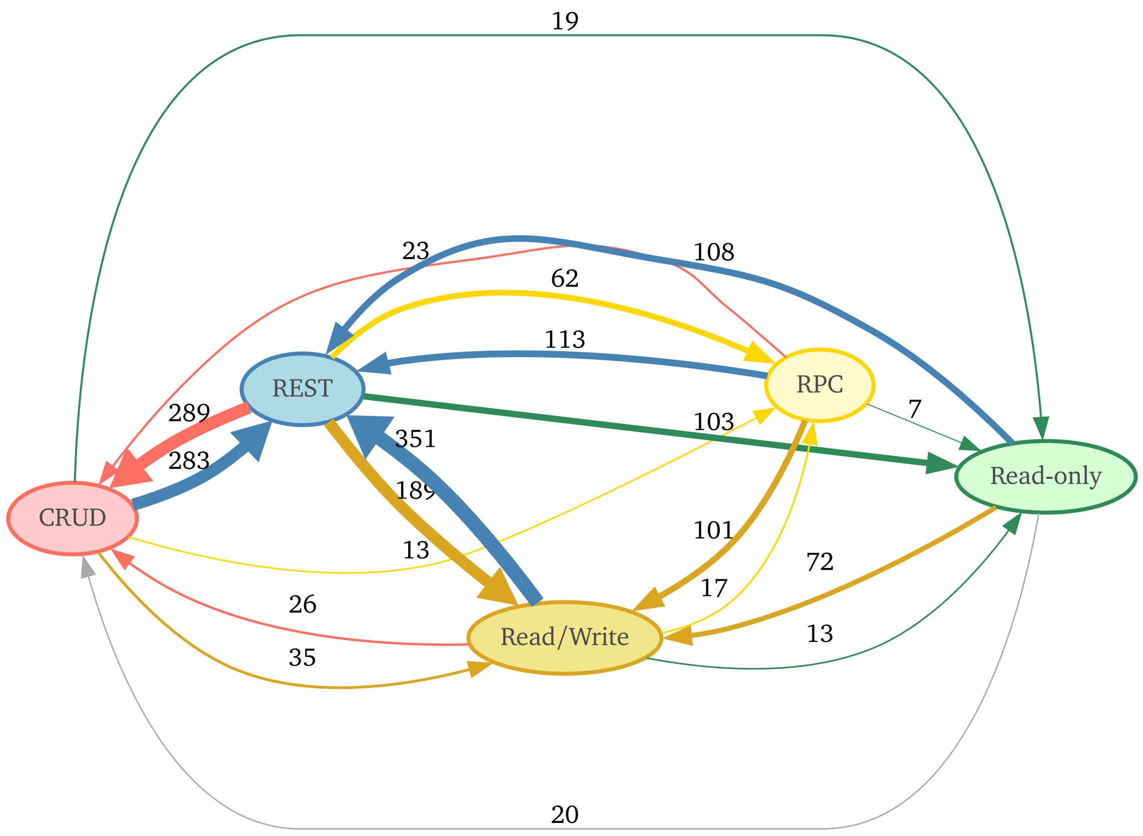


Figure 4.8. Visualization of API classification transitions: Nodes represent API classifications (CRUD, REST, RPC, Read-only, Read/Write), and directed edges indicate transitions between classifications from a commit to another. Edge thickness corresponds to the frequency of transitions, highlighting the most common paths such as CRUD to REST and Read/Write to REST

Similarly, RPC APIs, characterized by POST methods, maintain a stable presence, indicating their continued relevance for task-based and command-driven interactions.

Read-only APIs, which are limited to GET operations, hold a smaller but steady share, likely due to their specific use cases in data retrieval and monitoring. This stability highlights their niche role in modern API development.

The graph in Figure 4.8, visualizes the transitions that occur when APIs change classification during their evolution. REST emerges as the central hub, acting as the most frequent destination and source of transitions, with 351 transitions from Read/Write and 289 from CRUD. This highlights REST's dominant role in API design evolution.

CRUD APIs often transition to REST (283) and Read/Write (35), suggesting a shift from standardized operations to more flexible designs. Conversely, Read/Write APIs frequently evolve into REST (189) and occasionally into RPC (17), reflecting their adaptability for task-based or RESTful functionality.

RPC transitions are notable for their connections to both REST (113) and Read/Write (101), indicating that RPC APIs sometimes adopt hybrid capabilities or RESTful designs. Read-only APIs show fewer transitions overall but tend to evolve into REST (108) or remain isolated, emphasizing their niche, data-retrieval role.

Table 4.3 provides interesting insights into how APIs evolve and change over time. The most common transition is from *CRUD* to *REST*, with 241 APIs making this shift. This transition often happens when developers need to go beyond basic operations like **GET**, **POST**, **PUT**, and **DELETE**, which are typical of *CRUD*, and adopt a broader set of functionalities. These APIs have an average active time of about 1,019 days, meaning they tend to be stable and actively maintained for nearly three years on average. This highlights that moving from *CRUD* to *REST* is a natural step as systems grow more complex and require richer functionality.

Interestingly, the reverse transition, from *REST* back to *CRUD*, is also quite common, with 239 APIs undergoing this change. This may happen when systems simplify their operations, focusing on essential actions. These APIs show a slightly shorter active time of 961 days, suggesting that simplifying an API's design might be a response to changing priorities or a need for easier maintenance.

The heatmap of transition times gives additional context. Moving from *CRUD* to *REST* is one of the quickest changes, taking an average of 14 days. This suggests that such transitions are relatively straightforward and likely involve extending existing functionality rather than completely overhauling the design. On the other hand, transitions like *Read-only* (APIs using only **GET**) to *RPC* (APIs using only **POST**) take longer, averaging 51 days. These slower changes likely reflect more significant shifts in how the API is used, such as moving from primarily reading data to focusing on operations or commands.

Another notable trend is the stability of APIs that stick to simpler designs. For example, *CRUD* APIs, with 394 instances, are the backbone of many systems. These APIs have an average active time of 1,017 days and an average age of 1,760 days. Their long lifespan shows that basic operations are fundamental to many applications and are often preserved as core components over time.

Meanwhile, APIs moving from *CRUD* to *REST* and then to *RPC* are much rarer, with only two examples in the dataset. These more complex transitions involve significant design changes and longer active times, averaging 1,565 days. Such transitions suggest a shift in the API's role, possibly adapting to new architectural styles or system requirements.

Table 4.3. Transition chain analysis for APIs with more than 50 commits. Avg. AT (Average Active Time) is the average number of days between the first and last commits, while Avg. A (Average Age) is the average number of days from the first commit to December 31, 2024. The table also includes the maximum, minimum, and standard deviation of ages.

Transition Chain	#APIs	#Commits	Avg. AT	Avg. Age	Max Age	Min Age	Std Dev Age
CRUD	394	40779	1017.2	1760.1	3363.0	25.0	761.6
CRUD → REST	241	32353	1019.3	1818.3	3332.0	22.0	811.4
CRUD → REST → RPC	2	173	1565.1	1832.4	2385.0	979.0	688.7
CRUD → REST → RPC → Read/Write	3	264	170.8	1266.6	1573.0	1159.0	151.5
CRUD → REST → Read-only	4	610	641.5	1678.2	1987.0	740.0	417.2
CRUD → REST → Read-only → Read/Write	1	264	1392.0	1509.0	1509.0	1509.0	0.0
CRUD → REST → Read/Write	14	2001	1334.1	2114.9	2227.0	1154.0	328.2
CRUD → RPC	4	1502	1439.7	1993.0	1993.0	1993.0	0.0
CRUD → RPC → REST	3	1414	731.6	892.6	2371.0	469.0	775.3
CRUD → RPC → Read/Write → REST	1	40	854.0	1977.0	1977.0	1977.0	0.0
CRUD → Read-only	4	574	1097.0	1452.7	2226.0	714.0	469.3
CRUD → Read-only → REST	3	318	1102.7	1458.6	2126.0	928.0	366.2
CRUD → Read-only → REST → Read/Write	1	385	1265.0	1608.0	1608.0	1608.0	0.0
CRUD → Read-only → RPC → REST → Read/Write	2	207	1026.2	2328.4	2767.0	1054.0	749.4
CRUD → Read-only → Read/Write	1	24	1351.0	2589.0	2589.0	2589.0	0.0
CRUD → Read-only → Read/Write → REST	4	247	339.7	1075.4	2196.0	637.0	619.2
CRUD → Read/Write	4	300	1038.4	2072.7	2227.0	1593.0	272.5
CRUD → Read/Write → REST	8	1213	1328.4	2110.6	2368.0	1028.0	433.7
CRUD → Read/Write → RPC	1	59	1128.0	1647.0	1647.0	1647.0	0.0
CRUD → Read/Write → Read-only	3	1347	2061.0	2243.0	2243.0	2243.0	0.0
REST	1292	198916	923.8	1557.6	3335.0	21.0	846.5
REST → CRUD	239	28960	960.8	1754.8	3596.0	227.0	633.3
REST → CRUD → RPC	1	121	1411.0	1663.0	1663.0	1663.0	0.0
REST → CRUD → Read-only → Read/Write	2	104	727.0	995.0	995.0	995.0	0.0
REST → CRUD → Read/Write	3	254	202.2	2340.5	3562.0	1433.0	830.4

Continued on next page

Table 4.3. Transition chain analysis for APIs with more than 50 commits

Transition Chain	#APIs	#Commits	Avg. AT	Avg. Age	Max Age	Min Age	Std Dev Age
REST → RPC	19	1922	922.8	1465.1	2932.0	611.0	453.7
REST → RPC → CRUD	8	1395	840.3	1545.0	1572.0	1048.0	116.0
REST → RPC → Read-only → Read/Write	2	154	29.1	882.9	884.0	882.0	1.0
REST → RPC → Read/Write	11	2355	459.2	804.8	1979.0	411.0	434.9
REST → RPC → Read/Write → CRUD	5	423	586.8	1221.0	1432.0	833.0	244.7
REST → Read-only	24	2265	875.2	1544.3	2622.0	107.0	804.6
REST → Read-only → CRUD	11	1769	1567.8	2154.5	3034.0	819.0	879.5
REST → Read-only → CRUD → Read/Write	1	66	593.0	654.0	654.0	654.0	0.0
REST → Read-only → RPC	1	15	288.0	1164.0	1164.0	1164.0	0.0
REST → Read-only → RPC → Read/Write → CRUD	1	108	1080.0	1175.0	1175.0	1175.0	0.0
REST → Read-only → Read/Write	16	1768	1021.4	1541.2	2678.0	718.0	500.2
REST → Read-only → Read/Write → CRUD	7	1083	1334.9	1553.3	3416.0	573.0	1127.8
REST → Read/Write	58	6861	1109.0	1507.0	3219.0	469.0	727.6
REST → Read/Write → CRUD	15	2344	643.2	1308.4	2869.0	725.0	564.9
REST → Read/Write → CRUD → Read-only	1	62	26.0	1344.0	1344.0	1344.0	0.0
REST → Read/Write → RPC	2	104	905.0	2069.0	2069.0	2069.0	0.0
REST → Read/Write → Read-only	2	311	363.0	1336.5	2014.0	789.0	610.0
REST → Read/Write → Read-only → RPC	1	107	1347.0	1953.0	1953.0	1953.0	0.0
RPC	146	13297	1202.4	1587.4	3118.0	469.0	534.4
RPC → CRUD	5	380	448.9	1262.6	1705.0	931.0	286.8
RPC → CRUD → REST	2	356	1694.3	1812.3	1993.0	1047.0	372.4
RPC → CRUD → Read/Write → REST	1	70	68.0	901.0	901.0	901.0	0.0
RPC → REST	50	5682	957.2	1650.1	2782.0	287.0	536.2
RPC → REST → CRUD	15	2344	379.1	1300.2	2460.0	560.0	562.6
RPC → REST → Read-only	1	88	497.0	931.0	931.0	931.0	0.0
RPC → REST → Read-only → Read/Write	1	100	308.0	1274.0	1274.0	1274.0	0.0
RPC → REST → Read/Write	17	1430	607.6	1828.3	2394.0	721.0	609.3
RPC → REST → Read/Write → CRUD	2	170	789.4	1194.8	1687.0	654.0	517.5
RPC → Read-only → REST → Read/Write → CRUD	2	133	1460.6	1894.0	1894.0	1894.0	0.0
RPC → Read-only → Read/Write	1	64	27.0	1177.0	1177.0	1177.0	0.0
RPC → Read-only → Read/Write → CRUD → REST	1	63	781.0	1120.0	1120.0	1120.0	0.0
RPC → Read-only → Read/Write → REST	1	90	102.0	1315.0	1315.0	1315.0	0.0
RPC → Read-only → Read/Write → REST → CRUD	1	63	781.0	1120.0	1120.0	1120.0	0.0
RPC → Read/Write	59	8085	558.6	1205.1	2184.0	480.0	491.9
RPC → Read/Write → CRUD	5	388	968.9	2064.4	2574.0	1352.0	444.7
RPC → Read/Write → CRUD → REST	2	217	77.5	1114.3	1594.0	918.0	307.6
RPC → Read/Write → CRUD → Read-only → REST	1	52	250.0	629.0	629.0	629.0	0.0
RPC → Read/Write → REST	29	3352	979.0	1468.6	2394.0	693.0	454.9
RPC → Read/Write → REST → CRUD	21	3587	740.4	1497.4	1925.0	776.0	310.5
RPC → Read/Write → REST → Read-only	3	1777	2403.0	2468.1	2540.0	1000.0	325.1
RPC → Read/Write → Read-only → REST → CRUD	1	250	381.0	602.0	602.0	602.0	0.0
Read-only	109	12070	867.6	1357.3	3247.0	469.0	582.8
Read-only → CRUD	11	1704	1166.5	1695.0	2867.0	663.0	560.6
Read-only → CRUD → REST	14	1559	1351.3	1787.6	2840.0	340.0	602.7
Read-only → CRUD → Read/Write	1	56	178.0	967.0	967.0	967.0	0.0
Read-only → CRUD → Read/Write → RPC → REST	1	87	2968.0	3013.0	3013.0	3013.0	0.0
Read-only → REST	77	8446	1075.4	1587.3	3516.0	602.0	596.5
Read-only → REST → CRUD	31	3786	882.3	1692.1	3111.0	622.0	612.5
Read-only → REST → RPC → Read/Write	2	126	40.0	963.0	963.0	963.0	0.0
Read-only → REST → RPC → Read/Write → CRUD	1	118	140.0	1250.0	1250.0	1250.0	0.0
Read-only → REST → Read/Write	33	3775	1412.6	1958.9	3486.0	774.0	740.5
Read-only → REST → Read/Write → CRUD	3	265	1181.3	2565.7	2806.0	1722.0	413.3
Read-only → REST → Read/Write → RPC	1	153	314.0	2702.0	2702.0	2702.0	0.0
Read-only → REST → Read/Write → RPC → CRUD	1	57	25.0	1781.0	1781.0	1781.0	0.0
Read-only → RPC	5	294	1325.0	1559.7	1613.0	1537.0	34.9
Read-only → RPC → REST	1	294	1474.0	1516.0	1516.0	1516.0	0.0
Read-only → RPC → REST → Read/Write	1	57	2.0	629.0	629.0	629.0	0.0
Read-only → RPC → Read/Write → REST → CRUD	2	200	920.6	1804.4	2244.0	1535.0	345.0
Read-only → Read/Write	59	4858	933.2	1612.3	3170.0	94.0	637.5
Read-only → Read/Write → CRUD	6	434	934.5	1917.1	2670.0	941.0	590.7
Read-only → Read/Write → CRUD → REST	6	477	1192.5	2182.1	3266.0	941.0	703.1
Read-only → Read/Write → REST	69	6323	938.6	1691.4	3483.0	321.0	735.5
Read-only → Read/Write → REST → CRUD	37	4033	681.9	1434.1	3099.0	234.0	647.8
Read-only → Read/Write → REST → CRUD → RPC	1	79	265.0	731.0	731.0	731.0	0.0

Continued on next page

Table 4.3. Transition chain analysis for APIs with more than 50 commits

Transition Chain	#APIs	#Commits	Avg. AT	Avg. Age	Max Age	Min Age	Std Dev Age
Read-only → Read/Write → RPC	6	1060	1326.2	1498.5	1595.0	1047.0	195.9
Read-only → Read/Write → RPC → REST	1	95	813.0	861.0	861.0	861.0	0.0
Read-only → Read/Write → RPC → REST → CRUD	2	113	538.3	1643.0	2378.0	811.0	785.5
Read/Write	215	29305	896.2	1530.5	2888.0	23.0	516.9
Read/Write → CRUD	20	2024	791.2	1555.4	3275.0	772.0	830.6
Read/Write → CRUD → REST	12	1210	566.3	1896.3	2765.0	1111.0	598.7
Read/Write → REST	241	28727	1001.0	1702.9	3528.0	199.0	692.6
Read/Write → REST → CRUD	94	9902	1075.8	1831.7	3471.0	698.0	755.5
Read/Write → REST → CRUD → Read-only	1	70	2569.0	2765.0	2765.0	2765.0	0.0
Read/Write → REST → RPC	5	635	698.2	1867.7	2831.0	1356.0	520.8
Read/Write → REST → RPC → CRUD	1	221	2765.0	3412.0	3412.0	3412.0	0.0
Read/Write → REST → RPC → Read-only	1	63	136.0	742.0	742.0	742.0	0.0
Read/Write → REST → RPC → Read-only → CRUD	1	150	1638.0	2345.0	2345.0	2345.0	0.0
Read/Write → REST → Read-only	7	468	622.9	1708.9	2907.0	1383.0	563.4
Read/Write → REST → Read-only → RPC	2	340	915.9	1065.8	1110.0	1042.0	32.5
Read/Write → RPC	10	613	612.4	3210.6	6435.0	715.0	2049.6
Read/Write → RPC → REST	4	352	948.2	1446.0	1925.0	948.0	342.0
Read/Write → RPC → REST → CRUD	3	235	1054.1	1360.2	1417.0	1291.0	62.8
Read/Write → RPC → Read-only → REST	1	51	82.0	1730.0	1730.0	1730.0	0.0
Read/Write → Read-only	8	1207	913.3	1313.5	1680.0	713.0	325.6
Read/Write → Read-only → CRUD	1	147	452.0	851.0	851.0	851.0	0.0
Read/Write → Read-only → REST	24	2537	792.4	1538.2	2056.0	690.0	323.2
Read/Write → Read-only → REST → CRUD	7	627	999.8	2412.2	3193.0	1303.0	617.3

☛ Most APIs with long histories (more than 50 commits) have simple functions, like CRUD-only (394 APIs) or REST-only (1292 APIs), meaning they stick to basic operations. However, some APIs are becoming more complex, with transitions like CRUD → REST → RPC or REST → Read/Write → CRUD, though these are less common. This suggests that while most APIs stay simple, a few are adding more features and becoming more advanced over time, or simplifying existing operations.

4.1.3 Data model alignment considerations

The level of data normalization is considered.

Paths and responses are often aligned with the normalization state of the data:

- Normalized resources are typically exposed through separate endpoints, such as `/products/{id}` for individual products.
- Embedded-related data may be included in responses when appropriate to reduce client-side API calls. For example, order details might include embedded user information.

Conforming to data normalization in APIs means ensuring that the structure of paths and responses aligns with the normalized state of the data. This involves exposing individual resources through dedicated endpoints, such as `/products/{id}` for accessing specific items, which reduces redundancy and provides a clean, scalable design. Additionally, related data may be embedded in responses where appropriate to optimize client-side performance and minimize multiple API calls. For example, an `/orders` endpoint could include embedded user details to avoid the need for a separate request to `/users/{id}`. APIs that follow this principle achieve a balance between normalized resource access and efficient data delivery, improving performance, scalability, and usability while reducing redundancy.
We verify in our analysis conformance to data normalization by evaluating two key principles:

Table 4.4. Analysis of alignment of API datamodel

Metric	**Count**
APIs with Schemas	3377 (out of 3711)
APIs with Embedded-Related Data	3128
APIs with Normalized Resources	3202
APIs with Normalized Resources and Embedded-Related Data	2993

1. **Normalized Resources:** Endpoints must follow a standardized path structure, such as `/resource/{id}`, which is used to access individual items.

    - For example, the endpoint `/products/{id}` retrieves details for a specific product.

    This is achieved by matching API paths against a regular expression that identifies such patterns.

2. **Embedded-Related Data:** Response schemas are examined for embedded data, such as nested objects (`type: object`) or arrays (`type: array`), that represent related resources. Including such data optimizes client-side performance by reducing the need for multiple API calls.

    - For example, an `/orders` endpoint might include user details directly as an embedded object to avoid an additional request to `/users/{id}`.

The analyzer ensures that an API conforms to normalization if it satisfies at least one of the following conditions:

- It exposes resources through normalized endpoints (e.g., `/resource/{id}`).
- It includes embedded-related data in its responses.

By combining these checks, the script determines the degree to which APIs align with data normalization principles, ensuring a balance between resource separation and efficient data delivery. Table 4.4 summarizes the analysis output.

Figure 4.9 shows the distribution of the number of distinct schemas (left) and the number of distinct properties of the schemas (right) between conforming and non-conforming APIs. Both plots show that the data model of the conforming APIs tends to have a more diverse and rich data model in terms of number of schemas and diverse properties.

Immutable resources are not expected to expose update methods.

Resources representing entities that should be immutable for clients, such as event logs or transaction records, are generally read-only. Methods like **`GET`** are employed for retrieval, and **`POST`** is used for creation.

The verification of the adoption of this principle uses a heuristic approach to identify immutable resources by matching their paths with predefined keywords like `logs`, `transactions`, or `history`, which typically represent entities that should not be modified after creation. For each identified resource, the HTTP methods are analyzed to ensure that only `GET` (for retrieval) and `POST` (for creation) are used. The presence of update methods (`PUT` or `PATCH`) or delete methods (`DELETE`) on these resources is flagged as a violation, as such operations contradict the immutability principle. This heuristic efficiently detects potential design flaws in APIs handling immutable entities.

The results in Table 4.5 show that out of the 4599 paths that seem to handle immutable resources, about 12% of them also allow **`PUT`**, **`PATCH`** or **`DELETE`**.

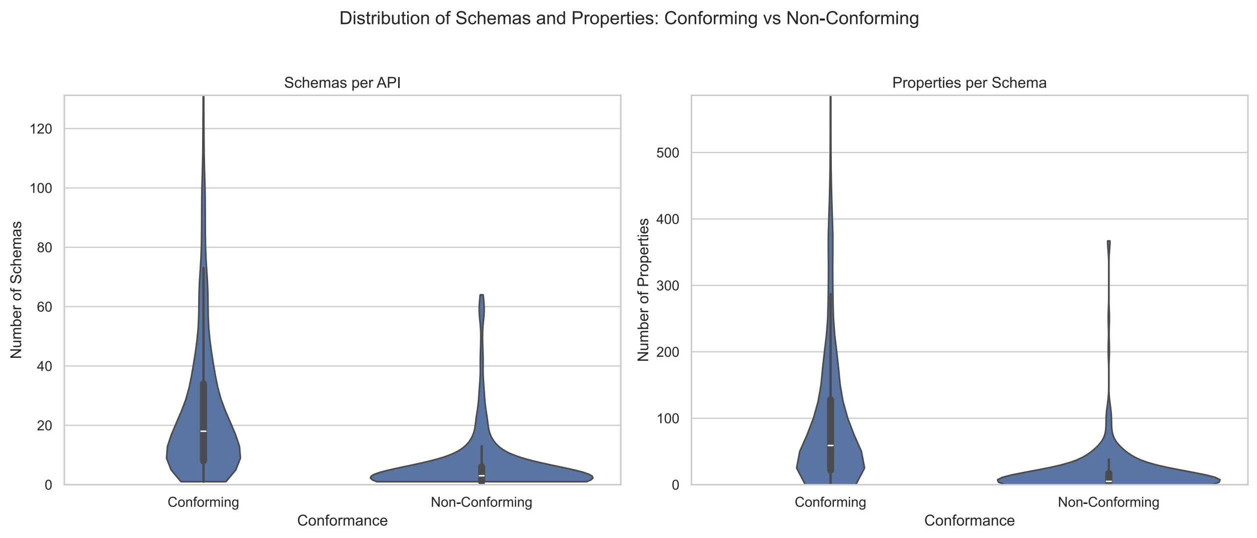


Figure 4.9. Distribution of Schemas and Properties in Conforming vs Non-Conforming APIs. The left plot illustrates the number of schemas per API, while the right plot shows the number of properties per schema. Conforming APIs exhibit higher variability and concentration in both metrics compared to Non-Conforming APIs.

Table 4.5. Analysis of operations applies on potential immutable ressources

Metric	**Count**
Total APIs Analyzed	3711
Total Paths Analyzed	110398
Paths Matching Immutable Resources	4599
Violating Paths (PUT, PATCH, DELETE on Immutable Resources)	549

4.2 Towards Assessing Web API Complexity

While size metrics defined in Chapter 3 helps to measure the size of a Web API in terms of its structure. They can also be used to evaluate the complexity of the API.

Striking the right balance between functionality and simplicity is critical to ensuring that APIs are powerful yet user-friendly, enabling seamless integration and long-term scalability. Moreover, adhering to the guidelines and design rules[94, 89, 75] positively impacts the understandability of the API [23].

The complexity of an API should be controlled and driven by the design decision adopted taking into account the alignment of the paths and operations with the underlying data model, answering the defined requirements with the highest performance.

4.2.1 Existing definitions

Previous researchers suggested different methods to measure Web API complexity. The authors Jonnada and Joy [77] represent each API resource as an undirected graph. In this graph, nodes represent the resource, its methods (API calls), and the input parameters for these methods. Edges define the relationships between these components, such as links between the resource and its methods or between methods and their input parameters. The complexity of this graph is quantified using Shannon's entropy, which measures the information content based on the connectivity of the nodes. The formula for the complexity of a node is:

$$\text{Node Complexity} = -\sum_{i=1}^{N} \frac{d_i}{D} \log_2\left(\frac{d_i}{D}\right)$$

where N is the total number of nodes, d_i is the degree (number of connections) of node i, and D is the total adjacency, calculated as the sum of the degrees of all nodes. The total complexity of the resource (TR_c) is the sum of the complexities of its resource node, method nodes, and parameter nodes. By summing the complexities of all resources, the total API complexity (API_c) is determined. This method provides a structured way to quantify the complexity of an API based on its structure and relationships.

This complexity metric is meaningful in the context of graphs because it captures structural properties like the number of nodes and edges, degrees of connectivity, and information content using Shannon's entropy. In graph theory, these properties are sufficient for understanding complexity since graphs are abstract representations of relationships and do not require semantic or functional interpretation. For instance, the metric can effectively compare graphs with different topologies by quantifying their structural differences.

However, in the case of Web APIs, the metric becomes less meaningful because APIs are not just structural entities but also semantic and functional systems. Web APIs involve logical relationships between resources, hierarchical path structures, and contextual interactions that go beyond simple node and edge connections. For example, a deeply nested path like `/models/{model_id}/versions/{version_id}` represents a logical dependency that adds semantic complexity not captured by graph-based metrics. Moreover, the uniform treatment of parameters and methods in the metric overlooks critical aspects such as the variability of data types, the dynamic behavior of APIs, and their state transitions. As other software, APIs are also designed with usability, scalability, and maintainability in mind, which are qualitative dimensions that a purely structural metric cannot measure. Which makes this metric insufficient to measure these aspects when used solely.

In the case of De Souza and Bentolila [40] who defined complexity metrics for APIs (rather than Web API), the complexity is quantitatively assessed using structural metrics that evaluate the interface specifications of methods, classes. The *Interface Size Metric* assesses the complexity of a method based on the number and types of its parameters, with methods having more and complex parameter types being considered more complex. The *Interaction Level Metric* measures the degree of interconnectivity between components by evaluating the potential interactions within a class or method, where higher interaction levels indicate greater complexity. Additionally, the *Operation Argument Complexity Metric* evaluates the overall complexity of method parameters by assigning weights to different parameter types based on their simplicity or complexity. These metrics are aggregated to determine the overall complexity of an API, which is further benchmarked statistically against other APIs, placing their complexity scores into deciles. This metric helps to evaluate the complexity from both the perspective of user and maintainer.

Complexity is also introduced with a high degree of feature diversity, especially when these features are not intuitively designed or aligned with user and developer expectations [192]. Feature diversity, while potentially powerful, increases cognitive load if users are required to navigate through disparate functionalities without clear or predictable patterns. It also increase the maintainability costs since developers are supposed to assist a larger set of functionalities. Intuitiveness is compromised when the tool deviates from established conventions, uses inconsistent terminology, or presents unclear workflows. Complexity further escalates when features interact in unexpected or non-transparent ways, requiring users to invest additional effort to understand dependencies and outcomes. Thus, a tool's complexity is not solely determined by the breadth of its capabilities, but also by how well these capabilities are integrated into a cohesive, user-friendly experience.

4.2.2 A Specific complexity metric for web APIs

Since Web API code is generally only accessible by its developers, we aim to propose a metric that can still reflect the maintainability cost but does not require code-based measuring, such as known software complexity metrics (e.g., Maintainability Index, Cyclomatic Complexity [173, 151, 102]). We define the complexity of a Web API as a combination of the cognitive load introduced by a high number and diversity of features (endpoints), the complexity of its data model (schemas size), and the divergences from expected design conventions, measured using specific metrics. The API complexity score (C_{API}) is calculated as:

$$C_{API} = w_f \cdot F_C + w_{DM} \cdot DM_C + w_S \cdot S_C + w_d \cdot D_C$$

Where:

- **Feature Complexity (F_C):** The feature complexity metric ($F_C = \frac{N_o}{N_p}$) focuses on the operational diversity per path. It is calculated as the ratio of the total number of operations (N_o) to the total number of paths (N_p) in the API. This metric, referred to in Chapter 3 as *Number of Operations per Path*, provides insight into how diverse the operations are for each path.

- **data model Complexity (DM_C):** This metric measures the complexity of the API's data model as the number of properties per schema:

  $$DM_C = \frac{N_{pr}}{N_{sc}}$$

  Where:

  - N_{pr}: Total number of unique properties across all schemas.
  - N_{sc}: Total number of schemas defined in the API's specification (e.g., request/response models).

  This metric, referred to as *Number of Properties per Schema* in Chapter 3, reflects the density and complexity of schema definitions.

- **Size Complexity (S_C):** This metric quantifies the API's size by combining the total number of paths and schemas, with weights reflecting their importance:

  $$S_C = w_{pr} \cdot N_p + w_{sc} \cdot N_{sc}$$

  Where:

  - N_p: Total number of paths.
  - N_{sc}: Total number of schemas.
  - w_{pr}: Weight for the importance of paths (default $w_{pr} = 1$).
  - w_{sc}: Weight for the importance of schemas (default $w_{sc} = 1$).

  By adjusting the weights, this metric can prioritize either paths or schemas based on the API's domain and use case.

- **Design Divergence Complexity (D_C):** This metric measures deviations from expected design conventions, which can increase cognitive load and make the API harder to use [23]. It is calculated as:

  $$D_C = \sum_{i=1}^{N_e} EC_i$$

  Where:

- N_e: Total number of endpoints.
- EC_i: The divergence score for the i-th endpoint:

$$EC_i = \sum_{j=1}^{N_d} w_{d_j} \cdot D_j$$

- N_d: Number of design rules evaluated for the endpoint. This allows for flexibility in adding or modifying design rules.
- D_j: Deviation of the j-th design element. Examples include:
  - ⋆ Path and Naming Consistency: Plural resource names returning arrays (+0) vs. single objects (+1).
  - ⋆ Parameter Design: Query parameters as objects (+1) or nested structures (+1 per level).
  - ⋆ Response Format: Inconsistent or deeply nested responses (+1 per extra nesting level).
  - ⋆ HTTP Method Semantics: Incorrect usage, such as a 'GET' operation modifying server state (+1).
- w_{d_j}: Weight reflecting the importance of each design rule.

Weights (w_f, w_{DM}, w_S, and w_d): These weights allow prioritization of different aspects of complexity. For example:

- A data-heavy API might assign a higher weight to w_{DM}, emphasizing data model complexity.
- An API with many endpoints might prioritize w_f to reflect the impact of feature diversity.
- If design consistency is critical, w_d can be increased to penalize divergences more heavily.

By adjusting these weights, the metric can be tailored to reflect specific API design priorities, ensuring that the complexity score aligns with the API's intended purpose and domain.

This interpretation of the metric is meaningful when used to compare two APIs or as suggested by Lanza and Marinescu[88] to compare an API with a set of pre-computed complexity values of a large number of APIs.

Example

As an example, in Table 4.6 we compute the complexities of the API described in the snippets in Listings 4.2 and 4.5.

Listing 4.5. OpenAPI schema for a collection of users with metadata and possibility to fetch users by their ids

```
paths:
 /users:
  get:
   summary: Retrieve a list of users
   parameters:
    - name: page
     in: query
     schema:
      type: integer
     description: Page number
    - name: limit
     in: query
     schema:
      type: integer
     description: Number of results per page
   responses:
    '200':
     description: A JSON array of user objects
     content:
      application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    id:
                      type: string
                      description: Unique identifier for the user
                    name:
                      type: string
                      description: Name of the user
                    email:
                      type: string
                      description: Email of the user
                    address:
                      type: object
                      properties:
                        street:
                          type: string
                        city:
                          type: string
                        zipcode:
                          type: string
  /users/{id}:
    get:
      summary: Retrieve details of a specific user
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: string
          description: ID of the user
      responses:
        '200':
          description: A JSON object representing the user
          content:
            application/json:
              schema:
                type: object
                properties:
                  id:
                    type: string
                  name:
                    type: string
                  email:
                    type: string
                  address:
                    type: object
                    properties:
                      street:
                        type: string
                      city:
                        type: string
                      zipcode:
                        type: string
```

The Shannon entropy-based complexity is computed as:

- For the simple API (Listing 4.2):

$$N = 4, \quad D = 6$$
$$d_i = \{3, 1, 1, 1\}$$

  Substituting into the formula:

$$\text{Node Complexity} = -\left(\frac{3}{6}\log_2\frac{3}{6} + \frac{1}{6}\log_2\frac{1}{6} + \frac{1}{6}\log_2\frac{1}{6} + \frac{1}{6}\log_2\frac{1}{6}\right)$$

Metric	Simple API (Listing 4.2)	Complex API (Listing 4.5)
F_C (Feature Complexity)	1	1
DM_C (Data Model Complexity)	2	4
S_C (Size Complexity)	2	4
D_C (Design Divergence Complexity)	0	1
C_{API} (Total Complexity)	$2 \cdot 1 + 1 \cdot 2 + 1 \cdot 2 + 1 \cdot 0 = 4$	$2 \cdot 1 + 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 1 = 11$

Table 4.6. Comparison of complexity metrics for two API examples

$$\text{Node Complexity} = -(0.5 \cdot (-1) + 0.167 \cdot (-2.585) + 0.167 \cdot (-2.585) + 0.167 \cdot (-2.585))$$

$$\text{Node Complexity} \approx 1.0 + 0.861 + 0.861 + 0.861 = 3.583$$

- For the complex API (Listing 4.5):

$$N = 12, \quad D = 16$$
$$d_i = \{4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$$

Substituting into the formula:

$$\text{Node Complexity} = -\left(\frac{4}{16}\log_2\frac{4}{16} + \frac{3}{16}\log_2\frac{3}{16} + \frac{2}{16}\log_2\frac{2}{16} + 7 \cdot \frac{1}{16}\log_2\frac{1}{16}\right)$$

$$\text{Node Complexity} = -(0.25 \cdot (-2) + 0.1875 \cdot (-2.415) + 0.125 \cdot (-3) + 7 \cdot 0.0625 \cdot (-4))$$

$$\text{Node Complexity} \approx 0.5 + 0.453 + 0.375 + 1.75 = 3.078$$

Compared to our proposed metrics, the Shannon Entropy does not reflect differences in complexity, as the simple API appears slightly more complex. This is due to its inability to account for semantic depth or design intricacies by considering all the nodes as similar API elements.

In Summary

The goal of the analysis presented in this chapter is to combine the metrics defined in Chapter 3, to be used as indicators and study more complex features such as the correlations between data model and API structure, and the relationships between the HTTP methods employed in these structures. Key findings reveal that while API designs might be technically "RESTful", they still exhibit unexpected design decisions, such as poor utilization of logical relationships between path segments and inadequate labeling practices (e.g., using plural forms to retrieve single resources or vice versa). These design inconsistencies with established expectations highlight the need for API design tools that can proactively guide API designers by identifying potential design smells and providing actionable warnings, thereby promoting more consistent, maintainable, and semantically aligned API designs.

Chapter 5

OAS2Tree: Visualizing Web APIs as Trees

Web APIs are essential for enabling seamless communication between software systems [189]. However, their sheer size and complexity [139] can make them challenging to understand [58]. While visualizing the overall architecture can significantly aid in understanding the interactions between a system's components, focused visualizations of crucial elements within a complex system, such as web APIs, can clarify the structure, help ensure the correct flow of data, and determine which operations should be exposed.

In this chapter, we introduce a visualization that emphasizes the tree structure of a web API. This representation, which we refer to as the *web API tree*, highlights the branching structure between paths and the operations, parameters, and responses associated with them, using a selected graphical notation. *The web API tree* is supported by a tool called OAS2Tree, which generates the tree structure from OpenAPI specifications. We present the design and implementation of OAS2Tree, showcasing its features and use cases. In addition, we discuss the limitations of the current version and explore potential future improvements and extensions.

5.1 Representing the API Structure as a Tree

5.1.1 API Tree Model

We transform the textual documentation related to the resources and the methods supported by the API into a tree data model, to represent the nesting relationships between the API endpoint URIs, enumerated as paths in the OpenAPI specification. This model has two purposes:

1. It can be used to visualize the functional characteristics of the APIs graphically, to provide a quick overview supporting the understanding of the APIs structure.

2. The second purpose of this tree data model, described in Figure 5.3, is to help to rapidly spot commonly used patterns by analyzing reoccurring fragments found within a large set of APIs. The elements colored in gray in Figure 5.3 are the ones being mapped to graphical notations for being visualized in the API Tree representation.

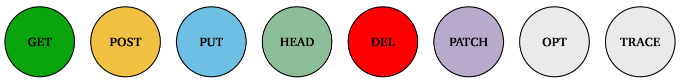
An example of an API Tree model visualization is in Figure 5.4. Each API operation, originally listed in the OpenAPI file, can be enumerated by following the path from the root until reaching a leaf of the tree. These later represent the HTTP methods, enumerated in each path in the Open API description. The nodes within the tree are labeled with the corresponding URI path segment and labeled depending on the type of the path segment (Table 5.1). The types of nodes are explained in Section 6.5.2.Due to this graphical representation, we can also visually detect a repetitive usage of an API Fragment in an API. This same fragment can also reoccur in other APIs, with different labels.

In Table 5.1, we summarize the notation used in our APIs Tree visualization.

Table 5.1. API Tree notation

Name	Notation	Signification
Root	●	The root of the API Tree.
Method	○	The HTTP methods, where each method has a specific color.
Static path segment	□	Path segment with no parameter
Parametric path segment	■	Path segment with single {parameter}
Unusual path segment	▨	Path segment mixing parameters with static labels
Query Parameter Collection	?	A node connected the method to indicate the existence of a query parameter. It is colored with the same color as the method.
Responses Collection	>	A node connected the method to indicate the existence of response schema object documentation. It is colored with the same color as the method.

The HTTP Methods are visualized in a circular shape. We attribute a specific color to each HTTP method to make it easier to identify them in the visual representation. The colors we adopted are closely similar to the color coding used in Postman [126]. The colors are as follows:



Listing 5.1. Example of API paths description ins an OpenAPI document

```
paths:
  /users:
    get:
      summary: "Retrieve␣the␣list␣␣of␣all␣registered␣users."
      responses:
        "200":
          description: "A␣list␣of␣users."
  /users/{id}/details:
    get:
      summary: "Retrieve␣details␣of␣a␣user␣by␣their␣ID."
      responses:
```

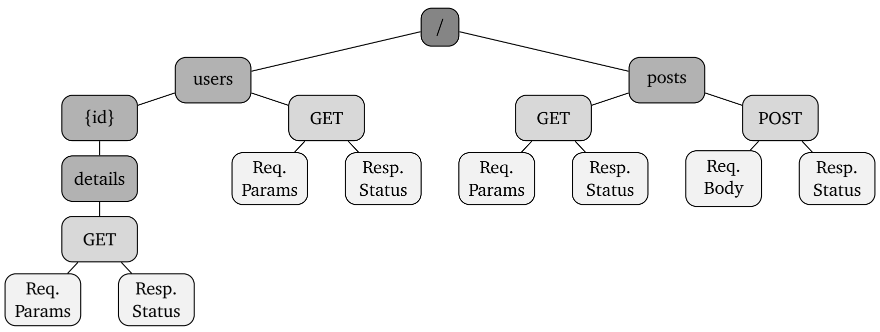


Figure 5.1. Extracted API tree structure

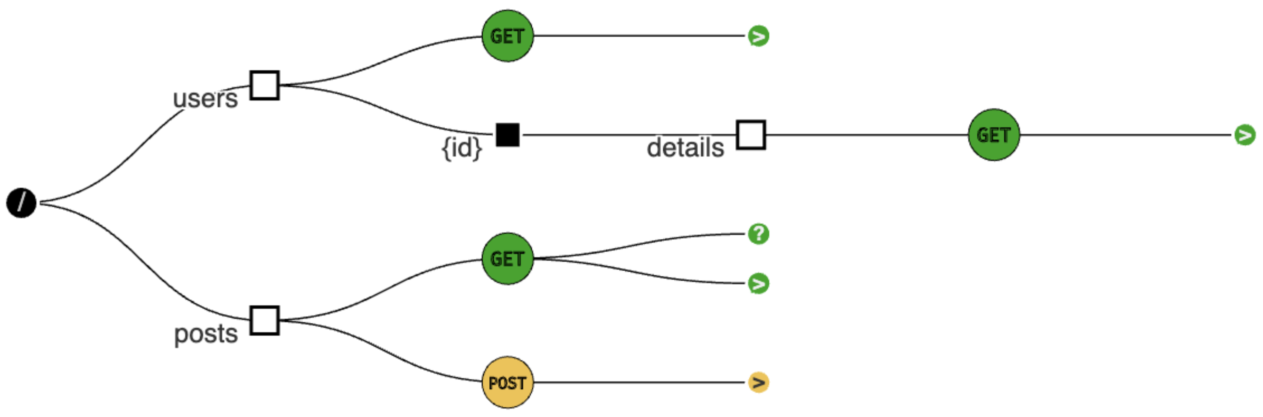


Figure 5.2. OAS2Tree visualization of API in Listing 5.1

```
    "200":
      description: "The␣user␣information."
    "404":
      description: "User␣not␣found."
/posts:
  get:
    summary: "List␣all␣posts"
    responses:
      "200":
        description: "A␣list␣of␣all␣available␣posts."
        parameters:
        - name: limit
          in: query
          description: "The␣number␣of␣posts␣to␣return."
          type: integer
        - name: offset
          in: query
          description: "The␣number␣of␣posts␣to␣skip."
          type: integer
  post:
    summary: "Create␣a␣new␣post."
    responses:
      "201":
        description: "The␣created␣post."
```

5.1.2 OpenAPI to Tree model transformation

To visualize the OpenAPI specification as a tree structure, we transform the flat list of paths extracted from an API description into a hierarchical structure by breaking down the paths into segments. When a segment is shared between different paths, we check if these segments share the same sequence of parent segments. If they do, the segments are merged into a single node, as they refer to the same container resource. Once the path segments tree structure is complete we attach to each node the set of HTTP methods of the corresponding endpoint.

OpenAPI also includes details related to the responses of each endpoint and the parameters that can be passed to the endpoint. We attach these details to the corresponding HTTP method node. In the current version of OAS2Tree, we only visualize the response status codes but do not further drill down to show the data models of the request or response schemas.

While the YAML or JSON syntax adopted by OpenAPI can make it hard to locate the operations that are applicable over the same resources but with different HTTP methods, in the case of large APIs with many paths. This

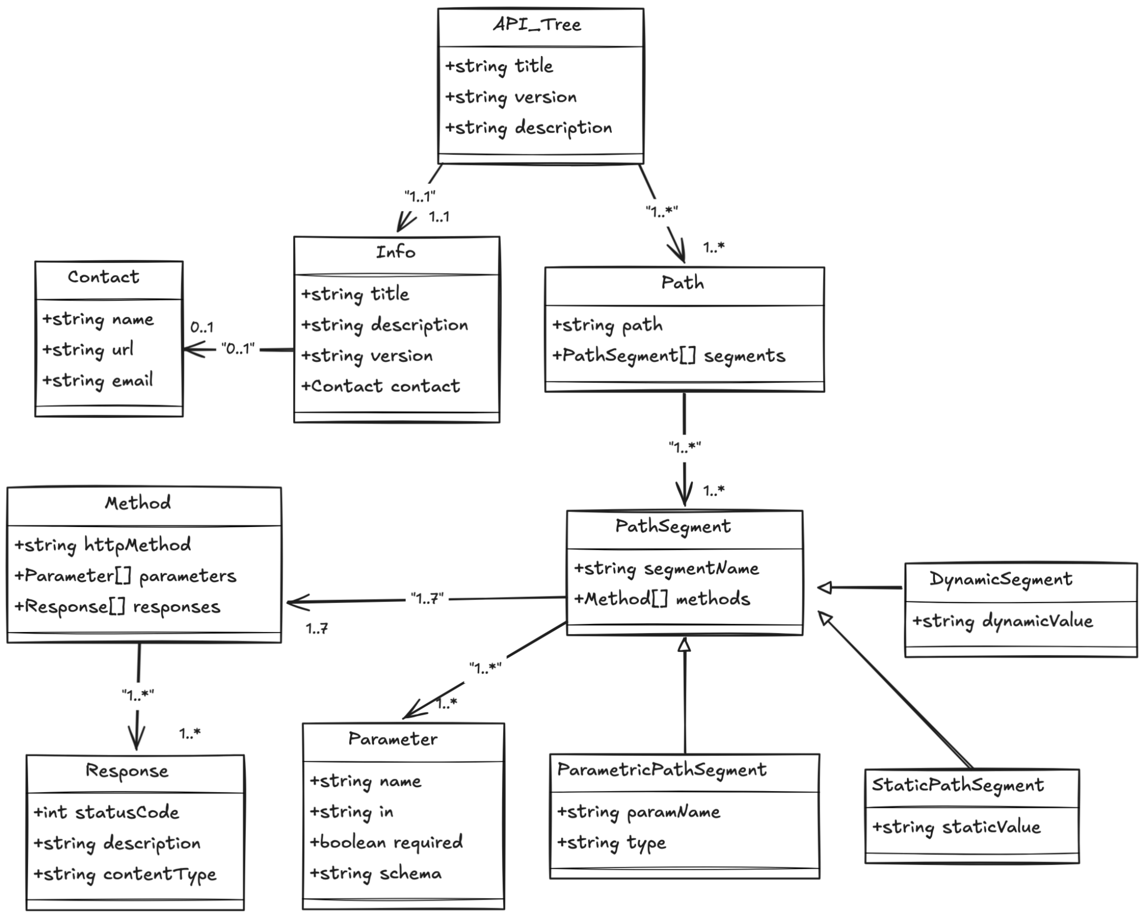


Figure 5.3. Excerpt of the API Tree metamodel, highlighting the visualized elements

hierarchical representation clearly shows the shared segments and their relationships within the API structure.

For explaining the model transformation, responsible of producing the tree visualization of the OpenAPI descriptions, we use the description example in Listing 1, which is an excerpt extracted from the OpenAPI description of Apacta Web API whose API Tree is shown in Figure 5.4.

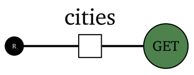
The path `/cities` only contains one path segment {`1:cities`} labeled `cities`. In our transformation, each segment is transformed to a PathSegment object (Figure 5.3). We always connect the first path segment to the Root object ●, an added graphical element which helps to visualize the API model as a tree. The {`1:cities`} path segment has no in-path parameters, thus it is mapped to the static path segment notation □ (Table 5.1). As a result, the obtained first portion of the tree is $\bullet\overset{\text{cities}}{\longrightarrow}\square$, where we label the path segment node 'cities'. Moreover, the PathSegment object contains fields holding some original information such as the summary, description and the parameters information, for further usages. In this study, we only distinguish between paths that are having in-path parameters and the one that don't have them. However we plan to extend the graphical visualization to include also the other type of parameters and the responses details.

This path provides only one GET operation, which allows to get a city by its zip code. The HTTP methods are transformed to the `Method` object, which also keeps most of the original information about the method, such as

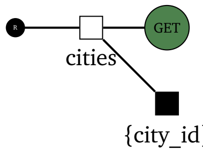
```
paths:
  /cities:
     get:
        parameters:
          - description: Search for a city with specific zip code
          in: query
          name: zip_code
          required: false
          type: string
        responses:
          '200':
           description: OK
           schema:
              ...
          '404':
           description: Not found
           schema:
              ...
        summary: Get list of cities supported in Apacta
  /cities/{city_id}:
     get:
        parameters:
          - in: path
          name: city_id
          required: true
          type: string
        responses:
          '200':
           description: OK
           schema:
              properties:
                 data:
                   ...
           success:
              default: true
              type: boolean
          '404':
           description: Not found
           schema:
               ...
```

Listing 1. Excerpt from the OpenAPI description of the Apacta API shown in Figure 5.4

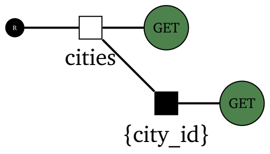
the summary, description, and the response details. This Method object is mapped to the graphical notation: ○, which contains as a label the name of HTTP method and colored in specific color depending on the method. In this case, the notation should be

. And as a result, the whole path visual representation is: 

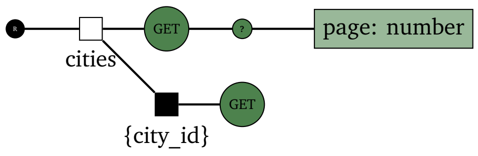
Once all the methods of a path are all transformed, the algorithm jumps to the next path and start extracting the path segments, and put them in a list, respecting their original order. In our example the second path is `/cities/city_id`. It contains tow path segments { 1: `cities`, 2: `city_id` }. The path segment { 1: `cities` } has already been created. Knowing that each next path segment is a child of the previous one, the path segment {2: `city_id`} should be then connected to { 1: `cities` }, which is already created and added to the tree. This new node is also mapped to the `PathSegment` object, and more specifically to the `ParametricPathSegment` object, which is associated to the notation: ■. As consequence, the tree becomes :



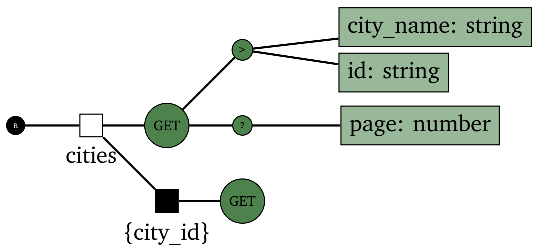
Same as for the previous path, this path also provides only one Get HTTP method. So the API Tree corresponding to the whole OpenAPI description example is:



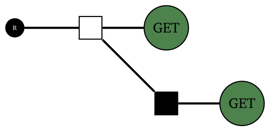
When a method has query parameters, this is visible threw the other node attached to the method with the “?" mark. In this example the **GET**/cities uses pagination to list the cities using the parameter `page`.



When the



The corresponding API Tree Structure for this API Tree is simply obtained by removing all path labels and ignoring the responses and query parameters details:



Looking at the tree model visualization in Figure 5.4, we can notice this same portion of the tree, constructed from the example in Listing 1, appears multiple times with different labels. For computing exactly how much frequently, a specific structure of a tree fragment appears in the set of APIs in our collection, we proceed to apply the fragmentation and matching technique presented in Section 6.5.

5.2 OAS2tree: Tool Support for API Tree Visualization

OAS2Tree dynamically generates visual representations of APIs described using the OpenAPI Specification (OAS) [118]. By offering developers a visual representation instantaneously synchronized from the OpenAPI description text, the tool facilitates obtaining valuable insights and a deeper understanding of the API structure. Furthermore, our tool goes beyond basic visualization capabilities by proactively identifying and highlighting design smells—common issues or inefficiencies in API design. These design smells have been extensively discussed in our research study on API structural patterns and design flaws [148] detailed in Chapter 6.

By providing real-time visual representations of API endpoint and operation structures, the tool facilitates validating the consistent and regular design of API endpoints as a tree of URL paths with color-coded HTTP methods as tree leaves. Additionally, its integration of design smell detection capabilities assists developers in identifying potential issues and refining their APIs. Ultimately, our visualization tool aims to enhance the collaboration and efficiency of incremental API design and review processes, resulting in the creation of well-designed APIs.

The tool is available as both a standalone web app [113] and as a Visual Studio Code (VSCode) extension [114], catering to the diverse needs of developers. The web app version is designed for those who want a quick and convenient way to visualize an API without the need for local tool installation and setup. On the other hand, the VSCode extension version is intended for developers who prefer to have the visualization tool seamlessly integrated into their development environment.

5.3 OAS2Tree Features

5.3.1 API Spec validation Design Smells Detection

OAS2Tree can be used to detect smells in the API design described in the specification. The tool currently supports the detection design smells we empirically identified in our research study on API collection resource patterns [148] presented in Chapter 6. The goal is to alert developers to potential design issues. The design smells detected by OAS2Tree include:

- **Ambiguous PUT and POST endpoints:** When an API contains both PUT and POST operations with similar paths, it can lead to confusion.
- **Create without delete:** When an API allows the creation of resources without providing a corresponding delete operation, it can result in data inconsistencies.
- **Delete without create:** When an API allows the deletion of resources without providing a corresponding create operation, it can lead to data loss.
- **Write-only endpoints:** When an API contains endpoints that only allow write operations without providing read capabilities, it can limit the usability of the API.

In addition, OAS2Tree validates the API specification against the OpenAPI Specification schema to ensure that it adheres to the standard, and highlights any errors or inconsistencies in the document and in the problems view as depicted in Figure 5.5. For both smells and validation errors, the user can navigate to the problematic element in the OAS document by clicking on the error message.

5.3.2 Navigation of API description through the tree visualization

The tree structure visualization allows users to navigate through the API description easily. By expanding and collapsing nodes, users can explore the API structure and view the details of each endpoint, operation, and parameter.

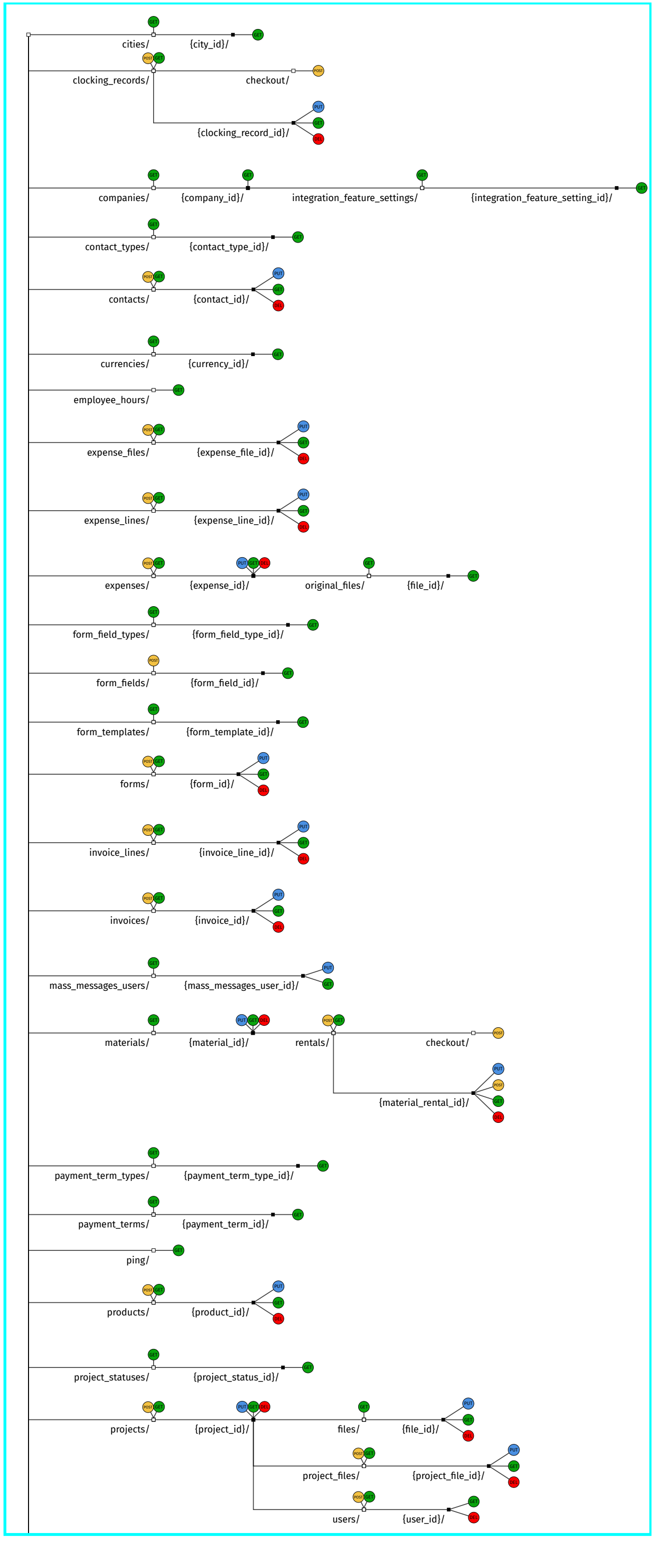
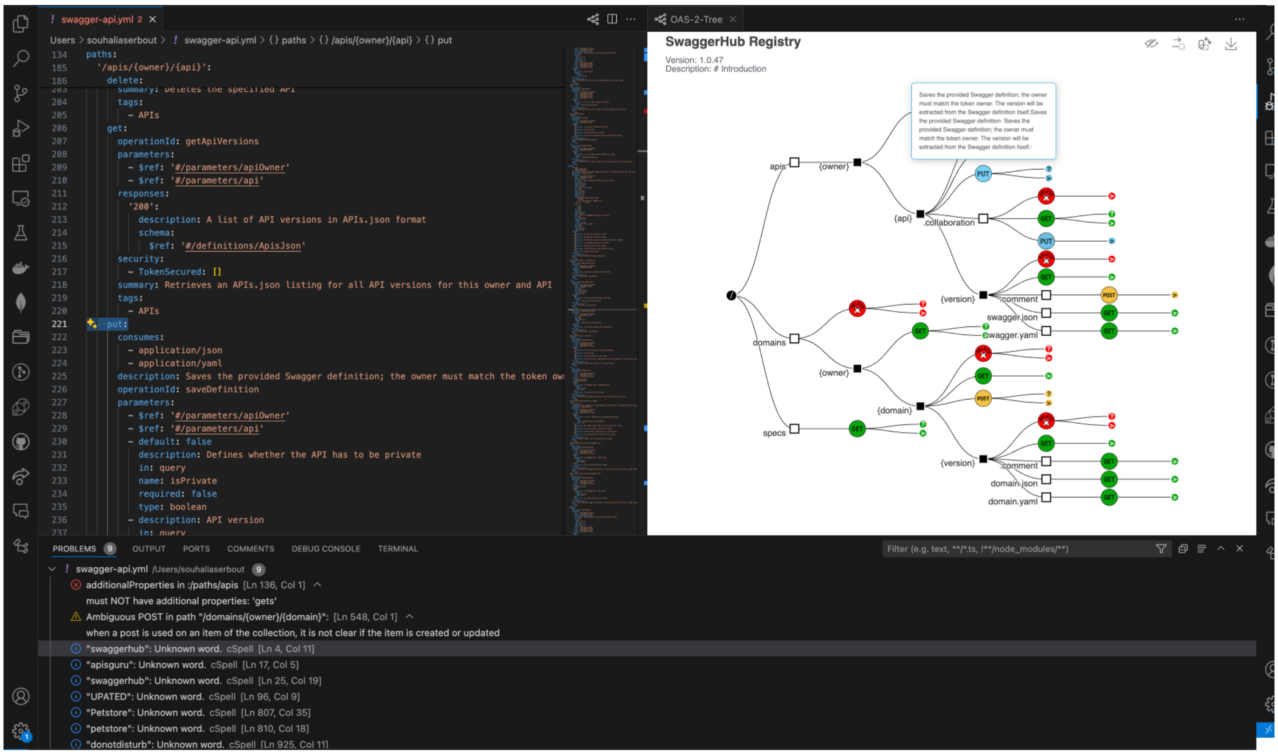


Figure 5.4. Visual representation of the Apacta API structure as a tree of resources and HTTP methods. This API tree includes many reoccurring subtrees, which we extract as API fragments (Click for OpenAPI source)



```
paths:
  '/apis/{owner}/{api}':
    delete:
      summary: Deletes the specified API
      tags:
        - APIs
    get:
      operationId: getApiVersions
      parameters:
        - $ref: '#/parameters/apiOwner'
        - $ref: '#/parameters/api'
      responses:
        '200':
          description: A list of API versions in APIs.json format
          schema:
            $ref: '#/definitions/ApisJson'
      security:
        - TokenSecured: []
      summary: Retrieves an APIs.json listing for all API versions for this owner and API
      tags:
        - APIs
    put:
      consumes:
        - application/json
        - application/yaml
      description: Saves the provided Swagger definition; the owner must match the token ow
      operationId: saveDefinition
      parameters:
        - $ref: '#/parameters/apiOwner'
        - $ref: '#/parameters/api'
        - default: false
          description: Defines whether the API has to be private
          in: query
          name: isPrivate
          required: false
          type: boolean
        - description: API version
          in: query
```

Saves the provided Swagger definition; the owner must match the token owner. The version will be extracted from the Swagger definition itself.Saves the provided Swagger definition- Saves the provided Swagger definition; the owner must match the token owner. The version will be extracted from the Swagger definition itself.-

```
swagger-api.yml /Users/souhaliaserbout 9
  additionalProperties in :/paths/apis [Ln 136, Col 1]
  must NOT have additional properties: 'gets'
  Ambiguous POST in path "/domains/{owner}/{domain}": [Ln 548, Col 1]
  when a post is used on an item of the collection, it is not clear if the item is created or updated
  "swaggerhub": Unknown word. cSpell [Ln 4, Col 11]
  "apisguru": Unknown word. cSpell [Ln 17, Col 5]
  "swaggerhub": Unknown word. cSpell [Ln 25, Col 19]
  "UPATED": Unknown word. cSpell [Ln 96, Col 9]
  "Petstore": Unknown word. cSpell [Ln 807, Col 35]
  "petstore": Unknown word. cSpell [Ln 810, Col 18]
  "donotdisturb": Unknown word. cSpell [Ln 925, Col 11]
```

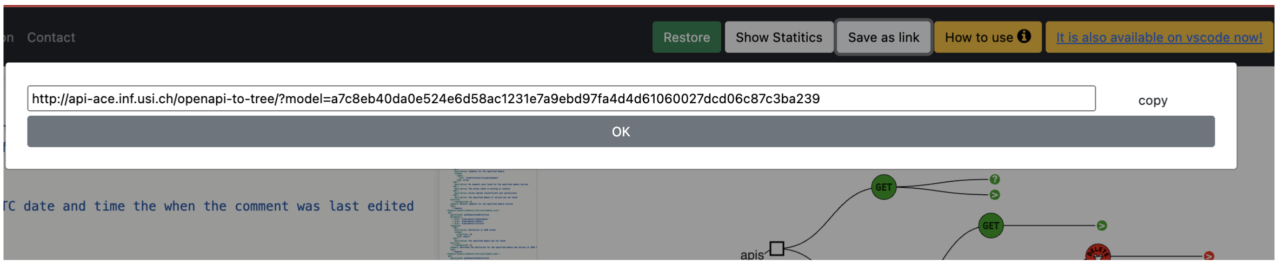
Figure 5.5. Navigation between the diagram and the OAS code

This interactive feature provides a comprehensive overview of the API architecture, enabling users to quickly locate specific endpoints and understand their functionalities. On mouse over, the editor highlights the corresponding element in the OpenAPI document in yellow. The description of the element is displayed in the tooltip. It is also possible to navigate from the problems view to the problematic element in the OpenAPI document by clicking on the error message.

5.3.3 Web version of OAS2Tree

OAS2Tree is also available as a standalone web application, allowing users to visualize API structures without the need for a development environment. The web app version provides the same features as the VSCode extension, including the ability to save the current specification being visualized and share it through a unique URL with other users. The web app version is particularly useful for users who want to quickly visualize or sketch an API design from their Web browser (Figure 5.6). An additional feature that the web version offers is the ability to navigate a collection of API specification examples, and visualize them in the tree structure format. This feature can be useful for users who want to explore different API designs and understand the common patterns and structures used in API development, and how some features are documented in OpenAPI by other API designers[1].

[1] http://api-ace.inf.usi.ch/openapi-to-tree/navigate-apis?limit=50&page=13



http://api-ace.inf.usi.ch/openapi-to-tree/?model=a7c8eb40da0e524e6d58ac1231e7a9ebd97fa4d4d61060027dcd06c87c3ba239

TC date and time the when the comment was last edited

Figure 5.6. Save as URL functionality in OAS2Tree Web App

5.4 Usage scenarios

OAS2Tree can be used by both API providers and API consumers in various use-case scenarios across the API development lifecycle.

As an API Designer & Developer:

- API Design and Documentation: During the initial stages of API design, OAS2Tree can be employed to visualize and refine the API structure. Since the visualization can also be generated from not fully complete OAS, it can serve as a tool for sketching partial API structures, and have an early overview of it.
- API Review and Design Validation: Designers can use OAS2Tree not only to ensure conformance to the OpenAPI standard, but also to ensure consistency in endpoint naming, parameter usage, and selection of HTTP methods.
- API Evolution: The tool can help find the adequate place of an extension.

As an API Consumer:

- Functionalities exploration: Client developers can use OAS2Tree to explore the endpoint structure and assess whether the API meets their requirements.
- Documentation navigation: OAS2Tree can be used to navigate the API documentation and quickly locate specific endpoints and operations, by hovering over the tree nodes, the corresponding element in the OpenAPI document is highlighted.
- Visual comparison: OAS2Tree can be used to visually compare the structure of different versions of an API, or the structures of different APIs.

5.5 Web API modeling tools

Most of the API lifecycle management tools currently widely used in the market[2][3][4], including paid ones, offer a Postman-like interface in their 'API Design environment', where the user can interact with the API, and test it. However, none of them offers a tree-like visualization of the API structure, that can be used to understand the overall API structure and navigate the documentation.

OAIE Sketcher[5] offers another way to visualize the endpoints by emphasizing the relationships between the schemas used in the request and response bodies. However, it lacks the hierarchical structure of the API paths and the HTTP methods, and the visualization is not kept synchronized as the corresponding textual specification changes.

The OpenAPItoUML tool [49] generates UML models from OpenAPI definitions, providing a means to visualize both API endpoint structures and API data models using class diagrams. However, it does not provide a tree-like

[2]https://xapihub.io/features/designAndDev
[3]https://stoplight.io/drive-api-results
[4]https://apigit.com/why-apigit/api-design
[5]https://raw.githack.com/OAIE/oaie-sketch/master/sketch.html

visualization of the API paths and operations, and it does not support the detection of any design smells in the API specification or data schema issues.

In Summary

OAS2Tree is a Web API visualization tool, available as a Visual Studio Code extension and a web application, that transforms OpenAPI specifications into a simple visual tree representation. It supports OAS v2.0, v3.0, and v3.1. It provides a side panel in the VS Code editor to interactively display the API structure as a tree. The tool can be employed by API designers to visualize and refine the API structure during the initial design phase, ensuring consistency in endpoint naming, parameter usage, and HTTP method selection. It can also be used to validate the API design and detect potential design smells early on. API consumers can use OAS2Tree to explore the functionalities of an API and understand whether it meets their requirements. In addition, it contains a navigation feature that can help to quickly locate the textual description of specific endpoints and operations.

The tool currently has about 600 users and has been used in teaching, research, and development projects.

In future work, we plan to extend the tool to support detecting additional design smells and enhance the representation of the detected issues on the tree visualization to ease their location in the overall API structure. We are also experimenting with embedding the visualization as part of the API documentation generated from the OpenAPI description.

Demo Video Link: `https://youtu.be/E48c9Rwntz8`

Chapter 6

Web APIs Structural Patterns and Smells: A pattern map of API fragments

Web APIs are theoretically designed to meet specific, predefined requirements. However, an important question arises: do APIs that provide the same functionalities exhibit similar structural designs? Furthermore, are there specific, repetitive components within these APIs that can be systematically extracted and analyzed?

In this chapter, we present the results of an analysis that draws an analogy between API structures and tree-like representations, where an API is conceptualized as a set of branches (functional paths) and nodes (connections between paths). Through this lens, we aim to explore the structural similarities and repetitive patterns in APIs by mining the recurrent API fragment composing Web APIs.

6.1 Mining Recurrent Web API Fragments

The primary objective of this research is to investigate recurring structural patterns in Web APIs, focusing on their design and evolution as described in OpenAPI specifications. Specifically, this study aims to identify and classify repetitive components, referred to as *API fragments*, and analyze their relationships with higher-level *API design patterns* and *design smells*. The overarching goal is to determine the extent to which these fragments can serve as building blocks for designing consistent and reusable API structures, and how deviations from such patterns may indicate inefficiencies or design flaws.

By conceptualizing API structures as tree-like representations presented in Chapter 5, where endpoints are modeled as nodes and their relationships as branches, the study seeks to address the following questions:

1. **Recurrent Structures :** Which fragments of API structures occur repeatedly across APIs, and what functional roles do these fragments serve?
2. **Structural Smells:** Are there structural anomalies, such as inefficient or ambiguous configurations, that may hinder usability, security, or maintainability?
3. **Semantic Variations:** Using the semantic closeness between labels used in similar API Fragments, how diverse is the semantic context in which a specific fragment is employed?

This research aims to provide both a theoretical framework and practical insights to assist API designers in creating consistent, reusable, and maintainable API structures. The findings also aim to contribute to the development of tools and methodologies that enhance API design quality, promote alignment with best practices, and enable pattern-driven API evolution.

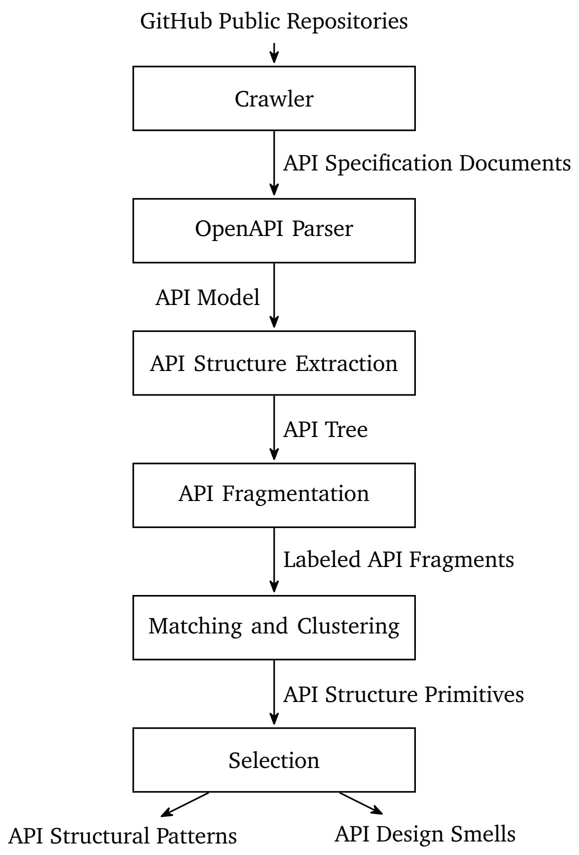


Figure 6.1. API Analytics Pipeline: From API Specifications to Patterns

6.2 API Fragments Mining Approach

Out of many existing kinds of APIs, in this research, we focus on Web APIs [158, 172, 121, 12, 132, 133] remotely accessible through the HTTP protocol and described using the standard OpenAPI specification language [164]. We do so because of the large number of API descriptions using this language which can be retrieved by crawling open source repositories (Figure 6.3). While the original purpose of OAS was to generate human-readable documentation, it can also be used to generate interactive test clients, as well as client-side and server-side stubs [84, 119, 161].

we statically analyze a large collection of real-world API descriptions looking for recurring structures that can play the roles of pattern primitives [180] which can be composed to obtain API design patterns [188]. In particular, we are interested in the resources exposed by the HTTP API naming and in the relationship between resource paths and the corresponding HTTP methods. This information can be used by clients to invoke the corresponding operations.

As illustrated in Figure 10.1, our process begins by crawling open-source code repositories to gather API description documents that adhere to the OAS. These documents are then parsed and fed into a model to extract API structure trees. These trees are fragmented, and the fragments are analyzed to identify recurring patterns. Subsequently, these fragments are clustered to discern prevalent uses.

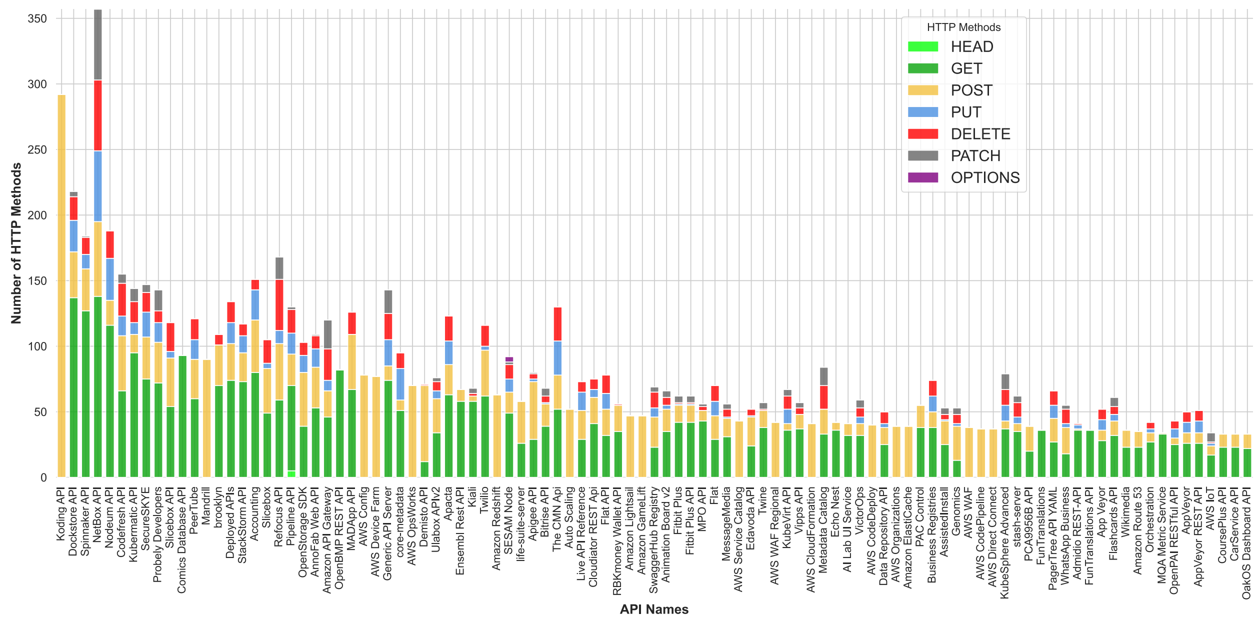


Figure 6.2. API Collection Sample (sorted by Number of Paths)

6.3 API Fragments Mining Outcome

The contributions of this research include:

1. A method to detect similar reoccurring API structures, which takes into account natural language labels associated with each path segment. This method can be also used to compare the structure of different Web APIs.

2. A collection of widely used API fragments, with a quantitative analysis of how frequently they occur across the same or different real-world APIs.

3. A collection of structural pattern primitives which have been used as building blocks for HTTP-based APIs. In particular, we selected API structures used to provide API clients with access to resource collections of related items (e.g., user accounts, purchase orders and their items, computational jobs, blog posts and their comments, videos, or audio tracks).

4. A classification of some design smells found across Web APIs.

5. Two composition operators for assembling the pattern primitives into larger API structures and a proposal for connecting them with API design patterns and service contracts.

6.4 Analyzed Data

In this study, we analyzed a snapshot of the dataset that included specifications pushed to GitHub between December 2020 and January 2021, belonging to 6919 APIs, with an average size of 22.8KB, including specifications of some well-known APIs such as Twilio, Slack, Flickr APIs, Google APIs, and Amazon APIs. All the APIs described in the

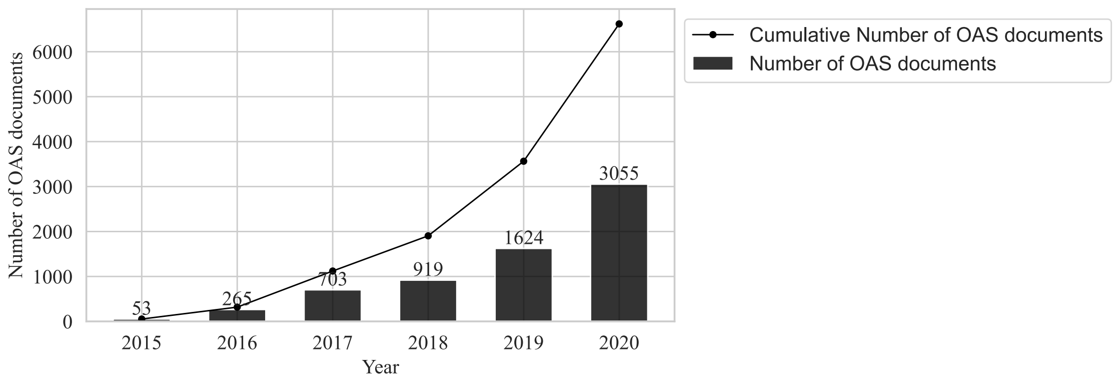


Figure 6.3. Yearly distribution of the specification in the analyzed snapshot

descriptions under study contain at least one method and one path. In this chapter, we include one visualization of one of the largest APIs in the dataset (Figure 5.4), and other examples of smaller APIs to show how the detected primitives are used as building blocks to construct the whole API's structure.

The yearly distribution of the age of the OpenAPI documents in our dataset is depicted in Figure 6.3. The horizontal axis refers to the year of the last commit that updated the document.

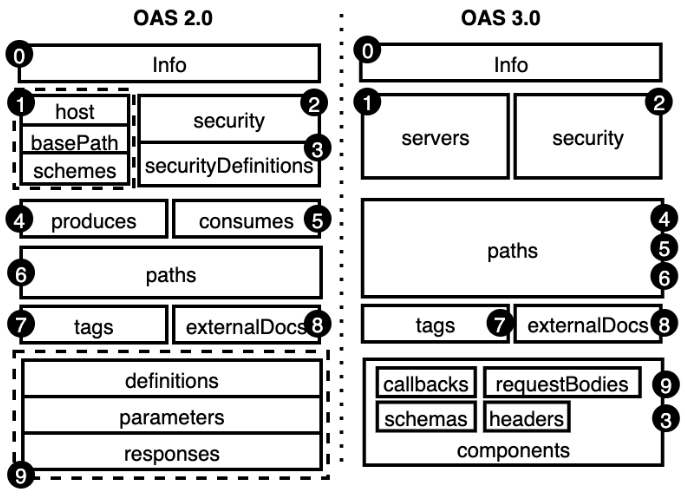


Figure 6.4. Open API Specification Metamodel Versions: 2.0 vs 3.0

6.4.1 OAS versions distribution

The collection studied in this work contains 6619 OAS descriptions. 28.9% are written in OpenAPI 3.0 and 71.1% are written in Swagger 2.0, coming from more than 600 different providers.

The two versions are slightly different from each other on the content level, but they both allow us to describe API structures with almost the same level of granularity. In Figure 6.4, we describe the main differences between the two versions. The numbers (1), (2),.., (9), show the mappings between the sections of a description written in

OAS 2.0 and their correspondents in a description document written in OAS 3.0. The first difference is in the servers details section. While in OAS 2.0 it was possible to include only one endpoint for an API, in OAS 3.0 it is allowed to include multiple server objects. Other structural rearrangements have been done in OAS 3.0 in order to increase the reusability of definitions, such as the inclusion of the Components section, where `securityDefinitions`, schema definitions, parameters, and responses are defined. In addition, in OAS 3.0 a Component object can also contain callback descriptions, which makes this version more efficient in describing asynchronous APIs. Moreover, OAS 3.0 improved the description of the parameters and supports more security schemes and bearer formats than OAS 2.0.

In our case, these differences between the two versions do not impact the results of the structural analysis and the API fragments extraction, because our study focused on the paths and methods provided by APIs, which are described in both versions.

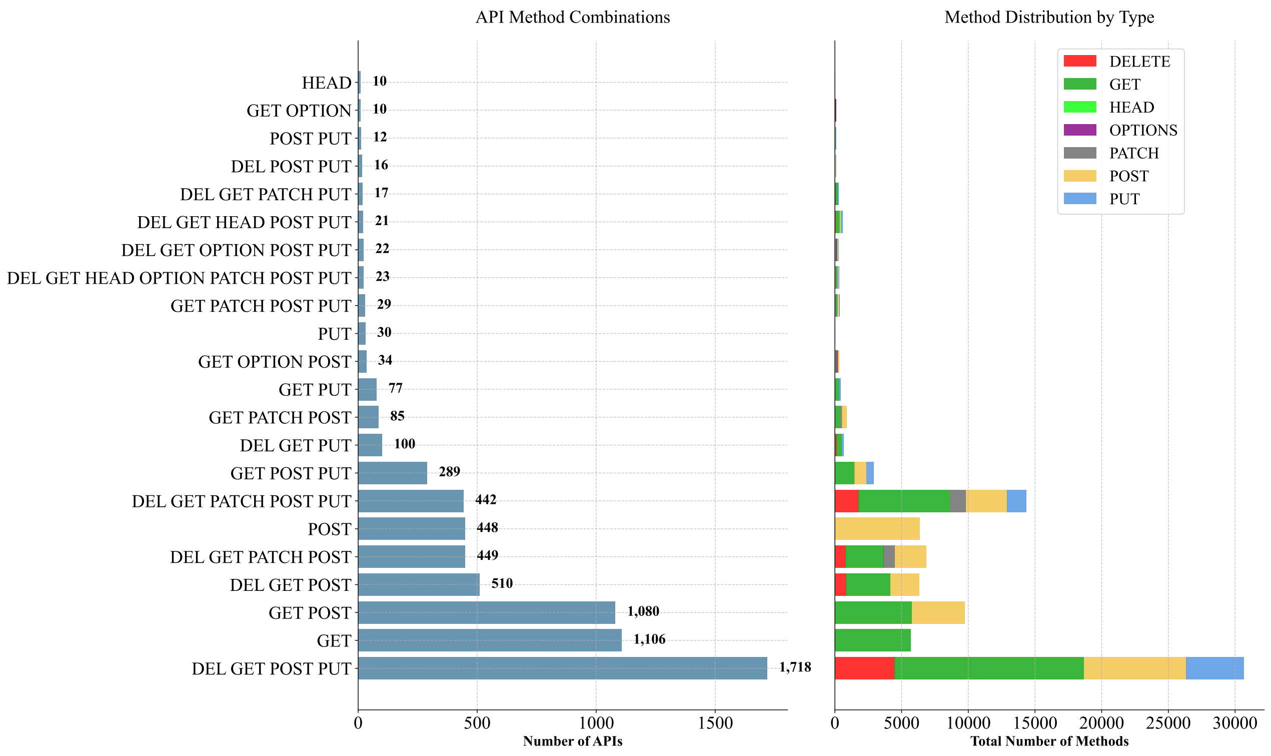


Figure 6.5. API Method Combination Overview

6.4.2 HTTP methods usage

In Figure 6.2 we show an overview of HTTP methods usage in a subset of the APIs under study. We classify APIs based on which HTTP methods they use following the RESTful maturity model [56], which distinguishes L0) APIs that use only one endpoint and one method from L1) APIs that use multiple endpoints and still one method associated to each endpoint, and L2) APIs which use multiple methods with multiple endpoints. Given the lack of support for describing hypermedia in OpenAPI documents, we are unable to distinguish the highest level of the maturity model L3, which includes the APIs that make use of the REST principle Hypermedia as the Engine of Application State (HATEOAS).

Still, we can clearly see different types of APIs emerging if we simply count how many HTTP methods are associated with each path enumerated in the API description (Figure 6.5). We have grouped the APIs into sets according to their HTTP method combinations and depicted the results in the bar chart in Figure 6.5. The most popular group makes use of the CRUD-like primitives of GET, PUT, POST, and DELETE. The second most popular group only uses the read-only GET method. This is closely followed in terms of size, by the APIs which use only the GET or POST methods. Another group of similar size can be observed by combining CRUD APIs that do not use the PUT method (so they alias update and creation operations under the same POST method) together with APIs that instead of using the PUT method replace it with the PATCH method. The next group includes the pure RPC APIs, which only use the POST method. The last group worth mentioning is the ones that use all five methods, which includes 442 APIs. The collection also includes about 500 APIs with different method combinations but of rather a small size.

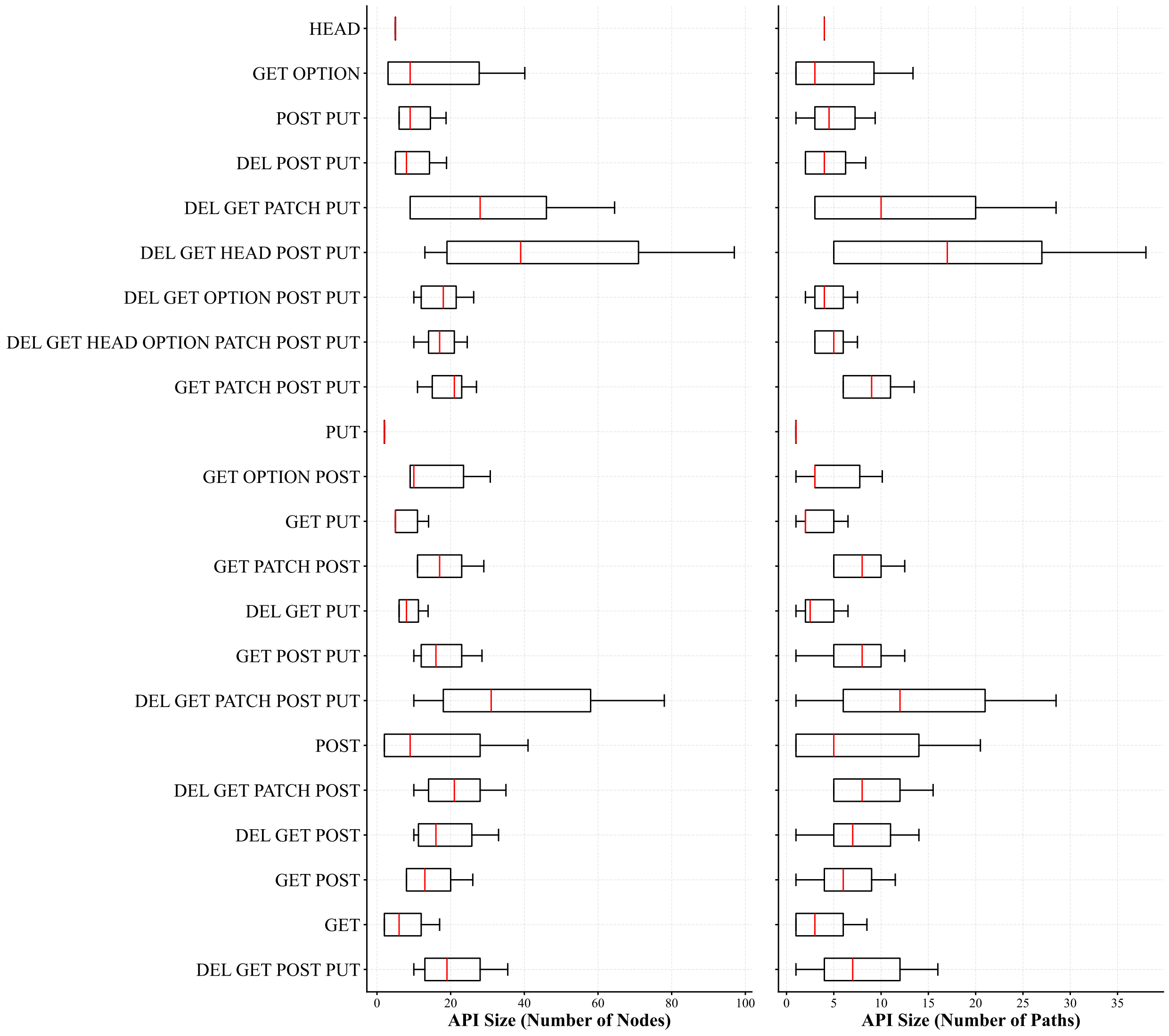


Figure 6.6. API Method Combination vs. API size

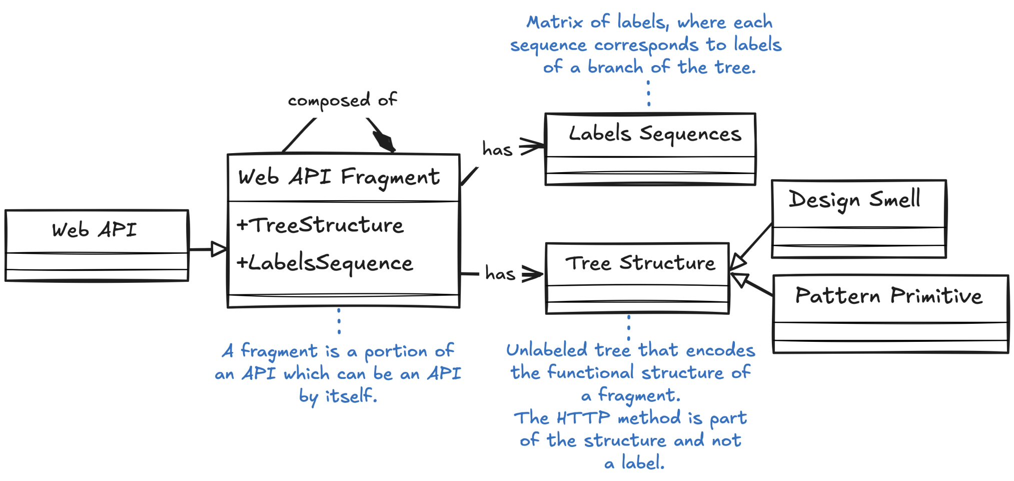

Matrix of labels, where each sequence corresponds to labels of a branch of the tree.
composed of
Labels Sequences
has
Web API Fragment
+TreeStructure
+LabelsSequence
Design Smell
Web API
has
Tree Structure
Pattern Primitive
A fragment is a portion of an API which can be an API by itself.
Unlabeled tree that encodes the functional structure of a fragment.
The HTTP method is part of the structure and not a label.

Figure 6.7. Domain concepts and their relations

6.4.3 API sizes distribution

Figure 6.6 presents an overview of the size of the APIs in the same groups, measured with two different metrics [65]: a) the number of paths listed in the API description and b) the nodes present in the API tree. The boxplots in Figure 6.6 represent the distribution of the size measurements for each API. Overall, the median values for the size of the APIs in the collection reach approx. 50 nodes and 20 paths.

6.4.4 Domain Concepts

In this work, we focus on analyzing the structures of Web APIs with the goal of detecting APIs with similar structures. Due to the granularity of API descriptions in OpenAPI models, we could create a tree model representation for each API in the collection, which we call from now on API Tree. For lifting the level of abstraction of the tree model, we unlabel all its nodes. We refer to the unlabeled tree model as API Tree Structure.

In our analysis, we aim to detect repetitive tree fragments in the API tree models. For that, we define an API Fragment object, a subtree of an API tree. As for an API, the fragment also has an unlabeled version, which we call henceforth Fragment Tree Structure. After matching and filtering the set of API tree structures extracted from the whole API collection under study, we obtained a list of API structure primitives and another for API Structure smells, as described in the domain concepts summary of Figure 6.7.

6.5 API Fragments

An API Fragment is any sub-tree (a connected sub-graph that includes some of the leaves of the original tree) of an API tree structure. A sub-tree is also a tree, therefore a fragment can be also seen as an API itself, which can be further decomposed. For instance, the excerpt in Listing 1, is an example of an API fragment extracted from the Apacta API (Figure 5.4).

To achieve our goal of detecting recurrent fragments in the API structures, we present a two-step approach that uses an algorithm that first extracts significant model fragments from a dataset of APIs models, and then compares them across multiple APIs to detect recurring ones.

6.5.1 APIs fragmentation approach

A tree T is a non-linear data structure, where each non-leaf node can be seen as a root of one or many sub-trees. The goal of the fragmentation function $\mathfrak{F} : T \rightarrow lf_1, lf_2, .., lf_n$ is to extract all the possible sub-trees $lf_1, lf_2, .., lf_n$ containing a sub-set of the ensemble of leaves of T. For collecting the nodes we walk T using Depth-First Search (DFS). In this way, we can extract all the trivial sub-trees, which are the ones having as root the different nodes of T. The algorithm extracts also non-trivial sub-trees , which are built by extracting all the branches of a sub-tree having as root a node N, then reconstructing the Tree Structures from all the possible combinations of the branches. Note that a branch starts from the root of the tree, and keeps all the methods attached to the deepest path segment node of the tree. Doing so, we obtain all the possible sub-trees having as a root the node N. Once a sub-tree is retrieved, it is serialized as JavaScript Object Notation (JSON) and stored in a MongoDB database. The same process is repeated over all the nodes of T until no node is left.

We analyse each API description in the collection and extract $T_1, T_2, .., T_m$, where m is equal to the size of our OpenAPI descriptions collection. Then we apply $\mathfrak{F}$ on each tree to extract all possible labeled sub-trees, or labeled fragments lf which include a subset of the leaves of the tree from where it was extracted T. While labeled fragments carry the original path segments labels, unlabeled fragments f_j only distinguish whether a path segment is parametric or not, and if it contains an unusual label. The leaves of both labeled and unlabeled fragments refer to the HTTP methods which can be applied to the corresponding sub-path.

We extract all fragments from all API trees in the collection and look for reoccurring ones. To speed up the process, we first match unlabeled fragments based on their topology, then we further compare the semantic similarity of labeled fragments sharing the same structure. To do so, we project the labeled fragments lf_j into *Label Sequences* which enumerate the labels found during the traversal of each node of the API fragment tree. In other words, we apply the projection function $\mathfrak{P} : lf_j \rightarrow (TS_j, LS_j)$, to obtain for each labeled fragment a Tree Structure TS_j (also called unlabeled fragment f_j) and a Labels Sequence LS_j. All the resulted output objects are also serialized as JavaScript Object Notation (JSON) and stored in a MongoDB database.

6.5.2 API Fragments Clustering and Selection

Having obtained the set of all labeled API fragments, which in our collection corresponds to 277'094 entities, we proceed to remove duplicates and cluster them.

For clustering the fragments, we followed a two-step similarity checking approach, which consists of exact topologies matching and labels closeness similarity scoring:

1. first by their common structure (i.e., the unlabeled fragment),
2. then, we compute the average label semantic similarity for each cluster of fragments sharing the same structure.

The output of structural clustering consists of a set of clusters where the elements of each cluster share the same API Structure, using different labels. We give higher priority to the larger clusters (more than 40 elements), knowing that the size of the cluster reflects how common is a specific structure. These known uses are then considered as candidate structural pattern primitives.

The goal behind semantically comparing the fragment sharing the same structure is to find out if there is a common use context of a highly recurring API Structure.

Tree Structures matching

In our approach, we see an API fragment as a sequence of labels LS placed on the nodes of a Tree Structure TS. A node of a TS can be either a path segment or a leaf representing an HTTP method. During our analysis, we decided to distinguish between three types of path segments: segments containing a parameter, noted as single-word labels between *{ }*, segments that do not include a parameter, and segments holding labels with more complex parameter notations, such as the example in Figure 6.8, which occurs 222 times. In our comparison approach, we consider the type of the path segment to be part of TS. Thus, API Fragments in Figure 6.9 and Figure 6.8 are detected to be distinct since the first structural clustering step. In this way, we already distinguish fragments, which even if they have the same tree topology, have parameters in different positions along the tree.

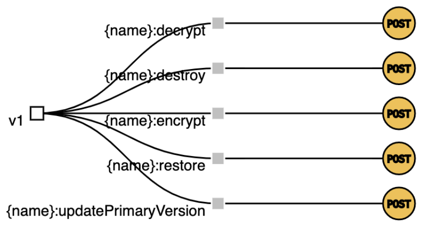


Figure 6.8. Example of a repetitive fragment with unusual parametric path segments labels

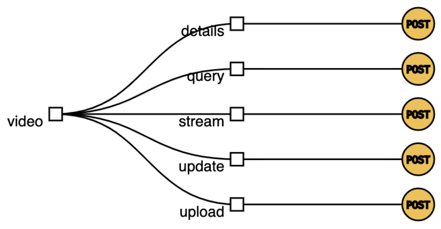


Figure 6.9. Example of a repetitive fragment with non-parametric path segments labels

Following our fragments TS comparison approach, we extracted, from a set of 277'094 labeled fragments, 79'728 TS unlabeled fragments sharing the same tree structures, considering also the type of path segment node, and the types of the HTTP method in the leaves.

Semantic closeness

Oftentimes, path segment labels carry some semantic meaning related to the resource handled by the path. For that reason, we considered taking into account the labels of the fragment nodes. By doing so, we involve the semantic context and have a better understanding of the common usage contexts of a specific fragment.

In our two-step similarity-checking approach, we first clustered the fragments by their TS, then extracted all ordered sequences of node labels found for each TS of the labeled fragment (Figure 6.10). Doing so, we obtain a collection of labels sequences for each TS. The size of the sequence is equal to the number of nodes of the TS, excluding the leaves.

To compute the similarities between the label sequences, we use spaCy[1], an open-source library for Natural Language Processing (NLP) in Python and Cython. In our case, we use a spaCy's trained model for English language [155], using the latest version of the "en_core_web_md" model package, multi-task CNN trained on OntoNotes, with GloVe vectors trained on Common Crawl for spaCy.

We distinguish the following types of labels:

1. Single words (i.e., stream, details, etc as in the example fragment in Figure 6.9),
2. composed labels, which concatenate single words using camelcase, or a "-" or a "_" symbol (Figure 6.22),

[1]spaCy: https://spacy.io/

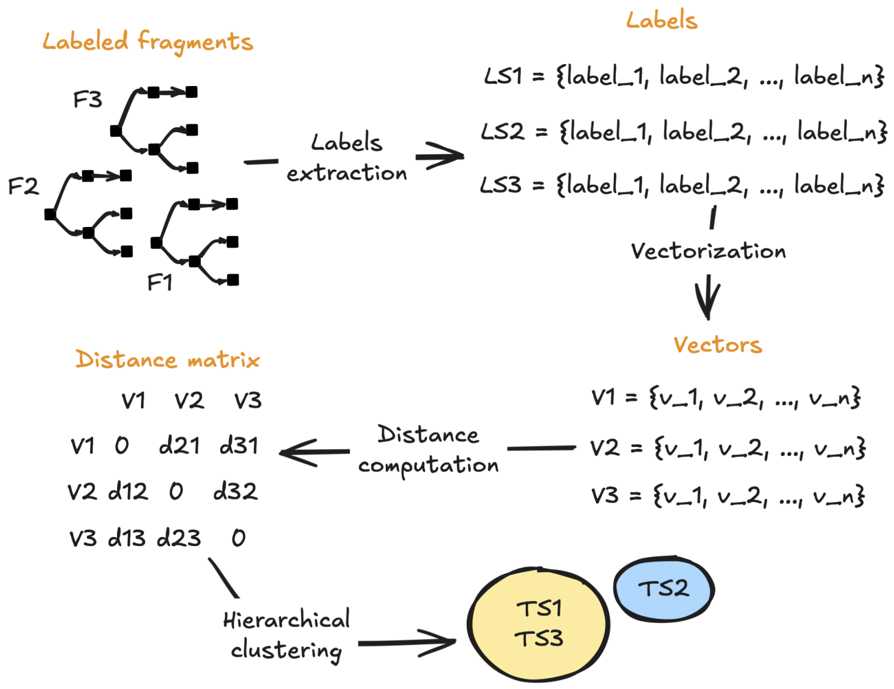

Labeled fragments
F3
F2
F1
Labels extraction
Labels
LS1 = {label_1, label_2, ..., label_n}
LS2 = {label_1, label_2, ..., label_n}
LS3 = {label_1, label_2, ..., label_n}
Vectorization
Vectors
V1 = {v_1, v_2, ..., v_n}
V2 = {v_1, v_2, ..., v_n}
V3 = {v_1, v_2, ..., v_n}
Distance computation
Distance matrix
V1 V2 V3
V1 0 d21 d31
V2 d12 0 d32
V3 d13 d23 0
Hierarchical clustering
TS1
TS3
TS2

Figure 6.10. Fragments semantic clustering pipeline

3. unusually long, complex labels (i.e., #x-amz-target=codedeploy 20141006deleteapplication),

We added a formatter to the spaCy's processing pipeline in order to cover the different labels types cases we have. We also added a filter at the end of the pipeline, which has a goal to exclude the labels that could not be matched to any semantic concept.

We define the distance between two label sequences $S = \{l_1, .., l_p\}$ and $S' = \{l'_1, .., l'_p\}$ as :

$$dist(S, S') = \sum_{i=1}^{p} \frac{sim(nlp(l_i), nlp(l'_i))}{p}$$

Where $sim(A, B) = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} (a_{ij} - b_{ij})^2}$ is the Euclidian Distance between the matrices $\mathbf{A} = (a_{ij})$ and $\mathbf{B} = (b_{ij})$. And where $nlp(l_i)$ is the vectorizer function of a label l_i in S. We normalize these distances to values between 0 and 10. As much is $d(S, S')$ closer to 0, S and S' are semantically close.

Doing so, within each TS cluster, we measure the semantic closeness of each sequence by calculating a similarity score between the labels attached to the same nodes of the tree. This score consists of the distance between the vectors representing the labels sequences of each fragment. Using Agglomerative Hierarchical Clustering, we obtain the semantic clusters for each set of structurally similar fragments, by setting a threshold depending on the similarity score distribution in each TS cluster.

Labels similarity results

We summarize the results with five metrics (Table 6.1):

1. The average distance between each couple of sequences: the goal of this metric is to depict how much are each two labels sequences are alike or similar. A low average means that most of the labels sequences are composed of semantically close elements.

Table 6.1. Overview of distances between all the label sequences of each primitive and its variants/smells.

The design smells are color-coded
🟪 Create without Delete, ⬜ Delete without Create, 🟧 Ambiguous POST, 🟦 Ambiguous PUT, 🟩 Write-only

Primitive	Variant/Smell	Labels sequences distances			Clusters	Threshold
		Average	Median	Max		
ENUMERABLE COLLECTION (P1)	GET (P1.v1)	3.07	5.87	9.97	218	5
	GET/PUT (P1.v2)	2.60	5.23	9.01	20	5
	GET/PUT/DEL (P1.v4)	2.89	5.80	9.79	34	5
	PUT/DEL (P1.v3)	2.94	5.51	9.03	13	5
	GET/POST (P1.s1)	3.26	7.53	9.42	14	5
	GET/DEL (P1.s2)	2.54	5.56	8.65	8	6
APPENDABLE COLLECTION (P2)	GET/PUT/DEL (P2.v1)	2.41	2.16	9.87	24	5
	GET/DEL (P2.v2)	1.90	0.00	9.86	23	5
	GET (P2.v3)	3.06	5.58	9.98	52	5
	PUT/DEL (P2.s1)	2.54	4.98	8.77	24	5
	DEL (P2.s2)	3.33	6.78	9.48	23	6
COLLECTION (P3)	GET/PUT/DEL/PATCH (P3.v1)	1.93	3.06	9.28	19	5
	GET/PUT/DEL (P3.v2)	2.62	5.02	9.81	120	5
	GET/DEL/PATCH (P3.v3)	2.74	5.31	9.84	12	5
	GET (P3.v4)	2.78	5.17	9.98	36	6
	PUT/DEL (P3.v5)	2.54	4.72	9.91	25	5
	GET/DEL (P3.v6)	3.07	5.77	9.98	39	6
	DEL (P3.v7)	2.59	5.34	8.50	39	5
	PUT-ONLY (P3.s1)	2.58	5.34	8.50	12	5
	GET/PUT (P3.s2)	2.32	4.41	9.16	24	5
MUTABLE COLLECTION (P4)	GET/PUT/DEL/PATCH (P4.v1)	1.22	0.00	8.08	19	4
	DEL (P4.s1)	2.55	5.49	9.70	10	5
	GET/DEL (P4.s2)	2.51	0.00	8.93	20	5

2. The median of these distances: the median gives an idea about the distribution of the distances. A high median means that the majority of the labels sequences are not semantically close.
3. The maximum distance between a couple of sequences.
4. The number of clusters that sequences were grouped by.
5. The threshold defining the maximum distances between all observations of two sets. This value was defines based on the the distribution of the values in the distance matrix.

Table 6.1 shows that label sequences in different collections of fragments are semantically similar. For each primitive, we will provide detailed examples of labels associated with each variant/smell in the next Section.

Labels usage

We calculated the occurrences of each label in each structural primitive. The labels are sorted first by their total number of occurrences, then alphabetically so that it is easy to find at the top of the table which are the most used labels in each primitive. The most used label for a specific variant is found by looking at the row which has the darkest color in the variant's column.

To know in which variant a specific labels sequences is used the most, it is enough to horizontally scan the row for the label looking for the highest value, ignoring the last column in which the total number of occurrences across all variants is reported. The heat maps only include the representative labels that we obtain using our labels merging approach described in Section 6.5.2.

We merged labels sequences with common container labels such as the ones in Figure 6.11. These labels are used

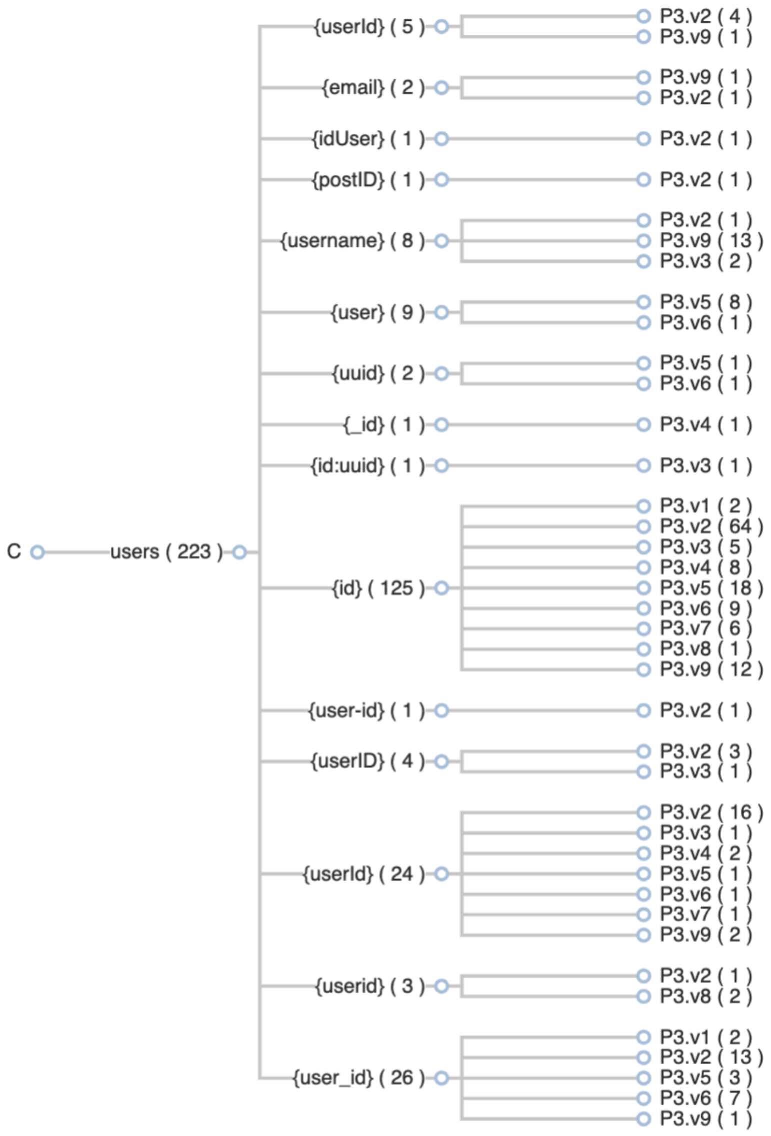


Figure 6.11. Label sequences with container label "users" in Collection primitive

in different variants of the Collection (P3) primitive and they share the same meaning, thus we decided merging them in order to have concise guidance tables. In our labels processing approach, we ignore the case of the container

Table 6.2. Known Uses of Selected Fragments: Number of Distinct Label Sequences (nDLS) and their Occurrences within APIs.

Design smells: The design smells are color-coded: 🟪 Create without Delete, ⬜ Delete without Create, 🟧 Ambiguous POST, 🟦 Ambiguous PUT, 🟩 Write-only.

Primitive	Variant	Occurrence	Size	nDLS	Most distant labels sequences	
					Sequence 1	Sequence 2
Enumerable Collection P1	GET (P1v1)	1588	4	744	$\{operations, \{operation\}\}$	$\{cidades, \{nome\}\}$
	GET/PUT (P1v2)	99	5	43	$acls, \{user\}$	$users, \{id\}$
	PUT/DEL (P1v3)	40	4	26	$\{song, \{id\}\}$	$\{oauth, \{provider\}\}$
	GET/PUT/DEL (P1v4)	176	6	71	$countries, \{country\}$	$namespaces, \{namespace\}$
	🟧 GET/POST (P1s1)	77	5	22	$msgs, \{username\}$	$servers, \{framework\}$
	⬜ GET/DEL (P1s2)	56	5	26	$\{serveurs, \{id\}\}$	$\{tasks, \{task\}\}$
Appendable Collection (P2)	GET/PUT/DEL (P2v1)	194	6	64	$item, \{itemid\}$	$bucketlist, \{id\}$
	GET/DEL (P2v2)	145	5	43	$vim, \{vim\_uuid\}$	$v1, \{album\}$
	GET (P2v3)	202	4	127	$\{user2, \{username\}\}$	$\{disease, \{disease\}\}$
	🟩 PUT/DEL (P2s1)	50	5	31	$rulesets, \{rulesetName\}$	$pelicula, \{peliculaId\}$
	🟩 DEL (P2s2)	69	4	51	$\{token, \{iat\}\}$	$\{jobs, \{id\}\}$
Collection (P3)	GET/PUT/DEL/PATCH (P3v1)	328	8	159	$lenses, \{key\}$	$movies, \{movie\}$
	GET/PUT/DEL (P3v2)	1123	7	574	$nodes, \{ip\}$	$tickets, \{tid\}$
	GET/DEL/PATCH (P3v3)	232	5	139	$rooms, \{key\}$	$taxrate, \{zipcode\}$
	GET (P3v4)	323	5	168	$\{deposits, \{depositor\}\}$	$\{txs, \{txid\}\}$
	PUT/DEL (P3v5)	169	6	84	$pedidos, \{numero\}$	$countries, \{code\}$
	GET/DEL (P3v6)	345	6	187	$applications, \{appid\}$	$caixa, \{codigo\}$
	DEL (P3v7)	201	5	87	$byon, \{id\}$	$client, \{pubkey\}$
	🟪 PUT-Only (P3s1)	78	5	38	$\{Chems, \{chemid\}\}$	$\{users, \{userid\}\}$
	🟪 GET/PUT (P3s2)	63	6	47	$manager, \{username\}$	$pessoas, \{idPessoa\}$
Mutable Collection (P4)	GET/PUT/DEL/PATCH (P4v1)	74	9	52	$workflows, \{name\}$	$boards, \{id\}$
	🟦 DEL (P4s1)	48	6	18	$progress, \{ordinal\}$	$beverages, \{beverage\}$
	🟦 GET/DEL (P4s2)	102	7	56	$ciudad, \{id\}$	$themes, \{uuid\}$

resource label (e.g : *dataPointer* is equivalent to *datapointer*). We also remove all the special characters and the spaces (e.g: *dataPointers*, *data* − *pointers*, *data_pointer* are considered equivalent). Moreover, ignore the singularity and plurality of the labels (e.g: *data_pointer*is equivalent to *dataPointers*). In the example of Figure 6.11 the label *users* originally appears in different formats (e.g: *user*, *Users*, *User*), our merging algorithms pick the most occurring form as a representative label of all the forms. The reason behind this clustering is to give more insight about the labels' usages and merge the ones that represent similar concepts in order to avoid redundancy.

6.6 Structural API Primitives

Out of the results obtained from the fragmentation and clustering process, we selected a set of most occurring fragments and classified them into four primitives (Figure 6.13), depending on their functionality based on their structures.

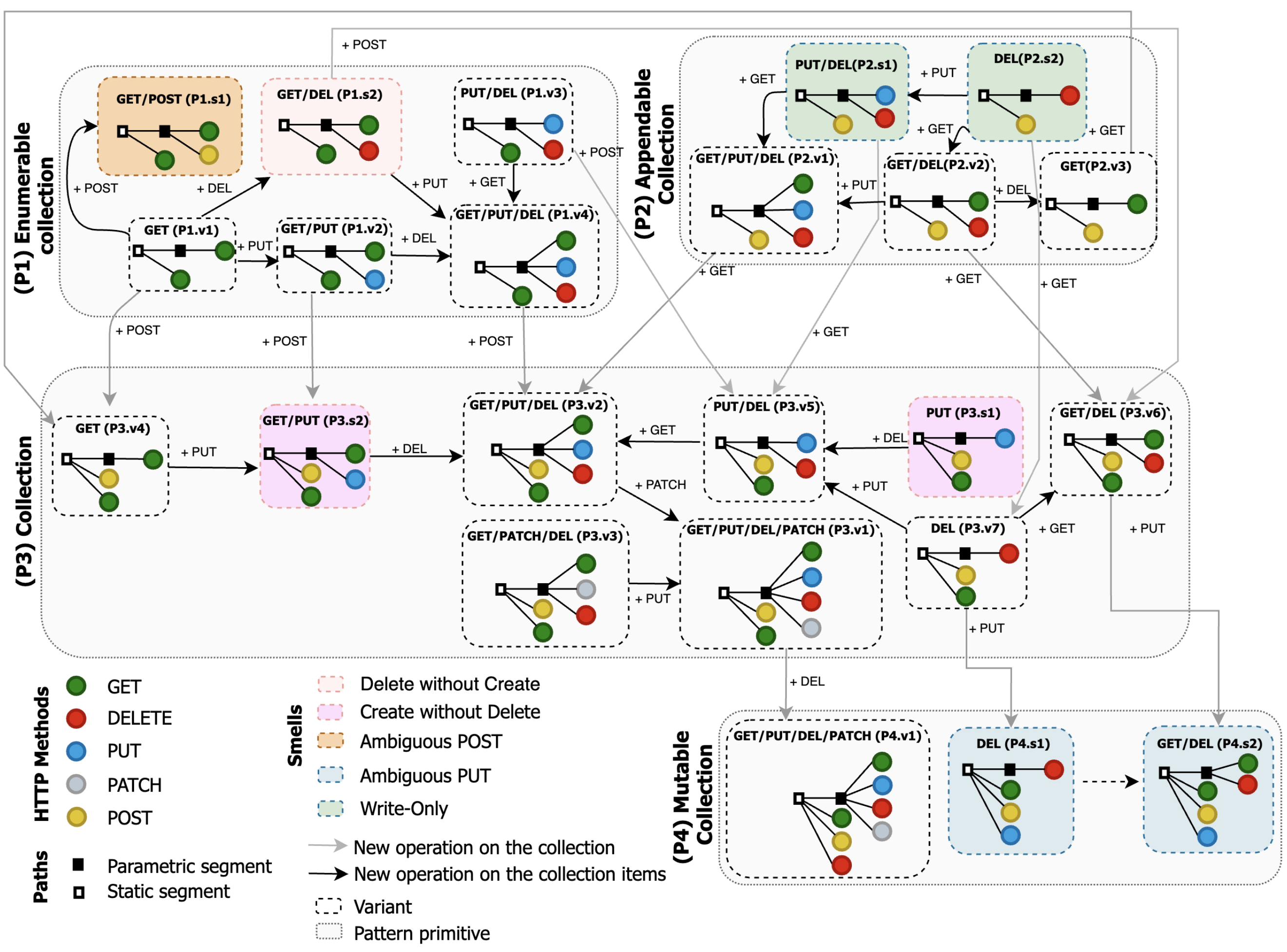


Figure 6.12. Overview: API Structural Primitives and their variants and design smells

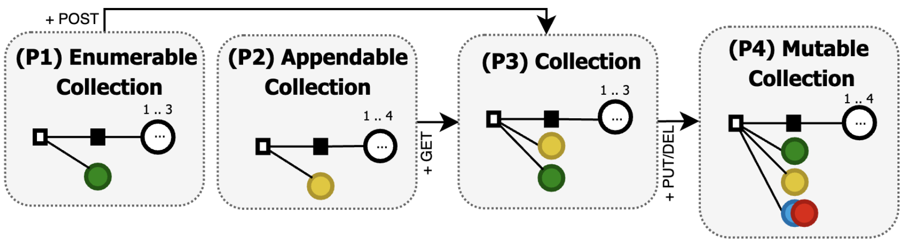
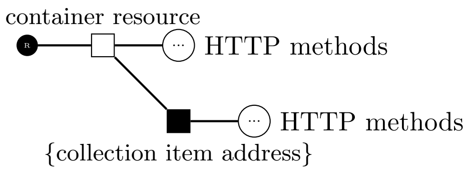


Figure 6.13. Overview: API Structure Collection Primitives

The *context* for all pattern primitives is the same: a designer needs to use an HTTP-based API to provide access to a collection of items which are stored on the server.

All the primitives are used to expose in the API collections of items, where each collection if identified by a statically-named container resource and its items are dynamically addressed within the container resource. We distinguish each primitive based on which combination of HTTP methods are attached to the container resource.



The ENUMERABLE COLLECTION (P1) primitive is used when clients can use the API to only discover the content of the collection by retrieving a list of their items. The APPENDABLE COLLECTION (P2) primitive makes it possible for clients to only append items into the collection exposed by the API. The COLLECTION (P3) primitive combines both features of the APPENDABLE COLLECTION and the ENUMERABLE COLLECTION, so that clients may use it to both append new items and list existing items. Since this primitive is the most commonly found one, we choose to name it with the simplest and shortest name, while adding qualifiers to the names of the other primitives. Finally, the MUTABLE COLLECTION (P4) primitive extends the COLLECTION with the ability to perform batch operations on the entire collection (e.g., to delete the entire content or replace the entire content of the collection).

Within each primitive, we have collected many variants and design smells depending on which combination of HTTP methods is attached to the collection item resource.

In Figure 6.12, we provide a more detailed overview showing for each primitive the corresponding variants and design smells. Each variant and design smell of the same primitive are encapsulated in a gray frame. We also show how each variant can be obtained by changing another one with the black and gray arrows. The black arrows trace the paths that allow moving from a structure primitive to another by adding an operation on the items of the collection. And the gray ones are showing which methods are added to the container resource. In the rest of this section we present overviews focused on each structural primitive.

During our analysis, we have also detected some structural design smells, which we highlight in Figure 6.12 with colored frames. We classified the detected smells into the following categories:

🟪 Create without Delete: API structures that allow the clients to create elements from a collection, but do not provide a possibility to delete elements from it.

▭ Delete without Create: API structures that allow the clients to delete elements from a collection, but do not provide a possibility to append elements to it.

🟧 Ambiguous POST: API structures that contain a POST operation on the items of a collection. Is this POST method used to append items to the collection?

🟦 Ambiguous PUT: API structures that provide a PUT operation on the collection. Is this PUT method really used to update the whole collection?

🟩 Write-Only: API structures that have no read operation neither on the whole collection nor on its items.

In Table 6.2, we show an overview of the selected collections of fragments, by listing their occurrences and the number of unique label sequences used by the same structures in the same API or across different ones. We also give an example of the most distinct sequences found among the unique labels sequences, in order to show the extreme use contexts for each structure.

The rest of this section details each of the selected primitives where we present for each primitive the different occurring variants.

In the Figures, we show the occurrences (counting how many times a Labels Sequence is used for the same TS) of each cluster of labels in a specific variant/smell.

The goal is to support designers who would like to introduce a collection for a specific class of items in their API. They can take advantage of the observations we have collected as they attempt to look up the collection label and see if there is a non-ambiguous mapping to a given primitive variant.

In order to give an idea about the yearly distribution of the variants ages and popularity, we calculate the number of APIs in which a specific variant appears (Tables 6.15, 6.26, 6.34, 6.47).

6.6.1 Enumerable Collection (P1)

Summary

Expose an enumerable set of items within their own container resource.

Problem

How to make the collection items discoverable by clients?

Solution

Provide a unique address for each collection item. Allow clients to read the content of each item applying the GET method to the address of the item. Group together related items under the same resource path prefix. And, allow clients to enumerate the items within the collection by applying the GET method to the container resource.

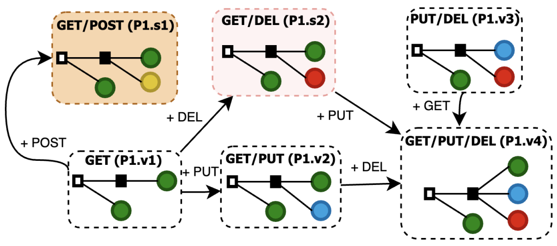


Figure 6.14. Enumerable Collection - Overview of Variants and Design Smells

	2015	2016	2017	2018	2019	2020
P1.s2	0	0	6	6	9	34
P1.s1	0	1	3	7	8	24
P1.v4	0	0	12	23	30	61
P3.v3	0	1	5	3	5	20
P2.v2	1	3	4	11	15	35
P1.v1	7	17	80	86	165	374

Figure 6.15. Yearly distribution of the API specifications where the Enumerable Collection (P1) variants appear

In Table 6.15, we can clearly see the increasing usage of the Variants and Smells in the API collection over time. This increase can be both because of the yearly distribution of the API specifications gathered in our data set, and to the popularity the structural primitives gained through the years.

- **Enumerable Collection Variants**

For the Enumerable Collection primitive we have identified 2 variants and 3 design smells (Figure 6.14).

GET (P1.v1)

The read-only variant is one of the most occurring structures, which allows clients only to enumerate the content of the collection and to read the corresponding items. APIs use it to publish one immutable set of related items. By setting a threshold of 5 obtained 218 Labels Sequences clusters. Which depicts the variety of usage contexts of this variant.

According to the whole label sequences set that we extracted, we noticed that this read-only structure is widely used for different domains. We can notice that all the most frequent labels in the Enumerable Collection (P1) are used by this variant, except 3 ones: *keys* and *episodes*, which are used by the variant GET/PUT/DEL (P1.v4) which allows also to update and delete the items of the container resource, and *client*, which is only used by GET/PUT (P1.v2).

An example of API where this structure primitive is present several times, in the Apacta API showed in Figure 5.4. In this API we can see clearly the high occurrence of GET (P1.v1) with different labels, combined with variants of other primitives.

Size: 4 — Occurrence: 1588 — Distinct Labels: 744

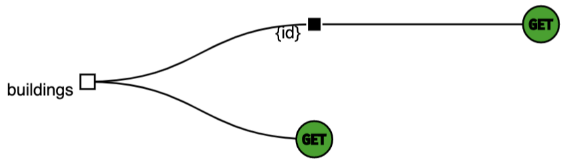


Figure 6.16. Enumerable Collection - GET Variant (P1.v1) Visualization

GET/PUT (P1.v2)

This variant allows clients to use the GET and PUT methods on the collection items. This makes it possible to read and update the content of individual collection items. This API structure also appears in Apacta API (Figure 5.4). Figure 6.18, is a use case example of this API structure. The GET operation in the resource handled by the path `/users/id/topics` allows the client to get all the topics of a specific user. The get operation in the path `/users/{id}/topics/{topic_id}` has as its goal to verify if a user is following a specific topic. The response is an object of boolean type. In this case, the PUT operation is for interpolating the FOLLOW / UNFOLLOW relationship between the user {id} and the topic {topic_id}.

Size: 5 — Occurrence:99 — Distinct Labels: 43

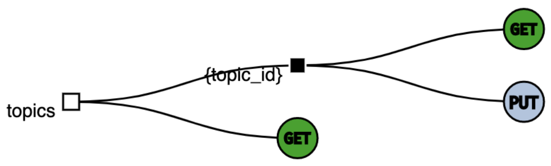


Figure 6.17. Enumerable Collection - GET/PUT Variant (P1.v2)

The Columba API[2] uses three instances of the GET (P1.v1) variant and one of the GET/PUT (P1.v2) variant.

[2]https://github.com/columbasms/columbasms.github.io

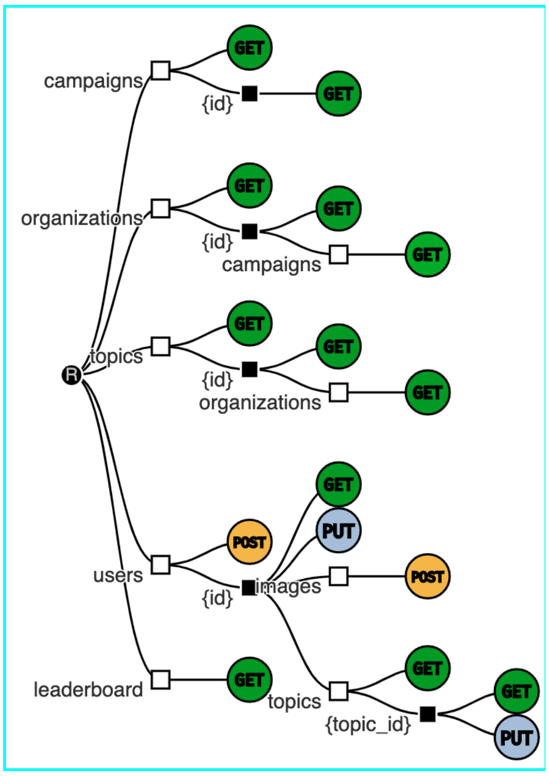


Figure 6.18. Tree Visualization of Columba API

PUT/DEL (P1.v3)

The particularity of this variant is that it allows to both update and delete items, however, it does not allow the client to create new items in the collection by using the container resource. Instead, it still allows them to do so by invoking the PUT method directly on the items to be created. In this case, clients themselves should provide the identifiers for the items to be added to the collection.

Size: 5 — Occurrence: 40 — Distinct Labels: 26

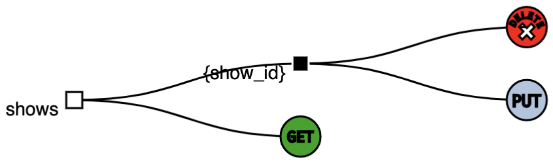


Figure 6.19. Enumerable Collection - PUT/DEL Variant (P1.v3)

This API structure appears one in the TVmaze user API showed in Figure 6.22. Where it is used for reading the collection of shows and deleting or updating each, another API example where this structure appears is Invotra API (Figure 6.20). In this case, adding a new user to the users' collection is possible due to the POST operation of the path `users`. However, it seems that the client is not allowed to add new user memberships to a specific team. While, according to the descriptions of the operations, it is possible to remove a user's membership of the team or update information about his team membership. Then, how can a team have new members? In this case, the user object schema is having a `teams` property of array type. Thus, adding a new member to a team is performed by means of the PUT operation provided in the path: `/users/{userId}`.

The Invotra API[3] can be seen as a combination of GET/PUT (P3.s2) (occurs twice) and PUT/DEL (P1.v3).

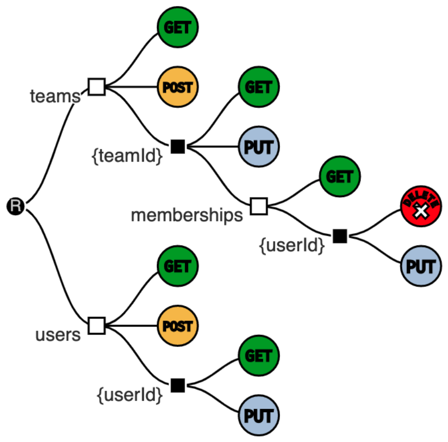


Figure 6.20. Invotra API Tree Visualization

GET/PUT/DEL (P1.v4)

The main characteristic of this structure is that, in addition to the GET and PUT methods, it also exposes a DELETE method on the collection items. This way, clients can not only read and write the associated content but can also remove items from the collection.

Size: 6 — Occurrence: 176 — Distinct Labels: 71

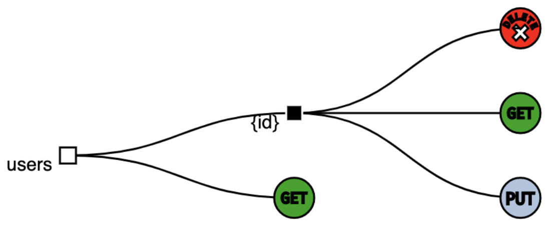


Figure 6.21. Enumerable Collection - GET/PUT/DEL Variant (P1.v4)

While in general, it can be useful to allow clients to remove items from a collection, it is not clear whether an API should support this for collections whose content can only be enumerated without providing the means for the service to mint identifiers for new items. Instead, new items can only be added by clients as long as they provide the new item's identifier.

[3] https://github.com/invotra/api

While this can lead to crashes when multiple clients attempt to invoke the PUT operation on the same item, we have observed different semantics for the PUT and DELETE methods. For example, some APIs use the DELETE method for something different: task cancellation. In this case, we assume that the tasks being performed within the server can be monitored by clients and when necessary can be interrupted.

Table 6.3. Content of the description field of the DELETE method of the variant GET/PUT/DEL (P1.v4)

- Close an existing position
- Delete Link - Will not delete the target object
- Delete a contact device for a user Delete a contact device for a user
- Deletes the given device, and invalidates any access token associated with it
- Delete a directory tenant under a resource group
- Delete a node - Remove the node identified by id. A node can only be deleted if it is currently offline
- Delete a node - Remove the node identified by id. A node can only be deleted if it is currently offline and does not host any master deployments
- Delete file - Delete file uploaded to a project from wall post or form
- Delete maintenance configuration
- Delete mock definition
- Delete snapshot repository - Deletes a snapshot repository configuration by name
- Delete the scheduled override assignment - Delete the scheduled override assignment
- Deletes a policy definition at management group level
- Deletes a policy definition
- Deletes a product package
- Deletes a server communication link
- Deletes a user from the list of registered users
- Deletes an acquired plan
- Deletes an existing server Active Directory Administrator
- Deletes single user
- Deletes specified file container - Delete an existing file container
- Deletes specified quota - Delete an existing quota
- Deletes the MariaDB Server key with the given name
- Deletes the MySQL Server key with the given name
- Deletes the PostgreSQL Server key with the given name
- Deletes the log profile
- Deletes the specified Azure key vault
- Deletes the specified application security group
- Deletes the specified public IP address
- Remove a CIDR Map
- Remove a Geographic Map
- Remove a Property
- Remove a Resource
- Remove a single task
- Remove an episode vote
- The operation to delete a container service
- Unfollow a network
- Unfollow a person
- Unfollow a show
- Unfollow a webchannel
- Unmark an episode
- Delete an Ad - you must own the Ad and be logged in to delete an Ad. Deleting an Ad will also erase all pictures uploaded to the API linked to it

For a better understanding, we have extracted the content of the description field of the DELETE method.

Looking at the descriptions of the delete method extracted from the OpenAPI documents in Table 6.3, it is clearly understandable that the DELETE operation is not always meant for clients to delete an item from the collection.

More in detail, in the description D-40, the DELETE method is allowing the client to delete a person from the list of followed people, but no append operation is provided. An example of an API where this fragment appears is in Figure 6.22. In this API example, we look at the fragment with labels sequence $S = people, \{people\_id\}$, in which we can notice that the following operation is done through the PUT method. In this case, when following a person, this new followed person is not appended to a collection of followed people, but instead, the followed person is updated through the PUT operation with the information about a new follower.

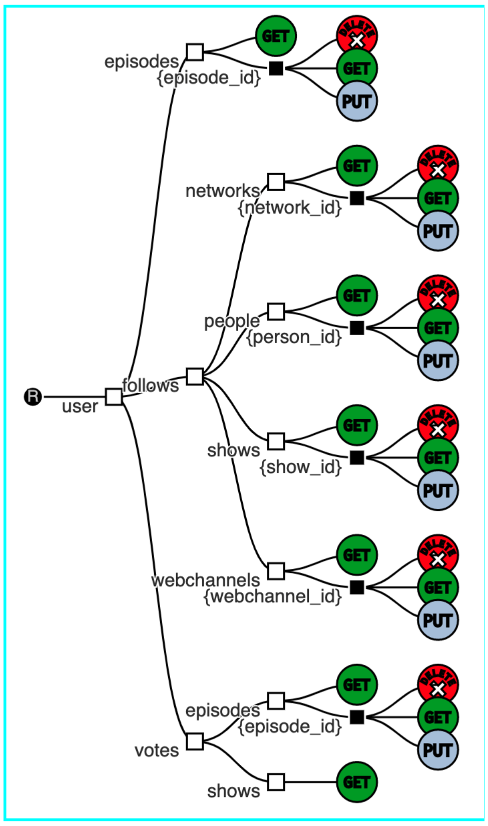


Figure 6.22. Tree visualisation for the TVmaze user API
(click for OpenAPI source)

In the TvMaze API[4] we can find six occurrences of Enumerable Collection - GET/PUT/DEL Variant (P1.v4), combined with one read operation on the path `/vote/shows`.

[4]https://static.tvmaze.com/apidoc/

Path segment	Method	Description
people	GET	List the followed people
{*people_id*}	GET	Check if a person is followed
	DELETE	Unfollow a person
	PUT	Follow a person

Table 6.4. Methods description of a fragment of Enumerable Collection - GET/PUT/DEL Variant (P1.v4), extracted from the OpenAPI description of TVmaze user API

Table 6.5. Extracted description of the POST method for Enumerable Collection - GET/POST Design Smell (P1.s1)

- Generates customized software development kit (SDK) and or tool packages used to integrate mobile web or mobile app clients with backend AWS resources
- "Write a range of table elements"
- Alert about something
- Builds templated versions of the challenge- Uses the flag format and seed to template out a new version of the challenge This may take a signficant amount of time-
- Create a deployment request
- Create a new user in system
- Generate token for valid user
- Perform pruning on input resource name
- Post message by username- Creates a message with the username as author
- Save a new revision of a page given in HTML format
- Set automation state- Set automation state for the given automation type
- Sets the value of a float variable
- Sets the value of a string variable
- Sets the value of an integer32 variable
- This endpoint returns the result of executing this operation
- This endpoint returns the result of executing this test
- Upload an Attachment- Upload an Attachment-

- **Enumerable Collection Design Smells**

GET/POST (P1.s1)

🟧 Ambiguous POST As opposed to updating the content of individual items of the previous variants, in this variant, the API makes it possible to fetch the current state of each item with GET and invoke some arbitrary operation on each of them with POST.

Size: 5 — Occurrence:77 — Distinct Labels: 22

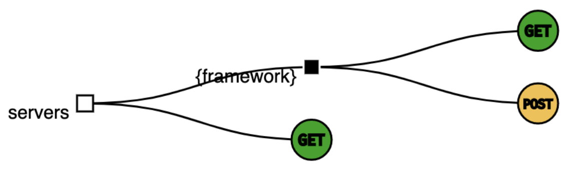


Figure 6.23. Enumerable Collection - GET/POST Design Smell (P1.s1)

Coming back to the OpenAPI descriptions of the APIs where this variant of fragments appears, we extracted the content of the summary and description fields for the POST method, which we list in Table 6.5. Based on the descriptions, we can detect two main use cases for the POST methods on the collection items:

(1) Appending an item to the collection: in this case placing the POST operation over the collection items can be seen as a common mistake.

(2) Updating an attribute of an existing item: in this case the POST is mistakenly used to perform the role of the PUT method.

GET/DEL (P1.s2)

▭ Delete without Create

This smell provides access to a collection whose items can be read and deleted, without offering clients the possibility to append new items.

We found that this smell only appears with 26 distinct labels. It appears 21 times out of 56 with labels represented by the label *instances*. This same label appears only once with the GET (P1.v1) variant. Other labels found in conjunction with this smell (e.g., tasks, jobs) would indicate uses for providing access to server-side resources which can only be monitored and eventually removed by clients, which do not have any control over their lifecycle.

Size: 5 — Occurrence: 56 — Distinct Labels: 26

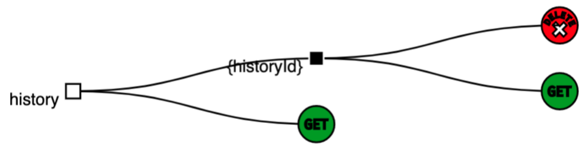


Figure 6.24. Enumerable Collection - GET/DEL Design Smell (P1.s2)

6.6.2 Appendable Collection (P2)

Summary

Append new items by posting them in the container resource

Problem

How to offer clients the ability to add new items to the collection?

Solution

Allow clients to use the POST method on the container resource to append new items into the collection. The address of the newly created items must be returned to the clients since this pattern does not feature the ability for clients to enumerate the content of the collection.

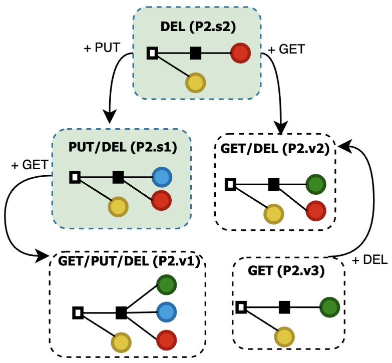


Figure 6.25. Appendable Collection Overview.

	2015	2016	2017	2018	2019	2020
P2.s1	0	0	0	4	12	25
P2.s2	2	2	3	4	8	38
P2.v3	1	3	13	20	34	104
P2.v2	5	10	14	16	25	68
P2.v1	4	9	19	25	53	63

Figure 6.26. Yearly distribution of the API specifications where the Appendable Collection (P2) variants and smells appear.

- **Appendable Collection Variants**

The common point between the variants of this primitive is that they all only allow the client to append on a collection, and to perform different operations on the items. Starting from the variant that allows all of GET/PUT/DEL operations, until the one that only allows reading the items. For this primitive, we have detected a design smell, where the client is not allowed to perform any read operation, neither of the collection nor on its items.

GET/PUT/DEL (P2.v1)

This variant allows clients full control over the items they have appended to the collection, as they can read, update and delete them.

Size: 6 — Occurrence: 194 — Distinct Labels: 64

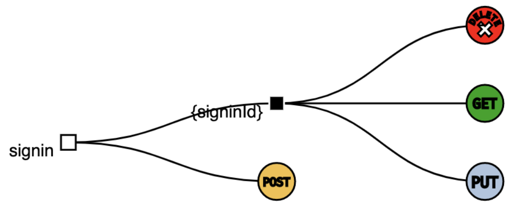


Figure 6.27. Appendable Collection - GET/PUT/DEL Variant (P2.v1)

To understand the reason for the absence of a read operation on the collection, we extracted natural language descriptions of the GET operation. We want to verify whether the designers mistakenly considered that the GET operation on the item would also serve to list all the collection content. In Table 6.6, we list some of the summaries and descriptions associated with the GET method. We can see from the descriptions that the GET operation is indeed used to retrieve specific elements from the collection.

Table 6.6. Description and summary of the GET method in GET/PUT/DEL (P2.v1)

- Find Tracks by ID - Returns a single Track
- Find ad by ID - Returns a single Ad
- Find ad_html_meta by ID - Returns a single AdHtmlMeta
- Find client by ID - Returns a single client
- Find course by ID - Returns a single course
- Find item by ID - Returns a single Item
- Find order by ID - Returns a single order
- Find pet by ID - Returns a single pet
- Find product by ID - Search one product by id
- Find provider by ID - Returns a single provider
- Find user - Returns a user
- Finds News by Id - Returns a single news
- Get Address by ID
- Get Student By Name - Get Student Details by name
- Get Usage by id
- Get a Client Registration for a given Client ID
- Get a Distribution - Get a Distribution
- Get a client by way of Client ID
- Get a project by project_id
- Get a single message
- Get a specific city
- Get a user by ID
- Get a user - Return a json object of the user
- Get an Assessment object
- Get bucketlist with given ID for loggedIn User
- Get details of an Order
- Get infos about a specific exam - Returns the exam id
- Get match
- Get one Product with specified ID
- Get provider by user code
- Get region by id
- Get scotch by id
- Get table
- Get team
- Get user by id - Get the user information by its id
- Get user by user id
- Get user by user name
- Gets Business Partner Object
- Gets an annotation Caller must have READ permission for the associated annotation set
- Gets an annotation set Caller must have READ permission for the associated dataset
- Gets the details for an order
- Look up a user by their user id
- Obter um momento
- Retrieve the information associated with a signin record
- Return a Question by ID - Returns a single Question object
- Returns a nomination based on a single ID - Returns the nomination identified by ‘nominationId‘
- Returns a single entry

Figure 6.28 shows an example of the use of this variant, where it is combined with the GET/DEL (P2.v2) variant. In this case, the PUT operation is used to update a sign-in record.

This is an API for the COVID-19 Contact Tracing QRCode Signin Server. It combines the GET/DEL (P2.v2) and GET/PUT/DEL (P2.v1) variants of the Appendable Collection (P2) primitive with a set of paths with a unique method (POST or GET).

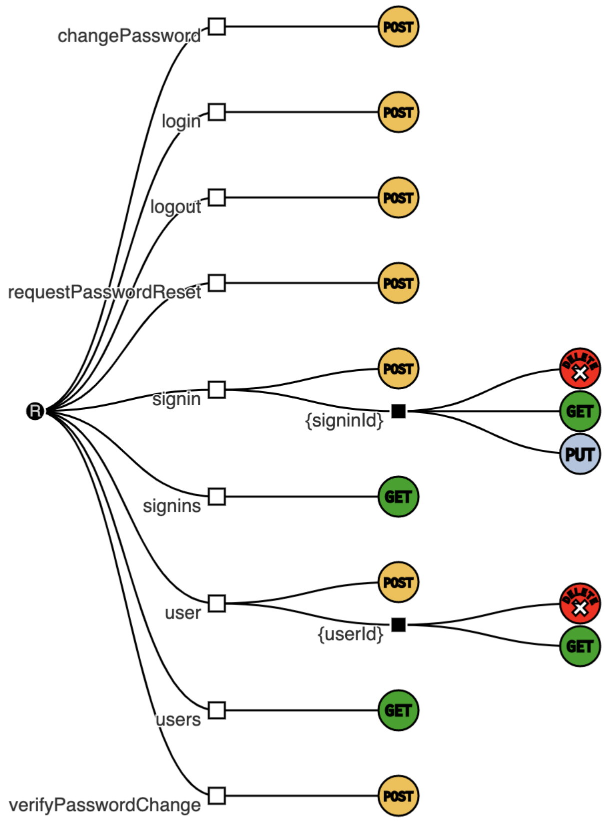


Figure 6.28. Tree visualization of an API for the COVID-19 Tracking QR Code Signin Server

GET/DEL (P2.v2)

This variant only allows you to read or delete individual collection items. It occurred 145 times, however, with only 43 distinct Label Sequences.

A concrete usage example of this primitive is in Passman API (visualized in Figure 6.39), an open-source developers API for Passman extensions. In the case of this example, the GET/DEL (P2.v2) variant is used in order to allow uploading and attaching a file to an item by means of the POST operation in the `/file` path. The client is also allowed to delete or get the content of a specific file, using, respectively the DELETE and GET operations allowed in the path `/file/{file_id}`. Another example is in Figure 6.28, where it is used beside the GET/PUT/DEL (P2.v1) variant, allowing to create a team member (user) record, to retrieve the information associated with a user's account, and finally to delete a team member's user record.

Size: 5 — Occurrence: 145 — Distinct Labels: 43

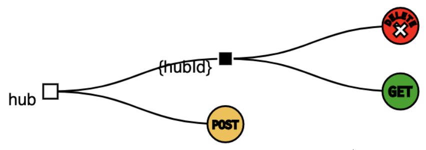


Figure 6.29. Appendable Collection - GET/DEL Variant (P2.v2)

GET (P2.v3)

This variant only allows the client to add elements to the collection, and then read each one, but it does not provide the ability to edit or remove items. Collections featuring this primitive contain resources which are garbage collected on the server-side, such as jobs, queries, or sessions. Another example is the append-only shopping cart in which clients can only add items without ever removing them.

Size: 4 — Occurrence:202 — Distinct Labels: 127

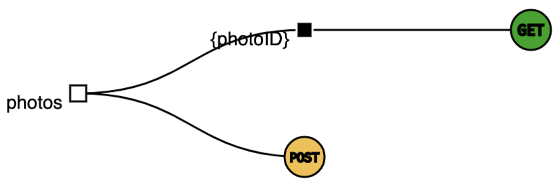


Figure 6.30. Appendable Collection - GET Variant (P2.v3)

- **Appendable Collection Design Smells**

PUT/DEL (P2.s1)

🟩 Write-Only

Instead of a GET operation, this variant introduces a PUT. However, it does not occur as frequently as the variants GET/DEL (P2.v2) and GET/PUT/DEL (P2.v1).

We have analyzed the 50 occurrences to attempt to determine how such a write-only API fragment would work, since it appears it is only possible to append new items, update or delete them. Indeed, no occurrence supports the ability to enumerate the content of the collection, nor it allows clients to read from its items.

Size: 5 — Occurrence:50 — Distinct Labels: 31

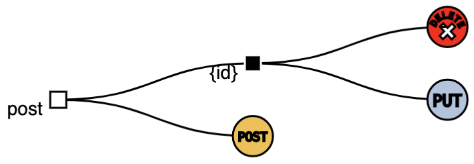


Figure 6.31. Appendable Collection - PUT/DEL Design Smell (P2.s1)

DEL (P2.s2)

🟩 Write-Only

Same as PUT/DEL (P2.s1), this variant does not provide the client the possibility of performing GET operations. Neither on the containers nor on the items. It only allows to append new items to the collection and delete them.

Such unreadable, write-only collection can still be useful, for example, to manage asynchronous jobs, or subscriptions or messages submitted into the API which can be only canceled from the clients. Since the collection cannot be enumerated, this works only if the address of the newly created items is returned to the client who created it using POST.

Nevertheless, we tag this variant as a smell, because of the strong limitations imposed by offering a write-only collection.

Size: 4 — Occurrence:69 — Distinct Labels: 51

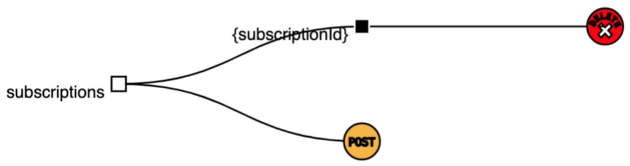


Figure 6.32. Appendable Collection - DEL Variant (P2.s2)

6.6.3 Collection (P3)

Also known as

Enumerable-Appendable Collection

Summary

Use the container resource to enumerate its content and add new items.

Problem

How to make the collection items discoverable by clients? How to let clients add items to the collection?

Solution

Group together related items under the same prefix. Allow clients to enumerate the items within the collection by applying the GET method to the container resource. Clients can use the POST method on the same container resource to add new items.

- **Collection Variants**

We present different variants featuring different method combinations on the collection item, starting from the one having four methods, all the way to fragments with a single method attached to the collection item.

In this primitive, we have detected two Design Smells (Figure 6.33), both are related to the ◻ Create without Delete smell.

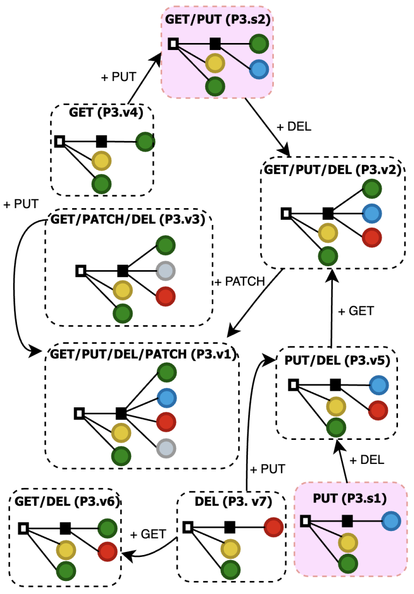


Figure 6.33. Collection – Overview of Variants

		2015	2016	2017	2018	2019	2020
	P3.s2	0	1	1	11	6	30
	P3.s1	0	1	6	11	14	17
	P3.v7	2	10	7	16	28	84
	P3.v6	1	3	33	43	40	134
	P3.v5	1	0	12	23	38	76
	P3.v4	1	17	33	36	51	112
	P3.v3	0	4	10	15	38	75
	P3.v2	6	89	53	91	157	287
	P3.v1	0	1	8	6	18	101

Figure 6.34. Yearly distribution of the API specifications where the Collection (P3) variants appear

Even the simplest variants with only one operation on the item to delete or update them would appear to lack the ability to directly read individual collection items. While this is the case, as opposed to the previously discussed Appendable Collection smells, clients can still fetch the content of the entire collection using the GET operation provided by the container resource and then extract the values for individual items from the result.

GET/PUT/DEL/PATCH (P3.v1)

The first variant in this collection is the one providing all of the GET, PATCH, PUT, and DELETE operations.

Although this variant includes most HTTP verbs and thus is the most expressive in terms of which operations clients can perform on collection items, it is far from being the most frequently used in practice. An example of use of this variant is in ID Vault API (Figure 6.36), where it appears 6 times.

Size: 8 — Occurrence: 328 — Distinct Labels: 159

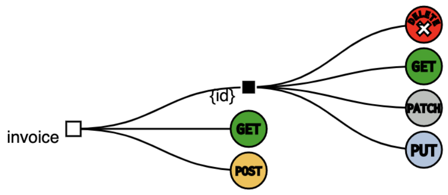


Figure 6.35. Collection - GET/PUT/DEL/PATCH Variant (P3.v1)

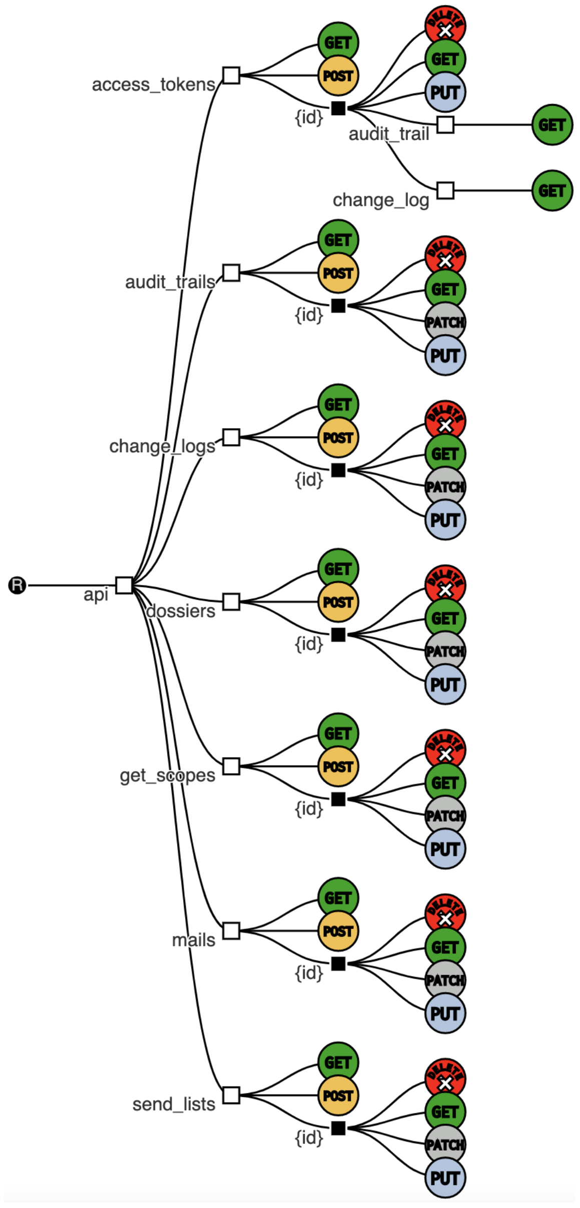


Figure 6.36. Tree Visualization of ID Vault API. This is an API example where the Collection - GET/PUT/DEL/PATCH Variant (P3.v1) appears several times, combined with one use of the GET/PUT/DEL (P3.v2) variant.

GET/PUT/DEL (P3.v2)

Fragments of this variant combine both the POST and GET operations on the collection. With more than one thousand occurrences, this variant (Figure 6.37) is the most occurring we have mined, not only within the variants of this collection but also among all the fragments having more than 3 distinct methods in their leaves. Several instances of this primitive can be found with different labels in the Apacta API (Figure 5.4).

Size: 7 — Occurrence: 1123 — Distinct Labels: 574

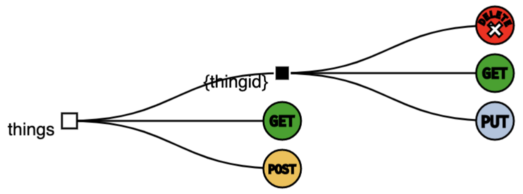


Figure 6.37. Collection - GET/PUT/DEL Variant (P3.v2)

GET/DEL/PATCH (P3.v3)

This variant uses a PATCH operation instead of the PUT as in variant GET/PUT/DEL. The Passman API (Figure 6.39) is an example of an API in which this variant appears. The Passman API[5] combines all of the GET/DEL/-PATCH (P3.v3) and GET/DEL (P2.v2).

Size: 7 — Occurrence: 233 — Distinct Labels: 139

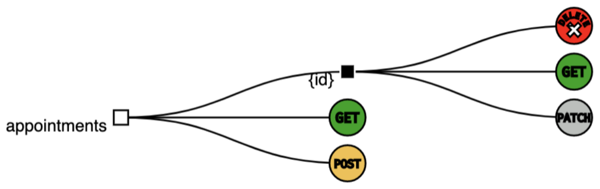


Figure 6.38. Collection - GET/DEL/PATCH Variant (P3.v3)

[5] `https://github.com/nextcloud/passman#api`

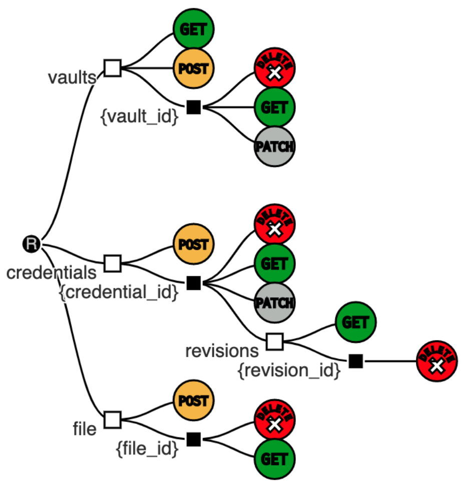


Figure 6.39. Tree Visualization of Passman Developers API.

GET (P3.v4)

This is the simplest variant of this collection. The client cannot perform any operation on the collection items, except to read their content. We have found examples of account collections, whose content cannot be modified by clients. Likewise, this is a common structure for long-running operations [122], which are started with a POST request used to transfer the input of the computation, while the status of the ongoing job and its result can be retrieved from the corresponding item.

Size: 5 — Occurrence: 323 — Distinct Labels: 168

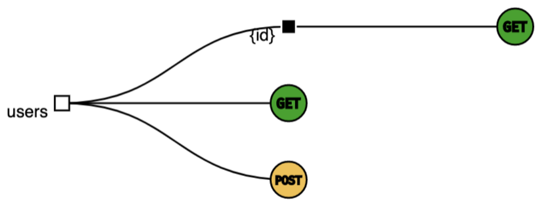


Figure 6.40. Collection - GET Variant (P3.v4)

PUT/DEL (P3.v5)

This variant makes the client unable to individually read each item of the collection. However, it is possible to list them all, insert new items, delete or update them. Examples of such collections with unreadable items would contain simple items whose address indicating their identity and existence is sufficient to control their lifecycle (e.g., using the PUT operation to control the video or audio track playback). Likewise, to set the quantity of individual order line items or remove them from the order altogether one does not need to be able to retrieve any information about them. Also because such information can be fetched when enumerating the content of the entire collection.

Size: 6 — Occurrence: 169 — Distinct Labels: 84

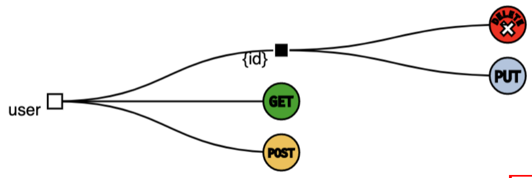


Figure 6.41. Collection - PUT/DEL Variant (P3.v5)

GET/DEL (P3.v6)

This variant only allows to read or delete individual collection items. This is one of the most frequently found variants, with a collection storing a wide variety of items. For example, once blog posts, comments, or questions are published, they cannot be updated but just removed. Likewise, it appears there is no need to update the ingredients of a recipe.

Size: 5 — Occurrence: 345 — Distinct Labels: 187

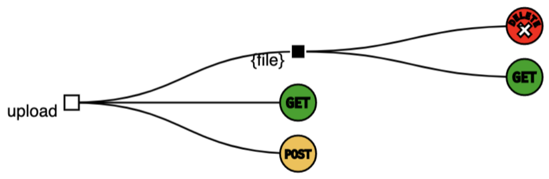


Figure 6.42. Collection - GET/DEL Variant (P3.v6)

DEL (P3.v7)

This is a simpler variant, where it is possible to the client to list the elements of the collection and insert elements into it. Once the items have been added, it is only possible to remove them. In addition to bookings, this variant has been frequently used for collections of blog post comments, product reviews, or favorite bookmarks, whose content can be shown when retrieving the entire collection, but for moderation purposes, it may be necessary to be able to remove individual items.

Size: 5 — Occurrence: 201 — Distinct Labels: 87

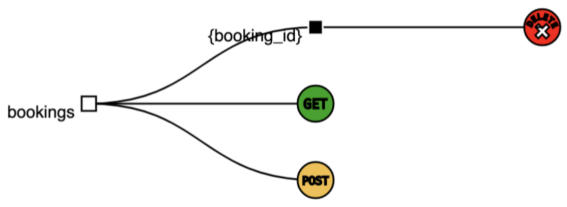


Figure 6.43. Collection - DEL Variant (P3.v7)

- **Collection Smells**

PUT-Only (P3.s1)

▭ Create without Delete

This smell provides only one operation to update individual items of the collection but lacks the affordance for deleting individual items. This is used with collections of items whose state should be controlled by clients, for example, to configure or simply switch on or off devices, gateways, or services through a management API.

Size: 5 — Occurrence: 78 — Distinct Labels: 48

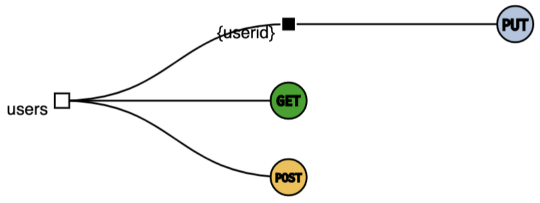


Figure 6.44. Collection - PUT-Only Design Smell (P3.s1)

GET/PUT (P3.s2)

▭ Create without Delete

Also, this smell does not allow clients to delete an item from the collection, however, it allows them to insert items, read them, and update them.

It has been used to design APIs which provide access to collections of users, user accounts, customers, employees, or withdrawals. These are resources which once they are created may need to be preserved forever for legal reasons, due to data preservation or retention regulations.

Size: 6 — Occurrence: 63 — Distinct Labels: 47

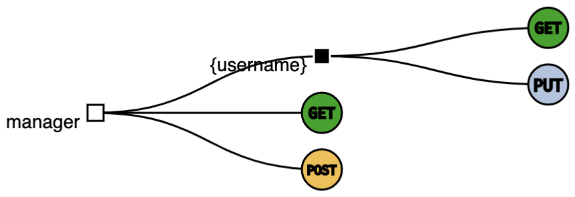


Figure 6.45. Collection - GET/PUT Design Smell (P3.s2)

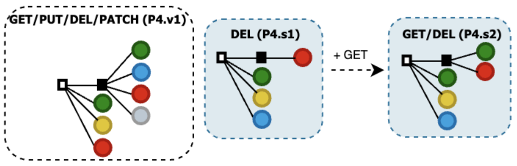


Figure 6.46. Mutable Collection (P4) Overview

	2015	2016	2017	2018	2019	2020
P4.s2	0	2	5	8	17	29
P4.s1	0	1	0	4	2	17
P4.v1	0	1	2	1	5	29

Figure 6.47. Yearly distribution of the API specifications where the Mutable Collection (P4) variants and smells appear

6.6.4 Mutable Collection (P4)

Summary

Replace the content of the collection (PUT) or clear the entire content of the collection (DELETE)

Problem

How to let clients bring the whole content of the collection to a known state?

Solution

Add DELETE or PUT method to the container resource.

- **Mutable Collection Variants**

The two variants with PUT do not seem to be used to replace the content of the entire collection (only a few exceptions). Instead, the PUT method is used to edit or update individual items, addressed by a query parameter that identifies the item to be replaced. (Query parameters are not shown in the figure).

GET/PUT/DEL/PATCH (P4.v1)

This variant provides only a delete operation on the collection. This variant occurs 74 times. It provides besides the DELETE and GET operations on the collection items, as well as PUT and PATCH operations. The second label for all the labels sequences is $\{name\}$ except one sequence $S = \{boards\ , \{id\}\}$

For verifying the purpose of use of the DELETE method over the collection, we have extracted the descriptions associated with that method in the OpenAPI specification of the APIs where that fragment occurs. Indeed, in this case, the DELETE operation is used to delete the whole collection. This "DELETE all" variant could be promoted to a separate primitive named “Erasable Collection”.

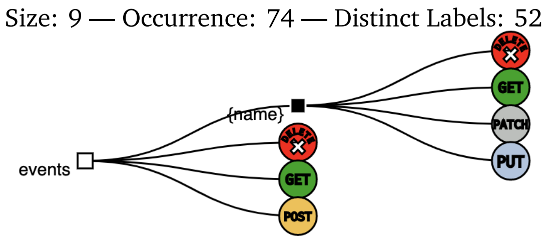


Figure 6.48. Mutable Collection - GET/PUT/DEL/PATCH Variant (P4.v1)

- **Mutable Collection Design Smells**

DEL (P4.s1)

▭ Ambiguous PUT This structure provides both read and append operations over the collection, in addition to a PUT operation. While the only operation that the client can perform over the collection items is a delete. For verifying the real purpose behind having a PUT operation over the collection resource we extracted the descriptions associated with this method in the OpenAPI documents. The text shows that, in reality, in almost all cases, the PUT is used to update an item of the collection. The address of the item is provided as a request parameter, as opposed to using the resource path as with most other primitives. Only in a few cases, it is actually used as one would expect, for updating the whole collection with a single batch operation to replace its content.

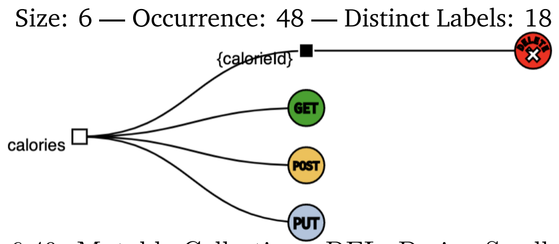


Figure 6.49. Mutable Collection - DEL Design Smell (P4.s1)

GET/DEL (P4.s2)

▭ Ambiguous PUT

The fragments of this variant combine all of GET, PUT, and POST operations on the collection. An example of the most occurring fragment in this variant is in Figure 6.50. This fragment occurs 102 times with 56 distinct Label Sequences.

Also, in this variant the PUT method on the container resource is used mostly to update the content of individual itemss, thus leading to some ambiguity as it should have been associated with the collection item resource.

Figure 6.51 show an example of API where this variant is used twice.

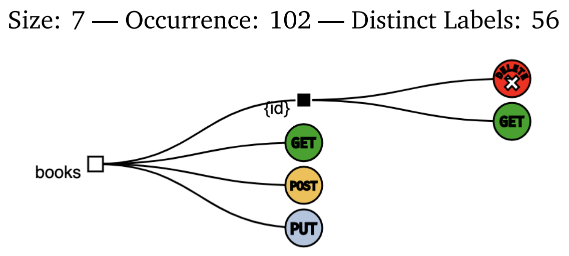


Figure 6.50. Mutable Collection - GET/DEL Design Smell (P4.s2)

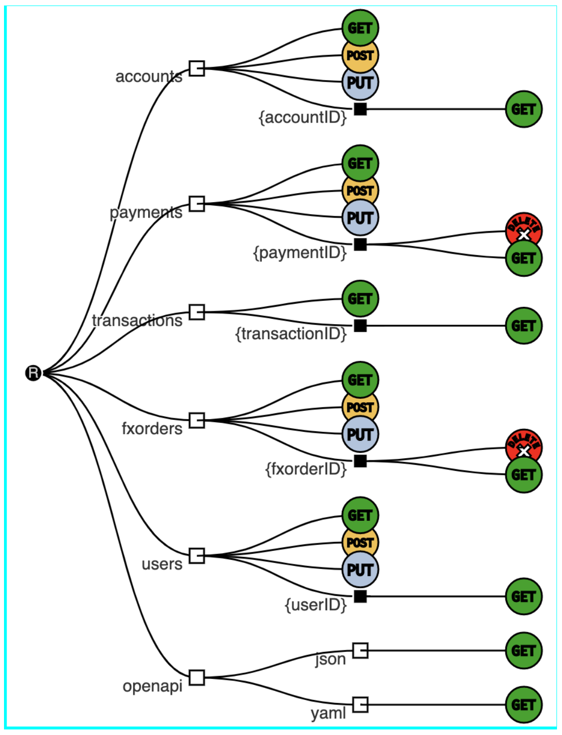


Figure 6.51. Tree visualization of AnyPay API. AnyPay service targets parents and children who are making payments. It is an example of the usage of the GET/DEL (P4.s2) variant.

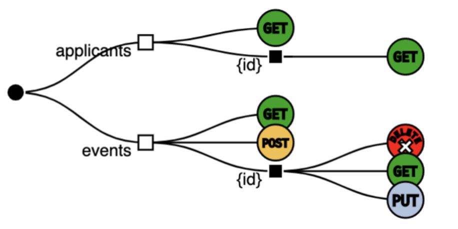
6.7 From Primitives to Larger Structures and API Responsibility Patterns

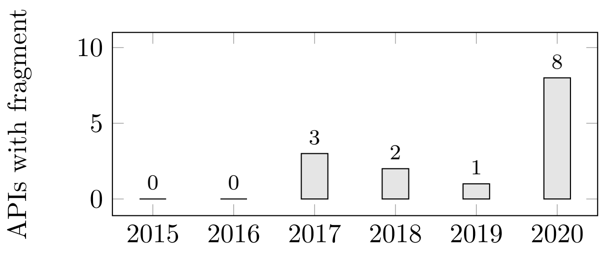
This section gives two examples of how the mined primitives presented in the previous sections can be used during API design and API reviews. First, we discuss primitive composition. Next, we briefly outline how the structural primitives from this chapter relate to previous work on API design patterns and interface description languages.

6.7.1 Composing Primitives

The basic collection primitives can be composed to form larger API structures in two ways:

Table 6.7. API Fragments Composing the Read-only Collection and the Collection primitives side by side





Size: 12 — Occurrence: 33 — Distinct Labels: 12

Label clusters of sample known uses:
(O: Occurrence, C:Cohesion)

Cluster	$Label_1$	$Label_2$	$Label_3$	$Label_4$	O	C
C1	*applicants*	{*id*}	*events*	{*id*}	2	8.33
	users	{*id*}	*teams*	{*id*}	1	8.16
	users	{*id*}	*tasks*	{*id*}	1	8.33
	gst	{*gst_id*}	*base_slice_des*	{*id*}	1	2.47
	application − templates	{*id*}	*applications*	{*id*}	1	4.08
C2	*contact_types*	{*contact_type_id*}	*contacts*	{*contact_id*}	6	8.04
	form_templates	{*form_template_id*}	*forms*	{*form_id*}	6	8.16
	time_entry_intervals	{*time_entry_interval_id*}	*time_entry_types*	{*time_entry_type_id*}	6	8.04
	time_entry_value_types	{*time_entry_value_type_id*}	*users*	{*user_id*}	6	8.04
C3	*resources*	{*location*}	*function*	{*func_id*}	1	2.66
C4	*cookings*	{*uuid*}	*programs*	{*uuid*}	1	8.04

1. Unrelated collections can be added to the API by adding the corresponding container resource on the same level as shown in the fragments of Table 6.7;
2. Related collections can be nested inside one another, by adding a sub-container resource within each item of the main collection, as shown in the fragment of Table 6.8.

In general, we found that both side-by-side composition and nesting can be used together in the same API. The Investra API shown in Figure 6.20 is an example of an API entirely composed of two primitives.

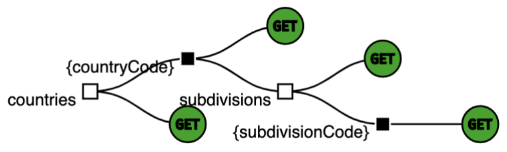
6.7.2 Relation to Architectural Patterns

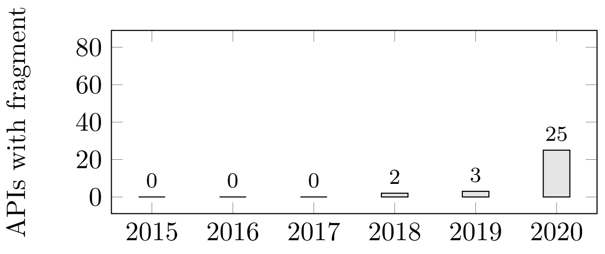
In the patterns community, technology- and platform-neutral interface representation and service design patterns have been mined and published. The Microservice API Patterns (MAP) language [189], for instance, focuses on the design of remote APIs — including but not limited to service-oriented architectures. MAP has two categories that complement the API primitives and fragments carved out in this chapter, *structure* [190] (of request and response message representations, not HTTP resource tree structures as covered in this chapter) and architectural *responsibility* [185].

The HTTP methods found in the resource trees in Sections 5.1.2 and 6.6 map to the MAP language as this:

- HTTP GET methods are "Retrieval Operations"[185].
- HTTP POSTs can be "State Creation Operations" but also "State Transition Operations" (partial update variant) [185].

Table 6.8. API Fragments Composing the Read-only Collection Primitives with Nesting





Size: 8 — Occurrence: 32 — Distinct Labels: 13

Label Clusters of Sample Known Uses: (O: Occurrence, C: Cohesion)

Cluster	$Label_1$	$Label_2$	$Label_3$	$Label_4$	O	C
C1	*companies*	*{company_id}*	*integration_feature_settings*	*{integration_feature_setting_id}*	5	6.66
	identity − classes	*{identityClassID}*	*levels*	*{identityLevelID}*	1	7.44
C2	*BRK_waardelijsten*	*{waardelijstidentificatie}*	*waarden*	*{code}*	2	4.98
	tabellen	*{tabelidentificatie}*	*waarden*	*{code}*	2	5.57
C3	*components*	*{component}*	*resource − managers*	*{resource − manager}*	2	6.05
C4	*stations*	*{stationId}*	*history*	*{columnName}*	1	5.65
C5	*entities*	*{entityName}*	*views*	*{viewName}*	3	7.64
C6	*countries*	*{countryCode}*	*subdivisions*	*{subdivisionCode}*	1	8.05

- HTTP PUTs are "State Transition Operations" (full replacement variant).
- PATCHes correspond to "State Transition Operations" (partial update variant).
- DELETE methods are represented as variants of State Transition Operations.

The remaining operation pattern from MAP, "Computation Function", can be mapped to HTTP GET (if its request parameters are simple) or POST (if request parameters are complex).

The collection primitives that we derived from URI structures in Section 6.6 correspond to the MAP endpoint pattern "Information Holder" and its specializations[187]:

- "Master Data Holders" expose many GET retrievals and only a few bulky create POSTs and update PUTs. They usually are enumerable, and also appendable (at least for certain clients).
- "Operational Data Holders" typically are Enumerable and Appendable Collections, often also Mutable Collections.
- "Reference Data Holders" are read-only and therefore Enumerable Collections.
- "Data Transfer Resources" are Mutable Collections decoupling multiple application clients.
- A set of related "Link Lookup Resources" forms a Collection as well; each item in such collection is mutable and so is the entire collection.

In Summary

This chapter presents a data-centric pattern mining approach. We applied it to find recurring primitive structures within Web API descriptions.

To do so, we extracted recurring fragments from a large collection of OpenAPI specifications gathered from open source repositories. Reflecting the hierarchical nature of HTTP-based APIs, these fragments are represented as trees. These trees are built out of resource identifiers; they can be traversed to obtain all paths that are present in the original OpenAPI specification. The leaves of the API trees refer to the HTTP methods that invoke the corresponding operations.

From a population of thousands of fragments, we selected those that a) frequently occur, b) have a relatively small size, and c) are centered around the notion of resource collection. As shown in Figure 6.13, we distinguish the following cases a) collections only offer operations on their items or on the collection level as well, b) their content can be enumerated, new items can be created, and/or both enumeration and creation are supported, and c) batch removal and update are provided.

For every primitive, we presented a selection of variants together with the corresponding label clusters and, in some cases, descriptions associated with the operations. A few variants can be also seen as design smells, for instance, if they use the HTTP method semantics incorrectly or inconsistently.

Our results are a collection of pattern primitives, which can be (and have been) composed to build larger API structures. For instance, this becomes evident when connecting the syntactical patterns that we mined here automatically with semantic architectural patterns previously mined manually by knowledge engineers [185, 187].

Part III

Web API changes analysis

Chapter 7

Web API Changes

This chapter lists the common Web API changes, empirically mined from the unique histories of nearly five thousand APIs evolving across over a hundred thousand commits. The goal is to systematically identify and classify these changes, shedding light on the evolution patterns of Web APIs and providing a deeper understanding of their dynamics. By examining common types of changes—whether breaking, non-breaking, or undecidable—this analysis aims to help developers anticipate potential changes they may need to address and highlights the importance of developing tooling capable of detecting such changes proactively, mitigating their potential impact on clients.

7.1 Web API Change

7.1.1 Definition

A key concept in this chapter is "change". A **Web API Change** refers to any modification in the structure, behavior, or permissions of an API' elements, which affects how clients interact with it. These changes can include modifications to endpoints, request methods (e.g., **GET**, **POST**), parameters (query parameters, request bodies), response formats, or security components.

In this context, we are specifically **focused on interface-level changes—the external contract that a Web API exposes to clients—rather than changes to any SDK (Software Development Kit) or client libraries** that might be generated or provided to consume the API. The key aspect is that these changes are analyzed independently of the underlying framework, language, or platform used to implement the API. Our goal is to study the evolution of Web APIs purely from an interface perspective, ensuring that the findings apply universally, regardless of the technical stack used.

7.1.2 Web API changes traceability

When a Web API provider introduces changes in a new release, they must communicate these updates through one or multiple channels. API consumers need to be aware of the methods the provider uses to stay informed and to track those changes, enabling safer transitions and more informed decisions [98].

We classify the source of Web API changes traceability into **explicit** and **implicit** ones. **Explicit** traces are intentional and well-documented changes made by developers to inform users about modifications, while **implicit** traces include unintentional or informal records that can still provide insight into the evolution of an API. We list in what follows some examples of these traces.

Explicit Traces

1. **Changelogs:** Developers might explicitly document updates, including added, modified, or removed features. These documents are meant to be used by the Web API consumer, so they might not depict all the changes that the API undergoes with a low level of granularity. In addition, they are often not machine-readable and do not follow a universal standard.

2. **Versioned Documentation:** Separate unstructured documentation for each version provides an intentional reference for changes.

3. **Semantic Versioning:** APIs adopting the "Semantic Versioning" evolution pattern [98] (e.g., v1.0.0, v2.0.0) explicitly indicate the type of change (major and breaking, minor and non-breaking, or bug fixes). This can reflect the impact of the introduced changes, but we still need to go over other traces to be informed about the changes. However, semantic versioning alone is not sufficient to be used to understand the potential impact of the changes on certain clients.

4. **Updated API Specifications (OpenAPI, Swagger):** Formal changes in specification files that are easy to compare across versions. Mainly because they are machine readable (YAML/JSON), which facilitates their comparability and ensures that they are often aligned with the code, making it a reliable source for describing the API's behavior.

5. **Release Notes:** Detailed descriptions of new features, improvements, or removed functionality in each API release. they often follow a structured format, but there is no standard to follow. The release notes can offer a clear idea of the different improvements, fixes, and new features, but they may omit minor changes or deprecated features.

6. **Deprecation Notices:** Notifications in the documentation and API response header signal the upcoming removal of certain features. API users can easily miss these notices if they do not go back to the documentation website or do not read the response headers. Links to details about formal procedures that outline when features will be removed and how users should transition are often provided [44].

Implicit Traces

1. **Commit Messages in Version Control (e.g., Git):** Developers may leave descriptions of changes in commit messages, but this is often for internal use rather than explicit API documentation.

2. **Git Tags and Branches:** While tags and branches can mark releases, they are often implicit and not always used to provide clear information about API changes.

3. **API usage logs:** Error codes (e.g., 410 Gone) can signal the removal of endpoints, but they may not explicitly describe what has changed.

4. **Codebase Comments and Documentation:** Comments in the code may explain changes but are generally intended for developers and are not a formal record of API changes.

7.2 Related Work

Researchers have distinct approaches to explore the changes in Web API histories. We distinguish in what follow between the ones that rely on explicit change traces and the ones that use implicit ones to mine changes from.

7.2.1 Change patterns extraction from explicit traces

To study Web API changes, we first studied the literature on Web API evolution to identify the common changes that researchers identified and the reasons behind each change.

In [154], Sohan et al. analyzed changes happening in a sample of nine famous Web APIs: Facebook Platform API, Twitter REST API, WordPress REST API, Salesforce & Chatter REST API, Google Calendar API, Stripe Payment Processor API, GitHub API, Google Maps API, OpenStreetMap API. The goal is to analyze how these APIs were versioned during their evolutions, and how the changes were documented and communicated, looking at the available public resources (Explicit Traces) such as the API homes pages and API changelogs from multiple versions. The authors analyze API evolution by tracking additions, deletions, and modifications that happen at the level of *Endpoints*, *Resources*, *Methods*, *Fields*, *Field Values*, and other types of changes such as modifications of *Error Handling*, or *Behavior Changes* which are not reflected by the interface, and *Security (Authentication/Authorization)*.

In what follows I list the identified changes and discuss how we attempted to improve this classification and the reasons behind that:

- **ET-Change 1:** Added API Element

  This includes any new addition to the API, such as introducing a new endpoint, parameter, or data property.

- **ET-Change 2:** Deleted API Element

  Refers to the removal of an endpoint, parameter, or data property. This change is significant as it can break client applications relying on the deleted element.

- **ET-Change 3:** Moved API Element

  Refers to any instance where an API element is relocated, such as moving a parameter from one endpoint to another or transferring a data property from one object to another.

- **ET-Change 4:** Renamed API Elements

  This refers to changes where an API element, such as a parameter or property, is given a new name.

- **ET-Change 5:** Behaviors Changes

  In this type of change, the authors mix up in the way they defined it both the backend functional or semantic changes with the schema changes of the data that the API outputs in responses. In our classification, we are contradistinguishing between these types of changes since we consider the data schema design a decision tied to the interface's design. Using our approach, we are not able to capture functional changes that are not reflected in the API.

- **ET-Change 6:** Post Condition Update

  This change doesn't alter the immediate result of the API call (e.g., the payment is still processed as usual). However, new obligations are imposed on the developer after the API call. The developer must now set up a system to handle post-call updates and comply with additional terms. For some reason, the researcher classified this type of change under this category, while it actually falls also under the *Behavior Changes*.

- **ET-Change 7:** HTTP Headers Change

  The header changes can also fall under the *Renamed API Elements* or *Moved API Elements* categories, but the authors decided to put it under a distinct category.

- **ET-Change 8:** Error Handling and Messages Changes

  These changes include modifications targeting the specific response code sent at each encountered error.

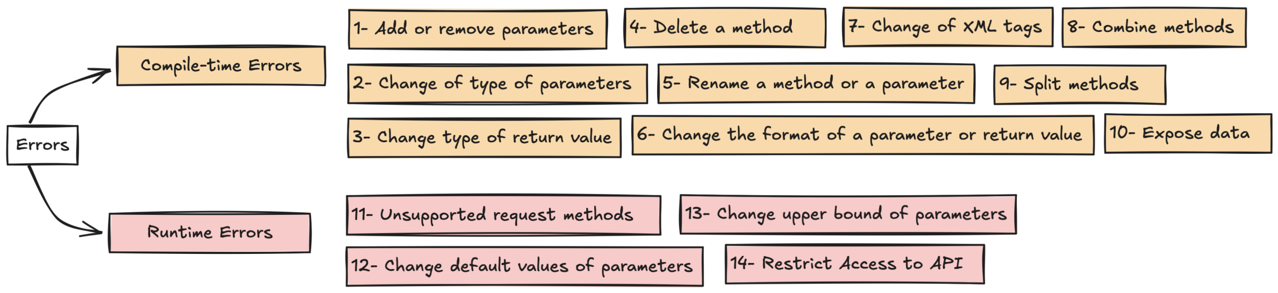

Errors
Compile-time Errors
Runtime Errors
1- Add or remove parameters
2- Change of type of parameters
3- Change type of return value
4- Delete a method
5- Rename a method or a parameter
6- Change the format of a parameter or return value
7- Change of XML tags
8- Combine methods
9- Split methods
10- Expose data
11- Unsupported request methods
12- Change default values of parameters
13- Change upper bound of parameters
14- Restrict Access to API

Figure 7.1. API Changes classification in [93]

- **ET-Change 9:** Change Field Datatype

  Refers to changes in the data type of an API element, such as modifying a field from a string to an integer. These changes can have significant implications for data validation and client compatibility.

- **ET-Change 10:** Authorization/Authentication Changes

  Includes updates to the security mechanisms of the API, such as introducing new authentication methods, updating token structures, or modifying required permissions.

In this classification, the authors mix up any addition/deletion/moving that happens in the elements of the API, such as moving a parameter from an endpoint to another one or moving a data property from a data object to another. Since these types of changes do not necessarily have the same impact, in our classification, we attempt to put under the same classification changes that will always have the same impact.

Same as for the *Added/Deleted/Moved API Elements* type of change, renaming a parameter does not necessarily have the same impact as renaming a property of a data object. The impact also depends on whether these elements are optional or required. In our case, we take into account this fine-grained classification.

The granularity level of this classification decision made by [154] is not uniform. While HTTP headers and error codes change and grouped under the elements type, other changes like parameters renaming or operation addition/deletion are grouped under the action: API element renamed/deleted/added. Another type of classification is suggested by Li et al. in [93], where they identified 14 types of changes by studying two consecutive versions of five well-known APIs from distinct providers: Google Calendar, Google Gadgets, Amazon MWS, Twitter API, and Sina Weibo. We aggregate these changes as follows depending on the API elements they target. The study was also done by looking at Web API documentation available in their websites (Explicit traces). We summarize in Figure 7.1 the changes that the authors have classified into errors that can happen at compile-time or run-time. Li et al. have not included in their change impact analysis the changes that do happen but do not cause in any error either at the level of the wrapper API or the HTTP API.

Li et al. identified two types of changes that were absent from the classification of Sohan et al. [154]:

- **ET-Change 11:** Split methods

  This change involves breaking down an existing method or operation into multiple smaller methods. This may occur to simplify the functionality, improve clarity, or better align with RESTful principles. For instance, an API operation handling both data creation and modification might be split into separate 'POST' and 'PUT' methods.

- **ET-Change 12:** Combine methods

  This change merges multiple operations into a single method. The intent is to reduce redundancy or consolidate related functionalities into one endpoint for ease of maintenance and usage. For example, merging separate **GET** operations for different filters into a single **GET** operation with query parameters to handle the variations.

Stocker and Zimmermann [157] included six additional change patterns in their catalogue, each varying in its impact on backward compatibility and client adaptation effort. These include introducing a Data Transfer Object to encapsulate internal data structures without breaking the API contract, adding a Wish List parameter to provide clients finer control over response content, and introducing pagination to deliver large datasets in manageable chunks, though not always recommended for backward compatibility. Other patterns include segregating commands from queries to independently optimize read and write models using tools like API Gateways or Version Mediators, introducing a Version Mediator to temporarily support deprecated APIs while ensuring a smooth transition for clients, and renaming, which was already listed by Li et al.. But in this case, Stocker and Zimmermann refer to renaming happening at the level of data model.

7.2.2 Change patterns extraction based on implicit traces: API usage logs

Changes can also be extracted from implicit resources, such as API call logs, which are rarely publicly available and difficult to obtain from API providers. Koçi et al. proposed in [81] a process mining-based approach taking as import API call logs to propose the next needed modifications, additions, or deletions in a web API. They have detected several different call patterns on which they based the next evolution possibilities. The changes extracted by Koçi et al. show how the changes identified by Sohan et al. [154] and Li et al. [93] can be combined to reach a specific functional feature change.

In what follows, I extract all the different types of changes that the Koçi et al. have identified and how these changes can be detected based on OpenAPI specification modifications[1].

Reflexive loop: This detected call pattern consists of an endpoint called several types with different values of query parameters/body payload. It can be improved by making the endpoint take an array of values instead of a single value – introducing a request bundle. However, this assumes that each subsequent call is not independent from the results of previous calls.

The category query parameter only accepts a single value. To retrieve items from multiple categories, the endpoint would need to be called multiple times, one for each category. In Listing 7.2 the categories query parameter now takes an array of values, allowing consumers to retrieve items from multiple categories in a single API call. This reduces the need for repeated calls and optimizes performance. This type of change can be detected in OpenAPI if a parameter schema has changed type from a string to an array.

IT-Change 1: Adding parameters to accept multiple values, reducing the need for repeated calls.

Risk: Unintended behaviors might occur if the feature is not used correctly. The client might send large data that exceeds the URL length limit (2048 characters) or the web server's request body size limit.

For instance, in the example of Listing 7.1, the **GET** `/items` endpoint allows fetching items classified in a specific category. It might be that the subsequent calls depend on the number of items listed in previously fetched categories.

[1]Note that in this thesis, I am not going to explain in detail the call patterns but I am going to focus on the modifications they suggested based on each detected pattern. And I will discuss the tradeoffs that each of these. And later explain how our analysis complete this approach.

Listing 7.1. Before IT-Change 1

```
paths:
 /items:
  get:
   summary: Retrieve items by a single category
   parameters:
    - in: query
     name: category
     schema:
      type: string
     required: true
     description: Filter items by a single category
   responses:
    '200':
     description: A list of items
     content:
      application/json:
       schema:
        type: array
        items:
         type: object
         properties:
          id:
           type: integer
          name:
           type: string
          category:
           type: string
```

Listing 7.2. After IT-Change 1

```
paths:
 /items:
  get:
   summary: Retrieve items by multiple categories
   parameters:
    - in: query
     name: categories
     schema:
      type: array
      items:
       type: string
     style: form
     explode: true
     required: false
     description: Filter items by a list of categories
   responses:
    '200':
     description: A list of items
     content:
      application/json:
       schema:
        type: array
        items:
         type: object
         properties:
          id:
           type: integer
          name:
           type: string
          category:
           type: string
```

Direct-follow nodes: This pattern describes the case where the set of endpoints are always called together in a specific order and never called independently. A suggested change would be to merge these endpoints in one that offers all the functionalities. The authors do not explicitly state whether the old endpoints can be removed or should be preserved.

IT-Change 2: Merging the functionalities of two or more endpoints in one endpoint.

This means that two endpoints will be deleted and a new one will be created.

Risks: Merging endpoints can increase complexity, reduce flexibility, and introduce backward compatibility and could also return unnecessary data (e.g., reviews when they are not needed). Additionally, it may complicate error handling, testing, and versioning if not carefully managed.

In the example of Listing 7.3 there are two separate endpoints for handling items and their reviews. The **`GET`** `/items` endpoint allows users to retrieve the details of a specific item bypassing the itemId as a query parameter, returning information like the item's ID and name. Meanwhile, the **`GET`** `/items/itemId/reviews` endpoint is used to fetch a list of reviews associated with that item, such as review IDs and comments. These endpoints are independent, and each serves a distinct purpose.

After the change, in Listing 7.4, the two endpoints are merged into a single endpoint **`/items`**. This new version retrieves both the item details and its associated reviews in one call. The **`itemId`** parameter still filters the data for a specific item, but now the response includes not only the item's basic information (ID, name) but also an array of reviews. This change streamlines the process, reducing the need for two separate API calls to get both the item details and its reviews.

This change is represented in the OpenAPI specification by the removal of one endpoint, while the schema of another endpoint is expanded to include the fields from the deleted endpoint's schema.

Listing 7.3. Before change IT-Change 2

```yaml
paths:
 /items:
  get:
   summary: Retrieve item details
   parameters:
     - in: query
      name: itemId
      schema:
        type: string
      required: true
      description: Get details of a specific item
   responses:
     '200':
      description: Item details
      content:
        application/json:
          schema:
            type: object
            properties:
              id:
                type: string
              name:
                type: string
 /items/{itemId}/reviews:
  get:
   summary: Retrieve reviews for an item
   responses:
     '200':
      description: A list of item reviews
      content:
        application/json:
          schema:
            type: array
            items:
              type: object
              properties:
                reviewId:
                  type: string
                comment:
                  type: string
```

Listing 7.4. After IT-Change 2

```yaml
paths:
 /items:
  get:
   summary: Retrieve item details along with reviews
   parameters:
     - in: query
      name: itemId
      schema:
        type: string
      required: true
      description: Get details of a specific item and its
          ↪ reviews
   responses:
     '200':
      description: Item details and reviews
      content:
        application/json:
          schema:
            type: object
            properties:
              id:
                type: string
              name:
                type: string
              reviews:
                type: array
                items:
                  type: object
                  properties:
                    reviewId:
                      type: string
                    comment:
                      type: string
```

Two-node loop This pattern describes the case where two nodes are always called together but in interchanged order. This indicates that the operations are complementing each other. A change that can minimize the number of calls in this case is to make one endpoint generic.

For instance, the two endpoints:

`GET /items/beauty?query=olive+oil` and `GET /items/home?query=olive+oil`

allow both to get the products that have olive oil in both categories Home and Beauty. In that case, the client has to make two calls. To avoid this, the API can instead offer a parametric endpoint `GET /items?category={beauty, home}&query=olive+oil`.

IT-Change 3: Replace a set of endpoints with a single generic endpoint that accepts a parameter or a request body instead of a static path segment.

Risks: While this change might help reduce the number of calls in some cases, it could harm performance due to added dynamic filtering logic. For example, clients who need only one category might experience slower response times.

Listing 7.5. Before change

```
paths:
  /items/beauty:
    get:
      summary: Retrieve beauty products matching a query
      parameters:
        - in: query
          name: query
          schema:
            type: string
          required: true
          description: Search for beauty products by query term
      responses:
        '200':
          description: A list of beauty products
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    id:
                      type: string
                    name:
                      type: string
                    category:
                      type: string

  /items/home:
    get:
      summary: Retrieve home products matching a query
      parameters:
        - in: query
          name: query
          schema:
            type: string
          required: true
          description: Search for home products by query term
      responses:
        '200':
          description: A list of home products
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    id:
                      type: string
                    name:
                      type: string
                    category:
                      type: string
```

Listing 7.6. After change

```
paths:
  /items:
    get:
      summary: Retrieve products from specified categories
          ↪ matching a query
      parameters:
        - in: query
          name: category
          schema:
            type: array
            items:
              type: string
              enum:
                - beauty
                - home
            required: true
          style: form
          explode: false
          description: Categories to filter products (e.g., beauty,
              ↪  home)
        - in: query
          name: query
          schema:
            type: string
          required: true
          description: Search for products by query term
      responses:
        '200':
          description: A list of products across the specified
              ↪ categories
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    id:
                      type: string
                    name:
                      type: string
                    category:
                      type: string
```

Fork[2] This pattern is seen when two different endpoints are called after the same initial sequence of calls. The authors suggest that these two endpoints share similar purposes and might be merged.

If after calling **GET** `/items`, users call either **GET** `/details` or **GET** `/reportTables`, merge them into **GET** `/visualization?type={chart,table}`.

Merging **GET** `/items/id/price` and **GET** `/items/id/stock` into **GET** `/items/id?include={price,stock}`

IT-Change 4: Merging endpoints that are called after the same initial sequence into a unified endpoint with

[2]Note that for these next identified API changes I am not going to show examples of how this changes are reflected in OpenAPI.

parameters to differentiate results.

Risks: This might introduce complexity in handling diverse data (price and stock), and increase payload size, which can slow down responses for clients that only need one piece of information.

Inverted fork In this case, two different endpoints are always called before the same set of following endpoints. This pattern suggests that the two starting endpoints may serve the same purpose and can be combined.

IT-Change 5: Merging starting endpoints that lead to the same set of subsequent calls into a single endpoint with parameters to differentiate the initial resource. Example: If both /rooms and /labs lead to the same endpoints like /reservations, merge them into /spaces?type=room,lab.

Merging `/items/home` and `/items/beauty` into:

`GET /items?category={home,beauty}` before calling endpoints like `GET /items/id/details`.

Risks: The distinction between categories could be lost, making the API less clear. Clients may struggle with the more generic endpoint, especially if specific logic applies to certain categories.

Choices This pattern occurs when two alternative endpoints are called in place of each other but within the same context. It indicates that these endpoints may serve similar roles or be substitutes.

IT-Change 6: Merging alternative endpoints into a single one with a parameter to differentiate between options.

Merging `GET /items/id/discounts` and `GET /items/id/promotions` into:

```
GET /items/id/offers?type={discounts,promotions}
```

Risks: The merged endpoint could lead to confusion between discounts and promotions, and it might require more complex validation logic to differentiate between types of offers.

Feed-forward This pattern is identified when two sequences of calls exist, where one sequence follows the full chain of calls (e.g., A → B → X), and the other sequence skips an intermediate step (e.g., A → X). This suggests that the intermediate step (B) might be optional or supplementary. A change could be to consolidate the functionality of the intermediate endpoint into the first one, or to make the intermediate step optional via a parameter.

IT-Change 6: Consolidate optional intermediate steps into the main endpoint or make them optional through parameters. Example: If some users call /competences → competences_type → subjects while others skip the middle call, merge competences_type into /competences?includeType=true.

Merging /items → /items/id/reviews and /items directly into /items?includeReviews=true.

Risks: Including optional reviews by default might increase response size and lead to unnecessary complexity for clients who don't always need the reviews, potentially causing performance degradation.

The pattern mines by Koçi et al. is a valuable input that can help interpreting the motives behind a combination of API changes.

It is not clear if Koçi et al. have intentionally focused on detecting the changes that break backward compatibility. The API changes we identified from Koçi et al.'s work are breaking changes, since they require the API clients to adapt to be integrated with the new way the API operations are supposed to be called. In our case, we start first by detecting all the changes, then once detected we analyze their potential impacts on the clients.

7.3 Web API Changes in a Large Dataset of Real-World Web API Histories

7.3.1 Change extraction from OpenAPI histories

In our case, we analyze API change from explicit traces consisting of git histories of OpenAPI specifications.

In the previous section, I showed how can the API changes that other researchers have identified, using different techniques, be mapped to the OpenAPI definition of the API.

In the approach we followed we extracted those changes by studying the differences between consecutive commits on OpenAPI specification. We have done this analysis at two levels:

- At the commit level: we have extracted changes happening at the commit level because they are associated with other metadata such as the commit messages and the commit-er identity. The results of this extraction are used in this chapter to create a taxonomy of recurrent API changes.
- At the version change level: the commits level might be very fine-grained because sometimes the commit change is to revert previous changes and it does not have as goal to introduce new changes. The version change level means the commits when the API changes the version. Looking at that level will help to verify the consistency between the change's impact and the versioning of the new release. The extracted results are employed in Chapter 10 to analyze the alignment of the changes with the versioning practices being followed.

For both, we follow the same change extraction and analysis pipeline described in Figure 7.2. Using this approach based on detecting changes based on documentation, we are able to track all functional changes, however, our technique does not capture semantic changes, which often involve the meaning and intended behavior of APIs. These semantic changes can sometimes be reflected in the `description` fields of the OpenAPI documentation but detecting them requires either manual analysis, which is infeasible for large-scale datasets, or the use of heuristics, which introduces uncertainty. Instead, we prioritize having a broader view of detectable API changes to create a taxonomy of possible API changes, enabling a structured analysis.

7.3.2 Web API changes extraction pipeline

As shown in Figure 7.2, to extract the change at the commit level, for each of the API we go over each couple of subsequent commits and compute the differentials between the specifications, then process these differentials to extract the actual aggregated API changes that look like the example in Listing 7.8.

Listing 7.7. Example of extracted changes from subsequent commits

```
[
 {
  id: "new-optional-request-property",
  text: "added␣the␣new␣optional␣request␣property␣'short_description'",
  operation: "POST",
  path: "/v1/clubs",
  section: "paths"
 },
 {
  id: "new-optional-request-property",
  text: "added␣the␣new␣optional␣request␣property␣'short_description'",
  operation: "PUT",
  path: "/v1/clubs/{clubUUID}",
  section: "paths"
 }
]
```

As illustrated in Listing 7.8, the extracted changes are aggregated by their types, which serve as unique identifiers for the change objects. Each change entry includes detailed information about the specific change and its location within the API. To facilitate efficient querying and analysis, the list of changes is stored in MongoDB alongside the corresponding commit specification. In this way, we not only enrich the database with detailed evolution and change analysis data but also complement the metrics described in Chapter 3.

In the example of Listing 7.8, while the addition of the `short_description` field appears as a one schema update in the specification, we trace its impact across all paths and operations using that schema. This identifies every affected endpoint (e.g., `POST /v1/clubs`, `PUT /v1/clubs/{clubUUID}`), providing a more accurate measure

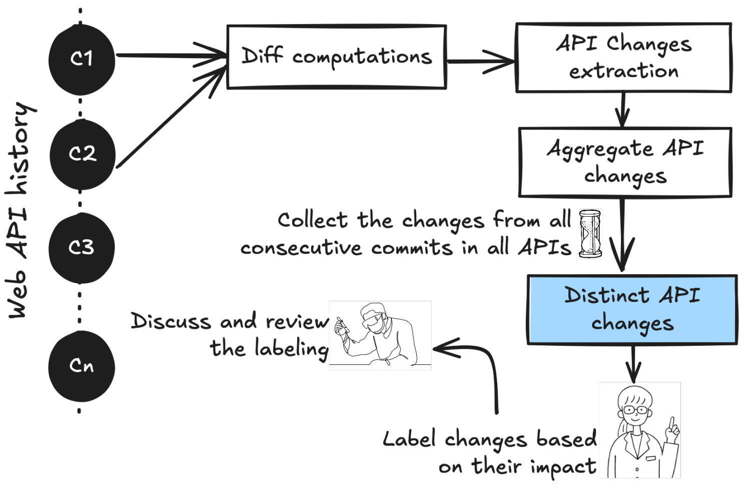

Web API history
C1
C2
C3
Cn
Diff computations
API Changes extraction
Aggregate API changes
Collect the changes from all consecutive commits in all APIs
Distinct API changes
Label changes based on their impact
Discuss and review the labeling

Figure 7.2. Changes extraction and classification pipeline

of the change's scope and its impact on the API. This approach highlights the cascading effects of schema changes on all related operations, ensuring a more accurate measure of their implications.

Once the whole extraction process is completed, and for all the APIs we have the list of changes that occurred between each subsequent commit, we collect the list of `ids` of all the captured types of changes, we manually label them to ● Breaking (BC), ● Non-breaking (NBC), or ● Undecidable (UC) depending on their perceived impact. This first labeling step is done by one person, then reviewed and discussed by another reviewers person.

Breaking Changes (**BC**) can disrupt existing client implementations and require clients to adapt to these changes. These changes include modifying existing properties or types (like changing types to enums), adding required to request body properties, required request parameters or deleting paths or properties from response payloads, and changing attributes values to nullable.

Non-Breaking Changes (**NBC**) do not require existing clients to change their implementations. These are generally additive changes such as adding new optional properties requests bodies or supporting additional media types, and changing types where backward compatibility is maintained (like integer to number).

Undecidable Changes (**UC**) refer to those modifications whose impact on the client varies depending on the client's or backend's tolerance level to dealing with unexpected message payloads [37]. For example, when removing authentication or authorization headers, old clients may not break if the security tokens they still send to a tolerant API are ignored. Likewise, properties that are added to API responses may break strict clients that reject unknown data elements. Undecidable changes, cannot be statically classified into breaking or non-breaking without making further assumptions about the client and the API tolerance level.

The same process is followed for both when extracting the changes between subsequent commits or subsequent commits where the version changes.

7.3.3 Data selection and preparation

Since we are interested in analyzing histories, we applied our analysis to specifications that had at least ten commits modifying them. We haven't applied any filtering criteria based on the repository characteristics because often the specification is present in a repository that is only dedicated to manage the specification. For instance, while OpenAI API is one the most popular APIs nowadays, its repository has, at the time of the writing of this thesis, only

1.2k stars ★ and 121 watchers 👁, which does not reflect the actual popularity of the API. Same thing for the GitHub API itself. While, in other cases, the specification is managed under the same repository containing the code such as the openapi-generator repository, which has more than 20k ★ but the specifications it contains are only test ones, and they have less than 10 commits changing them. The only criteria we take into account, beside being changes in at least ten commits, is to be a **valid specification**. This left us with 943 546 commits of APIs with at least 10 commits.

A **valid specification** is an OpenAPI document that conforms to the OpenAPI Specifications (3.0 or 3.1) or Swagger 2.0 standards and is parsable by standard OpenAPI tools. This means the document must adhere to the structural and syntactical rules defined by its respective version (e.g., OpenAPI 3.x or Swagger 2.0) and be free of critical errors that prevent it from being processed.

To validate our specifications, we utilized a Node.js validator and parser, specifically the @readme/openapi-parser. One key validity criterion we enforce is the ability to resolve all references within the specification. During the mining phase, if a specification referred to external resources (e.g., external schemas or components), these were downloaded and incorporated directly into the specification. As a result, by the time of processing, all references are internal, ensuring that the specification can be fully resolved and validated without dependency on external resources. This guarantees that each specification is self-contained and parsable according to OpenAPI standards.

7.3.4 Dynamics in Web API histories

We performed analysis, shown in Table 7.1, and found that there exist a commit that includes up to 3120 changes affecting different parts of an API. The changes in this API (Extrato Jurídico[3]) affected data model parts that were used in 159 operations, which explains the high number of detected changes.

For APIs with more than 10 commits, most commits do not include any changes. On average, about 9.5 changes happen in a single commit, showing that when changes occur, they are often grouped together in one commit. Interestingly, the time between commits with no changes, documentation changes, and API changes is nearly the same.

👉 One key finding is that in APIs with at least 10 commits, there is always at least one commit that introduces API changes. There exist also cases, where every single commit sometimes includes changes, which shows how active some APIs can be.

Table 7.1. Statistical Analysis of Web API Changes

Category	Max	Min	Mean	Median	Stdev
Changes Per Commit	3180	0	9.50	0.0	62.54
Distinct Changes Per Commit	43	0	0.91	0.0	1.82
Changes Per API	140267	0	417.55	51	3176.28
Distinct Changes Per API	106	0	10.65	8	11.70
Commits With Changes Per API	1449	1	43.93	21	94.27
Commits Without Changes Per API	34	0	1.02	1	0.54
Time to Next Commit (Days)	2329	0	15.66	0	62.37
Time to Next Commit With Changes (Days)	2329	0	15.02	0	59.40

[3] https://extratojuridico.com/

7.3.5 Types of changes and their impact on clients integration

Types of changes in Web API histories.

From the calculated diffs, we identified a total of **205 distinct changes**. Among these, eight were aimed at extending API functionalities (e.g., adding a path or operation), 31 focused on refining API functionalities by imposing restrictions on request parameters, and 87 involved restructuring the data model through modifications to request bodies. Additionally, 66 changes were made to response bodies to remove data that was no longer required.

We group the detected changes into four clusters depending on the API feature they affect:

- Endpoint-related changes: include additions, removals.
- Request-related changes: include modifications to parameters, properties, and body elements, including additions, removals, type alterations, constraints like length or patterns, and changes in optionality, default values, and discriminators.
- Response-related changes: include modifications to response properties, statuses, and body elements, involving additions, removals, type alterations, constraints on length or patterns, optionality, default values, discriminator mappings, and changes in read/write-only behavior.
- Security-related changes: involves applying or removing a new security measure either locally or globally over all the API endpoints. In addition, there can be find tuning of authorizations through changes of scopes of the applied security measure.

Table 7.2. List of API change types classified by their impact. Count represents the total presence of a change across all diffs, while Operations, APIs, Commits, and Classification provide additional insights.

API Feature	API Element	Change	APIs	Commits	Count
Security	Local security scope	🔴 Removed	130	7571	207920
	Local security scope	🔴 Added	116	7568	207723
	Local security	⚪ Added	663	1243	11058
	Local security	⚪ Removed	393	742	6677
	Global security scope	🔴 Added	12	15	504
	Global security scope	🔴 Removed	13	15	30
	Global security	🔴 Added	497	1007	1153
	Global security	⚪ Removed	344	804	929
	8 distinct changes	**Change types**	🔴 **5**	🟢 **0**	⚪ **3**
Endpoints	Endpoint	🟢 Added	3169	31247	138686
	Path	🔴 Removed without deprecation	2518	13800	96010
	Endpoint	🔴 Removed without deprecation	1060	2356	3455
	Path	🔴 Removed with deprecation	132	235	754
	Endpoint	🔴 Removed with deprecation	9	12	27
	5 distinct changes	**Change types**	🔴 **4**	🟢 **1**	⚪ **0**
Request parameters	Optional query parameter	🟢 Added	1763	7257	26013
	Parameter	⚪ Removed	1636	5536	21716
	Required query parameter	🔴 Added	912	2318	7158
	Parameter	🔴 type changed	435	698	2618
	In-path parameter	🔴 Added	340	636	2377
	Parameter	🟢 enum value added	217	639	4341
	Parameter	🔴 enum value removed	164	333	2252
	Parameter	🟢 Became optional	298	596	1471
	Parameter	🔴 Became required	326	541	1464
	Optional query parameter at path level	🟢 Added	107	236	1058
	Parameter	🟢 type generalized	206	332	1228

Continued on next page

Table 7.2. Top Changes by Count with Classifications (Continued)

API Feature	API Element	Change	APIs	Commits	Count
	Parameter	🔴 Pattern changed	42	100	683
	Parameter	🔴 Min length increased	37	56	554
	Parameter	🟢 Min length increased	37	56	554
	Parameter	🔴 Max length set	39	57	304
	Parameter	🔴 Min set	65	70	305
	Parameter	🔴 Pattern added	83	97	281
	Parameter	🔴 Max set	40	43	158
	Parameter	🟢 Default value added	98	152	597
	Parameter	🔴 Default value changed	81	192	599
	Parameter	🔴 Default value removed	47	78	229
	Required query parameter at path level	🔴 Added	40	49	450
	Parameter	🔴 Pattern removed	19	23	140
	Parameter	🟢 Min length decreased	19	26	93
	Parameter	🔴 Min increased	16	21	55
	Parameter	🟢 Min decreased	12	12	51
	Parameter	🟢 Max increased	20	30	51
	Parameter	🔴 Max decreased	10	10	15
	Parameter	🔴 Max length decreased	8	13	32
	Parameter	🟢 Max length increased	11	18	106
	Parameter	🔴 Min items increased	7	17	35
	Parameter	🟢 Min items decreased	7	15	25
	31 distinct changes	**Change types**	🔴 **15**	🟢 **13**	⚪ **3**
Request Body	Optional property	🟢 Added	1374	7423	56114
	Property	⚪ Removed	1324	5396	49498
	Property	🔴 type changed	833	2612	27472
	Property	🟢 Enum value added	548	1683	20639
	Property	🔴 Enum value removed	391	842	10458
	Required property	🔴 Added	672	1945	6466
	Property	🔴 type specialized	387	986	5821
	Property	🟢 Became optional	514	1106	5384
	Property	🔴 Became required	553	1067	5251
	Request body	🟢 Media type added	321	667	3142
	Property	🔴 type changed	242	662	3133
	Request body	🔴 Media type removed	306	629	2938
	Property	🟢 Became nullable	158	334	1845
	Property	🔴 Max length set	176	264	1724
	Property	🔴 Min length increased	132	195	1656
	Property	🔴 One of added	166	422	1535
	Request body	🔴 type changed	368	611	1325
	Property	🟢 Default value added	164	271	1121
	Property	🔴 Became not nullable	82	161	1121
	Property	🔴 Pattern changed	97	223	1088
	Request body	🔴 All of added	143	459	1027
	Property	🔴 Became enum	197	294	1025
	Required request body	🔴 Added	220	269	980
	Property	⚪ type generalized	112	181	909
	All of property	🔴 Added	97	227	907
	Property	🔴 One of removed	107	304	906
	Request body	🔴 All of removed	125	422	888
	Property	🔴 Default value removed	109	177	831
	Optional property	🔴 Became read only	59	107	813
	Property	🔴 Pattern added	116	174	798
	Property	🟢 Max length increased	43	74	725
	Optional request body	🟢 Added	176	267	606
	Request body	🔴 One of added	85	355	550
	Property	🟢 Min length decreased	84	131	540
	Request body	🔴 One of removed	82	323	499
	Request body	🔴 Became required	137	157	475
	All of property	🔴 Removed	76	179	445
	Property	🔴 Default value changed	74	182	444
	Property	🔴 Min set	89	107	407
	Any of property	🔴 Added	38	130	345

Continued on next page

Table 7.2. Top Changes by Count with Classifications (Continued)

API Feature	API Element	Change	APIs	Commits	Count
	Request body	🟢 Became optional	68	93	306
	Property	🟢 Pattern removed	68	98	259
	Property	🔴 Max length decreased	26	46	253
	Property	🔴 Max set	56	81	242
	Any of property	🔴 Removed	41	111	240
	Required property	🔴 Became read only	22	46	173
	Property	⚪ type generalized	20	30	166
	Request body	🔴 Any of added	15	73	144
	Required property	⚪ became not read only	12	28	144
	Required property	🟢 Added with default value	33	70	144
	Request body	🔴 Any of removed	15	67	141
	Property	🟢 Became required with default	37	57	140
	Property	🔴 Discriminator added	24	36	122
	Read only property	🔴 enum value removed	8	22	118
	Property	🔴 Min items increased	31	43	112
	Request body	⚪ type generalized	32	43	82
	Property	🔴 Discriminator removed	20	34	59
	Property	🔴 Discriminator mapping added	13	31	53
	Property	🟢 Max increased	10	14	50
	Optional property	🟢 Became write only	8	10	49
	Property	🔴 Min increased	12	13	46
	Request body	🔴 Discriminator mapping changed	3	7	38
	Request body	🔴 Default value changed	6	27	38
	Request body	🔴 Discriminator added	16	18	33
	Request body	🟢 Default value added	8	26	33
	Property	🔴 Discriminator mapping deleted	11	17	31
	Request body	🔴 Discriminator removed	10	12	27
	Property	🔴 Discriminator mapping changed	2	3	26
	Required property	🟢 Became write only	8	9	24
	Request body	🔴 Max length set	6	6	23
	Property	🟢 Min decreased	12	13	21
	Request body	🔴 Discriminator mapping added	7	18	21
	Request body	🔴 Default value removed	7	12	18
	Request body	🔴 enum value removed	4	4	17
	Request body	🔴 Min items increased	8	16	17
	Required property	🟢 Became not write only	5	5	15
	Request body	🔴 Became nullable	6	6	14
	Request body	🟢 Max length increased	1	1	12
	Request body	🔴 Became not nullable	3	3	12
	Property	🔴 Max decreased	9	9	12
	Request body	🔴 Min length increased	4	4	10
	Property	⚪ x extensible enum value removed	1	1	8
	Request body	🔴 Discriminator mapping deleted	4	8	8
	Read only property	🔴 Min increased	1	2	4
	Request body	🔴 Became enum	2	2	2
	Request body	🟢 Min length decreased	2	2	2
	Request body	🔴 Discriminator property name changed	1	1	1
	87 distinct changes	**Change types**	🔴 **60**	🟢 **21**	⚪ **6**
Response body	Optional property	🟢 Removed	1494	6755	246016
	Response optional property	⚪ added	1749	11267	214875
	Property	🔴 type changed	1185	4292	102348
	Property	⚪ Enum value added	658	2425	57977
	Required property	⚪ added	916	3916	45445
	Required property	🔴 Removed	881	3138	44174
	Property	🔴 Enum value removed	515	1238	35891
	Property	🔴 All of added	374	1101	28340
	Property	🟢 Became required	662	1663	19458
	Response media type	🟢 Added	766	1479	16687
	Property	🔴 All of removed	320	842	15324
	Property	🔴 Became optional	500	1061	14379
	Property	🔴 One of added	217	741	10043
	Property	🔴 Became nullable	241	599	9973

Continued on next page

Table 7.2. Top Changes by Count with Classifications (Continued)

API Feature	API Element	Change	APIs	Commits	Count
	Response media type	🔴 Removed	426	788	9536
	Response body	🔴 type changed	702	1449	6767
	Property	🔴 Pattern changed	79	178	5168
	Property	🔴 One of removed	173	604	3935
	All of	🔴 Added	246	854	3673
	All of	🔴 Removed	202	742	3173
	Property	🔴 Default value removed	94	147	2925
	Property	🟢 Default value added	134	197	2536
	Property	🔴 Any of added	68	243	2081
	Property	🟢 Pattern added	128	203	1983
	Property	⚪ Max length increased	65	144	1840
	Property	⚪ Max length unset	85	151	1787
	One Of	🔴 Added	132	503	1775
	Optional property	🟢 Became read only	91	176	1762
	One Of	🔴 Removed	112	470	1131
	Property	🔴 Min length decreased	81	134	1010
	Property	🔴 Any of removed	65	184	756
	Required property	🟢 Became read only	36	87	679
	Optional property	🟢 Became not read only	46	96	604
	Property	🔴 Default value changed	49	133	576
	Property	🔴 Pattern removed	69	93	491
	Property	🔴 Discriminator added	34	53	480
	Response mediatype	🔴 Enum value removed	10	18	454
	Required property	🟢 Became not read only	29	58	453
	Property	🔴 Discriminator mapping changed	6	9	175
	Property	🔴 Min items decreased	35	46	154
	Property	🔴 Discriminator mapping added	20	40	152
	Property	🔴 Discriminator removed	27	38	121
	Discriminator mapping	🔴 Added	6	12	85
	Property	🔴 Max increased	10	12	64
	Any of	🔴 Added	13	39	62
	Any of	🔴 Removed	13	36	61
	Property	🔴 Discriminator mapping deleted	12	23	61
	Discriminator mapping	🔴 Deleted	4	6	60
	Property	🔴 Min decreased	17	18	52
	Required write only property	🟢 Removed	10	10	42
	Discriminator	🔴 Added	17	24	34
	Discriminator	🔴 removed	15	23	27
	Required write only property	🟢 Added	4	4	21
	Response body	⚪ Max length increased	1	1	18
	Optional property	🔴 Became write only	3	3	17
	Response body	🔴 Min items decreased	10	10	14
	Response body	🔴 Became nullable	4	8	12
	Default value	🔴 Removed	2	2	7
	Required property	🟢 Became not write only	2	2	7
	Required property	🔴 Became write only	2	2	7
	Response body	⚪ Max length unset	5	6	7
	Response body	🔴 Min length decreased	5	6	7
	Optional property	🟢 Became not write only	2	2	6
	Default value	🔴 Changed	4	4	4
	Default value	🟢 Added	1	1	2
	Discriminator mapping	🔴 Changed	1	1	1
	66 distinct changes	**Change types**	🔴 **45**	🟢 **14**	⚪ **7**
Response code	Non success status	🟢 Added	1403	4027	25843
	Non success status	🔴 Removed	881	2146	11443
	Success status	🟢 Added	889	1749	3615
	Success status	🔴 Removed	686	1374	2851
	4 distinct changes	**Change types**	🔴 **2**	🟢 **2**	⚪ **0**
Request/response headers	Optional response header	🔴 Removed	120	263	2232
	Response header	🔴 Became optional	10	10	634
	Required response header	🔴 Removed	7	12	222

Continued on next page

Table 7.2. Top Changes by Count with Classifications (Continued)

API Feature	API Element	Change	APIs	Commits	Count
	Request header property	🔴 Became enum	3	4	18
	4 distinct changes	**Change types**	🔴 **4**	🟢 **0**	⚪ **0**
Totals across API elements	**205 distinct changes**	**Change types**	🔴 **135**	🟢 **51**	⚪ **19**
	785 803 Total changes	**Changes classified**	🔴 **404 551**	🟢 **278 813**	⚪ **11 932**

In Table 7.2, The table provides a detailed classification of 205 distinct API change types across various elements, categorized into 🔴 (Breaking), 🟢 (Non-Breaking), and ⚪ (Undecidable) based on their impact on backward compatibility. The "Count" column represents the total occurrences of each change, including repeated appearances within the same commit, capturing the full extent of API modifications impacts. Among the change types, 🔴 Breaking Changes dominate with 135 distinct types (65.85%), reflecting its significant role in altering API behavior and often requiring client adjustments. 🟢 Non-breaking ones accounts for 51 types (24.88%), focusing on extending or enhancing API functionality without disrupting existing clients, while ⚪ Undecidable changes make up 19 types (9.27%), representing cases with unclear compatibility impact.

The most frequent changes overall are concentrated in response body modifications, security adjustments, and endpoint restructuring. Notably, the removal of optional response properties (🟢 Removed) is the most common, occurring 246 016 times, representing 43.02% of all response body changes. Similarly, the addition of optional response properties (⚪ Added) appears 214 875 times, or 37.54%, highlighting the dynamic evolution of API response structures. In the security domain, local security scope changes dominate, with 207 920 instances of removal (🔴 Removed) and 207 723 instances of addition (🔴 Added), affecting over 7 568 commits, emphasizing the frequent adjustments in API access control mechanisms. Endpoint modifications also play a major role, with 138 686 new endpoints added (🟢 Added), making up 58.76% of all endpoint-related changes, and 96 010 paths removed without deprecation (🔴 Removed), showcasing the significant structural evolution of APIs.

Request and response parameters are another area of high activity, with 26 013 optional query parameters added (🟢 Added), comprising 31.53% of all parameter-related changes, and 21 716 parameters removed (⚪ Removed), at 26.33%. Additionally, property type changes are frequent, with 102 348 instances in response bodies (🔴 Type Changed) and 2 618 in request parameters (🔴 Type Changed), reflecting ongoing refinement of data models to improve functionality and clarity.

Looking at the "Count" column in Table 7.2, the most impacting changes are those affecting the request body. This is because modifications to the data model often propagate across multiple operations, as the same part of the data model is reused in different contexts. As a result, several associated changes are recorded together, making the total count of request body changes appear significantly high. However, when examining the number of impacted APIs by a specific type of change, it becomes clear that changes at the endpoint level are the most frequent. Notably, the addition of new endpoints stands out as the most common and distinct type of change.

☛ Breaking changes are approximately 2.44 times more frequent than non-breaking changes. However, the most recurrent change is non-breaking. Additionally, changes in response body data models are the ones with the most impact on the API backward compatibility, the addition of endpoints is the most common type of change shared between APIs.

Impacted HTTP methods for each type of API changes.

In the visualization of Figure 7.3, we show the impact of the 50 most frequent types of changes on HTTP methods, focusing on both the number of affected methods and their proportional distribution. Changes related to response

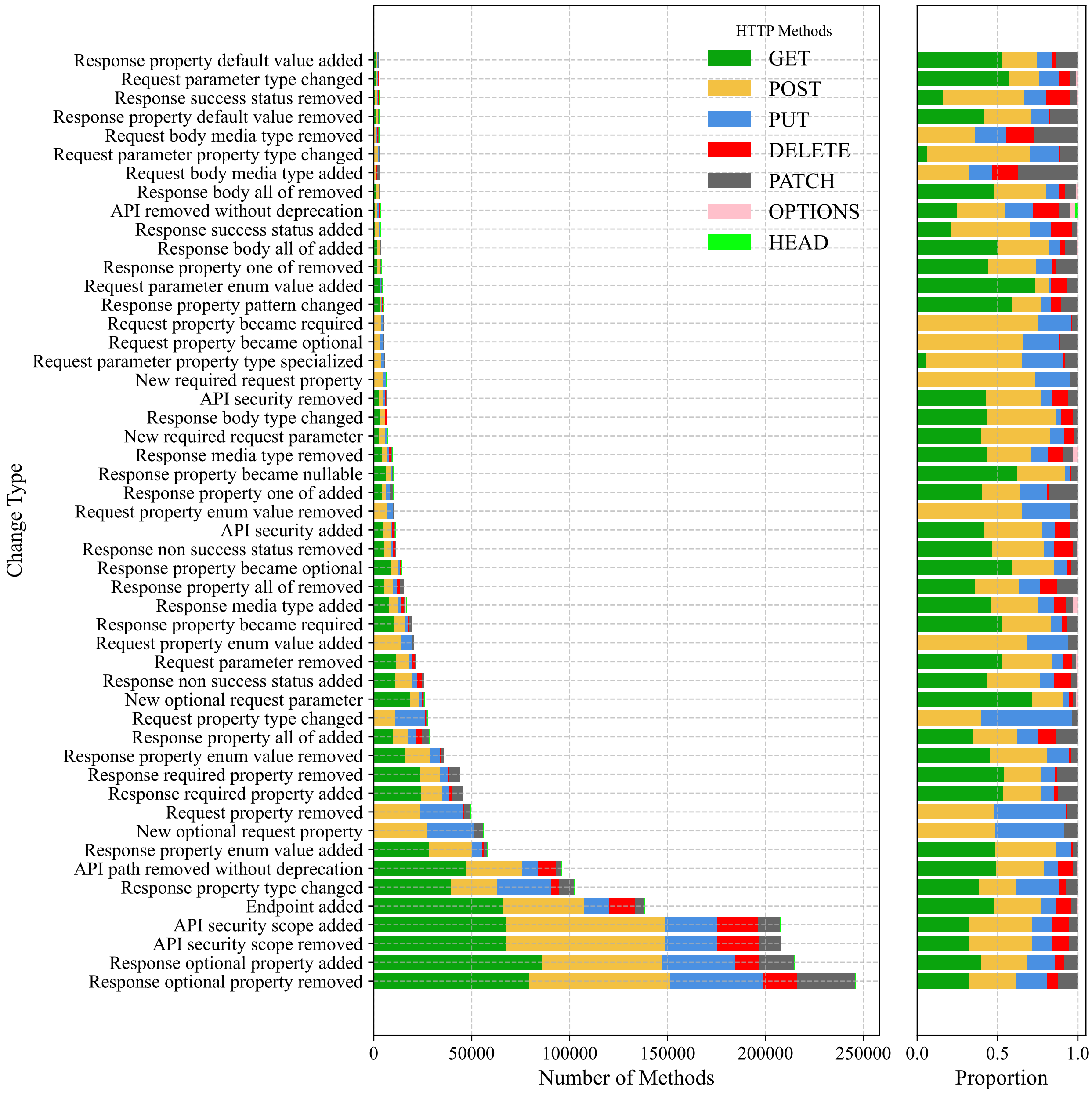


Figure 7.3. Affected HTTP methods in the top 50 detected web API changes

properties, such as adding or removing optional properties, modifying property types, or adjusting default values, dominate across all methods, particularly affecting **GET** and **POST**, which are the most widely used HTTP methods. PUT and **DELETE** are also notably impacted but to a lesser extent, while less common methods like **PATCH** **OPTIONS** and **HEAD** re minimally affected. Security-related changes, such as the addition or removal of API security measures or scope adjustments, highlight the evolving nature of API security, alongside functional changes like endpoint additions and path removals, which significantly reshape the API structure. The proportional analysis shows the widespread influence of these changes across HTTP methods, emphasizing the centrality of response and security modifications in driving the impact on API operations.

Web API change types co-occurences.

Changes in commits do not always come individually. 63 817 of the commit changing the API (80 963) had more than one change where 31 929 of them combined more than one change type.

It is worth highlighting that when using OpenAPI some of the co-occurring changes are for documentation purposes. For instance, since security in OpenAPI is documented using reusable components, the changes in Listing 7.8 are likely to co-occur when a new security requirement is introduced and the corresponding component is not already defined. OpenAPI's design encourages the use of a `components.securitySchemes` section to centralize the definition of security mechanisms, such as the Basic authentication scheme in this example. The first change, `api security component added`, adds this reusable definition, enabling it to be referenced by multiple endpoints or operations across the API. Without this component, applying the security scheme directly to an endpoint would violate the OpenAPI structure, as the security schemes must first be defined before they can be used. Consequently, the second change, `api-security-added`, which applies the Basic scheme to the `/login endpoint`, depends on the presence of the component. This sequential relationship makes the co-occurrence of these changes highly likely. While the addition of the security component is a documentation-level update, the application of the scheme to the endpoint is a breaking API change, as it alters the security requirements, potentially impacting clients. This example highlights how OpenAPI's modular documentation structure enforces a dependency between the definition and application of security schemes, driving such co-occurring changes when introducing new security measures. In our analysis of co-occurring changes we only take into account the change that are not only for documentation purpose. Meaning that if a commit has only the changes in Listing 7.8, we count it only as one change of one type.

Listing 7.8. Example of expected co-occurent changes

```
{
    "id": "api-security-component-added",
    "text": "the␣component␣security␣scheme␣'Basic'␣was␣added",
    "level": 1,
    "section": "components"
  },
  {
    "id": "api-security-added",
    "text": "the␣endpoint␣scheme␣security␣'Basic'␣was␣added␣to␣the␣API",
    "level": 1,
    "operation": "POST",
    "operationId": "login",
    "path": "/login",
    "section": "paths"
  }
```

The heatmap in Figure 7.4 provides a visual representation of the co-occurrence relationships between the top 50 co-occurring types of changes in APIs. Each cell indicates the relative frequency of two specific changes appearing together in the same commit or API update, with darker shades representing higher frequencies of co-occurrence. This pattern reveals clusters of changes that commonly occur together, suggesting possible dependencies or design trends in API evolution. For instance, changes related to response properties (e.g., "Response required property added" – the first in the x-axis– and "New required request property") are seen to co-occur frequently, likely due to

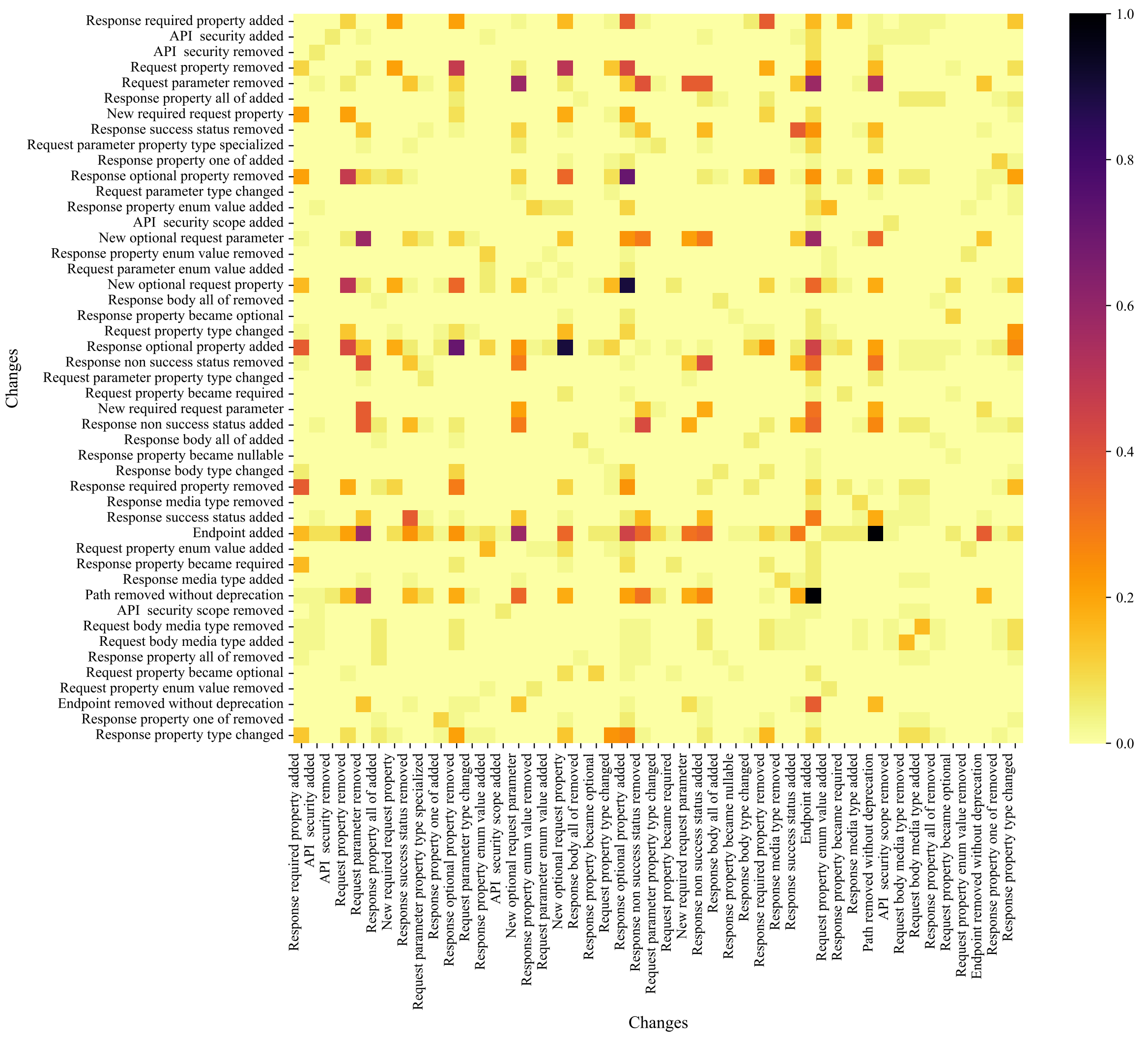


Figure 7.4. Heatmap illustrating the co-occurrence of changes in APIs, where darker cells indicate more frequent co-occurrence between specific changes

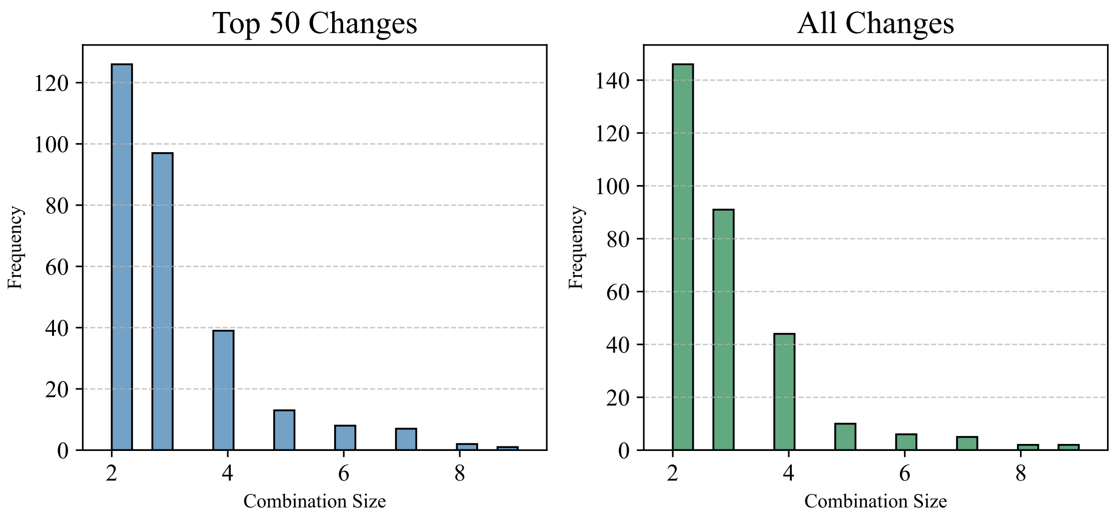


Figure 7.5. Distribution of combination sizes is presented in two plots. The left plot displays the sizes of combinations that include only the top 50 detected changes, while the right plot illustrates the distribution of combination sizes involving all detected changes.

updates in the datamodel where a property of a specific entity is used in both a request body and a response. Other frequent co-occurrences involving changes like the addition of new endpoints “Endpoint added” and “Path removed without deprecation” could reflect intentional API restructuring, such as deprecating and replacing outdated functionality, improving modularity, or implementing version updates. co-occurring changes such as “Response required property added” and “Response required property removed” might be due to renaming of properties.

Certain changes, such as adding an endpoint, often appear in combinations with a diverse range of other changes, reflecting their broad impact on the API structure. In contrast, changes like “Response success status removed” frequently co-occur with changes like “Response success status added”, indicating fine-tuning or adjustments to response codes rather than broader structural modifications. For instance, an API might simultaneously add a new success status while deprecating an older one to align with updated requirements or simplify response handling. This highlights how specific types of changes tend to cluster based on their functional purpose.

While the most common change combinations involve only two distinct types of change (26 105 out of 31 929 more than one change), Figure 7.5 shows that there exists commits that combine up to nine different types of changes all at once. We distinguish in the left plot only the combinations involving the 50 most occurring changes to see wether their presence impacts the distribution of the sizes of combinations.

Figure 7.8 that the most frequent changes tend to be also frequently used in combinations with other types of changes. The plots illustrate the relationship between the frequency of changes and each of the distinct combinations involving the change (left) and the number of commits containing those combinations (right). The left plot shows a moderate correlation (r = 0.63), indicating that more frequent changes are generally involved in more distinct combinations, though with some variability. The right plot reveals a stronger correlation (r = 0.81), demonstrating that changes with higher frequencies are consistently present in a larger number of commits containing those combinations. The narrower confidence interval in the right plot reflects greater certainty in this trend, suggesting that frequent changes play a key role in shaping the overall evolution of APIs through repeated inclusion in combinations of updates.

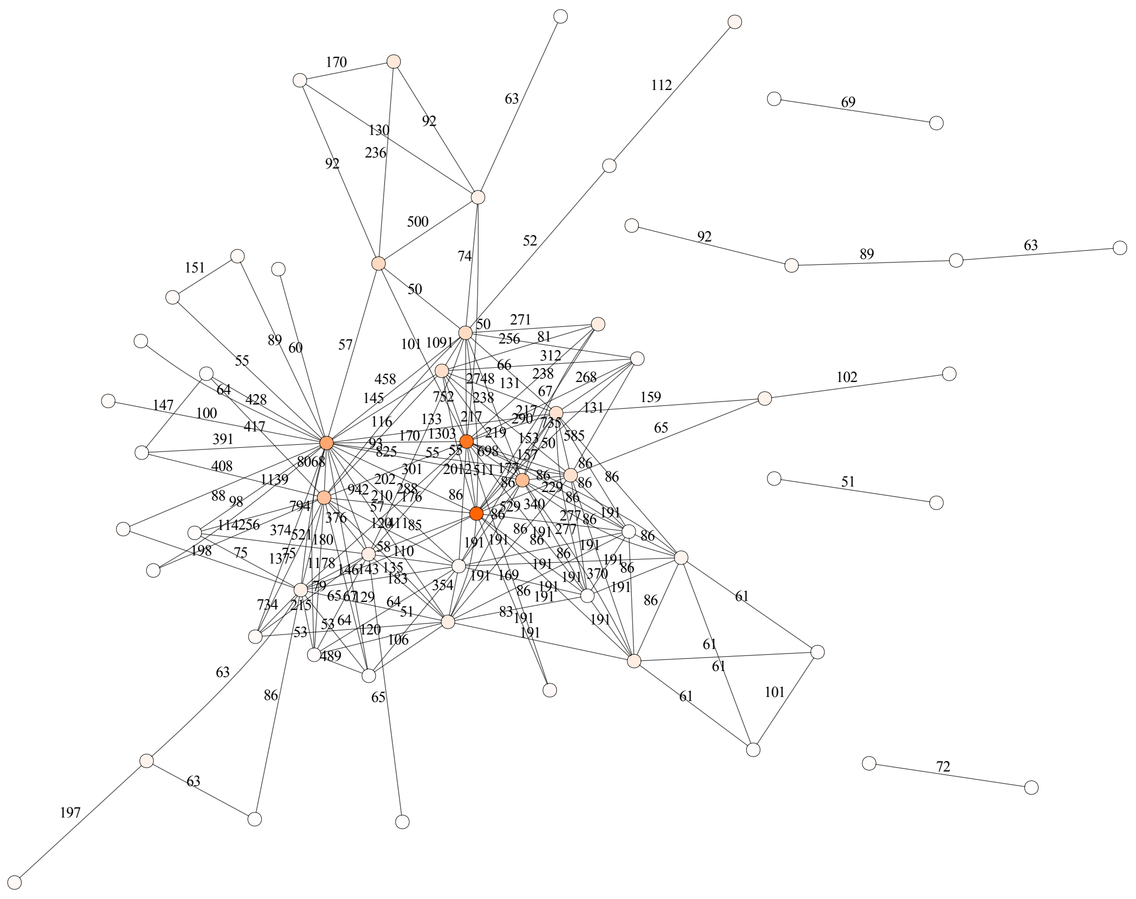


Figure 7.6. Graph visualization of Change Combinations with Normalized Coloring by Frequencies. For visibility, the graph shows only the changes that have been combined together with others at least 50 times

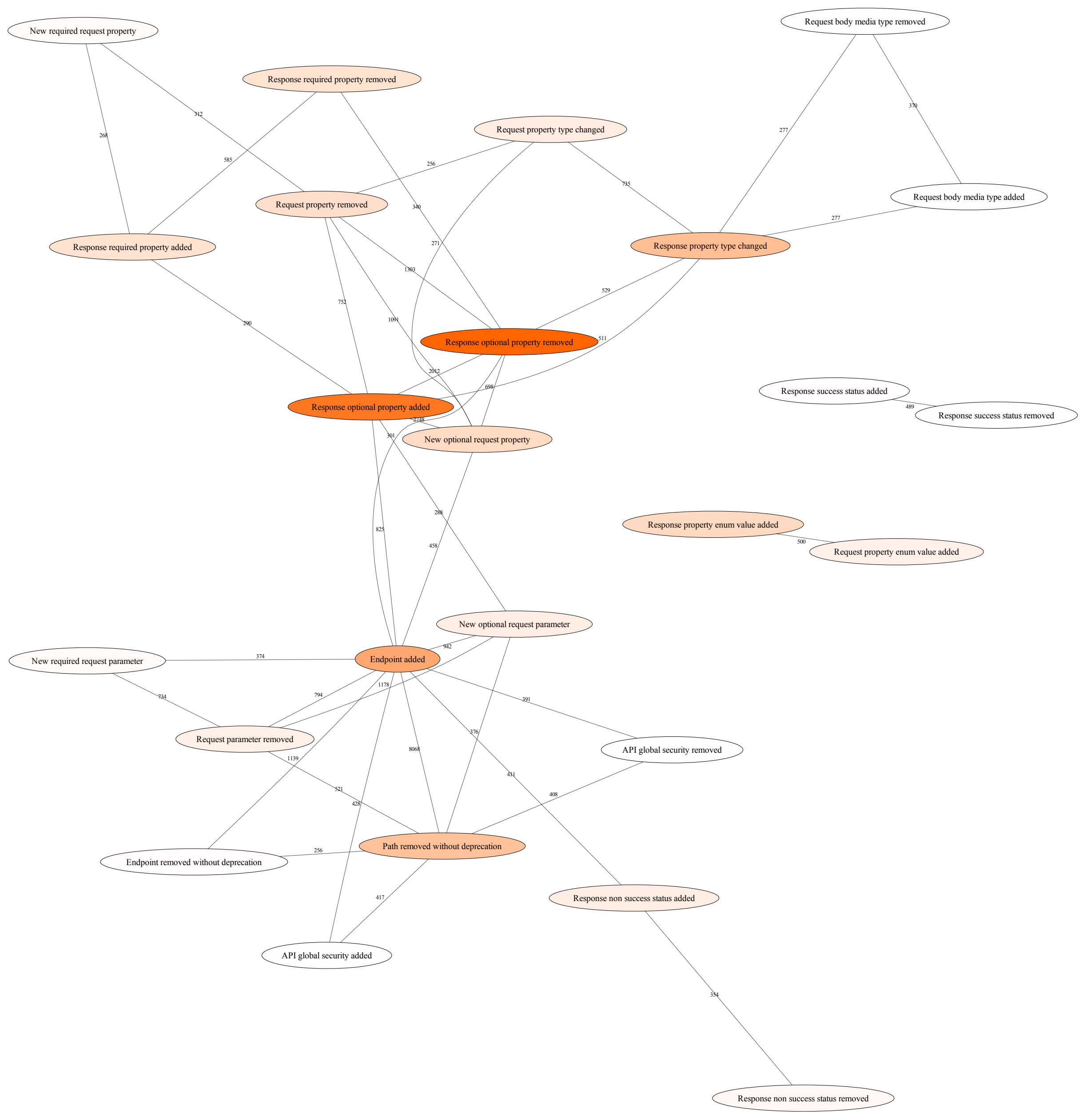

New required request property
Request body media type removed
Response required property removed
Request property type changed
Request body media type added
Request property removed
Response required property added
Response property type changed
Response optional property removed
Response success status added
Response optional property added
Response success status removed
New optional request property
Response property enum value added
Request property enum value added
New optional request parameter
New required request parameter
Endpoint added
Request parameter removed
API global security removed
Endpoint removed without deprecation
Path removed without deprecation
Response non success status added
API global security added
Response non success status removed
268
312
585
256
340
277
370
735
277
271
1303
529
752
1091
511
2012
698
489
301
500
288
825
458
942
374
1178
734
794
391
8068
376
1139
521
428
411
408
256
417
354

Figure 7.7. Graph visualization of Change Combinations with Normalized Coloring by Frequencies. The graph shows only the changes that have been combined together data at least 250 times

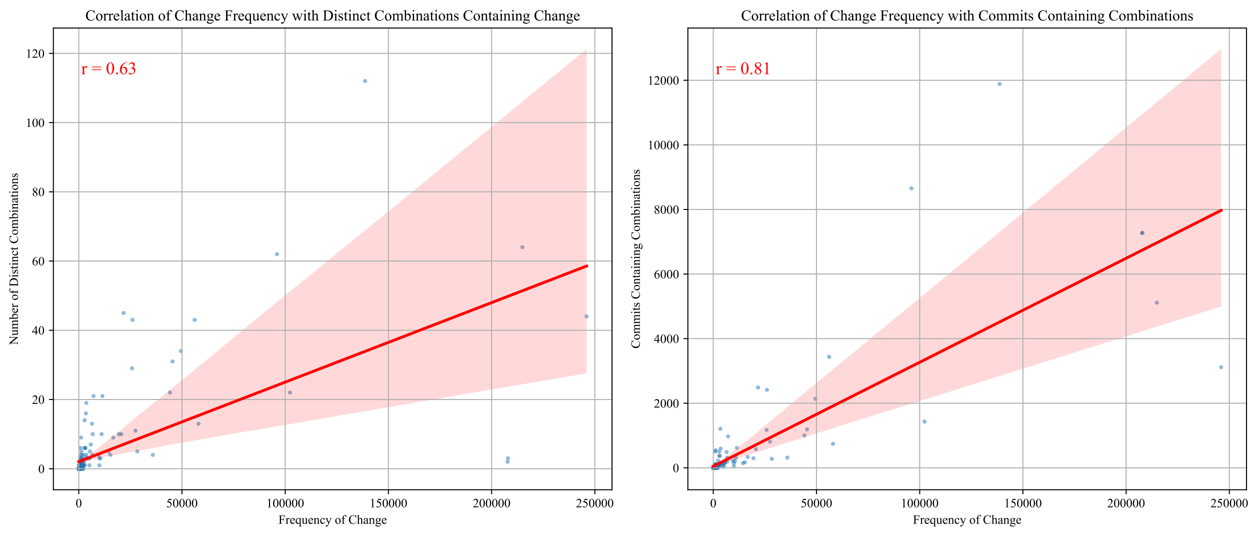


Figure 7.8. Correlation between the frequency of changes and its presence in combinations. The red area represents the confidence interval around the regression line. It indicates the range within which the true regression line is expected to fall with a confidence of 95%.

In Figure 7.6 we draw the network showing the occurrences of changes happening at least 50 times, where the nodes stand for changes and they are colored by their occurrences, and the edges represent a co-occurrence. The weights on the edge are the exact numbers of of occurrences. The network confirms the suggestion made earlier. The most frequent changes (darker orange) seem to be more connected to more diverse types of changes.

In Figure 7.7, we show a simpler network showing only occurrences that happen at least 250 time. For instance, some changes such as adding an enum value to a response property and adding it to a request property at the same time is more frequent to happen only together rather than being combined with another type of change.

☛ The correlation analysis between a change frequency and appearance combined with other changes shows that frequent changes are more likely to be involved in distinct combinations and appear consistently across a larger number of commits in combination with other changes.

In Table 7.3 we list examples of groups of changes that happened together in at least 10 APIs.

Table 7.3. Top 10 Largest Combinations Appearing in at Least 10 APIs, Sorted in Descending Order by Size

Combination	Size	APIs	Commits	Min	Max	Median	Mean
Path removed without deprecation	9	10	10	74	179	179.0	168.50
Endpoint added							
New optional request parameter							
Request parameter property type specialized							
Request parameter removed							
Response non success status added							
Response non success status removed							
Response success status added							
Response success status removed							
Path removed without deprecation	8	10	28	21	179	169.0	165.32
Endpoint added							
New optional request parameter							
Request parameter removed							
Response non success status added							
Response non success status removed							
Response success status added							
Response success status removed							
Request body media type added	8	10	19	992	992	992.0	992.00
Request body media type removed							
Response non success status added							
Response optional property removed							
Response property all of added							
Response property one of removed							
Response property type changed							
Response required property removed							
Request body media type added	7	89	172	293	2019	876.0	1085.65
Request body media type removed							
Response non success status added							
Response optional property removed							
Response property all of added							
Response property type changed							
Response required property removed							
Request body media type added	7	86	86	273	2338	864.0	1032.67
Request body media type removed							
Response non success status removed							
Response optional property added							
Response property all of removed							
Response property type changed							
Response required property added							
Path removed without deprecation	7	11	20	19	57	56.0	53.70
Endpoint added							
New optional request parameter							
New required request parameter							
Request parameter removed							
Response non success status added							
Response non success status removed							

Continued on next page

Table 7.3. Top Combinations by Size and Commits (continued)

Combination	Size	APIs	Commits	Min	Max	Median	Mean
New optional request property New required request property Request property removed Response optional property added Response optional property removed Response required property added Response required property removed	7	15	17	18	295	57.0	72.41
New optional request property Request property removed Request property type changed Response optional property added Response optional property removed Response property type changed Response required property removed	7	12	12	28	1486	37.5	174.42
Path removed without deprecation Endpoint added New optional request parameter Request parameter removed Response non success status added Response success status added Response success status removed	7	10	12	15	149	144.0	133.67
Path removed without deprecation Endpoint added New optional request parameter Request parameter removed Response non success status removed Response success status added Response success status removed	7	10	12	14	155	150.0	139.08

In Summary

Introducing a breaking change is 2.44 times more likely to happen than introducing a non breaking change. However, the most frequent one is non-breaking. Most of the changes occur at the level of request and response parameters and schemas. This is due to the wide variability in tuning their attributes and applying restrictions. While a single commit may include several changes of this type, at the API level, the most common changes are observed at the endpoint level, primarily involving the addition of new endpoints.

While constructing a taxonomy of fine-grained changes occurring in Web API histories, understanding the driving factors behind these changes is equally important. Our classification focuses on categorizing changes to assess their potential impact on integration during evolution. Examining the combinations of changes provides a broader perspective on their effects on API usability. For instance, co-occurring changes, such as the removal of an endpoint alongside the addition of a new one, might signify an intentional renaming of a path. This analysis is possible due to the provenance data persisted during our change extraction process, which meticulously tracks the exact location, path, and operations associated with each change, as shown in the examples of Listings 7.8.

In this chapter, I present only the changes extracted across commits. However, the same approach was used to extract changes occurring between version updates, enabling an analysis of the compatibility between API changes and the versioning strategy employed. The results of this analysis are discussed in Chapter 10.

Chapter 8

Web API Histories Visualization

Developers and API users often struggle to comprehend the entire history of an API and track its changes over time [52]. This lack of visibility can make it challenging to identify structural modifications and assess potential backward compatibility issues.

This chapter presents a concrete example of how the Web API change analysis can be exploited to provide a visual resource that developers can use to visually identify structural changes and potential backward compatibility issues. We suggest two visualization that make use of the changes extraction mechanism at the commit level and sunburst structure to capture historical information in a compact interactive artifact.

The visualizations, and evolution metrics reports, can be automatically generated using a publicly and open-source tool provided called APIcture.

8.1 Sunbursts of Web API histories

We use the sunburst visualization towards understanding the evolution of Web APIs, specifically concerning tracking the co-evolution of API structures and their versioning metadata. The main goal is to characterize and compare how different Web APIs evolve over large periods of time, in order to visually identify different API evolution patterns [98, 81, 156]. We introduce visualizations designed to distinguish which API elements have changed often, how such changes impact clients [93, 176, 70], and whether API developers consistently update versioning metadata to control client expectations about the impact of such changes [**?**]. The interactive visualizations and their scalability have been tested by using it to explore a large collection of 3, 271 API change histories mined from open source GitHub repositories. In this chapter we include a small sample, showing a rich diversity of Web API evolution histories reflected in our visualizations.

More in detail, the API VERSION CLOCK visualization shows when each API change happened, what is their impact on clients (e.g., classifying breaking vs. non-breaking changes), and their relationship with the API versioning metadata. The goal is to help API developers reflect on the pace of their API evolution and remind them to bump the versioning metadata to reflect the impact of changes on their API clients consistently. The API CHANGES visualization complements it by precisely representing which elements of an OpenAPI (OAS) description [118] have changed. The visualization can be applied both to study individual diffs, but also cumulatively over sets of changes up to the entire API history. The visualization highlights the unstable elements of an API description by showing how they have changed during a defined timeframe.

The visualization is supported by APICTURE, a CLI tool that offers the convenience of generating visualizations directly from git repositories that already contain an OAS specification. The tool can be embedded into DevOps build pipelines [18] to generate updated visualizations at every commit so that API developers can effortlessly track and understand the evolution of their Web APIs.

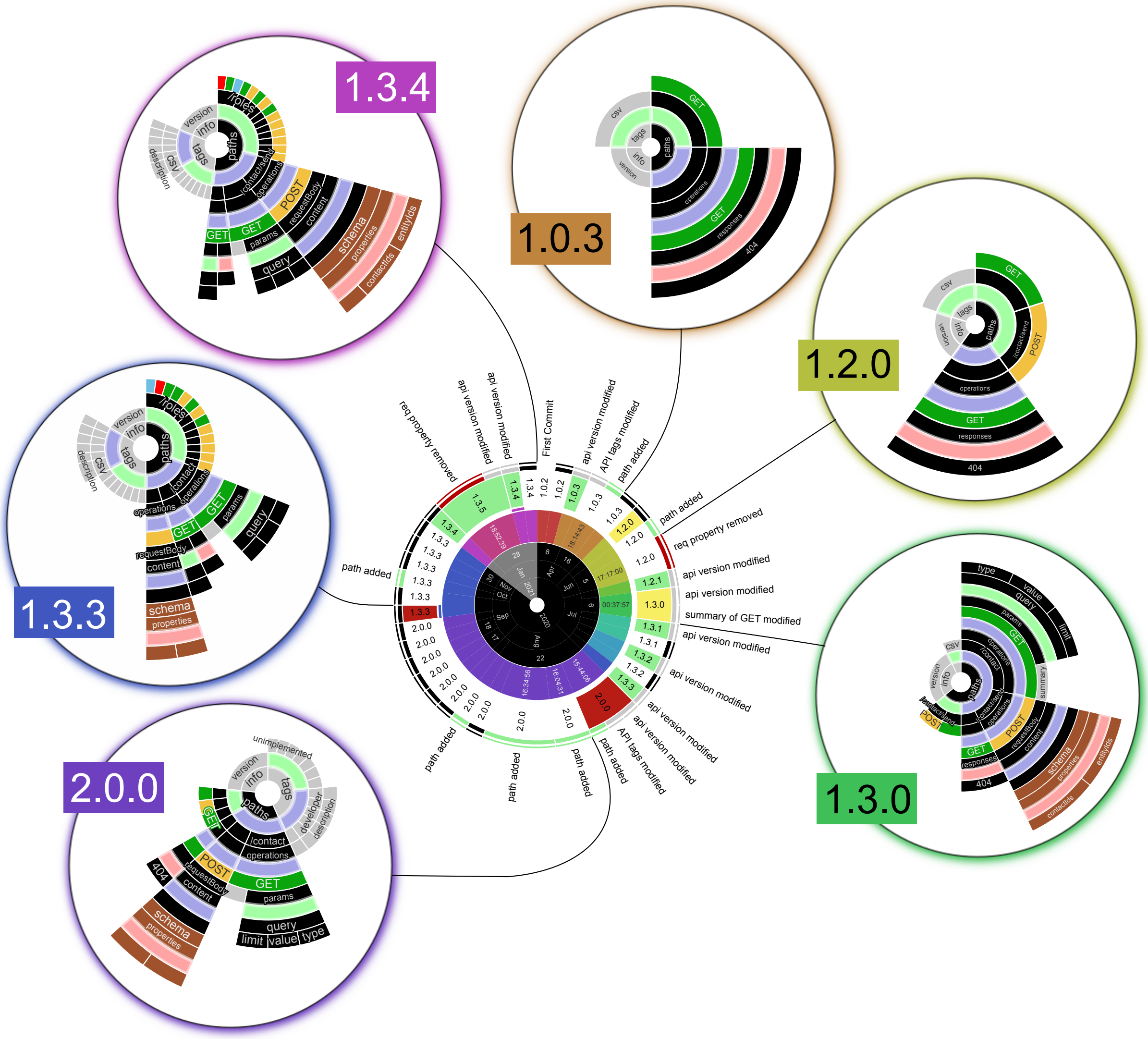


Figure 8.1. API Version Clock (center) and API Changes until a given version of the Bmore Responsive API. Legend explained in Figures 8.2 and 8.4.

This chapter makes the following contributions:

1) the API Version Clock visualization classifies all changes based on their impact on clients and captures them in chronological order while putting them in relationships with the different API versions.

2) the API Changes visualization provides a precise localization and analysis of modifications within the API description structure. This visualization aids in understanding the stability of specific API elements and fine-grained evolution patterns of the API design.

3) the integration of API Changes with API Version Clock to provide a holistic view of the API's evolution journey, supporting the analysis of versioning practices and the localization of breaking and non-breaking changes

on the API structure and data model.

4) a small gallery of API evolution visualizations, derived from real-world APIs that test the scalability of the visualization to increasingly larger histories (both in terms of versions and commits) and exhibit a variety of API evolution patterns.

5) the APICTURE tool, which automates the process of generating both visualizations, providing API developers and stakeholders with an interactive means to analyze, visualize and reflect on their API evolution and versioning strategies.

8.2 Use Case Scenarios and Example API

In this section we use a real-world API to introduce and explain the design of the two visualizations and how they are generated by leveraging the API's git historical record of changes. The Bmore Responsive API [7] is an emergency response and contact management API designed for monitoring and coordinating emergency responses to critical scenarios, such as managing the status and needs of local nursing homes during a global pandemic, identifying hospitals lacking power during natural disasters, and ensuring the safety of hikers in a national park during snowstorms.

The API history includes a total of 49 commits spanning from the first commit on *April 6, 2020* to the last commit on *February 28, 2022*. The API has been actively maintained and evolved over a period of *693 days*. Throughout its 693-day-long history, we found 13 distinct versions of the API (from `1.0.2` to `2.0.0`, later reverted back to `1.3.4`).

Developers are interested in reflecting on the history of the API to answer the following questions [104, 68, 51]:

- Q1: Did we correctly and consistently follow a semantic versioning [8] strategy?
- Q2: How often did we revert the API to a previous version?
- Q3: Did we follow a regular API maintenance and update cycle over time?
- Q4: Did we always ensure backwards compatibility of the introduced changes?
- Q5: Which are the stable and the unstable parts of the API structure and data model?
- Q6: Can we detect if our API followed a unique evolution path compared to other ones?

The interactive visualizations we propose are intended to help both API developers and developers of API clients to answer such questions.

In particular, the API VERSION CLOCK visualization uses the sunburst plot to provide a compact chronological view of the flow of changes over time. With it, developers can observe the progression of API versions. On the other hand, the API CHANGES visualization uses the sunburst hierarchy to offer a detailed representation of the accumulated changes that have occurred within a specific time frame. This visualization aids in localizing the specific areas of the API structure where the changes have occurred. Figure 8.1 exemplifies this integration, demonstrating how and when changes flow through specific time points in the history of the Bmore Responsive API. With the assistance of the API CHANGES visualization, it becomes apparent which parts of the API have been affected by these changes, enabling a more granular analysis of the API's evolution.

Both visualizations represent the evolution of an API from a different and complementary perspective. By default they display the entire API history, aggregating all changes into a single plot. If API designers are interested to observe which change happened when they can use the API VERSION CLOCK to select a specific commit so that the corresponding API CHANGES can show what API elements changed since the previous commit. Likewise, they can select to view all changes leading up to a certain version of the API, or all changes that happened between two

Algorithm 1: Web API repository analysis procedure

```
Input: repo_url
repo ← clone_repository(repo_url);
files ← find_OAS_specifications(repo);
if length(files) > 0 then
    user_selection ← select_file(files);
    if length(user_selection) = 1 then
        file ← user_selection[0];
        commits ← fetch_commits(file);
        foreach commit in commits do
            spec ← retrieve_file(commit, file);
            if is_valid(spec) then
                history_files ← fetch_history_files(commit, file);
                foreach consecutive f1, f2 in history_files do
                    diff ← compute_diff(f1, f2);
                    changes ← extract_changes(diff);
                    classification ← classify_changes(changes);
                    store_classification(commit, classification);
    build_sunburst_visualization();
else
    express_js_repo ← clone_express_js_repository(repo_url);
    history ← fetch_history(express_js_repo);
    foreach version in history do
        spec ← extract_OAS_specification(version);
        timestamps ← get_commit_timestamps(version);
        foreach timestamp in timestamps do
            commit ← find_commit(timestamp);
            spec_commit ← retrieve_file(commit, spec);
            if is_valid(spec_commit) then
                history_files ← fetch_history_files(commit, spec);
                foreach consecutive f1, f2 in history_files do
                    diff ← compute_diff(f1, f2);
                    changes ← extract_changes(diff);
                    classification ← classify_changes(changes);
                    store_classification(commit, classification);
    build_sunburst_visualization();
```

different releases. As mentioned above, each API modification represented in API CHANGES can be contextualized along the time dimension thanks to the API VERSION CLOCK, e.g., by highlighting the commit in which they occur.

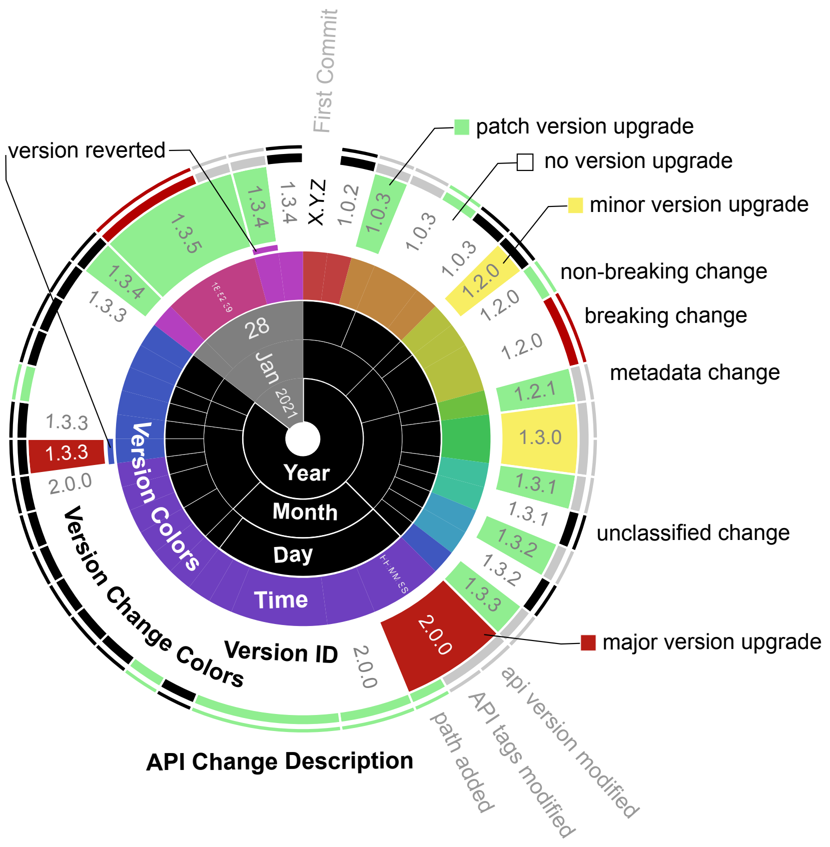


Figure 8.2. API Version Clock Design visualizing the Bmore Responsive API

8.3 API Version Clock Visualization

8.3.1 Visualization goal

This visualization serves as an evolutionary clock, providing a visual representation of the different types of changes that occur during a specific timeframe in an API's history. As depicted in Figure 8.2, in the *Time* ring – the fourth one from the center – each version upgrade is assigned a unique color, allowing for a clear depiction of the progression of version identifiers and the corresponding types of changes – shown in the outer rings – throughout the entire history of the API. Notably, for APIs that adopt the semantic versioning format, the visualization incorporates, in the *Version ID* ring, color coding to differentiate between major, minor, and patch-level upgrades.

The primary objective of API VERSION CLOCK is to assess the congruence between version identifier updates and the relative significance of breaking and non-breaking changes introduced in the API over time. Developers can leverage this visualization to gain insights into the adopted versioning strategy for a specific API and evaluate its adherence to the principles of semantic versioning.

In the case of the Bmore Responsive API, according to the API VERSION CLOCK visualization in Figure 8.2, semantic versioning is not properly adopted. Breaking changes were introduced however this was not reflected by the version identifiers. Moreover, the version identifier has been reverted twice during the API's history.

8.3.2 Building API Version Clock

Using the running example, we will demonstrate how the visualization aids in understanding the *chronological sequence of changes* within their *temporal context*, providing insight into the specific *API version* associated with each modification. Additionally, the visualization depicts the classification of these changes, allowing for a comparison to be made with the versioning strategy employed, thereby facilitating an assessment of the evolution and consistency of the API over time.

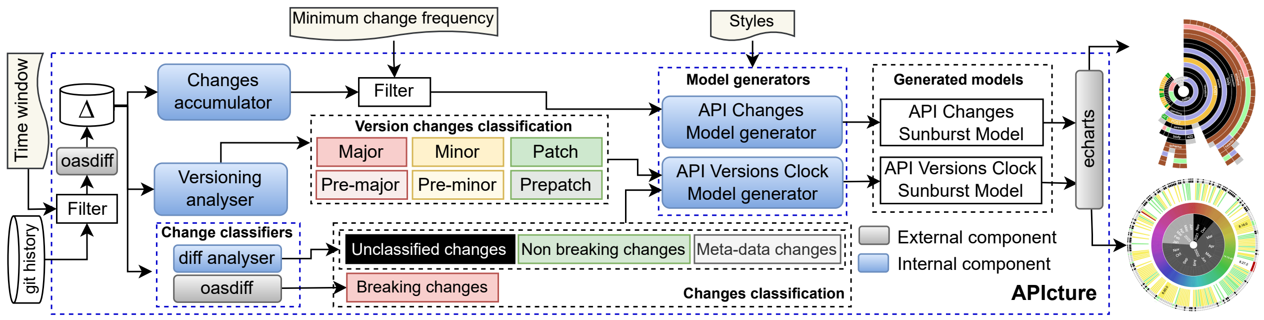


Figure 8.3. API Version Clock and API Changes analysis and rendering pipeline

As illustrated in Figure 8.3, the construction of the visualization involves retrieving all the git commits that made modifications to the OAS description of the API. We then compute the differences between each pair of consecutive commits, allowing us to identify the specific changes made during each commit. For extracting the changes we rely on *oasdiff* [115], an open-source command-line tool and Go package that compares two OAS descriptions.

The extracted differential data serves as input for both the *Versioning analyzer* and the *Change classifiers*, which generate classification results regarding version changes and the types of changes (breaking, non-breaking, meta-data, unclassified). These outputs, along with the necessary visual elements described in Figure 8.2, are utilized by the *API Version Clock model generator*, which then constructs the sunburst model. Finally, the generated model is rendered using ECharts [92].

The *Change classifiers* consists of two components: *oasdiff* [115], the external tool responsible for detecting breaking changes, and *diff analyzer* a tool developed as part of our work, which helps classify changes as non-breaking or unclassified.

Breaking changes are modifications made to an API that disrupt existing functionalities and result in backward compatibility issues with the latest deployed version [177]. These changes are identified and extracted from the differential data using again *oasdiff*. Conversely, non-breaking changes refer to modifications that do not introduce incompatibilities with existing functionality or the ability of clients to interact with the API. Unclassified changes, in the current version of the tool, are non-breaking changes which we could not precisely determine which parts of the API they affect.

8.3.3 Visualization Structure

In Figure 8.2, we illustrate the structure of the obtained API VERSION CLOCK visualization:

• *Localizing change in its temporal context:* The commits timestamps are mapped to the core rings of the sunburst visualization, starting from the year, month, day, and time of each commit. These rings provide a temporal context

Table 8.1. Color-coding for changes in the API Version Clock

Version Identifier Changes		**API Changes**	
⬜	The version identifier did not change.	🟥	Breaking change.
🟩	The patch version counter was increased.	🟩	Non-breaking change.
🟨	The minor version counter was increased.	⬜	Metadata change.
🟥	The major version counter was increased.	⬛	Unclassified change.

for the visualization, allowing users to observe the chronological order of the commits in a clockwise direction. They also allow users to hierarchically filter along the time axis by selecting to visualize only commits of a specific year, month, or day.

• *Localizing version change in its temporal context:* Since the evolutionary analysis of our study relies on the git commit history, the fourth ring of the version change visualization plays a crucial role in indicating the exact timestamp of each commit. This ring is color-coded based on the API version associated with that particular timestamp, allowing for a clear visual representation of version changes over time (Figure 8.2). A unique color is associated with each version identifier, currently based on a simple uniform mapping to the HUE component of HSL color values.

• *Discerning types of version change:* While the inner core of the visualization is dedicated to the time of each commit, the next layer represents the version identifier of the API at that time, extracted from the standard OAS metadata. While the fourth ring highlights the version change, in the case where semantic versioning is used, we distinguish in the *Version ID* ring the types of version change by assigning a specific color for each of the 🟥 major, 🟨 minor, and 🟩 patch releases.

With our analysis on version identifier changes [142], we could detect the presence of commits where the version was reverted to a previous identifier. To highlight such backward evolution steps. An example of this case is happening in Bmore responsive API, where the version was reverted from `2.0.0` back to `1.3.0` again in one of the commits in late September 2020. As can be seen from Figure 8.2, we added an extra thin ring sandwiched between the colored API version ring and the following one. The color of the ring is the same as the version to which the revert happened. This ring will remain empty for APIs with a monotonic version identifier evolution and highlight backward versioning steps otherwise.

• *Detecting API backward incompatibility:* Within the context of API elements, changes primarily pertain to modifications in the structure of endpoints, request/response schemas, and security/authentication mechanisms. These alterations directly impact the functional aspects of the API, potentially introducing *breaking changes* or enhancements to its capabilities. On the other hand, changes in the metadata predominantly involve updates to descriptive attributes that provide contextual information about the API. These include modifications to the API title, version number, server URL, and details about the API provider. Such changes are typically non-breaking in nature and focus on improving the clarity, documentation, or administrative aspects of the API [80]. Within each version segment, the sunburst chart displays on the outer rings, how many changes were detected by comparing the current commit against the previous commit. These changes are depicted first by distinguishing the breaking changes from the non-breaking ones and then – in the outer ring – by further indicating the presence and the number of specific types of changes.

• *Assessing Semantic Versioning Compliance:* To facilitate the assessment of compliance with semantic versioning practices, we intentionally utilize the same color scheme to represent both version identifier changes and the classification of changes as breaking or non-breaking. This design choice allows for easy visual identification of whether the API evolution aligns with correct semantic versioning principles. Commit nodes associated with breaking changes should correspond to major version upgrades (🟥), while non-breaking changes should be observed alongside patch-level version upgrades (🟩). By examining the placement and distribution of these node types, we

Listing 8.1. Tree model extracted from the diffs Δ1 (between first and second commit in Bmore Responsive API history) and Δ2 (between second and third commit)

```
name: root
value: 3
children:
  - name: paths
    children:
      - name: added
        children:
          - name: "/csv/{model_type}"
            value: 1
            children:
              - name: GET
                value: 1
 - name: info
   children:
    - name: version
      value: 1
 - name: tags
   children:
    - name: added
      value: 1
      children:
        - name: csv
          value: 1
```

expect users to readily detect instances where changes are not in accordance with the expected versioning rules.

8.3.4 API Version Clock Interactive Features

The visualization employs interactive ECharts features for dynamic exploration of the sunburst plot structure at various levels through *zoom in and out*. The *zoom* option aids in examining changes within specific timestamps or periods. Tooltips are essential, showing the change count for a slice. For time rings, the tooltip displays how many changes occurred at a given time.

8.4 API Changes Visualization

8.4.1 Visualization goal

The purpose of this visualization is to provide a clear understanding of the location of the changes occurring in the API structure, its data model as well as related metadata information. By visually representing the 🟥 deletions, 🟩 additions, and 🟦 modifications happening at the level of ⬜ metadata, ⬛ structural and 🟫 data model elements, it becomes easier to *localize* the frequently occurring changes and comprehend the overall stability of the API design (Figure 8.4).

One important aspect to highlight is that the changes visualization does not indicate when each change occurred. This deliberate omission contributes to the scalability of the visualization, as it allows for the accumulated changes across multiple API versions and commits within a specific timeframe to be captured in one visualization. This approach simplifies the visualization process and enables a more efficient analysis, particularly when dealing with APIs with long histories or many fine-grained granular changes applied to them.

Table 8.2. Color-coding for the API Changes visualization

Type of modification		**API Elements**		**HTTP methods**					
🟩	API element added	⬛	API structure element	🟩	GET	🟨	POST	🟦	PUT
🟥	API element deleted	🟫	API data model element	🟥	DELETE	▩	PATCH	🟩	HEAD
🟪	API element modified	⬜	API metadata element	☐	OPTIONS				

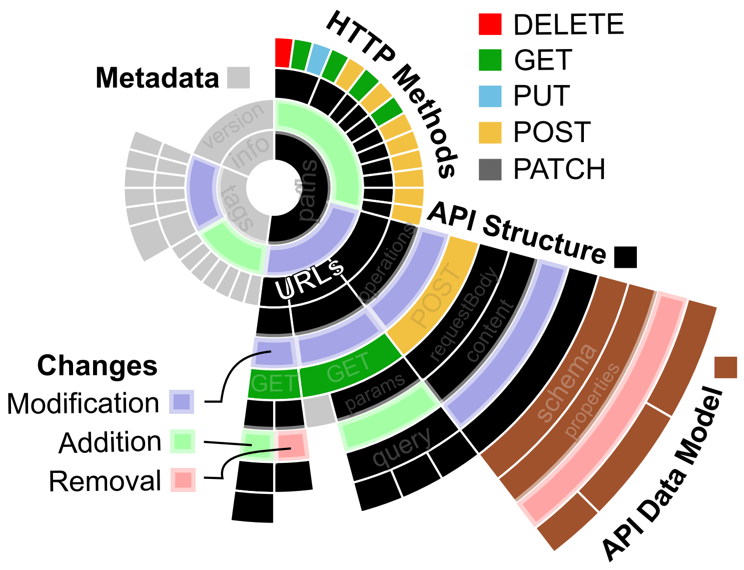


Figure 8.4. API Changes Design applied to the BMore Responsive API

8.4.2 Building API Changes

To build the visualization, we use the same differentials extracted for API Version Clock (See Figure 8.3) and transform them into a sunburst model so that it can be rendered using ECharts [92]. The structure of the sunburst tree reflects the OpenAPI specification structure. However, we only include OpenAPI elements which change at least once. As we accumulate all changes from multiple commits, we keep track of many times each element change, which is reflected in the relative ring sector angle. The elements represented in the sunburst plot are colored as defined in Figure 8.4.

In the API Changes visualization, instead of explicitly displaying individual changes like in the API Version Clock visualization, we employ an abstraction that focuses on measuring the frequency of different types of changes.

APIcture provides users with CLI options that offers the ability to customize the API history timeframe of interest. Specifically, in the context of API Changes visualization, APIcture allows users to filter out changes occurring less frequently than a defined minimum threshold. These are represented as *Filters* in Figure 8.3. Note that the *Time window Filter* can also be used to customize the history time frame for API Version Clock.

8.4.3 Visualization structure

We tailored the sunburst visualization to characterize the nature and also represent the magnitude of the changes occurring throughout the API lifespan and identify the unstable API elements that have undergone more frequent modifications.

• *API changes localization:* The nature of the changes shown in the visualization is contingent upon whether they occur within the *API structure elements*, *API datamodel* or the *metadata* elements, encompassing description fields, API title, API version, server URL, and API provider information. By distinguishing between changes in each of the later elements, the visualization enables a more comprehensive understanding of the different facets of changes occurring within the API evolution. This distinction allows developers and stakeholders to assess the amount of changes impacting both the functional behavior of the API and its associated contextual information.

By presenting the changes in a way mimicking a dereferenced version of the original OAS specification tree structure, it becomes easier to discern the exact areas or elements of an API that have undergone most alterations over time. The goal is to draw attention to the API elements affected by changes during its history. The wider the ring sector angular extent, the more frequently the corresponding API element changes. Different change actions affecting the API elements are highlighted using distinct colors (Figure 8.4): 🟥 Deleted, 🟩 Added, 🟦 Modified. These colored sectors refer to the type of changes applied to the element representing in their parent sector, the one found immediately above towards the center of the sunburst. These light-colored action sectors serve as a visual marker, drawing attention to the specific location within the API structure where the modification has taken place.

However, it should be noted that for non-object elements, such as the metadata element, and certain data types (e.g., enumeration), the only discernible alteration that can be detected is when the value itself changes. In these cases, there are no deeper levels or nested API elements to highlight, as the modification is confined to the value itself.

The diagram illustrated in Figure 8.4 can be read as follows. The initial structure-related components encountered when traversing the API structure from the root of the OAS description are the paths. The paths can undergo the tree types of changes 🟥 Deletion of a path, 🟩 Addition of a new path, 🟦 Modification of a specific path. The proportion of each type of change is shown in the next level, followed by a ring of sectors representing the actual paths being affected by the change. This is followed by the specific set of path elements which can be either: parameters, operations or metadata elements (summary and descriptions), in addition to extension elements (elements that start with an `x-` in an OAS specification).

In the case of modifications occurring at the path level, the subsequent rings within the visualization highlight the nested API elements that are affected by the modification.

The precise location of the modification is indicated by the appearance of a new action ring. This action ring serves as a visual marker, drawing attention to the specific location within the API structure where the modification has taken place. This applies on all the API structure-related components of an object type.

• *Quantifying change granularity:* Change granularity can be assessed by examining the *level* (depth of the *rings*) associated with a specific API element. For instance, if a specific endpoint is added (i.e., addition for a path), it represents a *less fine-grained change* as compared to the addition of a specific query parameter. This can be seen in API CHANGES of Bmore Responsive API in Figures 8.1 and 8.4.

Similarly, the addition of a new property to the schema of a response object for a specific operation signifies a *more fine-grained change* when contrasted with the addition of a parameter.

8.4.4 API Changes Interactive Features

Similarly to the API VERSION CLOCK visualization, the API CHANGES visualization capitalizes on the interactive capabilities provided by the ECharts sunburst. However, in this case, zooming helps to focus on the changes happening within specific API elements. The user can click on the API element to focus on, in order to expand all the subsequent rings and have a more detailed view to identify the nested API elements and the changes unfolding within them.

This capability can be exemplified by zooming in on the modifications occurring in the paths of the Bmore Responsive API.

Interactive tooltips provide insights into the frequency of specific element occurrences within a change, while also accommodating the display of labels identifying elements. To optimize readability, element labels are concealed when the available angle is insufficient to exhibit them without overlap. Furthermore, using APICTURE the generation of API CHANGES can be tailored by specifying a time frame and a minimum frequency value, granting users control over which changes are visualized based on their significance. And can be exported as a PNG, SVG, or an interactive HTML format.

8.5 API Evolution Gallery

In this section, we present five API evolution examples (Figures 8.5–8.9) growing from 24 commits up to 144 commits. They were selected out of a dataset of 3271 APIs, as they present different characteristics in terms of their evolution dynamics, their use of semantic versioning, the reached level of maturity, and their co-evolution with different repository artifacts. In the captions, we report the size of their evolution history (number of commits, versions, and duration in days) and the number of GitHub stars for the corresponding repository, which were also considered during the example selection process. The visualization gallery is obtained from a snapshot of the corresponding git repositories taken on 26 June 2023. It can be explored in: `https://souhailas.github.io/VISSOFT2023/`.

8.5.1 SunRocks API Evolution

The SunRocks's API VERSION CLOCK visualization provides a chronological depiction of the changes occurring in the SunRocks API from version `1.0.0` to version `4.0.0`, spanning over a period of more than 5 years (Figure 8.5 left). The majority of breaking changes took place within the first year, coinciding with two major version upgrades. No minor version increments were observed, and only a single patch version upgrade occurred (`3.0.1`), accompanied by a metadata change.

In all major version upgrades, there were always some unclassified or breaking changes, indicating that each upgrade involved modifications with a potential impact on the API backward compatibility. Upon closer examination through the API CHANGES Visualization (Figure 8.5 right), it becomes evident that the changes primarily manifested at the parameter level of POST methods of the `/claims` and `/sales` paths, with no significant alterations reaching the data model. This observation suggests that the developers focused on refining the API structure without requiring extensive modifications to the corresponding data representation.

8.5.2 xOpera REST API Evolution

Unlike the SunRocks API, the versioning strategy employed in the xOpera API (Figure 8.6 left) demonstrates a tendency towards minor version upgrades and patch upgrades. Interestingly, these version upgrades were consistently free of breaking changes. The largest evolutionary step happened on February 12th, 2021, when 15 paths were removed without deprecation and 17 paths were added. This change was applied to the version `2.0.0` without any immediate impact on the version identifier, which was changed 1 month later to `2.1.0`.

The API CHANGES shows that all three types of structural changes occurred with the addition of 25 paths, the modification of 21 paths, and the removal of 17 paths. Additionally, 26 changes impacted the API description metadata. Unlike the SunRocks API, some changes impacted the API data model, e.g., the removal or addition of response schema properties.

8.5.3 IPFS Pinning Service API Evolution

The IPFS Pinning Service API (Figure 8.7) illustrates the early preview release [98] phase of an API, which after almost one year of development reaches version `1.0.0`.

Most of the API structure and data model appears to be in place since the beginning, as there are no additions/removals neither at the level of paths or methods, nor at the level of the schema elements, as can be seen from the API CHANGES visualization. 20 out of 22 modifications affected the GET and POST operations of `/pins` path. The `depth` query parameter of the GET operation was added and subsequently removed.

Most of the breaking changes are concentrated in the early commits during the first few days of the project, while the remaining commits (largely affecting the natural language documentation) are backward compatible or unclassified. While version identifiers were gradually and regularly upgraded during the pre-release phase, the version `1.0.0` identifier remains fixed, even with minor structural modifications being still applied months after the initial release.

8.5.4 Xero Projects API Evolution

The Xero Projects API has been selected out of 11 APIs documented in a repository still under active development at the time of writing. Most of the structural changes occurred in the first two months of a 3-year long history. We can see 9 commits, from version `2.8.2` until `2.8.4`, during which 13 paths were removed and 7 paths were added. These breaking changes resulted in a patch version upgrade, up to version `2.8.5`. This version was however reverted back to the `2.8.4` identifier which remained constant until all remaining changes were committed. The version identifier then started to grow all the way to `2.38.0`, resulting in approx 50% of the API CHANGES, due to the co-evolution of this API with the others in the same repository: whenever one API description changes, developers bump the versions of all API descriptions in the same repository.

8.5.5 OpenFairDB API Evolution

The OpenFairDB API Evolution (Figure 8.9) is extracted from a repository in which both the API documentation and the backend implementation are found. Thus, developers will often use patch version upgrades without any API changes to track changes to the underlying backend implementation code. During the summer of 2020, a number of breaking changes were however introduced while keeping the same version `0.8.20` identifier. Moreover, commits of version upgrades never included structural changes.

Overall, there were more paths additions than deletions. And, most of the structural changes at the level of the paths are modifications. The most modified path is the `/search` endpoint (affecting both its response schema and query parameters), while the `/search/duplicates` path was added and later removed from the API.

8.6 Discussion

8.6.1 API Version Clock

One limitation of this visualization is that it may not scale well when dealing with longer histories of an API. The representation of individual commits and their associated changes can become overwhelming and cluttered, making it challenging to extract meaningful insights from the non-interactive visualization. The interactive version of the visualization can be helpful to zoom into specific sections of the timeline. Alternatively, an icicle plot [169] layout could be used so that the long history of APIs with a large number of commits can be displayed in a scrollable viewport.

To address this limitation and improve scalability, an abstraction technique can be applied to aggregate the changes over longer periods of time in the case of APIs with many frequent commits or long history. For example, all commits leading to a specific version change can be aggregated, considering that developers may first commit changes to the artifact and only update the versioning metadata in a separate commit. This would not fundamentally change the ring structure of the visualization, which would simply result in a less granular commit timestamp ring, where each sector would account for all changes occurring during a certain API version.

The version color on the fourth ring could better reflect semantic versioning. As opposed to uniformly assigning a distinct color to each version identifier, the mapping could use color shade variations for patch versions, similar colors for minor upgrades and different colors for each major release.

Furthermore, a prior investigation into web API versioning practices [142] unveiled that semantic versioning, following the `MAJOR.MINOR.PATCH` format, represents merely the most frequent one out of the 55 diverse API versioning formats. While the current version of the versioning analyzer already detects pre-release tags, it could be further extended to support a broader range of commonly adopted formats, such as calendar-based versioning. In this particular case, the same visualization would show the consistency between the commit timestamp and the version timestamp.

As APIs occasionally change title during their evolution, we plan to enhance the visualization to show during which timeframe each API title was in use. Likewise, it may be useful to filter the commits used to build the visualization based on the specific value given to the API title.

8.6.2 API Changes

One of the key design decisions of the API CHANGES visualization is the lack of precise time information for each individual change. While this enhances scalability by aggregating changes across multiple API versions, it can hinder the ability to analyze the chronological order of changes and understand their possible causal relationships. Given the atemporal nature of the API CHANGES visualization, it is not possible to perform such analysis with it. To find out when a certain change occurred and which version was affected, the commit corresponding to the selected change can be highlighted in the twin API VERSION CLOCK visualization.

The visualization's color scheme can be further enhanced to distinguish other frequently changing API features such as media types, security schemes, or other protocol-specific elements (e.g., response status codes). It would also be possible to apply a color layer to distinguish which changes did break compatibility with clients and which ones did not.

8.7 APIcture: Tool support

The creation of the visualization presented in this chapter is supported by a command-line interface (CLI) tool, we name APICTURE that allows generating interactive and non-interactive visualization from a git software repository that includes either the code implementation or the OAS documentation of an API. It is publicly available on NPM, facilitating easy access and utilization by researchers and practitioners alike. And, can be installed using the command: `npm install -g apict`.

The APICTURE tool functions as a visualization tool designed to maintain a comprehensive record of all previous versions of an OAS within a Git repository. It provides a command-line interface (CLI), build on top of ECharts [92], for seamless interaction. By executing the command *apict*, the tool retrieves the complete historical data of the OAS model from the repository, enabling the generation of three distinct visualizations that focus on key aspects: *changes localization*, *versioning* versus *changes types*, and *metrics evolution*.

In addition, APICTURE has the capability to generate evolution visualizations directly from API implementation in Express.js. This feature allows users to visualize and analyze the evolution of APIs even in cases where there is no existing OAS description available for the repository. Using this functionality, developers can gain insight into the evolution of APIs directly from the codebase, further enhancing the versatility and applicability of APICTURE.

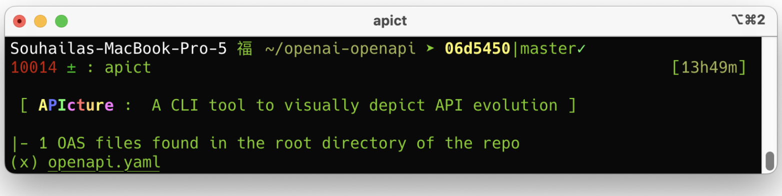
If the OAS specification is located at the root level of the repository, there is no need to specify the path of the specification. The following command lines can be executed:

APICTURE utilizes subcommands to allow focusing on specific perspective:

```
apict <spec-path>
apict changes <spec-path>
apict versioning <spec-path>
apict metrics <spec-path>
apict report <spec-path>
```

- `apict <spec-path>`: This command generates API evolution visualizations based on the OAS specification located at the specified `<spec-path>`. It retrieves the history of the OAS model from a Git repository and visualizes the changes that occurred over time.

  As shown in Figure 8.10 `apict` command can also be used without a spec path when the command is run from the folder containing the target API specification. In this case APICTURE will locate all the existing specifications and asks to select the wanted one.



```
Souhailas-MacBook-Pro-5 福 ~/openai-openapi ➤ 06d5450|master✓
10014 ± : apict                                                     [13h49m]

 [ APIcture :  A CLI tool to visually depict API evolution ]

|- 1 OAS files found in the root directory of the repo
(x) openapi.yaml
```

Figure 8.10. APIcture CLI when using the `apict` command

- `apict changes <spec-path>`: This command focuses specifically on the changes within the API. It generates visualizations that highlight and analyze the modifications, additions, and deletions made to the API endpoints, data models, and other components defined in the OAS specification located at `<spec-path>`. As for `apict` command without sub-commands, this command can also be run without a `<spec-path>`.

  Options:

  – *Details*: The 'changes' subcommand offers a `-details (-d)` option, which introduces an additional level of granularity to the generated visualization. By default, the visualization provides an aggregate representation of changes at the path level, offering a comprehensive understanding of the overall change patterns across all paths. However, by utilizing the `-details` option, users can access a more fine-grained view of changes occurring at each individual path. This feature proves especially valuable when dealing with APIs characterized by an extensive history. Analyzing changes at the path level facilitates the identification of shared evolution patterns among different paths, enabling a more nuanced exploration of the API's evolution.

  - *Changes Frequency*: The `changes` subcommand offers an additional option, `-frequency(freq)`, which allows the user to specify the minimum frequency of changes to be visualized. As not all changes have the same occurrence rate throughout the API's history, this option enables the generation of a focused visualization that includes only changes that have occurred a specified minimum number of times. For example:

    ```
    apict changes --frequency 100
    ```

    This command will generate a changes visualization that includes only those changes that have happened at least 100 times during the API's history.

- `apict versioning <spec-path>`: This command examines the versioning aspects of the API. It generates visualizations illustrating the version upgrades and changes made to the API over time. By using this command, you can analyze the adherence to semantic versioning principles and gain insights into the evolution of different API versions. As for `apict` command without sub-commands, this command can also be run without a `<spec-path>`.

- `apict metrics <spec-path>`: This command generates visualizations focusing on the metrics related to the API evolution. The `apict metrics` subcommand offers a range of options to enable users to select and visualize specific API size metrics and observe their evolution over time. It provides insights into various quantitative aspects of the API, such as the evolution of:

  - `--endpoints(-e)`: Count the number of endpoints at every commit timestamp.
  - `--paths(-p)`: Count the number of endpoints at every commit timestamp.
  - `--breaking-changes(-bc)`: Counts the frequent detected breaking changes in the API history.
  - `--methods(-m)`: The evolution of usage of HTTP methods in the API overtime.
  - `--breaking-methods(-bm)`: The types of API changes happening within specific HTTP method in the API.
  - `--parameters(-param)`: Depicts the number of parameters and parameterized operations present in the API and every commit timestamps. It also counts the number of distinct parameters used in the API in every commit.
  - `--datamodel(-d)`: Reflect the evolution of complexity of API data model, by counting the number of used schemas distinct and their properties, and the number of distinct proprieties at every commit timestamp.

  For instance, in the context of Web APIs primarily designed for accessing data in a fine-grained manner, visualizing the evolution of parametrized endpoints and schema properties can provide valuable insights into how the API enhances data access capabilities over time. By tracking the changes in endpoint parameters and schema properties, it becomes possible to observe the iterative improvements made to the API's data retrieval mechanisms. It can be also used as an indicator of a need of migration to a GraphQL API [28].

  Note that running the `apict metrics` command with no options generates a full report of metrics evolution.

  The tool generates an HTML page with interactive visualizations. The page allows downloading the visualization as SVG, or PNG.

  In this initial version of the tool, we currently support cases where the OAS specification has always resided in the same location during its evolutions.

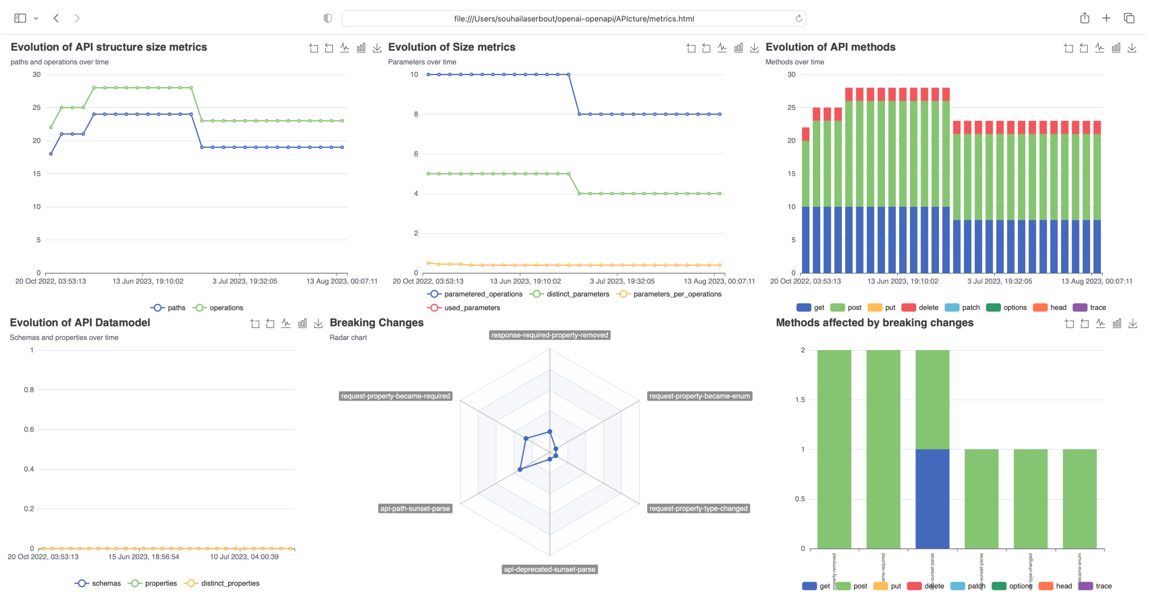
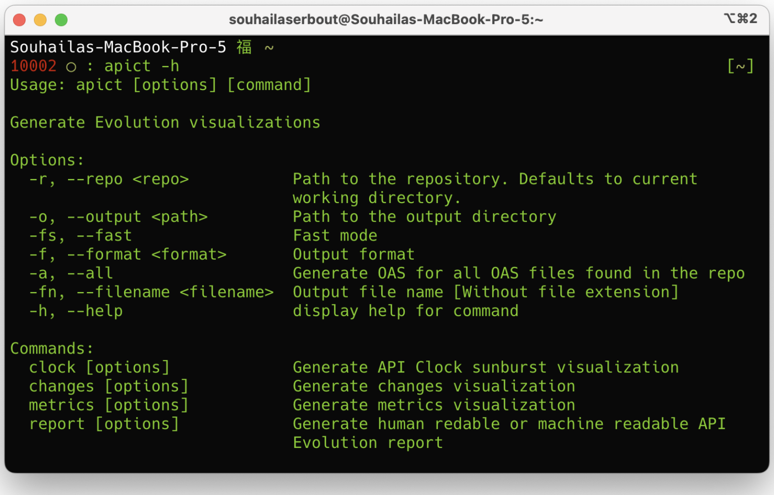

file:///Users/souhailaserbout/openai-openapi/APIcture/metrics.html
Evolution of API structure size metrics
paths and operations over time
paths
operations
Evolution of Size metrics
Parameters over time
parametered_operations
distinct_parameters
parameters_per_operations
used_parameters
Evolution of API methods
Methods over time
get
post
put
delete
patch
options
head
trace
Evolution of API Datamodel
Schemas and properties over time
schemas
properties
distinct_properties
Breaking Changes
Radar chart
response-required-property-removed
request-property-became-required
request-property-became-enum
api-path-sunset-parse
request-property-type-changed
api-deprecated-sunset-parse
Methods affected by breaking changes
20 Oct 2022, 03:53:13
13 Jun 2023, 19:10:02
3 Jul 2023, 19:32:05
13 Aug 2023, 00:07:11
15 Jun 2023, 18:56:54
10 Jul 2023, 04:00:39

Figure 8.11. Generated HTML metrics view using `apict metrics <spec-path>` without specifying a metric

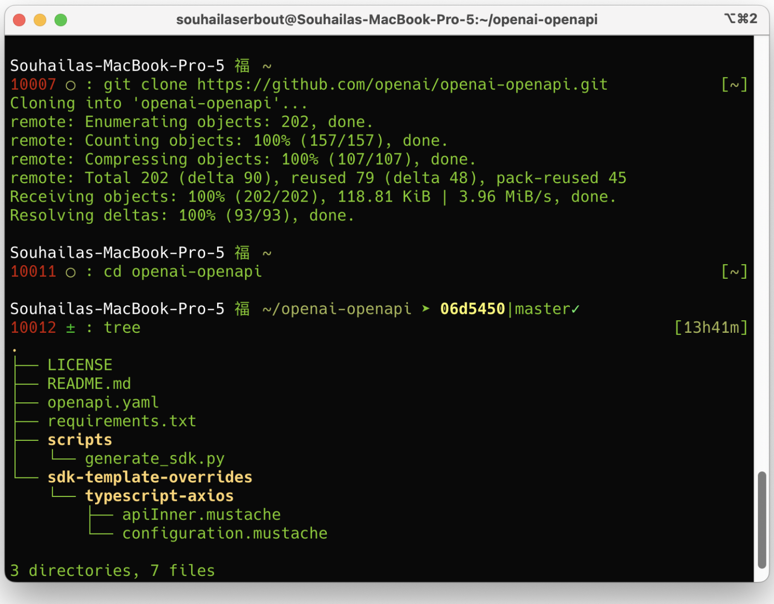
For all the commands, APICTURE provides several options (Figure 8.12) to customize the visualization process according to the users needs:

- **-r, –repo <repo>**: Specifies the path to the repository containing the API's version history. By default, the tool uses the current working directory as the repository location.
- **-o, –output <path>**: Defines the path to the output directory where the generated visualizations will be saved. If not specified, the output will be saved in the default directory.
- **-fs, –fast**: Activates the fast mode, which optimizes the execution for faster generation of visualizations. This mode can only be activated if the visualization has been already generated in normal mode. If not, this option is ignored.
- **-f, –format <format>**: Specifies the desired output format for the generated visualizations. The available formats include options such as PNG, SVG, and interactive HTML.
- **-a, –all**: Generates OpenAPI specifications for all OpenAPI files found in the repository. This option facilitates generating visualizations for multiple specifications within the same repository.
- **-fn, –filename <filename>**: Specifies the output file name for the generated visualization. The tool saves the visualization with the specified file name (without file extension) in the output directory.
- **-h, –help**: Displays the help information for the command, providing a concise overview of the available options.



```
Souhailas-MacBook-Pro-5 福 ~
10002 ○ : apict -h                                                              [~]
Usage: apict [options] [command]

Generate Evolution visualizations

Options:
  -r, --repo <repo>              Path to the repository. Defaults to current
                                 working directory.
  -o, --output <path>            Path to the output directory
  -fs, --fast                    Fast mode
  -f, --format <format>          Output format
  -a, --all                      Generate OAS for all OAS files found in the repo
  -fn, --filename <filename>     Output file name [Without file extension]
  -h, --help                     display help for command

Commands:
  clock [options]                Generate API Clock sunburst visualization
  changes [options]              Generate changes visualization
  metrics [options]              Generate metrics visualization
  report [options]               Generate human redable or machine readable API
                                 Evolution report
```

Figure 8.12. Help Command Output for APIcture Tool



```
Souhailas-MacBook-Pro-5 福 ~
10007 ○ : git clone https://github.com/openai/openai-openapi.git              [~]
Cloning into 'openai-openapi'...
remote: Enumerating objects: 202, done.
remote: Counting objects: 100% (157/157), done.
remote: Compressing objects: 100% (107/107), done.
remote: Total 202 (delta 90), reused 79 (delta 48), pack-reused 45
Receiving objects: 100% (202/202), 118.81 KiB | 3.96 MiB/s, done.
Resolving deltas: 100% (93/93), done.

Souhailas-MacBook-Pro-5 福 ~
10011 ○ : cd openai-openapi                                                    [~]

Souhailas-MacBook-Pro-5 福 ~/openai-openapi ➤ 06d5450|master✓
10012 ± : tree                                                            [13h41m]
.
├── LICENSE
├── README.md
├── openapi.yaml
├── requirements.txt
├── scripts
│   └── generate_sdk.py
└── sdk-template-overrides
    └── typescript-axios
        ├── apiInner.mustache
        └── configuration.mustache

3 directories, 7 files
```

Figure 8.13. Repository Structure: OpenAI API

8.7.1 Use case example

This section provides comprehensive instructions for generating visualizations from a real world repository using the commands and options listed earlier.

For illustrative purposes, we have included sample API GitHub repositories containing OpenAPI specifications in our GitHub Repository[1].

We pick the OpenAI API[2] as our example. Begin by cloning the repository to your local machine:

[1]https://github.com/souhailaS/APIcture/blob/main/vissoft/git_urls.json
[2]https://github.com/openai/openai-openapi.git

```
git clone https://github.com/openai/openai-openapi
```

Next, navigate into the repository:

```
cd openai-openapi
```

Figure 8.13 shows the structure of the OpenAI API repository, revealing the top-level placement of the OpenAPI specification file (`openapi.yaml`).

To generate all evolution visualizations at once, simply execute: `apict`.

Alternatively, utilize the '`apict`' command with the '`-r`' option, which will run the visualizations generation without need to navigate to the repository:

```
apict -r openai-openapi
```

In the absence of a specific file path, APIcture automatically locates all OpenAPI specification files within the repository and prompts the user to select one (Figure 8.10).

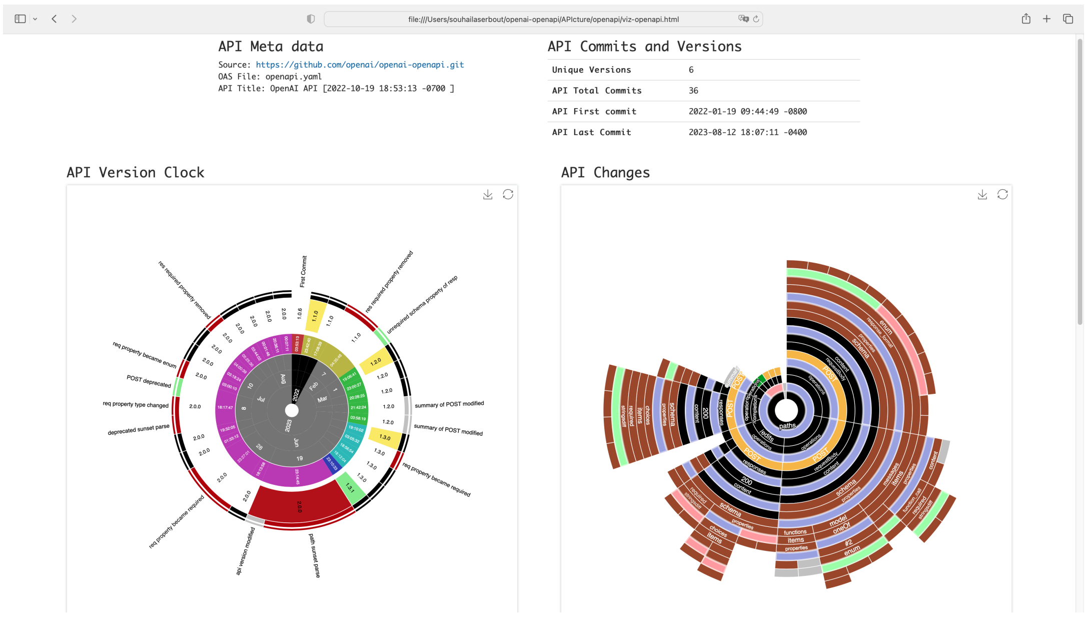
To generate visualizations for all OpenAPI specifications within the repository, apply the '`-a [-all]`' option:

```
apict -r openai-openapi -a
```

Generated visualizations are stored within the `APIcture` folder, organized under a directory named after the respective specification. For instance, in this case, the visualizations are located within `APIcture/openapi` (Figure 8.15). By default, if no format is given, the output format is HTML.

The generated HTML files are:

- `changes-<openapi api file name>.html`: an interactive format of the API Changes visualization.
- `version-clock-<openapi api file name>.html`: an interactive format of the API Version Clock visualization.
- `version-clock-<openapi api file name>.html`: an interactive format of six API evolution metrics plots (Figure 8.11).
- `viz-<openapi api file name>.html`: a single HTML page that includes all the previous interactive visualizations, in addition to a header showing history related metadata (Figure 8.14).



file:///Users/souhailaserbout/openai-openapi/APIcture/openapi/viz-openapi.html

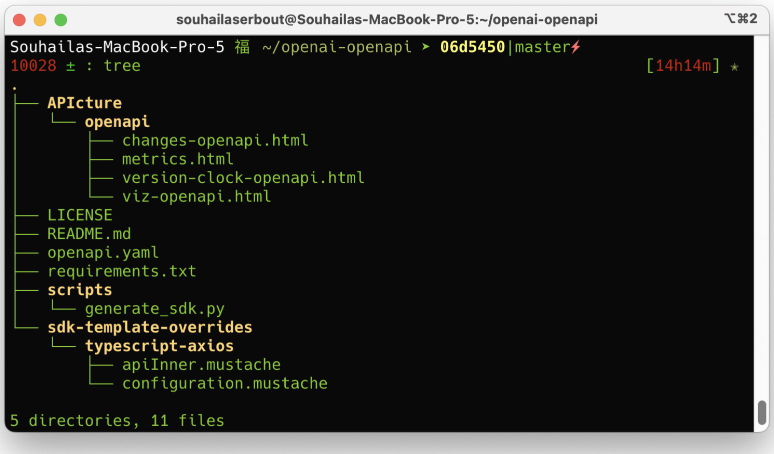
API Meta data

Source: https://github.com/openai/openai-openapi.git
OAS File: openapi.yaml
API Title: OpenAI API [2022-10-19 18:53:13 -0700]

API Commits and Versions

Unique Versions	6
API Total Commits	36
API First commit	2022-01-19 09:44:49 -0800
API Last Commit	2023-08-12 18:07:11 -0400

Figure 8.14. `apict` output evolution visualizations generated in `viz-openapi.html`



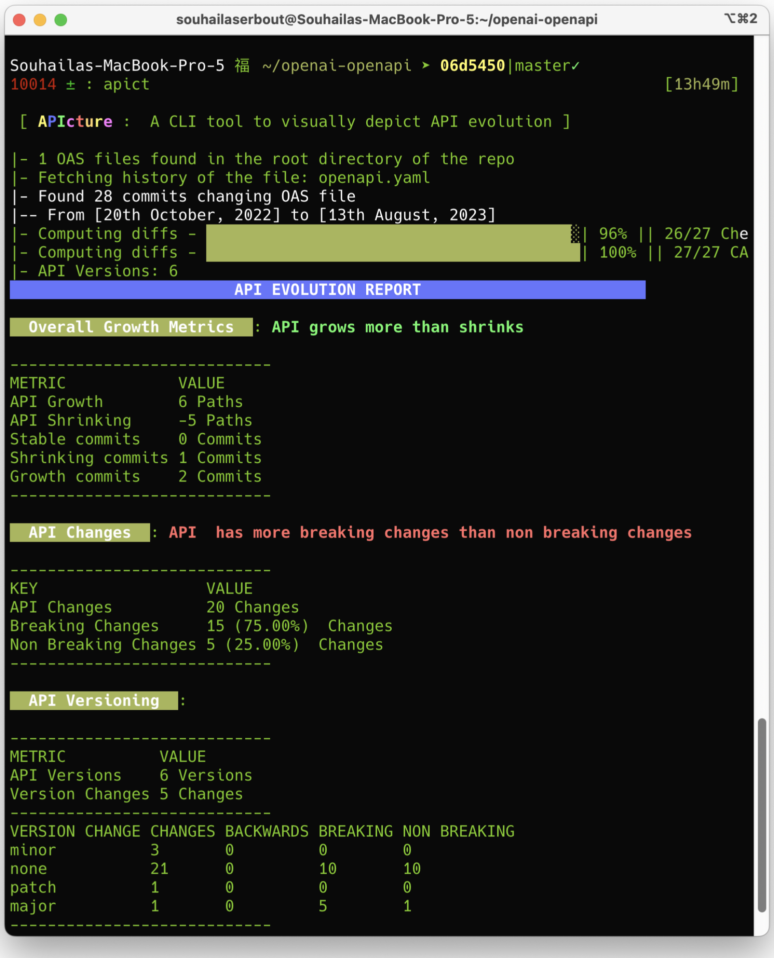
souhailaserbout@Souhailas-MacBook-Pro-5:~/openai-openapi

```
Souhailas-MacBook-Pro-5 福 ~/openai-openapi ➤ 06d5450|master⚡
10028 ± : tree                                                    [14h14m] ✭
.
├── APIcture
│   └── openapi
│       ├── changes-openapi.html
│       ├── metrics.html
│       ├── version-clock-openapi.html
│       └── viz-openapi.html
├── LICENSE
├── README.md
├── openapi.yaml
├── requirements.txt
├── scripts
│   └── generate_sdk.py
└── sdk-template-overrides
    └── typescript-axios
        ├── apiInner.mustache
        └── configuration.mustache

5 directories, 11 files
```

Figure 8.15. `apict` command output folders structure

When executing the `apict` command without specifying any additional options, it initiates the process of generating an evolution report directly within the terminal (Figure 8.16). Furthermore, for users seeking to access this report independently of the complete visualization generation process, the `report` subcommand can be employed.

```
apict report -r openai-openapi
```



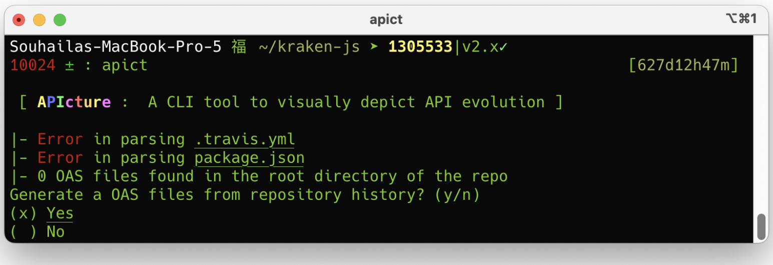
```
souhailaserbout@Souhailas-MacBook-Pro-5:~/openai-openapi                 ⌥⌘2

Souhailas-MacBook-Pro-5 福 ~/openai-openapi ➤ 06d5450|master✓
10014 ± : apict                                                   [13h49m]

 [ APIcture :  A CLI tool to visually depict API evolution ]

|- 1 OAS files found in the root directory of the repo
|- Fetching history of the file: openapi.yaml
|- Found 28 commits changing OAS file
|-- From [20th October, 2022] to [13th August, 2023]
|- Computing diffs - ████████████████████████████████████████░| 96% || 26/27 Che
|- Computing diffs - ████████████████████████████████████████| 100% || 27/27 CA
|- API Versions: 6
                        API EVOLUTION REPORT

  Overall Growth Metrics  : API grows more than shrinks

----------------------------
METRIC           VALUE
API Growth       6 Paths
API Shrinking    -5 Paths
Stable commits   0 Commits
Shrinking commits 1 Commits
Growth commits   2 Commits
----------------------------

  API Changes  : API  has more breaking changes than non breaking changes

----------------------------
KEY                  VALUE
API Changes          20 Changes
Breaking Changes     15 (75.00%)  Changes
Non Breaking Changes 5 (25.00%)  Changes
----------------------------

  API Versioning  :

----------------------------
METRIC          VALUE
API Versions    6 Versions
Version Changes 5 Changes
----------------------------
VERSION CHANGE CHANGES BACKWARDS BREAKING NON BREAKING
minor          3       0         0        0
none           21      0         10       10
patch          1       0         0        0
major          1       0         5        1
----------------------------
```

Figure 8.16. `apict` terminal prompt evolution report

Rather than being limited to using only the overarching `apict` command, users have the flexibility to employ focused subcommands. Each of these subcommands generates a specific output individually, allowing for a more tailored approach to visualization creation. In addition to this, the `-fast` option is provided to optimize the time taken in the generation process, particularly when users are exclusively interested in obtaining a particular output. This approach streamlines the generation time, making the process more efficient and relevant to the specific visualization needs of the user.

8.7.2 Other supported cases

In scenarios where no OpenAPI file is detected within the repository (Figure 8.17), APIcture employs a distinct approach for Express.js projects. It systematically generates a corresponding OpenAPI specification from the project's codebase for each commit existing in the repository's history. Subsequently, APIcture selects the specifications that exhibit differences from the specification of the preceding commit. This generation process leverages ExpressO [149][3] a CLI tool designed to validly generate OpenAPI specifications from Express.js code. The generated specifications are then utilized by APIcture to generate the intended visualizations.



```
Souhailas-MacBook-Pro-5 福 ~/kraken-js ➤ 1305533|v2.x✓
10024 ± : apict                                        [627d12h47m]

 [ APIcture :  A CLI tool to visually depict API evolution ]

|- Error in parsing .travis.yml
|- Error in parsing package.json
|- 0 OAS files found in the root directory of the repo
Generate a OAS files from repository history? (y/n)
(x) Yes
( ) No
```

Figure 8.17. APIcture in the case where no OpenAPI file is found in the project (run on the Kraken.js project)

In the case where the project's dependencies are not already installed, select `(x) No` then rerun the `apict` command.

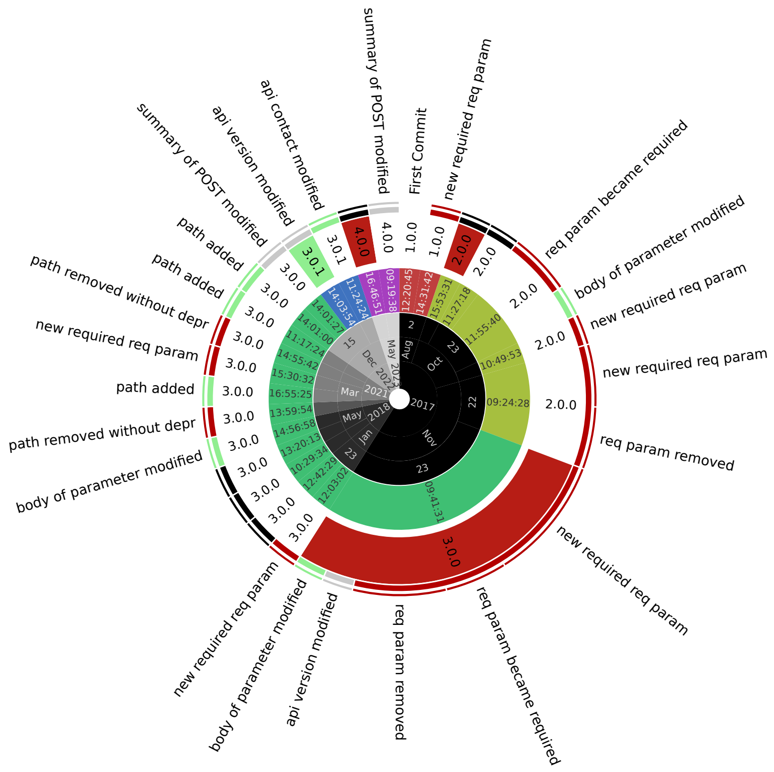
The showcased examples within our published gallery exclusively originate from projects that feature an OpenAPI specification present within the repository. In this version of APIcture, the effective generation of visualizations through Express.js code hinges on the capability of ExpressO to construct a specification from the underlying codebase.

In Summary

API Changes and API Version Clock are interactive visualizations tailored for developers, researchers, and stakeholders involved in API development, management, and evolution. The visualizations aim to offer valuable insights into the recurrent API changes, and versioning practices, aiding in understanding the evolution and backward compatibility between consecutive API versions and the adherence of the API to semantic versioning practices. The provided visualizations can be integrated into DevOps pipelines, helping to continuously gather awareness of the entire history of changes and see the evidence needed to enforce the chosen versioning strategies [71, 134].

The availability of these visualizations through the APIcture tool provides a tangible resource for API practitioners and researchers, allowing them to explore and analyze API evolution in a comprehensive and intuitive manner. It is released on NPM [2] as command-line tool that automatically generates the API Changes and API Version Clock visualizations from any git repository containing the history of an OAS specification. The tool can render each visualization separately as an SVG or PNG image, but also generate an HTML page with both interactive visualizations, individually or side by side, together with various evolution metrics visualizations and metadata. It can be installed using the command: `npm install -g apict`. A demo video of the interactive visualizations is available at: `https://www.youtube.com/watch?v=WtFm6VvKi20`

While the visualizations have been originally designed in the context of APIs described using the OAS standard, they can be generalized to other artifacts. The API Version Clock requires a stream of commits with the corresponding version identifiers together with metrics characterizing and classifying the changes w.r.t. the previous commit. The API Changes visualization is applicable to show how any nested object structure evolves, as it only requires a lightweight customization for color mapping different properties. We plan to broaden the scope of applicability of the visualization tool by decoupling it from its domain of inception in the near future.



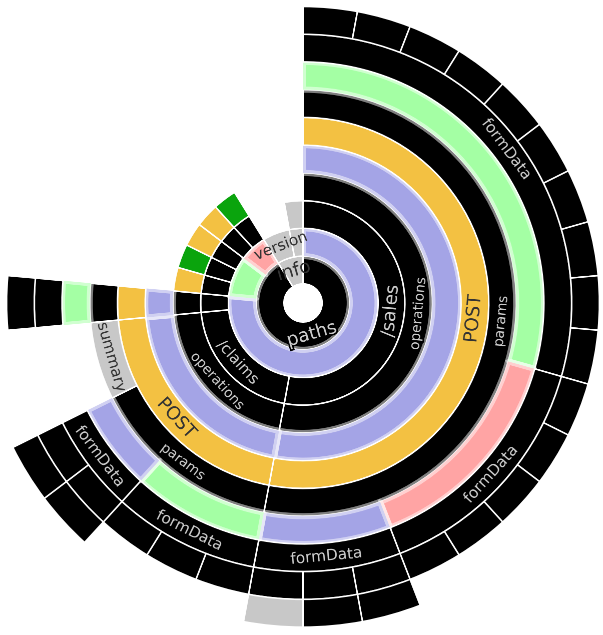
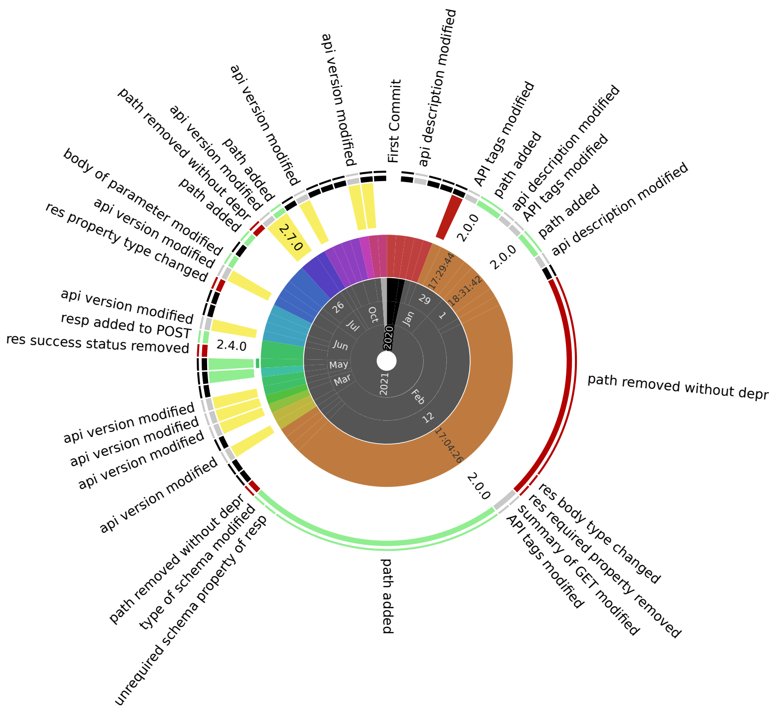


Figure 8.5. Visualizations of the SunRocks API Evolution (24 commits over 5 versions during 2114 days, 0⋆)



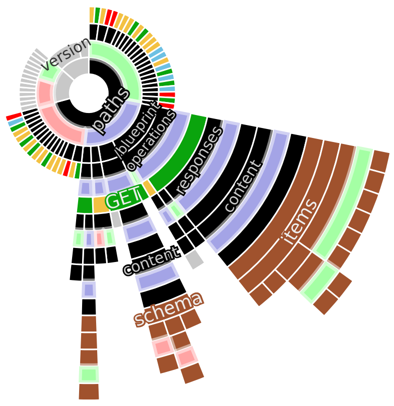
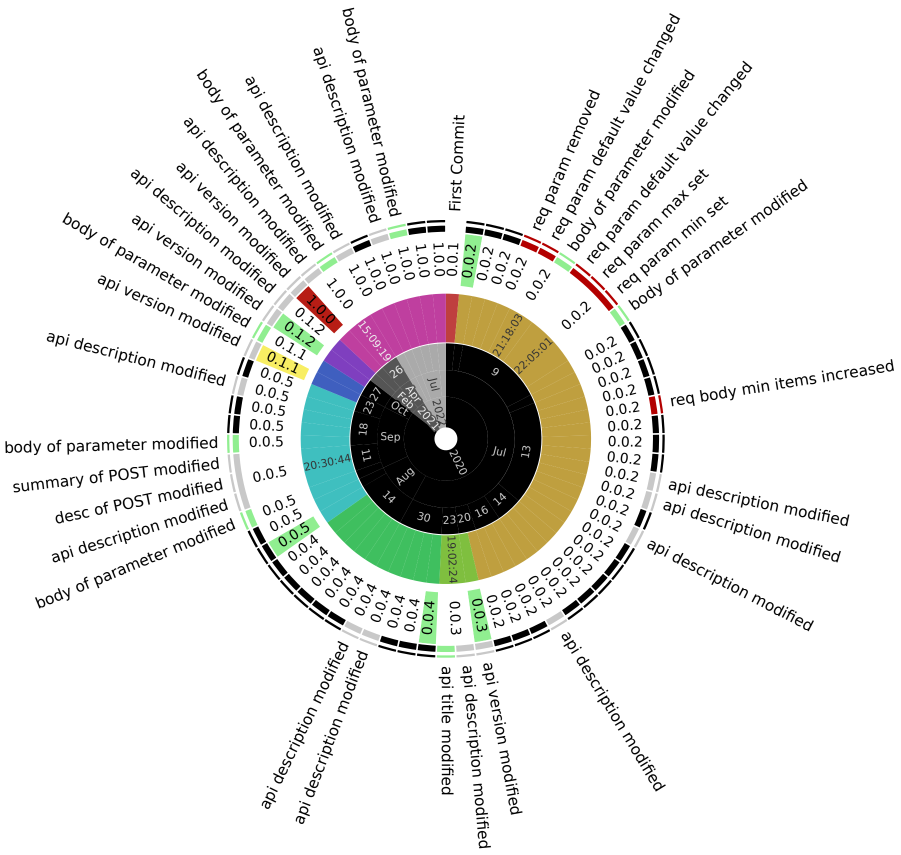


Figure 8.6. Visualizations of the xOpera REST API Evolution (42 commits over 14 versions during 408 days, 3⋆)



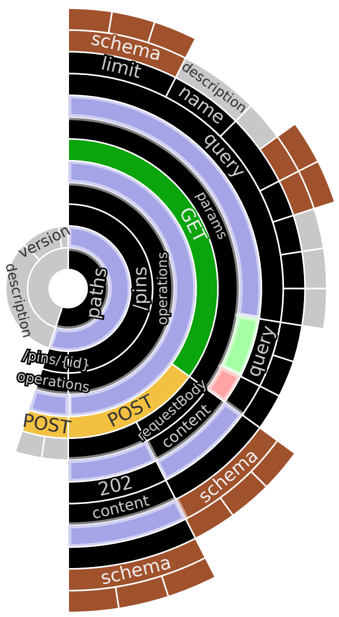


Figure 8.7. Visualizations of the IPFS Pinning Service API Evolution (61 commits over 8 versions during 773 days, 84⋆)

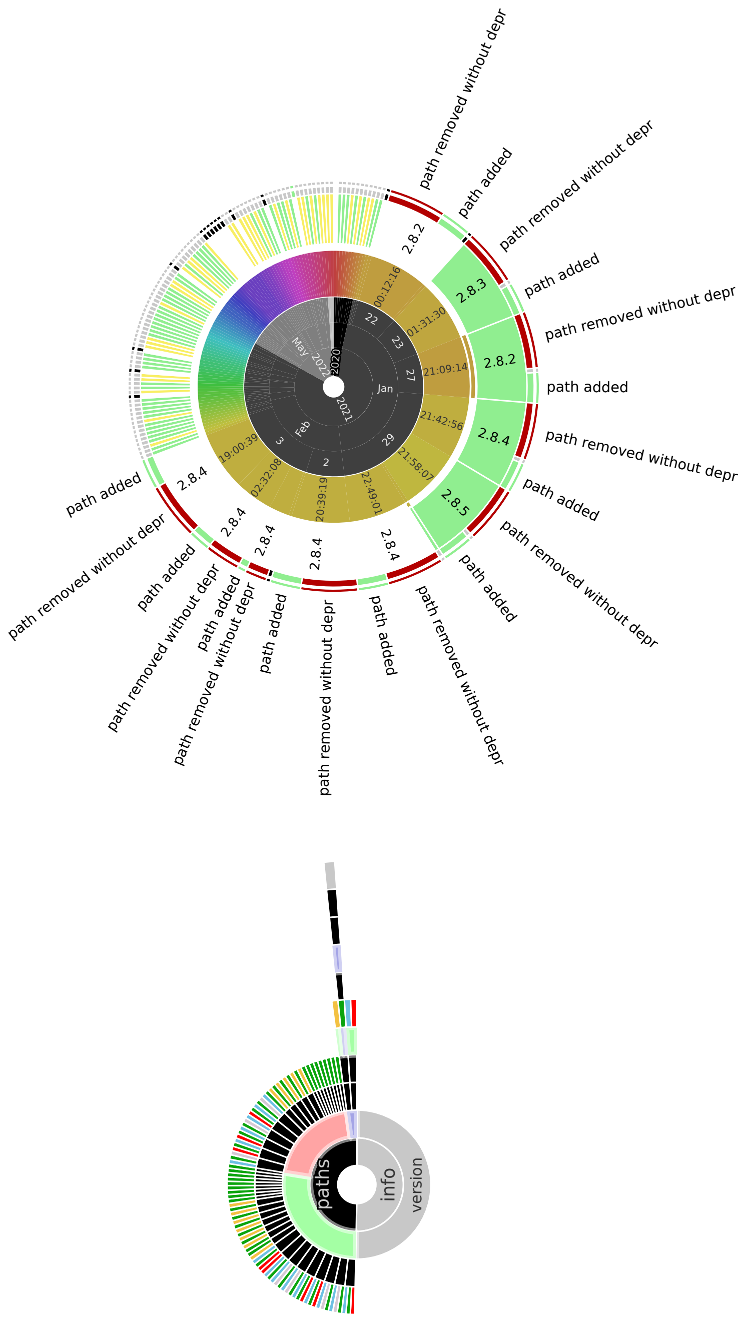


Figure 8.8. Visualizations of the Xero Projects API Evolution (125 commits over 93 versions during 999 days, 80⋆)

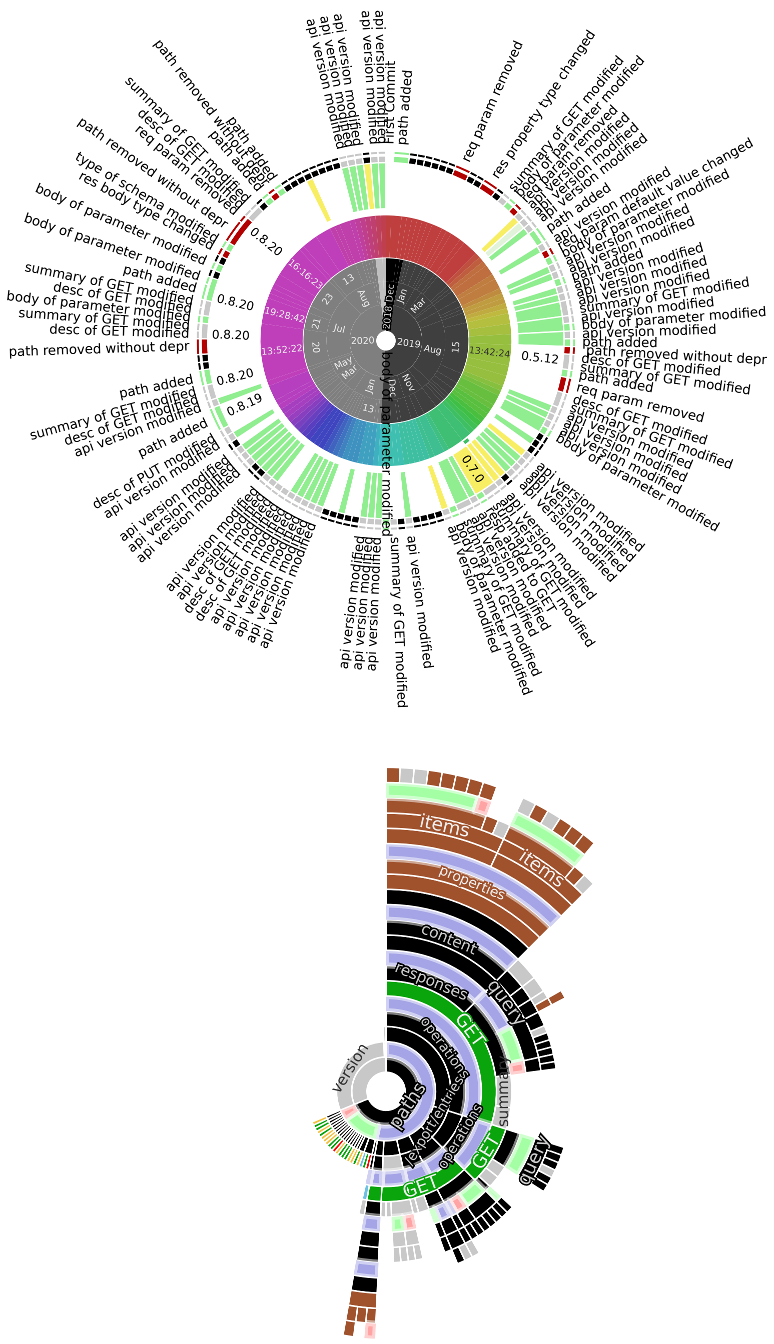


Figure 8.9. Visualizations of the OpenFairDB API Evolution (144 commits over 52 versions during 1563 days, 53⋆)

Part IV

Web API versioning analysis

Chapter 9

Web API Versioning

This chapter highlights our findings on the current state of Web API versioning using static analysis techniques applied to a snapshot of OpenAPI specifications comprising more than half a million specifications.

The analysis is performed over a snapshot of the specification dataset, including descriptions source from all of GitHub, SwaggerHub, APIsGuru and BigQuery.

9.1 Web API Versioning Challenges

API versioning [57] is a fundamental practice that enables API providers to manage change effectively while ensuring compatibility with existing clients. API providers often use version identifiers to make changes evident to clients, allowing them to refer to specific versions of the API on which they depend. In some cases, providers make multiple versions of the same API available to ease the transition for clients as they switch from retired versions to newer versions [98].

The lack of a centralized registry for Web APIs, combined with the flexibility for service providers to use their versioning approaches [127], has led to multiple and sometimes inconsistent practices in terms of discoverability and notification of breaking changes [22]. While versioning metadata is required when describing Web APIs according to the OpenAPI specification, developers use a variety of version identifier formats to express different concerns: when was the API released, whether the API version is stable or still a preview release, whether the changes introduced in the API are likely to break clients. Such variability in versioning practices raises questions about the prevalence of semantic versioning [8] adoption among Web APIs and how to dynamically discover and select which API version is available at runtime.

9.2 Web APIs Versioning Documentation

9.2.1 Version Identifiers in Web APIs

In the realm of Web APIs, there exist various options for including a version identifier, statically, as part of an API description, or dynamically, as part of messages exchanged with the API. In this study we expect to find evidence for the following practices:

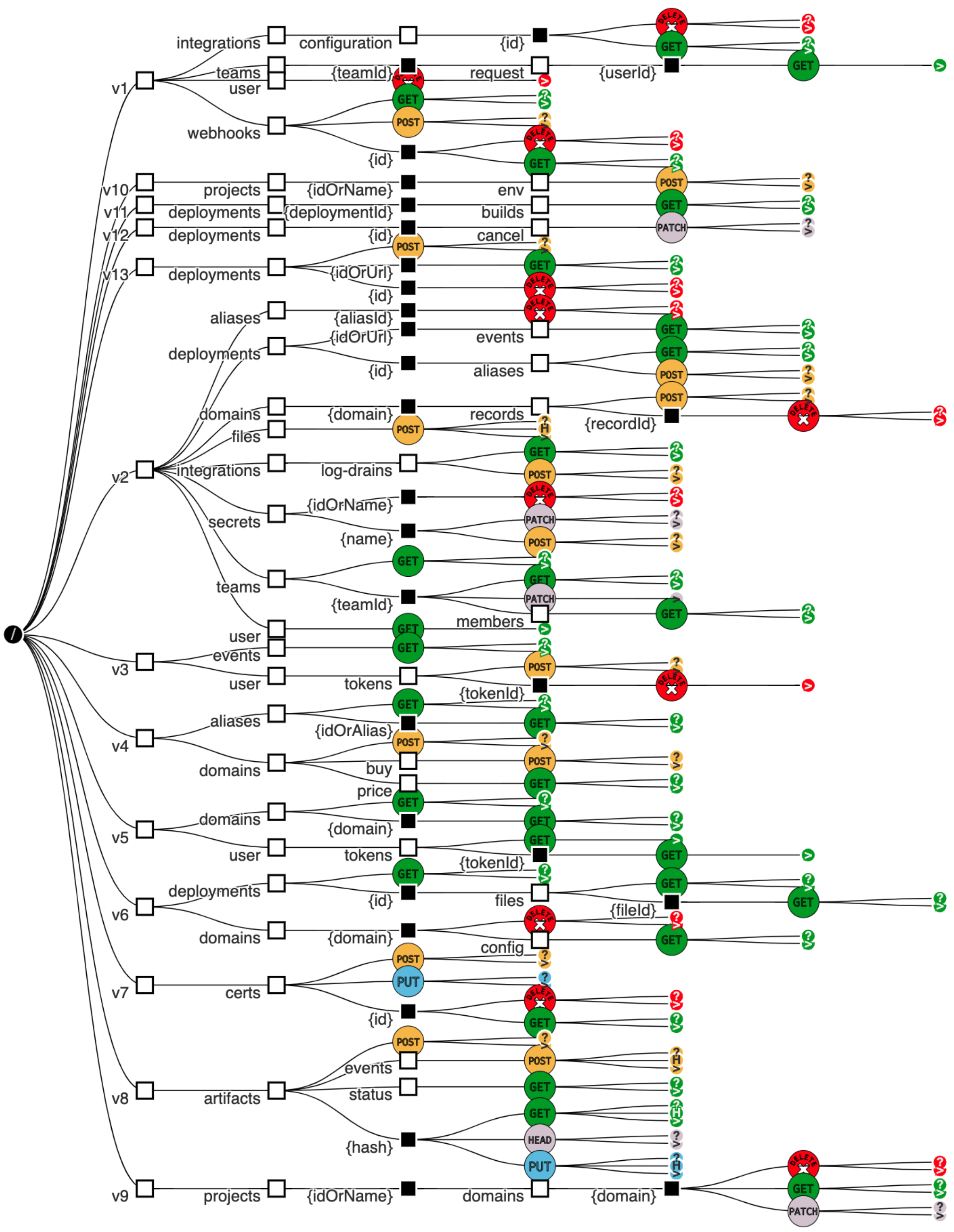


Figure 9.1. Tree visualization of the structure of a subset of the Vercel API [1]. Different version identifiers (v1-v12) are found in the path URL addresses.

• Metadata-based versioning: The version identifier is included within the API metadata. This can be achieved using industry-standard formats like the OpenAPI Specification, which provides a comprehensive description of the API, including versioning details, in a machine-readable manner or mentioning it on the API provider's website or documentation.

• URL-based versioning: The version identifier expected by the client can be embedded as part of HTTP request messages as a parameter or a segment in the endpoint path URL, such as:

```
https://<server-address>/<path>?<query>
```

where:

```
<server-address> = <version-identifier>.<dns-domain>
<path> = <path>/<version-identifier>/<path> || <path> || ""
<query> = <query>&version=<version-identifier>&<query> || <query> || ""
```

For example:
`https://v1.example.api/` | `https://api.example.com/v1` | `https://example.com/api?version=v1`

Embedding version identifiers in endpoint URLs is commonly used also when multiple versions of the API coexist simultaneously, known as the "two in-production" pattern [98]. The API server employs the version identifier found within the request to route the request to the appropriate API version. For example, in the Vercel API 12 different versions are accessible to clients (Fig. 9.1).

• Header-based versioning: Instead of embedding version information in the URL or other parts of the request, when an API uses header-based versioning the client specifies the desired version using an HTTP header. This type of versioning can be also applied at the operation level where the version is specified in the header of the request associated with the operation, such as the example in Listing 9.1.

Listing 9.1. Header-based versioning applied to certain API operations

```
name: x-ms-version
in: header
description: The version of the operation to use for this request See https://docs.microsoft.com/en-us/rest/api/storageservices/
     ↪ versioning-for-the-azure-storage-services for details
required: false
schema:
  type: string
```

E.g., within our dataset, the Amaysim[2] API serves as an illustrative example. It implements the utilization of the `accept-version` header as a mechanism for transmitting the version identifier conforming to Semantic Versioning (SemVer). In the event that no header is explicitly provided, the API defaults to invoking the latest available version:

One of the advantages of header based versioning is that the clients can seamlessly switch between versions without modifying the request structure. The server interprets the header to route the request to the appropriate version of the API.

• Dynamic versioning: In APIs using header-based versioning, developers must explicitly instruct API consumers on specifying the intended API version in request headers. This information, encompassing the designated header field (e.g., `x-api-version`) and the requisite version format (e.g., `v1`, `1.0`), must be meticulously documented. API consumers are then tasked with including the version data in the headers of their HTTP requests using the provided header name and value or the version query parameter. For instance, the case of the GitHub API, developers are informed about viable version header values through the invocation of a GET /versions endpoint. This endpoint facilitates the retrieval of a list encompassing available version identifiers. Developers can thus reference this endpoint to ascertain the valid version options for configuration within their request headers. This practice augments transparency and streamlines the process of selecting and incorporating appropriate API versions.

[2]https://www.amaysim.com.au/

9.2.2 OpenAPI Versioning Metadata

API service providers typically provide API clients with information on how to use the API through a description, which is often written in natural language [178] or based on a standard Interface Description Language (IDL), such as OpenAPI [118]. This later has seen a widespread adoption across industries [168, 60, 167], which underscores its pivotal role in modern API development and integration. It is also a form of documentation that is machine-readable, enabling systematic analysis on a large scale.

OpenAPI offer a standardized, language-agnostic framework for documenting RESTful APIs, which facilitates clearer communication among developers, accelerates development timelines, and ensures consistent API implementation. Moreover, it includes a specific required field `{"version": string}` in the `info` section pertaining to the API's metadata. However, there are no constraints on the format used to represent the version identifier. Additionally, version identifiers can be embedded in the API endpoint addresses, which are stored in the `server` and `path` URLs.

While the OpenAPI standard defines how developers describe their APIs, there is no centralized standard documentation manager service where developers can share API specifications. For example, SwaggerHub [3] does not impose any rules on the format of version identifiers, nor does it require developers to upgrade them when publishing a new version of the API description. We aim to study the resulting variety of version identifier formats found in a large collection of OpenAPI descriptions.

9.2.3 API stable releases

API stable releases represent the versions of the API that are deemed ready for use in production environments. These versions have undergone thorough testing and are considered reliable and stable for use by clients. The version identifiers for stable releases often convey important information about the changes introduced in the release, the compatibility with previous versions, and the maturity of the API.

In our study, we identified four primary classes of formats for stable release identifiers:

• *Major Version Number:* This format is characterized by a single integer value that increments with each major release. It is a simplified form of semantic versioning, focusing only on major changes that are likely to be incompatible with previous versions. This format is straightforward and easy to understand, but it does not provide detailed information about minor updates or patches.

• *Semver (Semantic Versioning):* The goal of semantic versioning [8] is to reflect the impact of API changes through the version identifier format `MAJOR.MINOR.PATCH`. The `MAJOR` version counter is incremented when incompatible API changes were introduced, the `MINOR` counter is upgraded when new functionalities were added without breaking any of the old ones, and the `PATCH` increases for backwards compatible bug fixes.
Several widely known package managers, such as NPM [9], Maven [182], and PyPI, adopt semantic versioning as a standard for package version identifiers. These package managers enforce the usage of semantic versioning and perform version increment checks every time the package is republished [45].
We put under this category all the version identifiers that follow the semantic versioning format, regardless of the number of digits used, starting from 2 digits.

• *Date:* Some APIs use the release date as the version identifier. This format can take various forms, such as YYYY-MM-DD or YYYYMMDD. It provides a clear timeline of API releases and is easy to understand. However, it does not provide any information about the changes introduced in each version.

• *Tag:* This format uses arbitrary word values as version identifiers, such as: “latest”, “newest”, “test", wich we found as the most common words. This format provides the most flexibility, but it can also be the most difficult to understand and manage, especially for large APIs with many versions.

[3]SwaggerHub API. https://app.swaggerhub.com/apis-docs/swagger-hub/registry-api/1.0.67

9.2.4 API Preview Releases

Test releases are often given specific marketing names to clearly reflect their purpose and distinguish them from stable releases. Marketing names help also to indicate the audience of the test releases, and allow users to understand that they should expect bugs [101][4][5].

In our datasets, we identified the following six types of usage for preview release tags:

• *Develop:* A version under development is still in the process of being created and is not yet complete or stable. It may contain new features or bug fixes that have not yet been fully tested, and may not be suitable for use in a production environment. Developers may use dev versions to test new features and make changes before releasing a final version to the public.

• *Snapshot:* These versions are automatically built from the latest development code and are intended to be used by developers.

• *Preview:* These are unstable versions that are made available to users before the final release. Preview versions are typically released to a small group of users or testers to gather feedback and iron out any bugs or issues before the final release. They can also be used to give users a preview about new features to expect to see in the next stable version.

• *Alpha:* These versions are considered to be very early in development and are likely to be unstable and contain many bugs. They are often released to a small group of testers for feedback.

• *Beta:* These versions are considered to be more stable than alpha versions and are often released to a wider group of testers for feedback. They may still contain bugs, but they are expected to be closer to the final release.

• *Release Candidate (RC):* These versions are considered to be very close to the final release and are often the last versions to be released before the final version. They are expected to be stable and contain only minor bugs.

Our goal is to quantify how often such types of stable and pre-release versions are found, and whether developers also use other kinds of tags to classify their API versions.

9.3 Methodology

9.3.1 Dataset preparation

Our analysis was performed on a snapshot of the OpenAPI files collection which comprised: GitHub (5 218 APIs, 165 939 commits); SwaggerHub (387 463 APIs), BigQuery (45 467 APIs), APIs.guru: (3 990 APIs), for a total of 602 859 API descriptions.

- GitHub: This historical dataset of 165 939 OpenAPI specifications, belonging to 5 218 APIs, was extracted from GitHub utilizing its API. Similar to previous works [43], also in this study we have included only APIs with the entire history of valid specifications and at least 10 commits in their version history, thereby filtering trivial or inconsequential repositories.

- SwaggerHub: Out of the 432 265 specifications, we could keep 387 463 unique and valid specifications. This collection has also time stamps of the update and creation of each entry.

- BigQuery: In this study, we used a total of 175 549 files, from which 45 467 represented unique and valid OpenAPI specifications.

- APIs.Guru: From APIs.guru, we used 3990 OpenAPI files that were all valid.

[4]Fedora versioning. `https://docs.fedoraproject.org/en-US/packaging-guidelines/Versioning/`
[5]Release naming conventions. `https://www.drupal.org/node/1015226`

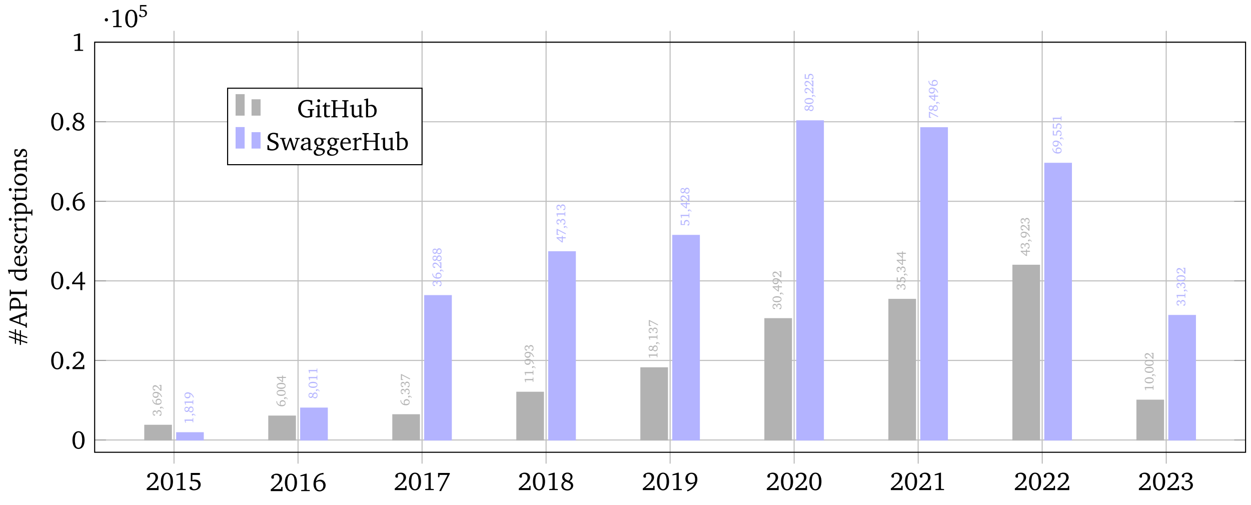
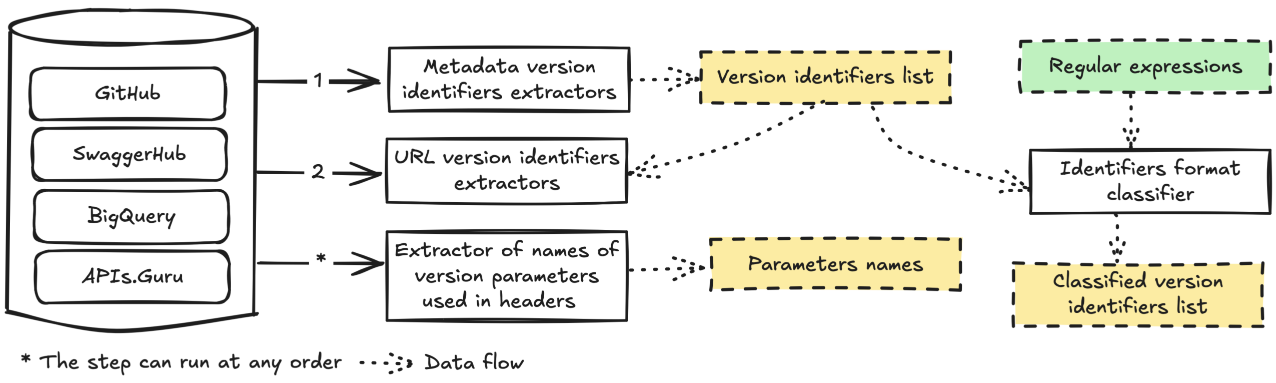


Figure 9.2. Number of artifacts in the GitHub and SwaggerHub datasets over the years



GitHub
SwaggerHub
BigQuery
APIs.Guru
1
2
*
Metadata version identifiers extractors
URL version identifiers extractors
Extractor of names of version parameters used in headers
Version identifiers list
Parameters names
Regular expressions
Identifiers format classifier
Classified version identifiers list
* The step can run at any order
Data flow

Figure 9.3. Versioning Analysis Pipeline

A distinctive feature of the GitHub dataset snapshot lies in the comprehensive historical record of API specification commits, complete with their respective timestamps. Conversely, the artifacts in the SwaggerHub collection include metadata such as their creation date and the last modification date of the specifications. In Figure 9.2, we give an overview of the yearly distribution of the yearly API commits in the case of the GitHub dataset and the number of APIs created every year in the case of the SwaggerHub dataset. This will make it possible to track the adoption of API versioning practices over the past years.

The approach of the analysis remains consistent across all the specification from all sources. Only the datasets obtained from GitHub and SwaggerHub provided the necessary timestamps for the creation of specifications. Utilizing these timestamps enabled a time-series analysis to observe the adoption patterns of dynamic versioning (referenced in section 9.4.6) and to track the evolution in the adoption of diverse formats over time (detailed in section 9.4.8).

9.3.2 Analysis methodology

To perform this study, we automated the extraction of versioning metadata and the detection of different versioning practices by analyzing 602 859 API specifications written in the OpenAPI description language [118].

Format	Regular Expression
integer	`/^(\d{3}|\d{2}|\d{1})+$/i`
v*	`/v\d*/i`
semver-3	`/^(v|)\d+\.\d+\.\d+$/i`
date(yyyy-mm-dd)	`/^\d{4}-\d{2}-\d{2}/`
date(yyyymmdd)	`/20[1-2][0-2](0[1-9]|1[0-2])(0[1-9]|[12][0-9]|3[01])$/i`
date(Month yyyy)	`/(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) 20[1-2][0-9]$/i`
date(yyyy.mm.dd)	`/\d{4}\.\d{2}\.\d{2}$/i`
semver-dev*	`/^(v|)\d+\.\d+(\.\d)*(\.|-)dev\d*$/i`
semver-snapshot*	`/^(v|)\d+\.\d+(\.\d)*(\.|-)SNAPSHOT\d*$/i`
date-preview*	[date]`(-|\.)preview$/i`
v*alpha*	`/^v\d+alpha\d*$/i`
v*beta*	`/^v\d+beta\d*$/i`
semver-rc*.*	`/^(v|)\d+\.\d+(\.\d)*-rc\d*\.\d+$/i`

Table 9.1. Some detectors are used to classify the version identifier formats. In the format name, * stands for an integer.

As depicted in Figure 9.3, we first retrieved 10 221 distinct version identifiers from the `info.version` field in each OpenAPI description in each dataset (see the third column of Table 9.2 for the number of unique version identifiers found in each dataset). We then searched for any of these identifiers in the URL addresses listed as part of the endpoints or server URL strings.

To classify the version identifiers, we employed a set of regular expression rules (Table 9.1). These detectors were iteratively defined based on our observations to ensure that most of the samples could be labeled. We also distinguished between version identifiers used to describe preview releases and stable versions of the APIs. The complete list of regular expression rules are included in the replication package.

Given such a variety of sources, we examine the specifications and present the results collectively and individually, based on the origin of the specifications, to determine how the outcomes vary according to their sources.

9.4 Results

9.4.1 Metadata-based versioning

Metadata-based versioning involves encapsulating the API version identifier within the API documentation itself. In the context of OpenAPI-documented APIs, this approach is facilitated by a designated `info.version` field within the specification. This field empowers developers to explicitly denote the version of the web API being documented. By articulating the version as a string in the OpenAPI specification, the practice of metadata-based versioning establishes a clear means to communicate and represent the API's versioning information.

Metadata-based versioning adoption overview

The `info.version` field, while obligatory for a valid specification, is found to accommodate various values including empty strings and certain other non-conforming string entries, such as: `""`, `"null"`, `"undefined"`, `"version` `unknown", "-", "_", "unknown", "VERSION_PLACEHOLDER", "no version"`, etc. These non-compliant entries were identified and subsequently excluded and considered as no metadata based versioning was used.

In Table 9.2, we report that the vast majority of artifacts (across all datasets, more than 90%) makes use of metadata-based versioning. The number of unique version identifiers detected within each dataset is listed in the third column. The most common version identifier is `1.0.0`, while `v1` is the mostly used one only in the APIs.guru, where 7% of the APIs which use metadata-based versioning have the `v1` identifier.

Version identifiers formats

Given that the version is represented as a string, discerning a consistent format for the extracted version identifier becomes a non-trivial task. To address this challenge, we developed a parsing mechanism leveraging 257 regular expressions. This tailored parser enables the detection and classification of diverse version formats employed within the dataset, enhancing our ability to systematically analyze and categorize the extracted version identifiers. In Table 9.4, we present the top 20 frequently employed version identifier formats observed in each of the the four study datasets.

The format `semver-3` was found to be most frequent format. But, looking at each dataset independently, we can see that the most frequently adopted version identifier format varies depending on the source. For the SwaggerHub dataset, the most common format is `semver-3`, accounting for 69.78% of the total. Similarly, the `semver-3` format is also the most prevalent in the GitHub dataset, representing 61.68% of the total.

In contrast, the BigQuery dataset primarily uses the `date(yyyy-mm-dd)` format, which constitutes 31.82% of the total. The APIs.guru dataset also favors a date-based format, specifically `date(yyyy-mm-dd)`, which accounts for 39.77% of the total.

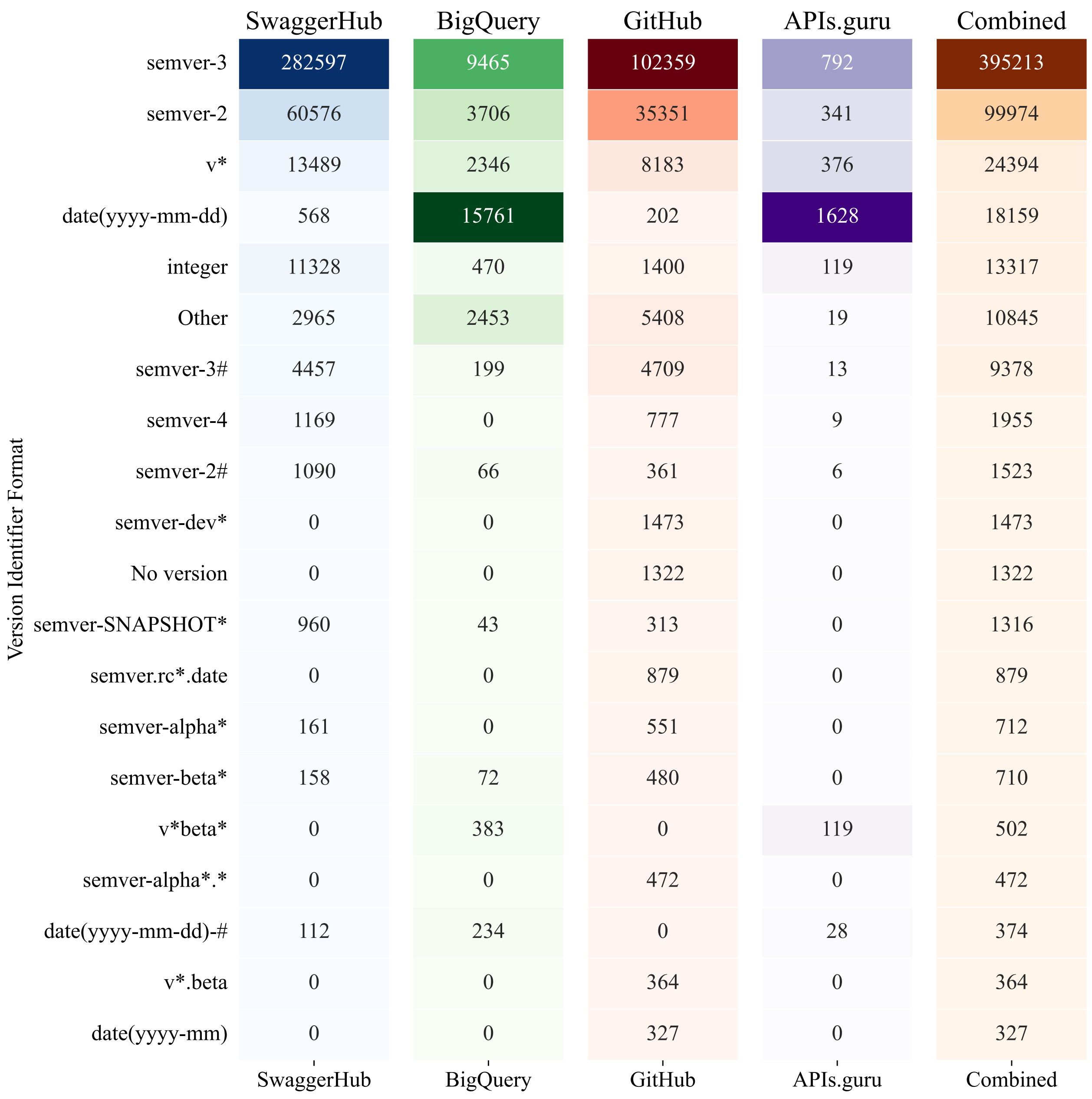
The `Other` category of formats encompasses all version identifier formats that could not be classified due to their non-uniformity. These formats do not adhere to any of the common versioning schemes such as Semantic Versioning or date-based versioning, and instead, they follow unique, custom formats devised by the API developers. Its presence highlights the diversity and complexity of versioning practices in the real-world APIs. It underscores the fact that despite the existence of widely accepted versioning schemes, a considerable number of APIs opt for custom, non-standard versioning formats. However, the use of such formats can lead to inconsistencies, make version management more complex, and potentially hinder the understanding and usage of the API for developers. Therefore, while these non classifiable formats represent a small proportion of the total, it is an important aspect of the versioning landscape that warrants further investigation and understanding.

The formats are categorized based on the versioning scheme they adhere to, such as Semantic Versioning (SemVer), date-based versioning, and others. For each format, Table 9.5 lists the number of occurrences in each dataset.

Table 9.3 and 9.4, provides a detailed breakdown of the version identifier formats used across the four datasets.

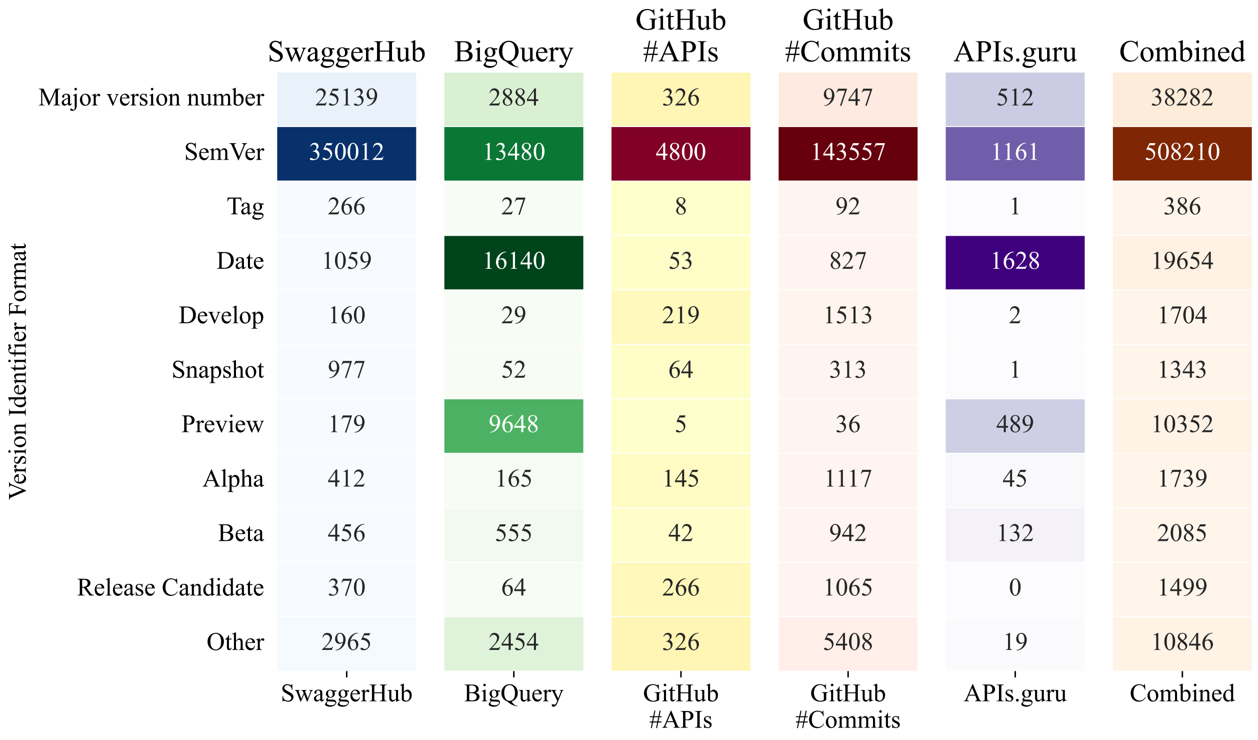
Dataset	#APIs	#Unique Version IDs	Most Used ID	#APIs Using Most Used ID
GitHub	5 107 (97.87%) 162 244 (97.77%) commits	7 020	`1.0.0`	2 242 42 416 commits
BigQuery	44 364 (89.94%)	1 941	`1.0.0`	4 987
SwaggerHub	381 437 (94.30%)	8 616	`1.0.0`	240 310
APIs.guru	3 988 (99.95%)	824	`v1`	275
Total	592 033 (98%)	10 221		

Table 9.2. Number of artifacts featuring metadata-based versioning.



Version Identifier Format	SwaggerHub	BigQuery	GitHub	APIs.guru	Combined
semver-3	282597	9465	102359	792	395213
semver-2	60576	3706	35351	341	99974
v*	13489	2346	8183	376	24394
date(yyyy-mm-dd)	568	15761	202	1628	18159
integer	11328	470	1400	119	13317
Other	2965	2453	5408	19	10845
semver-3#	4457	199	4709	13	9378
semver-4	1169	0	777	9	1955
semver-2#	1090	66	361	6	1523
semver-dev*	0	0	1473	0	1473
No version	0	0	1322	0	1322
semver-SNAPSHOT*	960	43	313	0	1316
semver.rc*.date	0	0	879	0	879
semver-alpha*	161	0	551	0	712
semver-beta*	158	72	480	0	710
v*beta*	0	383	0	119	502
semver-alpha*.*	0	0	472	0	472
date(yyyy-mm-dd)-#	112	234	0	28	374
v*.beta	0	0	364	0	364
date(yyyy-mm)	0	0	327	0	327

Figure 9.4. 20 most adopted version identifier formats used in metadata in each of the study datasets and combined



Version Identifier Format	SwaggerHub	BigQuery	GitHub #APIs	GitHub #Commits	APIs.guru	Combined
Major version number	25139	2884	326	9747	512	38282
SemVer	350012	13480	4800	143557	1161	508210
Tag	266	27	8	92	1	386
Date	1059	16140	53	827	1628	19654
Develop	160	29	219	1513	2	1704
Snapshot	977	52	64	313	1	1343
Preview	179	9648	5	36	489	10352
Alpha	412	165	145	1117	45	1739
Beta	456	555	42	942	132	2085
Release Candidate	370	64	266	1065	0	1499
Other	2965	2454	326	5408	19	10846

Figure 9.5. Number of artifacts with version identifiers used in metadata of stable and preview releases in each of the study datasets and combined

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Major version number	25 139	2 884	326	9 747	512	36 608
integer	11 328	470	85	1 400	119	13 402
v*	13 489	2 346	240	8 183	376	25 634
v*#	120	49	5	132	11	317
v*-#	202	19	1	32	6	260
SemVer	350 012	13 480	4 800	143 557	1 161	504 010
semver-2	60 576	3 706	1 146	35 351	341	101 120
semver-2#	1 090	66	19	361	6	1 542
semver-3	282 597	9 465	3 787	102 359	792	397 000
semver-3#	4 457	199	475	4 709	13	9 853
semver-4	1 169	39	13	777	9	2 007
semver-6#	2	3	-	-	-	5
semver-4#	49	-	-	-	-	49
semver-5	60	-	-	-	-	60
semver-5#	6	-	-	-	-	6
semver-6	6	-	-	-	-	6
Tag	266	27	8	92	1	394
latest*	124	13	8	92	1	238
test*	128	12	-	-	-	140
new*	11	-	-	-	-	11
Date	1 059	16 140	53	827	1 628	18 707
date(yyyy-mm)	22	4	19	327	-	372
date(yyyy-mm-dd)	568	15 761	16	202	1 587	18 134
date(yyyy-mm-dd)-#	112	234	3	18	28	395
date(yyyy-mm-ddThh:mm:ssZ)	-	60	16	262	3	341
date(yyyy.mm.dd)	124	46	1	8	7	186
date(yyyymmdd)	112	21	2	10	1	146
date(yyyy.mm)	10	3	-	-	1	14
vdate(yyyy-mm-dd)	8	5	-	-	1	14
date(yyyy)	87	-	-	-	-	87
date(yyyy-mm-dd).hh.mm.ss	1	-	-	-	-	1
date(yyyy-mm-dd hh:mm:ss)	5	1	-	-	-	6

Table 9.3. Number of artifacts with version identifiers used in metadata of stable releases in each of the study datasets and all combined

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Develop	160	29	219	1 513	2	1 704
dev*	55	1	2	39	-	95
develop*	10	-	1	1	-	12
semver-dev*	66	12	217	1 473	1	1 552
v*dev*	-	5	-	-	1	6
semver-dev*.*	5	-	-	-	-	5
Snapshot	977	52	64	313	1	1 343
semver-SNAPSHOT*	960	43	64	313	1	1 317
semver-SNAPSHOT*.*	2	-	-	-	-	2
Preview	179	9 648	5	36	489	10 352
date(yyyy-mm-dd)-preview#	77	9 374	1	10	480	9 941
semver-preview*	23	128	1	12	6	169
semver-preview*.*	10	56	3	14	1	81
date(yyyy-mm-dd)-preview*	1	40	-	-	-	41
preview*	10	12	-	-	1	23
semver-pre*.*	1	18	-	-	1	20
semver-pre*	3	-	-	-	-	3
Alpha	412	165	145	1 117	45	1 739
alpha*	50	5	2	27	2	84
semver-alpha*	161	18	25	551	-	730
semver-alpha*.*	45	1	118	472	-	518
v*alpha*	27	96	2	67	42	232
v*p*alpha*	-	2	-	-	1	3
Beta	456	555	42	942	132	2 085
beta*	91	3	2	68	2	164
semver-beta*	158	72	25	480	-	710
semver-beta*.*	27	3	3	10	-	40
v*.beta	3	-	11	364	-	367
v*beta*	24	383	2	20	119	546
semver (beta)	-	7	-	-	-	7
v*p*beta*	-	36	-	-	11	47
Release Candidate	370	64	266	1 065	0	1 499
semver-rc*	176	5	9	107	-	284
rc*	19	1	1	15	-	35
v*rc*	15	2	1	10	-	27
semver-rc*.*	160	56	255	933	-	1 353

Table 9.4. Number of artifacts with version identifiers used in metadata of preview releases in each of the study datasets and all combined

9.4.2 URL-based versioning

URL-based versioning is a method of version control for web APIs where the API version is incorporated directly into the URL structure. This version information can be embedded either in the API paths or within the DNS names. When versioning is integrated into paths, it's typically appended as a segment in the URL:

`api.example.com/v1/resource)`

Alternatively, with DNS-based versioning, the version is encompassed in the subdomain or domain:

`v1.api.example.com)`

The deployment implications of these approaches vary. Path-based versioning grants greater flexibility and straightforward resource grouping. However, it can potentially lead to longer URLs as versions accumulate. DNS-based versioning, on the other hand, offers cleaner URLs and enables physical separation of versioned APIs, but requires more elaborate DNS configurations and management.

9.4.3 Path-based versioning

Path-based versioning adoption overview

Dataset	Location	#APIs	#Unique	Most Used	#APIs
SwaggerHub	Only paths	57 518 (15,11 %)	464	v1	27 390
	Paths + Servers	72 951 (19,16 %)	957	v1	30 193
BigQuery	Only paths	6 001 (13,20 %)	203	v1	1 947
	Paths + Servers	7 599 (16,71 %)	261	v1	2 605
GitHub	Only paths	1 428 (27,37%)	124	v1	539
		39 860 commits (23,93 %)			16 519 commits
	Paths + Servers	1 793 (34,36%)	180	v1	861
		51 113 commits (30,80 %)			24 055 commits
APIs.guru	Only paths	935 (23,49 %)	138	v1	373
	Paths + Servers	1 381 (34,69 %)	186	v1	480

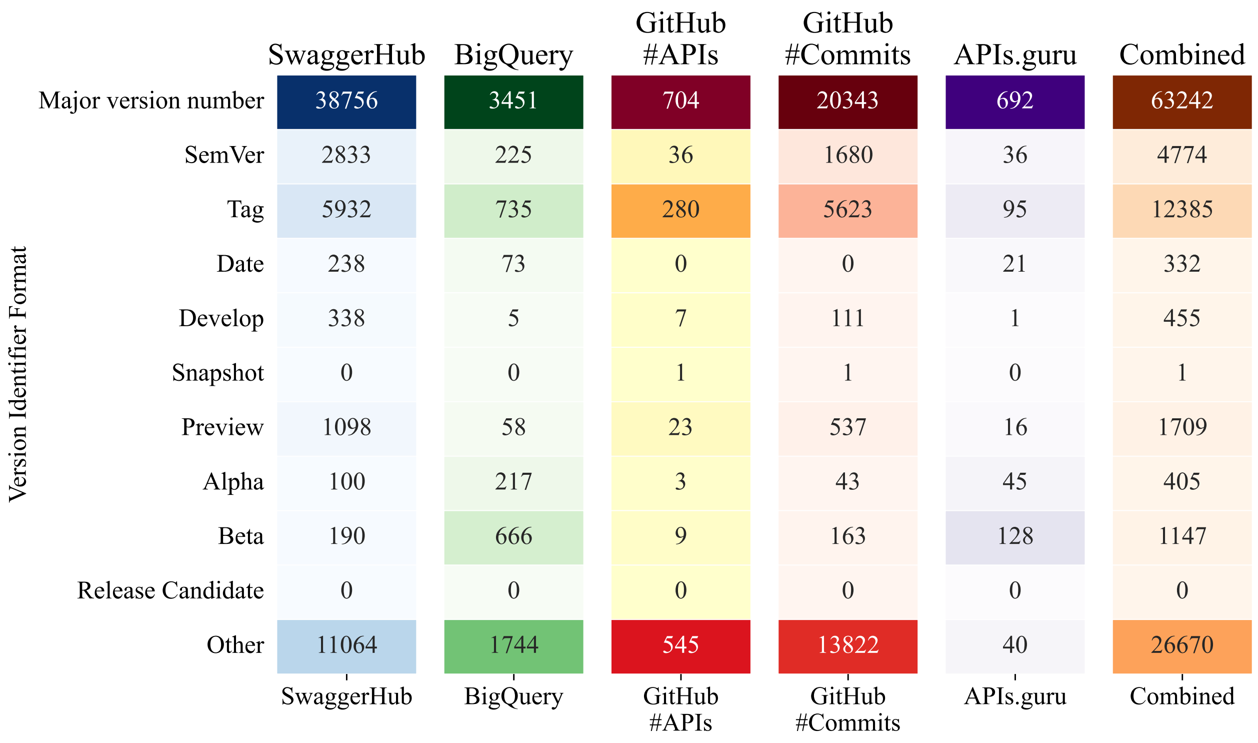
Table 9.5. Number of artifacts featuring Path-based versioning across datasets

Table 9.5 provides an overview of the adoption of path-based versioning across the four datasets, showing that the most commonly used version identifier in path-based versioning is 'v1'. The table differentiates between APIs that include version identifiers within each individual path and those that employ a global identifier attached to the server URL. The latter is located within the `servers` field.

• In the SwaggerHub dataset, 15.11% of APIs use path-based versioning with version identifiers in individual paths, and this percentage increases to 19.16% when considering APIs that also use a global identifier in the server URL.

• The BigQuery dataset shows a similar trend, with 13.20% of APIs using path-based versioning in individual paths, and 16.71% when including APIs with a global identifier.

• The GitHub dataset shows a higher adoption rate of path-based versioning, with 27.37% of APIs using version identifiers in individual paths at some point in their history, and 34.36% when considering APIs with a global identifier.



Version Identifier Format	SwaggerHub	BigQuery	GitHub #APIs	GitHub #Commits	APIs.guru	Combined
Major version number	38756	3451	704	20343	692	63242
SemVer	2833	225	36	1680	36	4774
Tag	5932	735	280	5623	95	12385
Date	238	73	0	0	21	332
Develop	338	5	7	111	1	455
Snapshot	0	0	1	1	0	1
Preview	1098	58	23	537	16	1709
Alpha	100	217	3	43	45	405
Beta	190	666	9	163	128	1147
Release Candidate	0	0	0	0	0	0
Other	11064	1744	545	13822	40	26670

Table 9.6. Number of artifacts with Path-based versioning of stable and preview releases Usage of one or multiple format categories in path-based versioning of APIs with multiple versions in production in each dataset and all combined

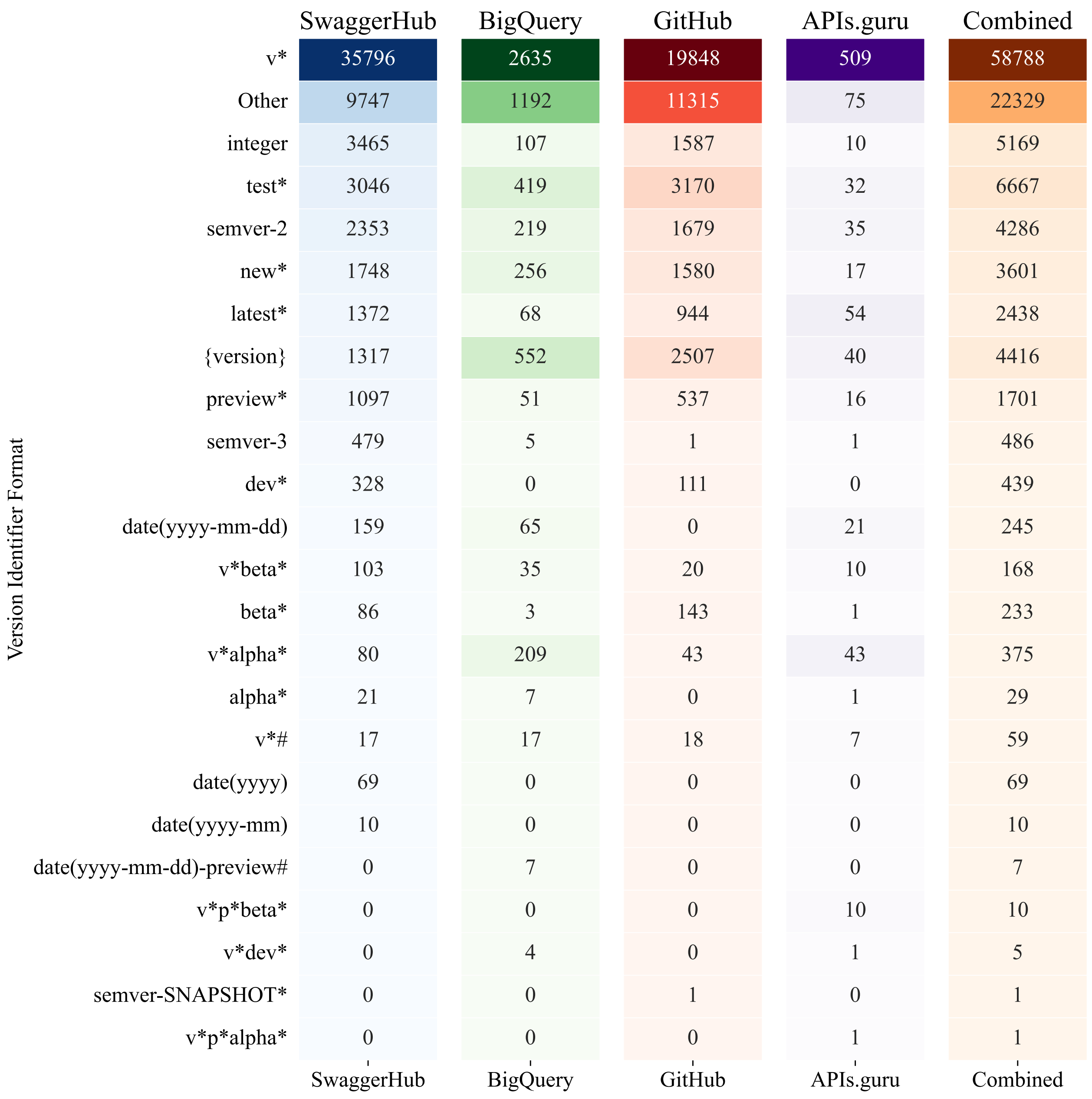
- The APIs.guru dataset also shows a substantial adoption of path-based versioning, with 23.49% of APIs using version identifiers in individual paths, and 34.69% when including APIs with a global identifier.

Path-based versioning identifiers formats

The results in Table 9.6 provide an overview of the most adopted version identifier formats appearing in paths across the study datasets. The table presents the formats along with their occurrence in each dataset, expressed as a percentage of the total number of APIs in the respective dataset.

The most common format across all datasets is v∗, which represents a version number prefixed with the letter 'v'. This format is prevalent in all datasets, with the highest adoption in the GitHub dataset (11.96%), followed by SwaggerHub (8.84%), APIs.guru (10.07%), and BigQuery (5.32%).

The "integer" format, denoting a numeric version identifier, is commonly utilized, particularly in the SwaggerHub dataset. Additionally, the "test*" format, likely indicative of versions used for testing, is prevalent in both the SwaggerHub and BigQuery datasets. Semantic versioning is a favored approach across all datasets, with "semver-2" and "semver-2#" formats frequently used. The "latest*" format, suggesting the most recent API version, appears less frequently in all datasets. The "version" format, acting as a placeholder for version identifiers, is employed across all datasets but with lower frequency. Notably, the "preview*" format, signifying non-finalized versions, is present in all datasets except BigQuery, while the "alpha*" and "beta*" formats, representing early version stages, see lesser usage across all datasets.



Version Identifier Format	SwaggerHub	BigQuery	GitHub	APIs.guru	Combined
v*	35796	2635	19848	509	58788
Other	9747	1192	11315	75	22329
integer	3465	107	1587	10	5169
test*	3046	419	3170	32	6667
semver-2	2353	219	1679	35	4286
new*	1748	256	1580	17	3601
latest*	1372	68	944	54	2438
{version}	1317	552	2507	40	4416
preview*	1097	51	537	16	1701
semver-3	479	5	1	1	486
dev*	328	0	111	0	439
date(yyyy-mm-dd)	159	65	0	21	245
v*beta*	103	35	20	10	168
beta*	86	3	143	1	233
v*alpha*	80	209	43	43	375
alpha*	21	7	0	1	29
v*#	17	17	18	7	59
date(yyyy)	69	0	0	0	69
date(yyyy-mm)	10	0	0	0	10
date(yyyy-mm-dd)-preview#	0	7	0	0	7
v*p*beta*	0	0	0	10	10
v*dev*	0	4	0	1	5
semver-SNAPSHOT*	0	0	1	0	1
v*p*alpha*	0	0	0	1	1

Figure 9.6. Most frequently adopted version identifier formats appearing in Path in each of the study datasets and all combined

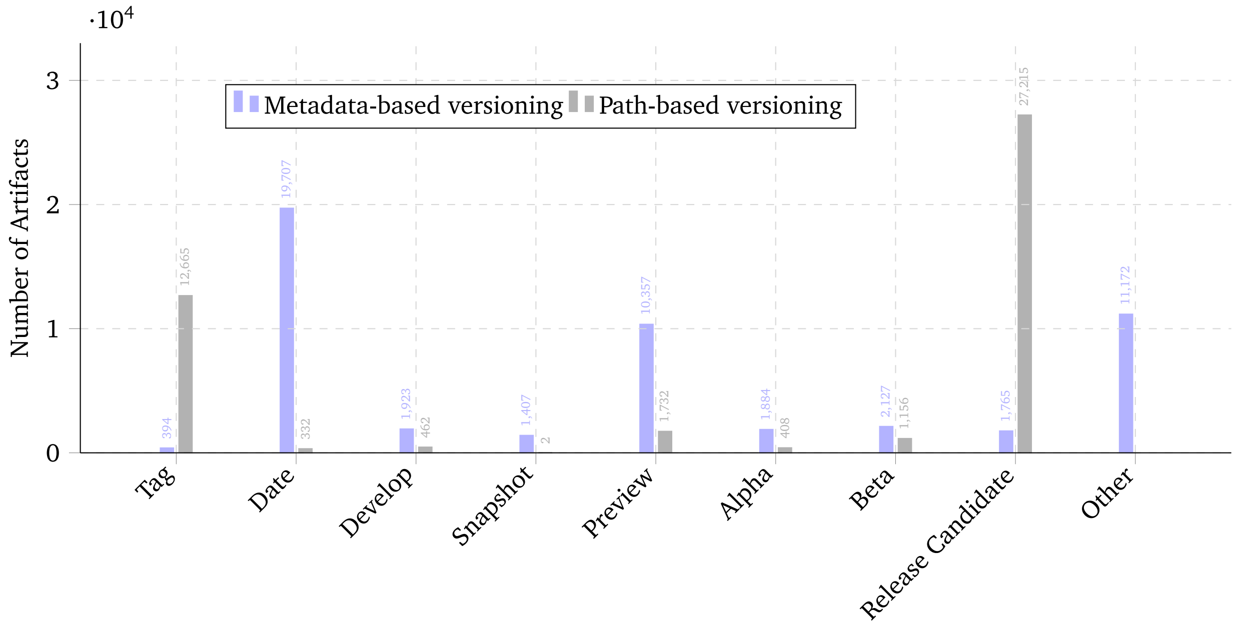


Figure 9.7. Comparing the adoption of the least used formats classed in metadata-based and path-based versioning in all datasets combined. (Semantic Versioning and Major Version Number have been omitted).

We systematically classified the identified versioning formats into distinct categories, distinguishing between stable and unstable release classes. The heatmap is Table 9.6 provides a detailed breakdown of the version identifiers used in stable and preview releases across the study datasets. The table categorizes the identifiers into different formats. The table presents the number of artifacts with each format in each dataset.

From Table 9.6, it is evident that “Major version number" and “Tag" are the most commonly used formats in stable releases across all datasets (Table 9.7). This suggests a preference for these formats in stable releases, possibly due to their simplicity and straightforwardness.

In preview releases (Table 9.8), the “Preview", “Alpha", and “Beta" formats are more prevalent. This indicates that these formats are commonly used to denote early stages of the version lifecycle, where the API is still under development and not yet finalized.

Regarding the “Others" category in Table 9.6 and Figure 9.7, it is worth noting that this category includes version identifiers that do not fit into any of the predefined formats. The high number of artifacts with the “Others" format suggests a diverse range of versioning practices across the study datasets or the fact that the paths are long enough to make it more probable to detect identifiers that are not meant to be used for versioning purposes. This applies also to the case of the “Tag” format.

Further analysis is required to understand the specific characteristics and patterns within the “Others" class. This could involve examining individual API documentation or conducting interviews with API developers to gain insights into their usage purpose.

9.4.4 DNS-based versioning

In DNS-based versioning, the version information is included in the server DNS name.

Table 9.9 presents the number of APIs that use DNS-based versioning across the four datasets, showing that DNS-based versioning is not as widely adopted as path-based versioning.

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Major version number	38 756	3 451	704	20 343	692	63 946
integer	3 465	107	48	1 587	10	5 217
v*	35 796	2 635	668	19 848	509	59 456
v*#	17	17	4	18	7	63
SemVer	2 833	225	36	1 680	36	4 810
semver-2	2 353	219	36	1 679	35	4 322
semver-3	479	5	1	1	1	487
semver-2#	4	1	0	0	0	5
semver-4	1	0	0	0	0	1
Tag	5 932	735	280	5 623	95	12 665
latest*	1 372	256	58	944	54	2 684
new*	1 748	68	88	1 580	17	3 501
test*	3 046	419	141	3 170	32	6 808
Date	238	73	0	0	21	332
date(yyyy-mm-dd)	159	65	0	0	21	245
date(yyyy)	69	0	0	0	0	69
date(yyyy-mm)	10	0	0	0	0	10
date(yyyymmdd)	1	0	0	0	0	1
date(yyyy-mm-dd)-#	0	1	0	0	0	1

Table 9.7. Number of artifacts with Path-based versioning of stable releases in each of the study datasets

Our analysis revealed that not many of the APIs documentation follow the standard URL format RFC 3986. In OpenAPI formats without DNS name, such as "`/v1/users`" or "`/`" are valid values for server URLs.

The low adoption rate of DNS-based versioning showed in Table 9.9 can be attributed to the fact that not all developers use a URL with the DNS name in the server field.

9.4.5 Header-based versioning

In Table 9.10 and Figure 9.8, we analyzed the approach's prevalence across our four datasets, investigated the header names used to denote API versions, and identified the most common ones.

The results of our analysis indicate that the header-based versioning approach is not as prevalent as path-based versioning. A wide variety of header names used in the SwaggerHub dataset (Figure 9.8), indicating a lack of standardization among the APIs adopting that practice.

In Figure 9.8 we depict the adoption of header-based versioning in the SwaggerHub dataset over the years.

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Develop	338	5	7	111	1	462
dev*	328	1	7	111	-	447
develop*	10	-	-	-	-	10
v*dev*	-	4	-	-	1	5
Snapshot	0	0	1	1	0	2
semver-SNAPSHOT*	-	-	1	1	-	2
Preview	1 098	58	23	537	16	1 732
preview*	1 097	51	23	537	16	1 724
date(yyyy-mm-dd)-preview#	-	7	-	-	-	7
semver-preview*.*	1	-	-	-	-	1
Alpha	100	217	3	43	45	408
v*alpha*	80	209	3	43	43	378
alpha*	21	7	-	-	1	29
v*p*alpha*	-	1	-	-	1	2
Beta	190	666	9	163	128	1 156
beta*	86	3	5	143	1	238
v*beta*	103	628	4	20	117	872
v*p*beta*	-	35	-	-	10	45
v*.beta	1	-	-	-	-	1
Release Candidate	0	0	0	0	0	0

Table 9.8. Number of artifacts with Path-based versioning of preview releases in each of the study datasets and all combined

9.4.6 Dynamic versioning

In this chapter we examine the prevalence of APIs that offer endpoints for retrieving either the current version or the list of available versions of the API. Our work also delves into understanding the potential correlations between the adoption of dynamic versioning strategies and the utilization of header-based versioning in real-world APIs. In Table 9.11 we show the number of APIs having one endpoint dedicated to fetch the current version/versions of the API.

The `GET /version` endpoint, which retrieves the current version of the API, is more prevalent across all datasets. In the SwaggerHub dataset, 1185 APIs provide this endpoint, while in the BigQuery, GitHub, and APIsguru datasets, 435, 67, and 11 APIs provide this endpoint, respectively. On the other hand, the `GET /versions` endpoint, which retrieves the list of available versions of the API, is less common. In the SwaggerHub dataset, only 153 APIs provide this endpoint. Similarly, in the BigQuery, GitHub, and APIsguru datasets, only 17, 4, and 8 APIs provide this endpoint, respectively. These results suggest that while some APIs provide dynamic versioning capabilities, the majority of APIs prefer to provide only the current version information. This could be due to the simplicity and lower maintenance overhead of only managing a single current version. However, providing a list of available versions can offer more flexibility to the clients, allowing them to choose the most suitable version for their needs.

Figures 9.9 and 9.10 illustrate the adoption of dynamic versioning over the years in the Github and SwaggerHub datasets, respectively. Figure 9.9, the adoption of the `GET /version` endpoint in the Github dataset has been relatively very low and non-stable over the years. On the other hand, the adoption of the `GET /versions` endpoint has been minimal, with a small increase starting from 2020. In Figure 9.10, we can observe that the adoption of

Dataset	#APIs	#Distinct Version IDs	Most used ID	Occurrence (#APIs)
GitHub	3 (215 commits)	1	`{version}`	3
BigQuery	21	2	`{version}`	19
SwaggerHub	64	8	`{version}`	57
APIs.guru	10	1	`{version}`	10

Table 9.9. Number of APIs with DNS-based versioning

BigQuery	#APIs	**APIs.guru**	#APIs	**GitHub**	#APIs	#Commits
x-ms-version	6	x-ms-version	3	basiq-version	3	180
x-api-version	2	x-readme-version	1	cdi-version	1	39
x-ph-api-version	1	trakt-api-version	1	x-myobapi-version	1	12
x-amz-fwd-header-x-amz-version-id	1	x-amz-fwd-header-x-amz-version-id	1	apiversion	1	6
accept-version	1	x-je-api-version	1	version	1	3
-	-	zuora-version	1	x-api-version	1	30

Table 9.10. Adoption of header-based versioning across the study datasets: BigQuery, GitHub, and APIs.guru.

the `GET /version` endpoint in the Github dataset has been relatively very low and stable over the years. On the other hand, the adoption of the `GET /versions` endpoint has been minimal all the time.

It is also worth noting that the adoption of dynamic versioning strategies does not seem to correlate with the use of header-based versioning. We anticipated discovering a correlation between the utilization of header-based versioning and dynamic versioning. However, our analysis revealed that this correlation was relatively scarce, with only seven APIs (comprising six from BigQuery and one from SwaggerHub) where these two practices were employed concurrently.

We looked at the correlation between the usage of dynamic versioning and query parameters where the clients can send the version identifier. Within the APIs that feature dynamic versioning, we found 146 APIs in SwaggerHub, 320 APIs in BigQuery, 12 APIs in GitHub dataset (223 commits), and three APIs in APIs.guru dataset that have version query parameters in at least one operation.

9.4.7 “Two in production" Evolution Pattern

We analyzed the usage of the “two in production" evolution pattern [98?] across the four study datasets by examining the APIs that have paths with distinct version identifiers.

The examination involved an analysis of specifications containing descriptions of various API versions. This analysis was predicated on the assumption that the presence of multiple versions within these specifications implied the coexistence of these API versions in a production environment.

As demonstrated in the bar charts of Figures 9.11 and 9.12, our analysis revealed the presence of 22 632 adoptions the "two in production” evolution pattern: 11 870 in SwaggerHub, 9 465 commits in GitHub, 1 139 in BigQuery, and 158 in APIs.guru collection (See Figures 9.13 and 9.14). in productions APIs across all the collections. Notably, among these APIs of having more than 2 versions concurrently active. 419 APIs from BigQuery and 219 APIs from SwaggerHub exhibited the noteworthy characteristic of using different formats for each version.

As illustrated in Figure 9.11, for APIs maintaining two versions in production, approximately 51% employ the Major Version Number as the format for the version identifier within the paths. This rate of adoption remains consistent for scenarios involving three to six concurrent versions. Beyond this range, the Major Version Number becomes the sole versioning format utilized.

Header Name	**#APIs**
api-version	84
x-api-version	73
app-version	22
x-app-version	17
Accept-version	13
appversion	9
accept-version	8
x-version	8
app_version	6
x-version-api	5
126 Distinct Names	**549**

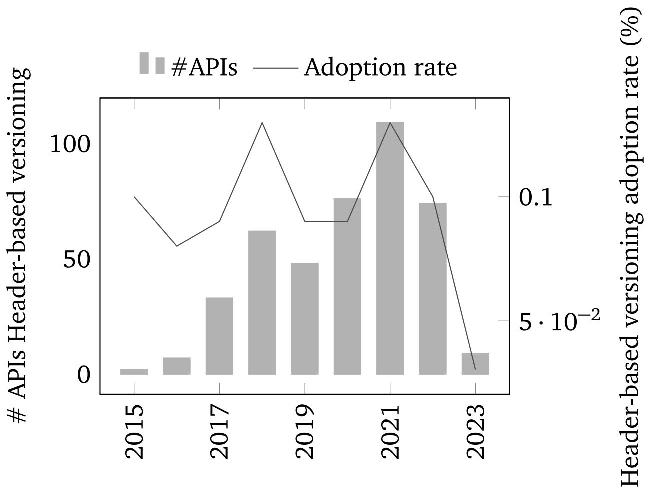


Figure 9.8. Header-based adoption in SwaggerHub Dataset over the year over the years

Endpoint	SwaggerHub	BigQuery	GitHub		APIsguru
			#API	#Commits	
GET /version	1185	435	67	2585	11
GET /versions	153	17	4	438	8

Table 9.11. Number of artifacts where dynamic version information endpoints is detected

The results presented in Figure 9.13 provide insights into the usage of multiple format categories in path-based versioning of APIs that adopt the "two in production" evolution pattern in each dataset sparately, where we can see that the adoption of Major Version Number slightly differs in the case of BigQuery.

In the APIs.guru dataset, the majority of APIs (79) use only one format category, while a smaller number (28) use two format categories. Only a very small number of APIs (2) use three or more format categories.

A similar pattern is observed in the BigQuery dataset, with a majority of APIs (563) using one format category, a smaller number (320) using two format categories, and a very small number (8) using three or more format categories.

In the GitHub dataset, the majority of APIs (5 120) use one format category, while a smaller number (902) use two format categories. Only a very small number of APIs (31) use three or more format categories.

In the Swagger dataset, the majority of APIs (6641) use one format category, while a smaller number (1339) use two format categories. A slightly larger number of APIs (87) in this dataset use three or more format category compared to the other datasets.

Wile the use of multiple format categories in path-based versioning is not uncommon, the majority of APIs prefer to use a single format category. This could be due to the simplicity and consistency offered by using a single format category.

In Figure 9.12, we quantify the number of APIs employing precisely one, two, or three or more format combinations for APIs with more than one version in production. It is evident that, in the majority of instances, APIs tend to use no more than one format for versioning.

The results presented in Figure 9.14 provide a comprehensive overview of the version formats used in APIs that

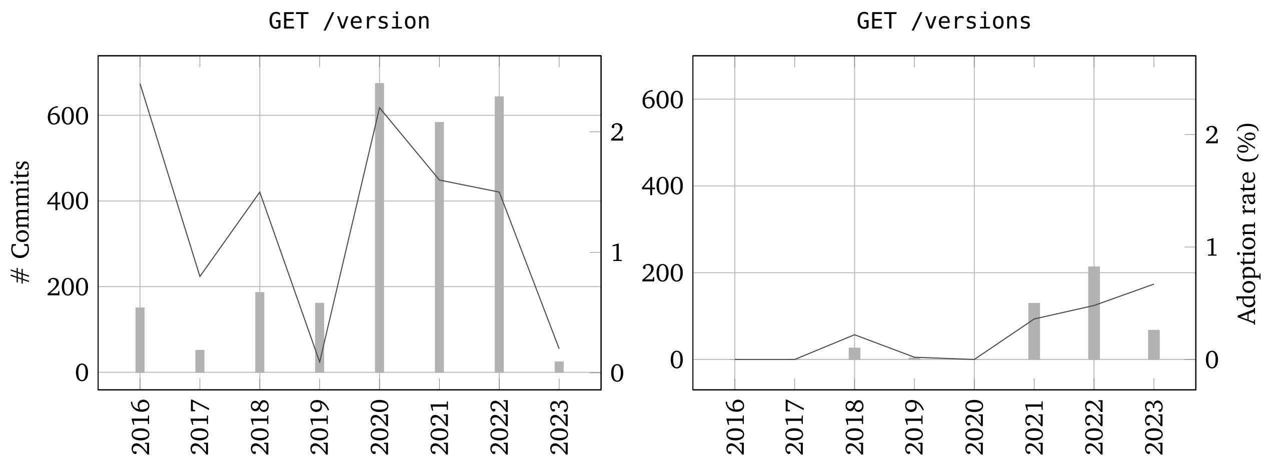


Figure 9.9. Dynamic versioning over the years in Github Dataset

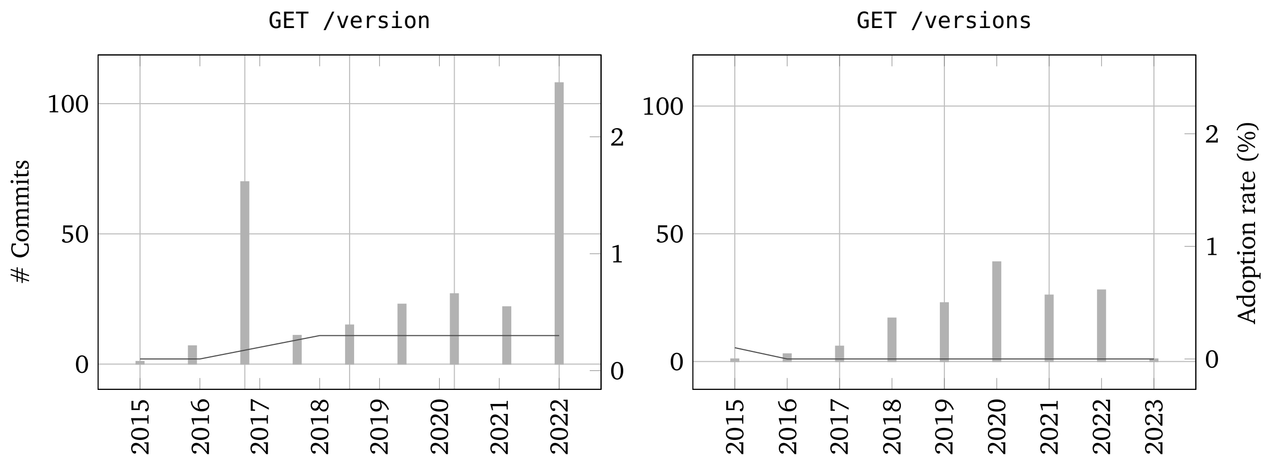


Figure 9.10. Dynamic versioning over the years in SwaggerHub Dataset

adopt the "two in production" evolution pattern in each of the datasets separately.

In the APIs.guru dataset, the majority of APIs (4470) use the "Major version number"format, while a smaller number (3511) use other formats. This trend is also observed in the BigQuery dataset, with a majority of APIs (2611) using the "Major version number"format, and a smaller number (3411) using other formats.

In the GitHub dataset, a similar pattern is observed, with a majority of APIs (199) using the "Major version number"format, and a smaller number (692) using other formats. However, in the Swagger dataset, the use of the "Major version number"format (4470) is almost equal to the use of other formats (3511).

These results suggest that while the `Major version number` format is the most commonly used format in path-based versioning, a significant number of APIs also use other formats. This could be due to the flexibility and adaptability offered by these other formats, allowing API developers to tailor their versioning strategy to the specific needs and requirements of their API.

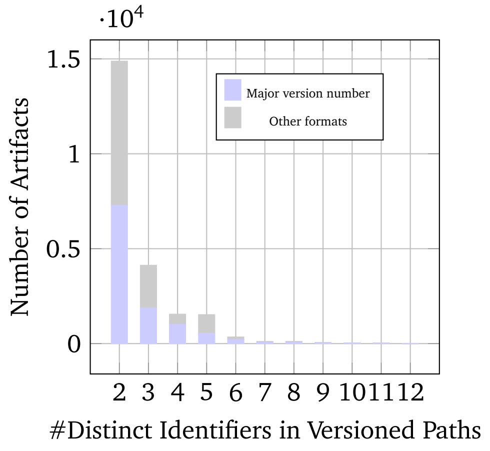


Figure 9.11. The adoption of major version number vs other formats in identifiers found in the paths of APIs with multiple versions in production in all datasets combined

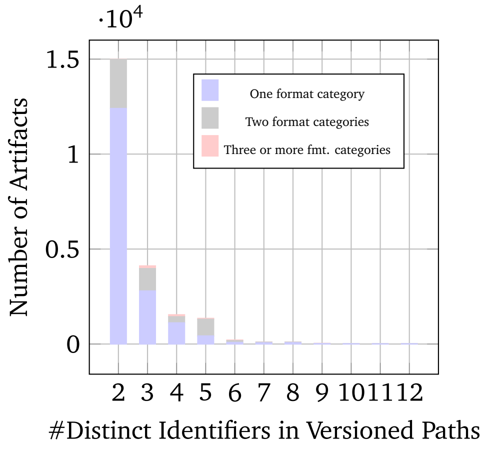


Figure 9.12. Usage of one or multiple format categories in path-based versioning of APIs with multiple versions in production in all datasets combined

9.4.8 Version Formats adoption over the years

Figure 9.15 shows that in 2015 Semantic Versioning (SemVer) held sway as the predominant versioning format, constituting the choice in 59% of the analyzed APIs, while the utilization of the Major Version Number format accounted for 38.11% of the cases. However, an observable shift occurred in the subsequent years. Notably, there was a conspicuous decline in the adoption of the simplified format, characterized by solely the major version number, accompanied by a notable resurgence in SemVer adoption during the year 2017. Nonetheless, the substantial surge in SemVer adoption observed in 2017 was not sustained in the subsequent years. Instead, SemVer's adoption exhibited a relatively stable trajectory over the years, punctuated by occasional slight declines noted in 2021 and 2022.

9.5 Results structuring

Q1: *What are the commonly adopted practices for Web APIs versioning?*
Based on our analysis investigating the adoption of the versioning practices: metadata-based, URL-based, header-based, and dynamic versioning, we detected the usage of these practices with different frequencies across the four study datasets. Metadata-based versioning, where the version information is included in the API metadata, is prevalent in 98% of the APIs. This practice is favored due to its simplicity and the ease of managing version information in a centralized location. URL-based versioning, where the version information is included in the URL of the API, is adopted in 26.23% of the APIs. This practice offers the advantage of making the version information immediately visible and accessible to the clients. Header-based versioning, where the version information is included in the HTTP headers, is less common, being used in only 0.17% of the APIs. This could be due to the additional complexity it introduces in managing version information. Lastly, dynamic versioning, where the version information is discovered dynamically at runtime, is used in a minority of APIs (1088 APIs in all datasets).

Q2: *How do developers distinguish stable from preview releases?*
Developers distinguish between stable and preview releases primarily through the use of specific version identifier formats. Our analysis of the study datasets revealed that the“Major version number" and “Tag" formats are the

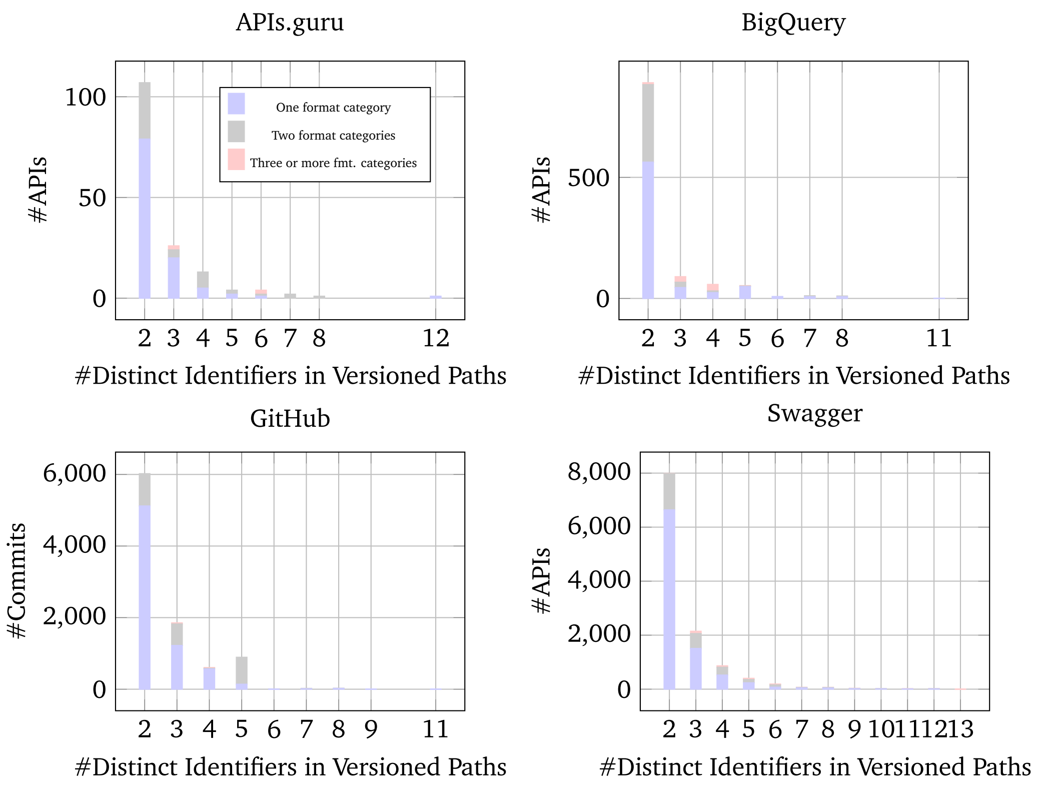


Figure 9.13. Usage of one or multiple format categories in path-based versioning of APIs with multiple versions in production

most commonly used in stable releases across all datasets. This suggests a preference for these formats in stable releases, possibly due to their simplicity and straightforwardness. In contrast, the “Preview", “Alpha", and “Beta" formats are more prevalent in preview releases. This indicates that these formats are commonly used to denote early stages of the version lifecycle, where the API is still under development and not yet finalized. For instance, in the SwaggerHub dataset, the “semver-beta*.*" format was used in 27 APIs, the “v*.beta" format in 3 APIs, and the “v*beta*" format in 24 APIs.

Q3: *To what extent is the practice of semantic versioning adopted in Web APIs, and are there alternative versioning schemes in use?*

Semantic versioning (SemVer) is found to be widely adopted practice in Web APIs. Our analysis shows that in 2015, SemVer was the predominant versioning format, used in 59% of the analyzed APIs. However, its adoption has seen some fluctuations over the years. For instance, in 2017, there was a significant increase in SemVer adoption, reaching 89.12% of the APIs. However, this surge was not sustained in the subsequent years, with a slight decline noted in 2021 and 2022. Despite these fluctuations, SemVer remains a popular choice, with its adoption rate in 2023 standing at 87.60%.

In terms of alternative versioning schemes, the Major Version Number format is the second most common, used in 38.11% of APIs in 2015. However, its adoption has seen a decline over the years, dropping to 5.70% in 2023. Another alternative is the Date format, which, although less common, has seen a tiny slight increase in adoption, from 0.17% in 2015 to 0.36% in 2023.

These findings suggest that while SemVer is the most prevalent versioning scheme, there is a diversity of practices

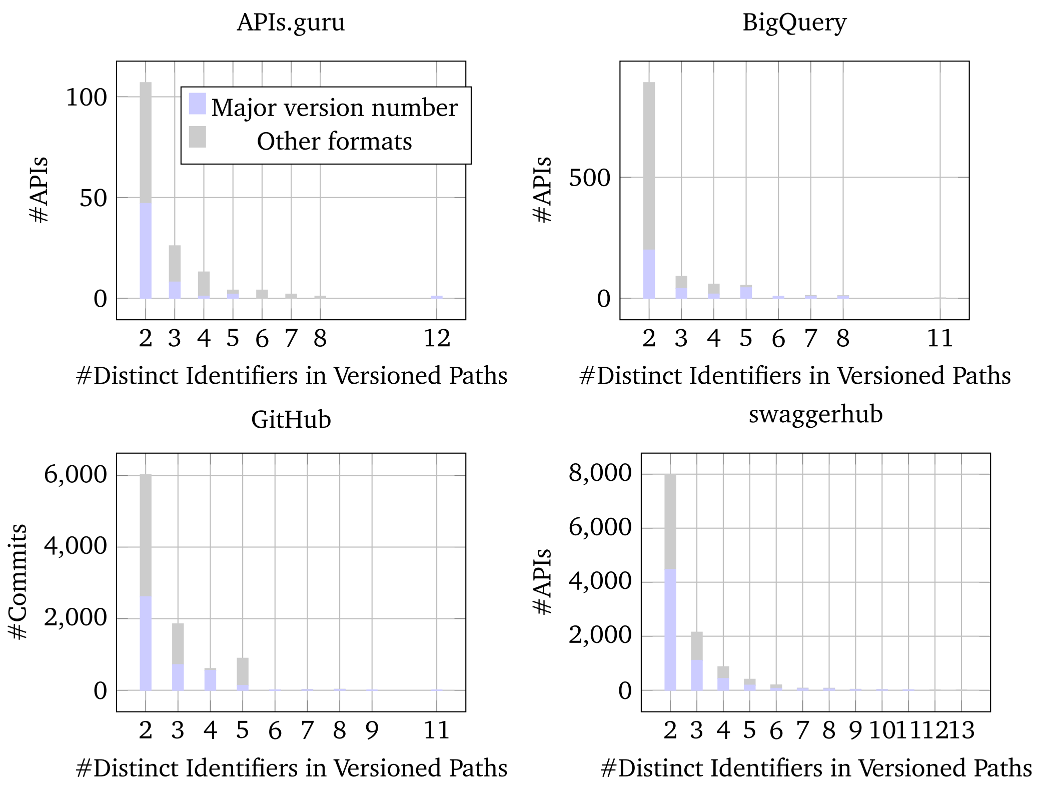


Figure 9.14. Comparing the adoption of major version number vs other formats in identifiers found in the paths of APIs with multiple versions in production

in Web API versioning, with some APIs opting for alternative schemes such as the Major Version Number or Date formats.

Q4: *What is the prevalence of APIs with multiple versions in production? how many concurrent versions exist?* Our analysis shows the presence APIs have multiple versions in production concurrently in all the dataset. Specifically, 14.29% of the APIs in the SwaggerHub dataset, 5.50% in the BigQuery dataset, 6.99% in the GitHub dataset, and 3.96% in the APIs.guru dataset have multiple versions in production.

In terms of the number of concurrent versions, our analysis reveals a wide range. The majority of APIs with multiple versions in production have between 2 to 5 concurrent versions. However, there are also APIs with a high number of concurrent versions. For instance, in the SwaggerHub dataset, the maximum number of concurrent versions found in an API is 13. This suggests that some APIs maintain a large number of versions in production, possibly to cater to a wide range of clients with different version requirements.

Q5: *How has the adoption of dynamic versioning and header-based versioning practices evolved over time?* Our analysis reveals interesting trends in the adoption of dynamic versioning and header-based versioning practices over time. Dynamic versioning, despite its potential benefits of flexibility and adaptability, is used in a minority of APIs across all datasets. This could be attributed to the additional complexity and overhead associated with managing dynamic version information.

On the other hand, header-based versioning, where the version information is included in the HTTP headers, is even less common. This could be due to the additional complexity it introduces in managing version information. However, it is worth noting that the adoption of dynamic versioning strategies does not seem to correlate with the

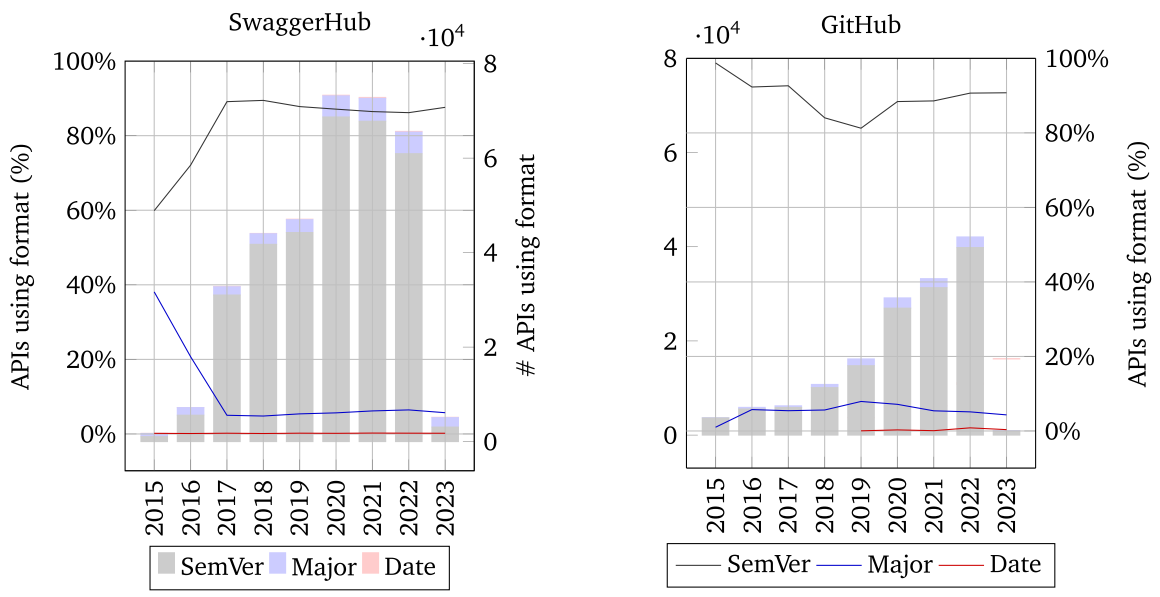


Figure 9.15. Adoption of Semantic Versioning, Major version number and Date formats over the years

use of header-based versioning. We anticipated discovering a correlation between the utilization of header-based versioning and dynamic versioning. However, our analysis revealed that this correlation was relatively scarce, with only seven APIs (comprising six from BigQuery and one from SwaggerHub) where these two practices were employed concurrently.

Although there is a diversity of practices in Web API versioning, the adoption of dynamic and header-based versioning practices has remained relatively low over the years. This highlights the need for more research to understand the factors that influence these adoption trends.

Q6: *How sensitive are the results to the source of the API descriptions?*

The results show some sensitivity to the source of the API descriptions. For instance, the adoption rate of Semantic Versioning (SemVer) in the SwaggerHub dataset was 87.60% in 2023, while in the GitHub dataset, it was 90.77%. Similarly, the adoption rate of the Major Version Number format was 5.70% in the SwaggerHub dataset and 4.32% in the GitHub dataset in 2023.

Figure 9.16, depict the adoption of identifiers format in Metadata-based versioning in each of the four dataset. The adoption dominance of each format follows the same order in the case of BigQuery and APIs.guru datasets, where the most common one is “Date”, followed by “Semver". Another particular noticed aspect is the presence of a relatively hight number of “Preview" identifiers.

The higher prevalence of the “Preview" identifiers in the BigQuery and APIs.guru datasets could be indicative of the experimental nature of many APIs on these data sources.

On the other hand, the SwaggerHub and GitHub datasets show a higher prevalence of the “SemVer" identifiers.

In terms of the “Major Version Number" format, its adoption is relatively consistent across all all of SwaggerHub, GitHub and BigQuery datasets, ranging from 5.82% in the GitHub dataset to 6.25% in GitHub. While in APIs.guru dataset, the adoption rate of this format goes up to 12.83%.

Figure 9.17 shows the the common traits between versioning formats in some of the datasets in Meta-data-based versioning are not present in the case of Path-based versioning.

These differences suggest that while the overall trends in versioning practices are similar across different sources of API descriptions, there are some variations in the specific adoption rates. This could be due to differences in the communities of developers contributing to these sources, their preferences, and their familiarity with different versioning schemes.

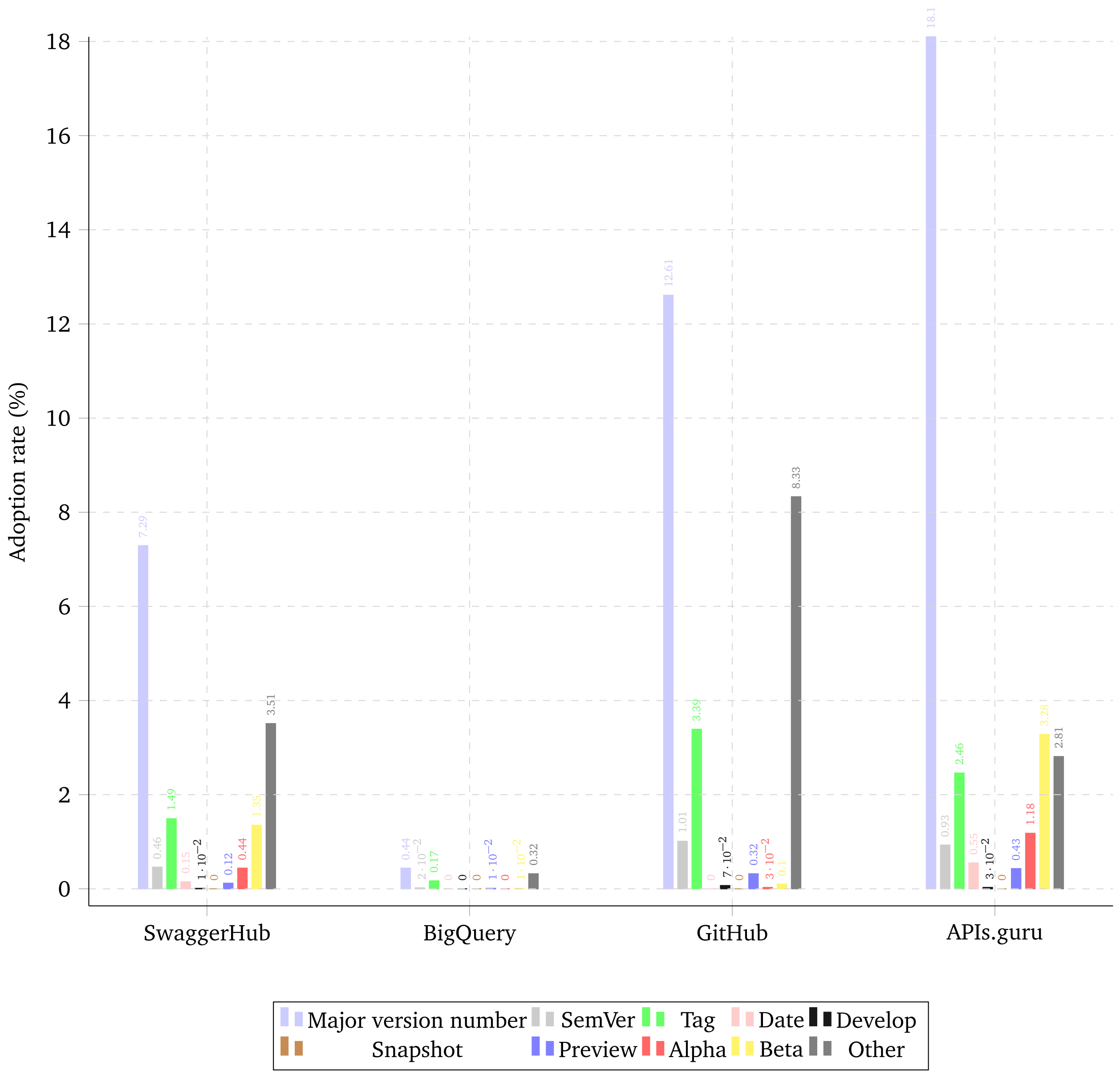


Figure 9.17. Adoption of different format categories in each dataset in Path-based versioning

9.6 Web API versioning in OpenAPI 4.0: proposal

The diversity of formats found in in the `info.version` field of the OpenAPI specification is due to the way to document API versioning in OpenAPI, which does not provide any information about the type of versioning adopted by the API, such as semantic versioning, date-based versioning, or custom versioning. It also does not explicitly support the use of multiple versions of the API specification in the same document, which can be useful for documenting deprecated or experimental features.

Introducing standardized metadata fields for version identifiers in the OpenAPI specification would significantly enhance *clarity* and *interoperability* across web APIs. By clearly defining the type of versioning adopted—be it semantic, date-based, or custom—developers and tools can more easily understand and manage API versions. This standardization would facilitate automated tools in accurately interpreting version changes, thereby improving API documentation, and would aid in the seamless integration of APIs with differing versioning schemes.

In our proposal in Listing 9.2, the existing `info.version` field would change type from string to an object that comprises the `value` field to specify the version identifier string, the schema field to document and enforce a precise, structured version format, and the upgrade field to define the version upgrade rules that should be followed, depending on the chosen format. In addition, for recording the release date, we introduce a timestamp so that the age of the API release can be tracked explicitly. Likewise, tags can represent the lifecycle phase to which the artifact belongs. A separate build counter can complement the timestamp so that DevOps pipelines can use a fine-grained identifier to stamp each artifact version without affecting the main version identifier.

Listing 9.2. Proposal for web API version in OpenAPI

```
version:
   semantic-identifier: 1.2.3 # Semantic version identifier value
   lifecyle: "stable" # stable, preview, rc, alpha, beta
   timestamp: YYYY-MM-DD HH:MM:SS # optionally track the API release age
   build: NNNN # an integer build counter
```

By knowing the versioning strategy (such as semantic versioning, date-based versioning, or custom versioning), API consumers can better anticipate the nature of changes and updates. This aids in planning for potential compatibility issues and migration efforts. The specification can serve as a reference point for all API development team members, making it clear how version numbers are assigned and what each increment signifies.

In Summary

Versioning in Web APIs is a fundamental practice to ensure their compatibility and ease their maintainability. In this empirical study we focused on version identifiers, observing their representation formats, static or dynamic discoverability, and purpose across 4 different datasets of 602 859 OpenAPI descriptions. The vast majority utilized static versioning in the API metadata (592 033; 98%), while only a subset include version identifiers embedded in the API endpoint path URL addresses (133 456; 22%). Only a small fraction (4219; 0.7%) supported dynamic discovery of the current version through a dedicated endpoint.

In terms of version format, we identified 10 221 distinct version identifiers and 59 distinct formats (28 stable, 30 preview, and other) used to distinguish stable and preview releases, with 23 251 pre-release versions across different stages of the API release lifecycle. While most APIs use semantic version identifiers to indicate the expected impact on clients of changes with respect the previous version, a few instead use version identifiers to track the age of the API.

We also observed the usage of the “two in production” evolution pattern in 13501 APIs (4 250 with more than 2 versions): 158 in APIs.guru, 1139 in BigQuery, 365 (with 9 465 commits) in GitHub, and 11 839 in SwaggerHub. In these cases, the most prevalent format for version identifiers attached to the path was to reference only the major version.

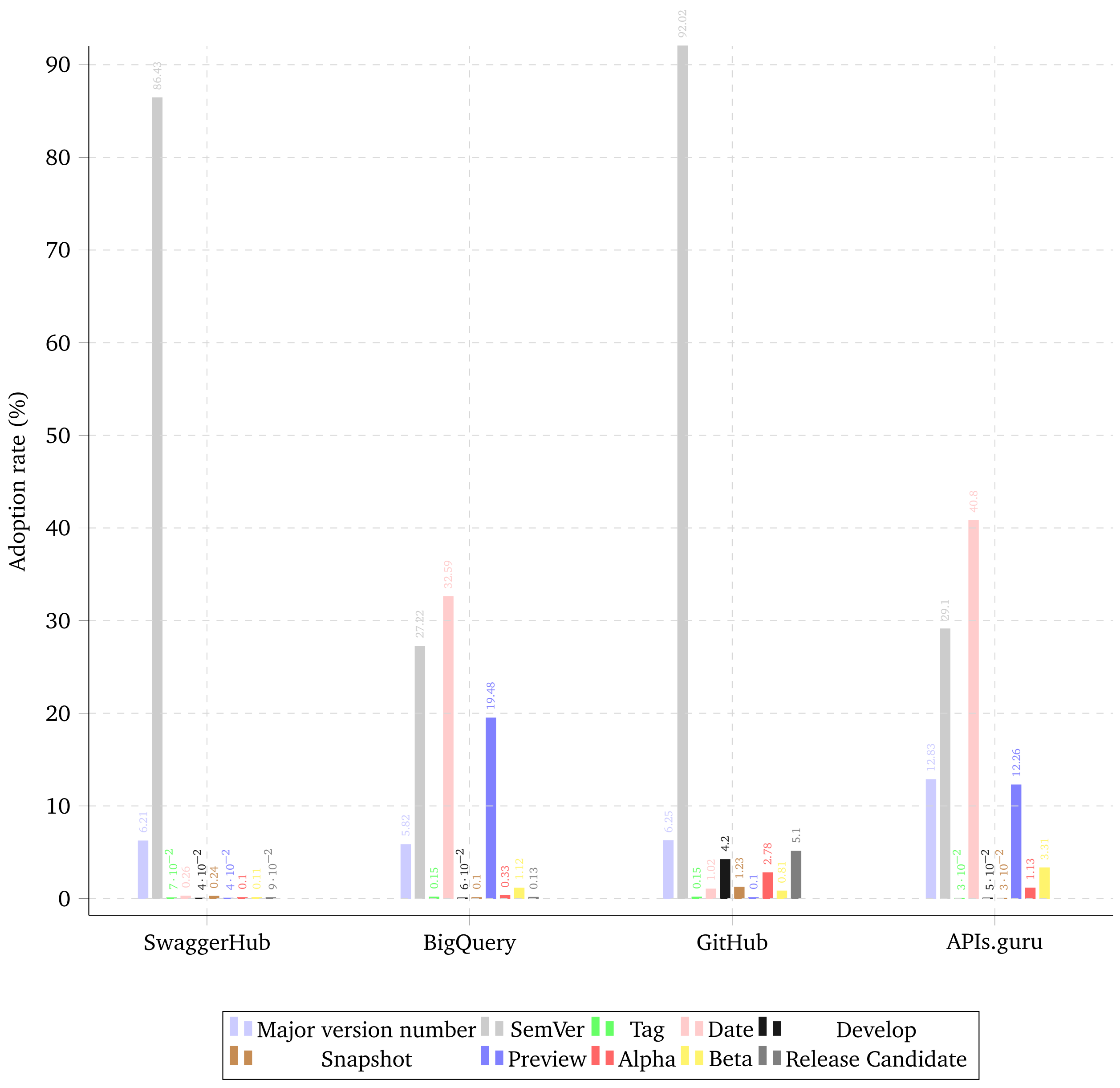


Figure 9.16. Adoption of different format categories in each dataset in Meta-data based versioning

Chapter 10

Web API Changes and Versioning Consistency

Given the public nature of Web APIs, the expectation is that their developers carefully assess the impact of every change as they strive to avoid breaking their clients. But if breaking changes are introduced, are semantic versioning rules properly followed? How often can clients rely on semantic versioning identifiers to set their expectations about the impact of new releases they depend on?

Breaking changes are found to occur approximately twice as often as non-breaking changes (Chapter 7), setting an expectation for providers to update the semantic versioning identifier accordingly.

This chapter presents the results of our analysis to evaluate this hypothesis, along with the approach we employed to carry out the evaluation.

10.1 Research Approach

In this research, we performed a data analysis method to statically classify 195 different types of changes that can be detected by comparing OpenAPI [118] descriptions and predict whether they are likely to break clients with different tolerance levels [37]. We apply the method to a collection of 3 075 API evolution histories mined from open-source GitHub repositories. The main findings are that, in the best case, 1) almost one-third of APIs in our sample (927) evolve in a backward-compatible way; 2) a minority of APIs (517) that introduce breaking changes do so by consistently adhering to semantic versioning rules.

Before filtering, the snapshot we used in this study included a snapshot containing 915 885 valid specifications from 270 578 APIs committed to GitHub between 2015 and January 2024. As described in Table 10.1, our analysis focuses on the evolutionary aspect of APIs. Therefore, we specifically looked at APIs with a history of at least 10 commits, all containing valid OAS documents. Considering the goal of this study, to examine the practical adoption of semantic versioning, we filtered for APIs that consistently use identifiers compatible with semantic versioning throughout their entire history. Additionally, to be able to check the level of compliance with semantic versioning rules, we identified APIs that have released at least one new version during their history that included some modifications impacting the functionalities of the API. As a result, our study includes the history of 3 075 APIs, with a total of 15 856 versions, corresponding to 506 273 changes introduced in their documentation.

10.1.1 Semantic Versioning Change Classification

Due to the lack of widely accepted semantics for arbitrary version identifiers, in this chapter we focus exclusively on APIs which make consistent use of semantic versioning throughout their evolution history, in both stable and preview releases.

More precisely, we analyzed API descriptions versioned with four different schemes: `X` (Major), `X.Y` (Major.Minor), `X.Y.Z` (Major.Minor.Patch), and `X.Y.Z-LABEL` (Major.Minor.Patch-Release Type). Where the release type, if present, labels the maturity of the artifact along the API release lifecycle. The version identifiers have been matched with the following regular expression:

```
        /^(?i)(v)?\d{1,3}(?:\.\d{1,3})?(?:\.\d{1,3})?(?:-LABEL ))?$/
```

Where `LABEL` can be: alpha, beta, dev, snapshot, rc, preview, test, or private.

We limit the size of the numbers to three digits because we want to avoid catching identifiers using dates [57], which are often used in versioning but do not provide a clear, incremental progression of versions, reflecting the expected change impact between releases.

Based on the previous regular expression, the parsing operation p transforms a version string into a structured tuple $(X, Y, Z, Label)$. E.g,: p(`v1`) → (1,0,0,∅) and p(`v3.0.1-alpha`) → (3,0,1,alpha).

In Table 10.2 we report the occurrence of each version identifier format.

To detect the type of semantic version change, we use a classification function c defined as follows. The function reads the tuples $V_1 = (X_1, Y_1, Z_1, Label_1)$ and $V_2 = (X_2, Y_2, Z_2, Label_2)$ representing two distinct version identifiers. It detects the following version changes:

Major (`X.y.z`): Incremented for incompatible API changes, signaling significant modifications that may require client adjustments.

$$\text{if } X_1 \neq X_2, \text{ then: } \begin{cases} \text{Major Upgrade, if } X_1 < X_2 \\ \text{Major Downgrade, if } X_1 > X_2 \end{cases}$$

Minor (`x.Y.z`): Incremented for adding backward-compatible features, indicating enhancements without breaking existing functionalities.

$$\text{if } X_1 = X_2 \text{ and } Y_1 \neq Y_2, \text{ then: } \begin{cases} \text{Minor Upgrade, if } Y_1 < Y_2 \\ \text{Minor Downgrade, if } Y_1 > Y_2 \end{cases}$$

Patch (`x.y.Z`): Incremented for backward-compatible bug fixes, often associated with routine maintenance updates.

$$\text{if } X_1 = X_2 \text{ and } Y_1 = Y_2 \text{ and } Z_1 \neq Z_2, \text{ then: } \begin{cases} \text{Patch Upgrade, if } Z_1 < Z_2 \\ \text{Patch Downgrade, if } Z_1 > Z_2 \end{cases}$$

Label change (`x.y.z-LABEL`): Updated to reflect the current (e.g., `alpha`, `beta`, `rc`) pre-release stage, indicating the API is not yet ready for production.

$$\text{if } X_1 = X_2 \text{ and } Y_1 = Y_2 \text{ and } Z_1 = Z_2, \text{ then: } \begin{cases} \text{Label Change, if } Label_1 \neq Label_2 \\ \text{No Change, if } Label_1 = Label_2 \end{cases}$$

Table 10.1. Data cleaning steps

Filtering Step	**# APIs**	**# Commits**
all valid commits	270 578	915 885
at least 10 valid commits	16 401	490 526
always use semantic versioning identifiers	14 489	413 463
have at least one version change	3 075	132 909

Table 10.2. Semantic versioning formats breakdown for all APIs adopting semantic versioning in all releases (above) and for the subset with at least one version change (below)

Format	**# APIs**	**# Commits**	**# Distinct IDs**	**Most Common**	**Occurrence**
X	1 323	41 067	49	`v1`	26 596
X.X	3 510	99 633	370	`1.0`	51 300
X.Y.Z	10 261	257 854	6 444	`1.0.0`	100 732
X.Y.Z-Label	620	10 485	645	`1.0.0-oas3`	3 078
Any	14 454	409 039	7 508		
X	193	4 114	36	`v0`	1 103
X.X	890	33 408	361	`1.0`	4 384
X.Y.Z	4 050	124 163	6 434	`1.0.0`	16 250
X.Y.Z-Label	518	7 947	638	`1.0.0-oas3`	2 138
Any	4 529	169 632	7 469		

Table 10.3. Classification of Version Changes

Version Change Type	Version Identifier Change		
Major Version Change	X → X'	X.Y → X'.Y	X.Y.Z → X'.Y.Z
Minor Version Change		X.Y → X.Y'	X.Y.Z → X.Y'.Z
Patch Version Change			X.Y.Z → X.Y.Z'
Label Version Change			X.Y.Z-label → X.Y.Z-label'

☛ The emphasis on semantic versioning is twofold: first, its primary purpose is to indicate the types of changes in terms of backward compatibility; second, our empirical analysis revealed that it is the most widely adopted versioning scheme in the context of web APIs.

10.2 Consistency Metrics

We implemented a systematic approach to assess consistency between changes detected across API releases and the corresponding types of semantic version identifier changes . The results of the analysis have been obtained by running a pipeline with the following steps (Figure 10.1).

For each API, we retrieve the complete commits history from its respective GitHub repository. We then ensure that the API meets the filtering criteria as detailed in Table 10.1. Following, we meticulously sift through the commits to isolate the ones where a version identifier change has happened. The detected version change is then classified to distinguish whether developers have made a Major, Minor, Patch-level release or simply changed the release type label. Following this, we extract the differences between the two consecutive versions of the API, we compare their respective specifications using the `oasdiff` library [115]. The extracted changes are then abstracted by matching them against the known list of 195 change types, which have been pre-classified into the Breaking, Non-Breaking, and Undecidable categories.

The outcome of the pipeline is a table listing, for all APIs and all their releases, the API version change classification with the corresponding API changes. To give a quantitative assessment of the consistency between the two according to semantic versioning rules we compute the following metrics:

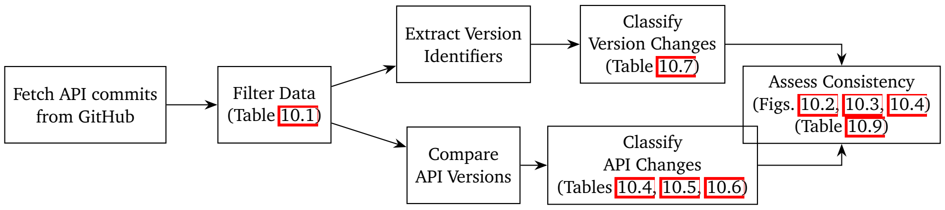

Fetch API commits from GitHub
Filter Data (Table 10.1)
Extract Version Identifiers
Classify Version Changes (Table 10.7)
Compare API Versions
Classify API Changes (Tables 10.4, 10.5, 10.6)
Assess Consistency (Figs. 10.2, 10.3, 10.4) (Table 10.9)

Figure 10.1. Data Analytics Pipeline

• Number of version changes (#VC), further subdivided into the number of Major, Minor, Patch and Label changes (#Major, #Minor, #Patch, #LC)
• Number of API changes (#C), comprising the number of breaking changes (#BC), non-breaking Changes (#NBC), undecidable changes (#UC).
• Proportion of Breaking Changes (BC%):

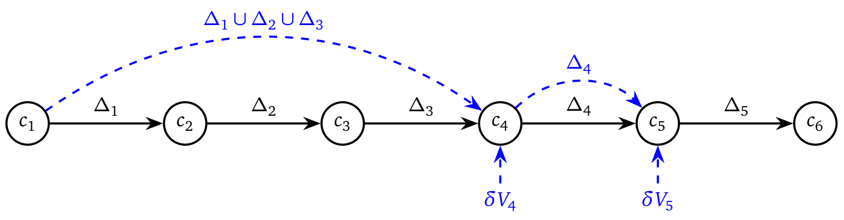
$$BC\% = \frac{\#BC}{\#C}\text{(Best Case)} \qquad BC\% = \frac{\#BC + \#UC}{\#C}\text{(Worst Case)}$$

We assess adherence to semantic versioning by examining if version updates involving at least one breaking change ($\#BC > 0$) or, in the worst-case scenario, at least one undecidable change ($\#UC > 0$), have been accurately categorized as Major. For each API, we define its compliance ratio as $CR = \frac{\#V}{\#VC}$ where $\#V$ is the number of versions which comply with semantic versioning, according to the following rules:

$$\#BC > 0 \implies \text{Major upgrade} \ \ \text{(Best Case)}$$

$$\#BC > 0 \lor \#UC > 0 \implies \text{Major upgrade} \ \ \text{(Worst Case)}$$

This definition permits developers to produce Major releases without introducing breaking changes, as the incompatibility indicated by the version identifier may be due to changes that do not visibly affect the API interface itself.
Version Change Commits Identification. For each API we identify among all the commits the the set of commits where a version change has happened. These particular commits are then used for the diff computation in order to get a coherent view of the changes that happened between the the two consecutive versions of the APIs.
Diff Computation: For each pair of successive commits with version changes, we compute the differentials between the specifications $\Delta_i = \text{diff}(c_i, c_{i+1})$. This represents the differences or modifications between commit c_i and its immediate successor c_{i+1}. To compute the diffs we use the `oasdiff`[1] library.
Changes Extraction: We abstracted all the changes that were detected in the diffs and built for each diff the corresponding list of API changes, excluding those types of changes affecting parts of the spec that have no impact on the API Strcuture, Datamodel, or Security.



$\Delta_i = diff(c_i, c_i + 1)$ and δV_j is the version change between c_3 and c_4

[1]https://github.com/Tufin/oasdiff

Classification of Changes. We manually classified 195 unique API change types by assessing their impact on clients. This meticulous process allowed us to understand how each change potentially affects client integration.

Mapping API Changes to Version Changes. Our objective is to associate these identified changes with specific version transitions, denoted as δV_j. Each δV_j represents a version change pinpointed at a particular commit. In instances where a single version change is recorded in a day, we attribute all API changes identified on that day, alongside the version change, to δV_j.

Change Metrics Computation. For every new API version, we calculated the total number of breaking, non-breaking, and undecidable changes. Additionally, we estimated the ratio of breaking changes under two scenarios: the optimistic scenario (assuming undecidable changes do not result in breaking changes) and the pessimistic scenario (assuming undecidable changes lead to breaking changes). Based on mapping between the changes and the version transitions we computed the breaking change proportions for each type of version change

10.3 Consistency Assessment Results

We present the results of the analysis at two levels of granularity. First we quantitatively study each API release independently by characterizing its type of version identifier change and the types of changes introduced in the API itself, by classifying whether they are expected to break or not break clients. This allows us to determine whether the release complies with semantic versioning. Then we proceed to aggregate each release along the history of the corresponding API. This will make it possible to classify the APIs in the dataset according to various facets: which type of changes they underwent at some release in their history, which type of version identifier change, as well as to which extent the API consistently adhered to semantic versioning throughout its entire history. The raw results are publicly shared in a replication package in GitHub.

10.3.1 Change-level compliance

Types of version changes. While the most frequently occurring type of version change (Table 10.7) is the "Patch Upgrade", "Minor Upgrades" can be found more widely across more than half the APIs in the dataset. Overall, the 14 204 Upgrades outnumber the 1 131 Downgrades. As expected, major releases are the least frequent (both concerning upgrades and downgrades). Among the 3 075 APIs, 2 198 APIs have combined at least two types of version changes during their change history. 764 has only one version change. 133 APIs have more than one version change, but they are all of the same type.

Types of API changes against types of version changes. In Table 10.4 we list the most recurrent breaking changes (out of 96). The analysis of breaking change within our dataset prominently highlights "Response property type changed" as the most frequently occurring type of change, followed by the removal of values from enumerated type definitions. The most widespread change affecting 1211 APIs at least once is the removal of paths. Path removal is the complementary change to Path addition, the most prevalent non-breaking change both according to the number of occurrences but also the number (48.14%) of impacted APIs (Table 10.5).

There is no clear correlation between the presence of specific API changes (e.g., the addition or removal of paths) and the corresponding version identifier changes (listed in the last four columns in the Tables 10.4 and 10.5). For example, the removal of paths without deprecation is detected in 246 major releases, which correctly represent the impact of such major change. However, also 629 minor and even 557 patch-level upgrades do include at least one path removal, a clear violation of semantic versioning rules.

Table 10.4. Most frequent breaking changes in the selected dataset snapshot

Breaking Change	**Occ.**	#APIs	#VC	#Major	#Minor	#Patch	#LC
Response Property Type Changed	23 048	714	872	100	335	406	31
Response Property Enum Value Removed	21 210	319	377	43	182	136	16
Path Removed Without Deprecation	15 877	1 211	1 463	246	629	557	31
Response Required Property Removed	12 587	409	547	68	244	205	30
Request Property Enum Value Removed	9 438	223	252	25	114	107	6
Path Parameter Removed	7 019	678	819	118	330	330	41
Response Media Type Removed	5 744	154	168	27	54	77	10
Response Property Pattern Changed	5 032	96	100	6	34	59	1
Response Property Became Optional	4 341	286	333	41	139	135	18
Response Property All Of Removed	4 261	185	240	27	119	87	7
Response Body Type Changed	3 906	351	380	37	170	163	10
Request Property Type Changed	3 872	448	502	40	188	259	15
Response Property Min Length Decreased	3 758	69	73	1	18	51	3
Request Required Property Added	2 524	339	394	62	169	154	9

Table 10.7. Classification of version changes (VC) indicating their occurrence (#VC), the total number of breaking, non-breaking and undecidable changes detected in conjunction with each type of version change, as well as their prevalence within all APIs and within how many APIs with breaking changes

	#VC	Upgrade	#APIs	#APIs with BC	#BC	#NBC	#Undecidable
Major	1 296	1 058	883	14 797	18 416	10 132	450
Minor	6 496	6 037	1 806	58 955	75 975	89 723	1 284
Patch	7 541	7 107	1 692	69 643	80 997	70 088	1 232
Label Change	718	N/A	345	7 938	6 512	3 085	85
Total	16 051	14 202	4 533	179 727	181 900	173 028	3 051

	#VC	#APIs	#BC	#NBC	#UC	#APIs w/BC
Patch Upgrade	7 108	1 669	63 541	77 490	67 032	1 198
Minor Upgrade	6 038	1 774	54 443	70 820	87 471	1 240
Major Upgrade	1 058	808	11 920	14 854	7 866	375
Label Change	718	345	7 938	6 513	3 085	85
Minor Downgrade	459	265	4 508	5 160	2 252	210
Patch Downgrade	434	249	6 102	3 519	3 056	210
Major Downgrade	238	163	2 877	3 560	2 266	132
Total	16 053	3 075	137 842	169 677	165 454	2 487

	#VC	#UC	#BC	#NBC	#APIs	w/BC	
					Total	Best	Worst
Patch Upgrade	7 108	67 032	63 541	77 490	1 669	1 198	1 498
Minor Upgrade	6 038	87 471	54 443	70 820	1 774	1 240	1 412
Major Upgrade	1 058	7 866	11 920	14 854	808	375	422
Label Change	718	3 085	7 938	6 513	345	85	96
Minor Downgrade	459	2 252	4 508	5 160	265	210	233
Patch Downgrade	434	3 056	6 102	3 519	249	210	231
Major Downgrade	238	2 266	2 877	3 560	163	132	150
Total	16 053	173 028	151 329	181 916	3 075	2 148	2 487

Table 10.5. Most frequent non-breaking changes in the selected dataset snapshot

Non-Breaking Change	**Occ.**	#APIs	#VC	#Major	#Minor	#Patch	#LC
Path Added	37 928	2 182	2 881	405	1 201	985	290
Response Optional Property Removed	34 172	826	1 011	117	458	413	23
Request Optional Property Added	19 814	1 019	1 259	105	482	627	45
Response Property Became Required	18 112	507	647	80	280	259	28
Request Property Enum Value Added	15 853	324	399	35	178	174	12
Request Optional Parameter Added	12 794	1 343	1 737	447	775	490	25
Response Media Type Added	10 604	334	360	51	148	149	12
Response Non Success Status Added	8 452	704	790	96	362	317	15
Response Optional Header Removed	2 332	73	79	14	52	11	2
Response Property Pattern Added	2 316	81	88	14	31	41	2
Request Parameter Enum Value Added	2 063	148	171	17	77	74	3
Request Parameter Became Optional	1 523	161	165	14	86	65	0
Request Property Became Nullable	1 493	111	135	8	76	39	12
Request Property Became Optional	1 433	257	293	36	131	112	14
Request Optional Default Parameter Added	1 122	69	75	7	29	38	1
Response Success Status Added	1 052	315	336	53	129	151	3
Response Required Property Became Not Read-Only	925	15	21	0	9	8	4

Version Changes classification by API Change Type. How many major releases contain at least some breaking changes? According to the aggregated results in Table 10.7 – listing the total number of breaking, non-breaking and undecidable changes for each type of version change – there are 1 058 major upgrades with 7 866 breaking changes in total. While according to semantic versioning, there should be no breaking changes for patch and minor upgrades, we can read that the highest number of breaking changes (87 471) is actually detected in conjunction with minor upgrades. Notably, label changes, despite their lower frequency, also account for a significant number of breaking changes, indicating that clients can and will be broken as an API `alpha` release is updated to `beta`.

The total number of breaking changes listed in Table 10.7 is further decomposed in Table 10.8 with some statistics. It stands out that the worst major release introduced 723 breaking changes. This is a small number, however, if compared to the 2 508 breaking changes applied to one *minor* release. We also spot that the minimum number of breaking changes is 0 across all version change types. This means that there at least some minor releases without breaking changes. How many? Only 32% of the minor releases and 27% of the Patch releases do include exclusively non-breaking changes as we can see from Fig 10.2, showing a complete, detailed map of the major, minor and patch version changes classified according to the corresponding mix of API change types. For example, we can see that while 705 major releases of 375 APIs contain at least one breaking change, 75 releases contain *only* breaking changes. In the worst case, 813 major releases of 422 APIs contain both at least one breaking and one undecidable change. There, we also observe that 37% of major releases include only non-breaking changes, all of which are listed in Table 10.6.

Non breaking changes in Major releases. While it is not a violation of semantic versioning to launch a major release that is fully backwards compatible, we observed that there is only a limited number of 12 non-breaking changes when this happens (Table 10.6). Predominantly, the most frequent changes pertained to modifications in the API structure, such as the inclusion of new paths or the addition of optional request parameters.

Version Change vs. Breaking Change Proportion. While in 594 major, 2 778 minor, and 3 220 patch releases do include changes of exactly one type, 54% of major releases (57% of minor and also 57% of patch) do include a mix of changes. It is thus worth to investigate how the proportion of breaking changes (BC%) relative to all changes influences the decision for a version upgrade. Figure 10.3 illustrates the BC% distribution both for the best and

Table 10.6. All the non-breaking changes that were associated with a Major version change during which no breaking changes occurred

Non-Breaking Change	**Occurrences**	**#APIs**	**#VC(=#Major)**
Request Optional Parameter Added	917	333	334
Path Added	763	118	141
Response Non Success Status Added	214	20	21
Response Optional Property Removed	10	5	5
Response Success Status Added	6	3	4
Request Parameter Became Optional	6	3	3
Request Optional Default Parameter Added To Existing Path	5	1	1
Response Media Type Added	3	3	3
Request Optional Property Added	2	1	1
Request Property Became Optional	2	1	1
Request Property Enum Value Added	2	1	1
Request Parameter Enum Value Added	1	1	1

Table 10.8. Number of breaking changes detected for each type of version change

#BC (Best)	Max		Min		Average		Median		StdDev	
#BC+#UC (Worst)	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best
Major Upgrade	723	509	0	0	18.70	11.27	1	0	64.97	42.77
Minor Upgrade	2 508	2 508	0	0	23.50	9.02	2	0	101.37	57.34
Patch Upgrade	2 308	1 692	0	0	18.37	8.94	2	0	94.25	57.29
Major Downgrade	553	518	0	0	21.61	12.09	5	3	50.60	39.11
Minor Downgrade	362	246	0	0	14.73	9.82	4	3	32.27	20.14
Patch Downgrade	559	349	0	0	21.10	14.06	4	3	58.60	43.12
Label Change	2 637	2 596	0	0	15.35	11.06	0	0	110.29	105.17

worst cases, with the APIs segmented according to the type of version change involved (Major, Minor, Patch, Label Change) as well as whether the version was upgraded (top) or downgraded (bottom). The 'Normalized Frequency' plots within the main histograms provide a relative comparison, allowing for the visual assessment of the impact of the proportion of breaking changes on the decision to launch a major or minor release irrespective of the absolute number of version changes.

In both the best and worst-case scenarios, the histograms show that most version changes have a null proportion of breaking changes, as evidenced by the high bars at the left side of the histograms (BC% = 0%). This observation is consistent with the fact that 54.68% of the APIs exclusively undergo non-breaking changes, thus maintaining backward compatibility. The presence of bars across all intervals indicates that breaking changes are spread across the entire spectrum, becoming more and more prevalent, up to thousands of releases which include only breaking changes. The normalized plots reveal that, regardless of whether updates are classified as upgrades or downgrades, the proportion of breaking changes does not significantly affect the assignment of a new version number to the API. This trend persists even in cases where breaking changes constitute 100% of the alterations, indicating scenarios where all the changes were breaking and developers still assigned a non-major version to the release. In the worst-case scenario, we identified that there were 66 distinct types of breaking changes that were applied in the absence

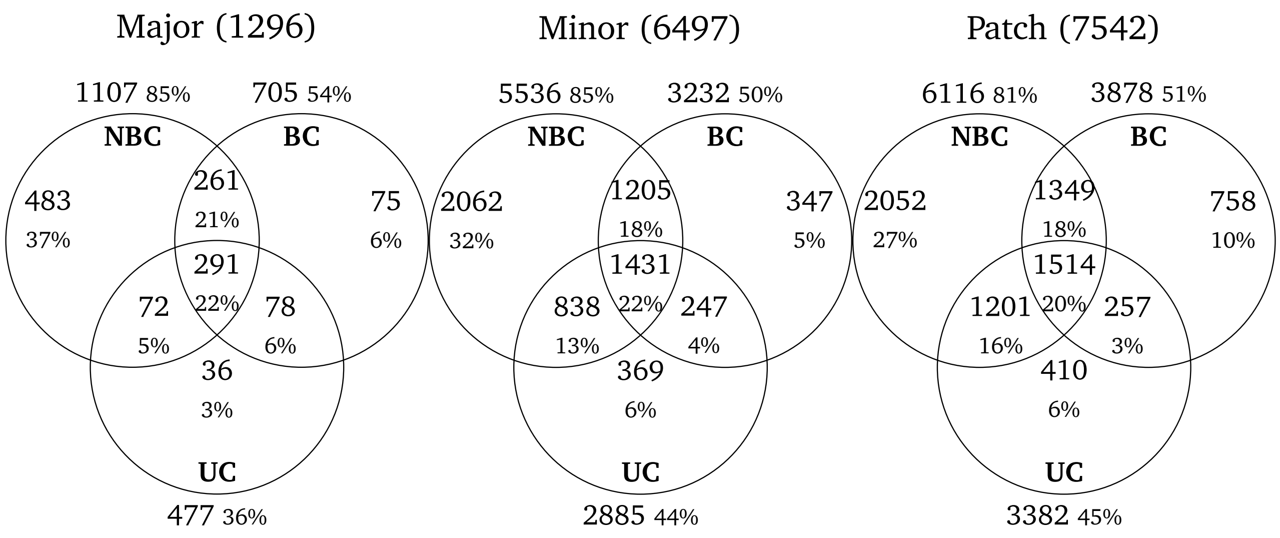


Figure 10.2. Classification of the Major, Minor and Patch-level releases according to their mix of breaking (BC), non-breaking (NBC), and undecidable (UC) changes. The values outside the circles refer to the number of version changes with at least one type of API change

of any non-breaking ones.

☛ At release level, no significant variation in the proportions of change types was observed across different version update types. Breaking changes account for 50–54% of changes across major, minor, and patch version updates.

10.3.2 API-level Compliance

APIs that adhere to semantic versioning are those that have consistently maintained backward compatibility or have appropriately notified clients of any compatibility breaks through version identifiers. Within our dataset, under the best case scenario, we identified a total of 962 adhering APIs (out of 3075 that experienced at least one instance of breaking changes (BC), non-breaking changes (NBC), or undecidable changes (UC)). In the worst-case scenario, this number decreases to 588 APIs. When examining the subset of 2487 APIs that introduced breaking changes, we found that 517 APIs in the best case and only 180 in the worst case have adhered to semantic versioning principles (Table 10.9). These APIs have the highest average number of major releases. The highest average number of releases (#VC) overall, however, is found within the non-compliant APIs. These also underwent a significantly larger number of changes (484 221) than the APIs which adhere to semantic versioning (31 244).

Figure 10.4 provides a nuanced view of the compliance ratio for both best and worst-case scenarios also distinguishing upgrades from downgrades. It illustrates that only some APIs do consistently adhere to (1 444 in the best case, 768 in the worst) or always deviate (532 in the best case, 766 in the worst) from compliance across all releases. Instead, there is a non-empty subset of 1541 APIs with partial compliance in the worst case. The central peak with 50% compliance ratio accounts for the 582 APIs with two releases, out of which only one is compliant.

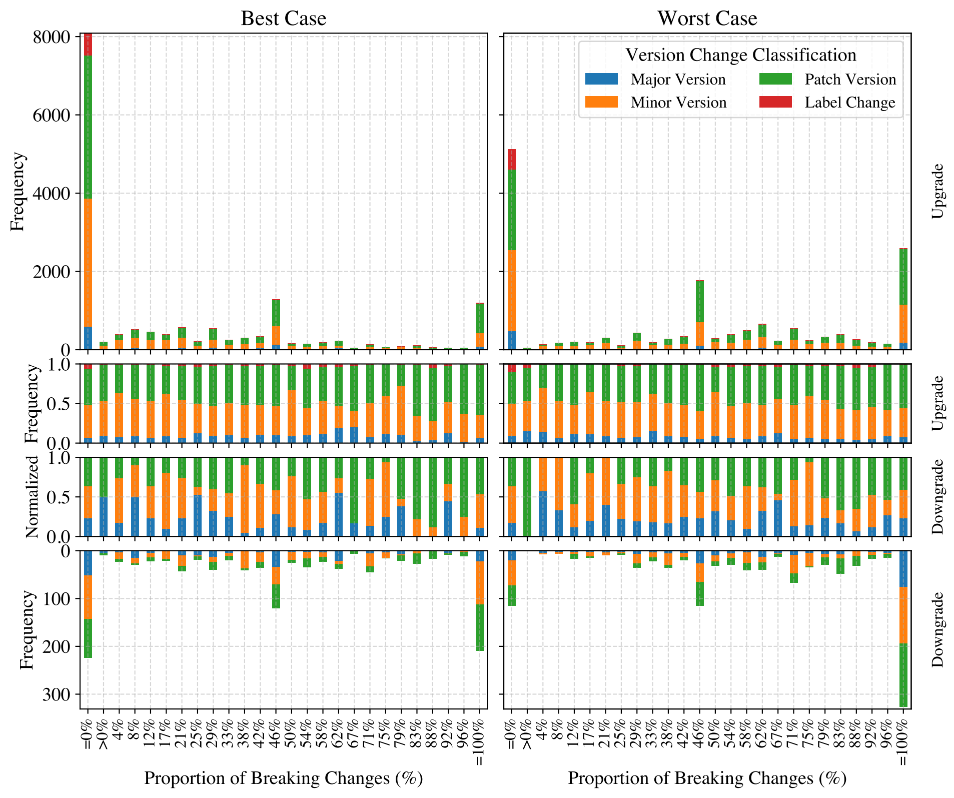


Figure 10.3. Breaking changes proportion distributions for Upgrades (above) and Downgrades (below), categorized by each type of version change

☛ In the best case, when considering that the undecidable changes are non breaking, only 15% of the APIs whose evolution contains breaking changes use version identifiers consistent with semantic versioning at every release. In the worst case, when assuming that the undecidable changes are breaking, this proportion of complying APIs drops to 6%.

10.4 Results Discussion

How often APIs introduce breaking vs. non-breaking changes?

The analysis of histories of 3,075 APIs that experienced changes affecting their functionalities, revealed that 80.87%, included backward incompatible changes. This finding reveals the considerable challenge developers face in maintaining backward compatibility. The prevalence of such changes underlines the critical need for effective versioning strategies and comprehensive documentation to mitigate potential disruptions and ensure a smoother transition for API consumers.

Are there many Web APIs which consistently follow semantic versioning rules across their entire history?

Contrary to theoretical expectations, the study uncovered that only 577 APIs with breaking or potentially break-

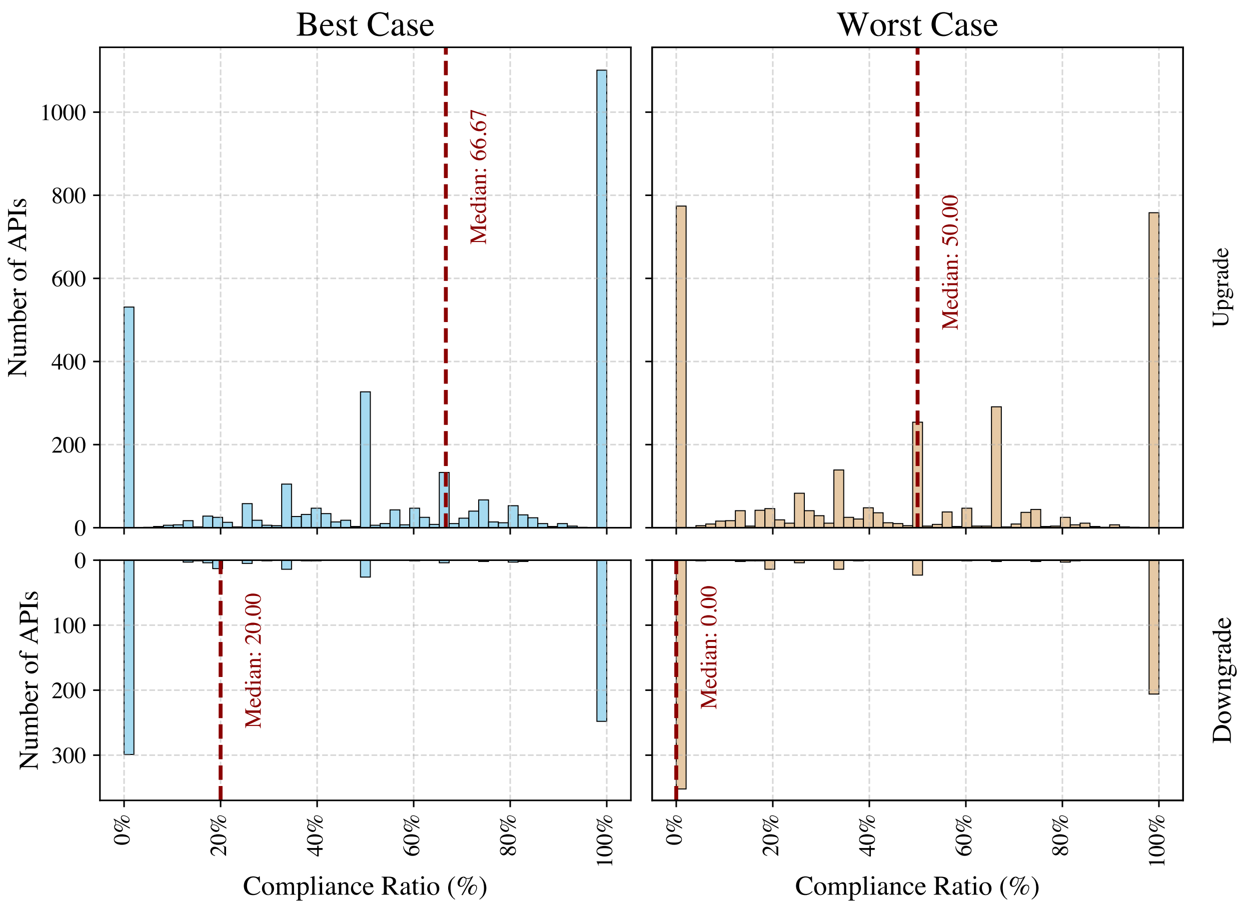


Figure 10.4. Compliance ratio distribution

ing changes adequately reflected these alterations by launching a major release, adhering to semantic versioning principles in practice. Moreover, despite SemVer guidelines suggesting that minor versions should only introduce backward-compatible features, 2 282 APIs did release breaking changes as minor or even patch-level updates. This deviation could be due to a misinterpretation of what constitutes a breaking change or a desire to push new features quickly without incrementing the major version.

We also found 910 APIs where Major version updates did not introduce any breaking changes. Interestingly, these non-breaking changes (NBC) were categorized into exactly 12 distinct types. This observation suggests a nuanced approach to versioning, where developers might choose to launch major releases for reasons other than breaking changes, such as significant feature additions or improvements meant to attract new clients without breaking existing ones.

In Summary

The results of the study presented in this chapter underscore a critical need for tools and guidelines tailored specifically for correctly applying semantic versioning to Web APIs. With an empirical analysis tracking the evolution

Table 10.9. Metrics comparison for APIs classified according to their compliance

	Adhering to Semantic Versioning				Not Adhering		Total
	BC%= 0		BC%> 0		BC%> 0		
Metric	Best	Worst	Best	Worst	Best	Worst	
#APIs	927	588	517	180	1 970	2 307	3 075
#VC	2 190	1 089	1 527	413	14 423	15 537	16 053
Avg #VC	2.36	1.85	2.95	2.29	7.32	6.73	5.22
Avg #Major	0.17	0.32	0.24	0.86	0.04	0.04	0.08
Avg #Minor	0.27	0.32	0.20	0.10	0.45	0.44	0.42
Avg #Patch	0.33	0.26	0.29	0.04	0.49	0.48	0.46
#BC	0	0	2 723	2 665	148 606	148 664	151 329
#NBC	8 226	4 558	7 584	3 896	169 774	173 462	181 916
#UC	5 523	0	7 188	1 440	165 840	171 588	173 028
Avg BC% (Best)	0.00	0.00	8.81	31.46	24.75	23.00	11.19
Avg BC% (Worst)	12.73	0.00	30.20	42.37	46.38	44.90	29.77

histories of 3 075 Web APIs, we found that in the worst case (assuming clients and backends perform strict checking of message payloads) only 768 (25%) APIs consistently comply with Semantic Versioning by always releasing major upgrades for breaking changes (180), or never breaking their backward compatibility (588). This number grows to 1444 APIs (46%) when assuming clients and backends follow the "tolerant reader" pattern [37].

This finding highlights a discrepancy between the theory [8] and the state of the practice of semantic versioning within the Web APIs described using OpenAPI specifications, tracked using GitHub open source repositories. Based on these results, there is a need for establishing standardized versioning protocols which can be embedded into semantic versioning calculators [86, 182] to mitigate the observed inconsistencies, benefiting both Web API developers and consumers by enhancing predictability, reducing potential disruptions, simplifying dependency management, and fostering a more resilient Web API ecosystem.

Part V

Conclusions

Chapter 11

Conclusions

This chapter outlines the key contributions of this thesis through answering the research questions introduced earlier. It also highlights the primary limitations of the study and presents research publications in which the findings and contributions of this dissertation have been published.

11.1 Summary of Research Contributions

This research produced three types of contributions, consisting of empirical insights, tool research prototypes, and data artifacts.

Our initial efforts focused on conducting extensive empirical studies of API Descriptions using OAS, a widely adopted language for defining contemporary Web APIs. These studies analyze the decision-making processes of Web API designers and identify structural patterns in existing API models through pattern mining in tree structures.

By adopting a metrics-based approach, we created a comprehensive picture of the current Web API landscape based on metrics computed over nearly a million API specifications. This vast dataset facilitates segmentation based on specific metrics, enabling the derivation of generalized, broadly applicable results. We also incorporated a temporal aspect to gain deeper insights into evolutionary patterns and addressed challenges developers face during API evolution, particularly in reflecting changes through versioning. To aid designers adopting an API-first strategy in their design and maintenance tasks, we have developed prototype tools that offer guidance during API development: • visualizing API structure and highlighting design smells, • generating comprehensive documentation from code, • visualizing API history and highlighting changes for semantic versioning consistency.

This research allowed answering the questions raised in the introduction part.

1. **What identifiable structural patterns within Web APIs can serve as modular and reusable building blocks?**
   The analysis in this thesis identifies four recurring structural patterns, termed **API primitives**, that frequently appear across Web APIs:

   - **Enumerable Collection (P1):** Allows clients to retrieve and enumerate items using GET methods.
   - **Appendable Collection (P2):** Enables clients to append new items to collections with POST methods.
   - **Collection (P3):** Combines the functionalities of P1 and P2, offering enumeration and appending capabilities.
   - **Mutable Collection (P4):** Extends P3 with batch operations, such as deleting or replacing entire collections.

These primitives were mined from a dataset of **277,094 API fragments** extracted from OpenAPI Specifications, representing common and reusable structures in Web API design. In addition to these patterns, the analysis uncovers **five design smells** prevalent in API structures, including issues such as "Create without Delete" and "Ambiguous POST," which hinder usability, maintainability, and security.

These mined smells are integrated into a Web API visualization tool (OAS2Tree) that alerts users to the presence of any identified smells in the API design. The tool also helps users pinpoint the exact location of these smells within the API structure. This tool is meant to be used at any phase of the API life-cycle, whenever modification need to be inserted.

2. **How are Web APIs and their data models interconnected, and to what extent is this relationship considered in designing APIs that align with expected design principles?**
The research establishes a connection between API structures and their underlying data models, revealing misalignments with design principles like logical structuring and naming conventions. While we detected a strong correlation between the API structure and its size, we found that there is often no paths structural hierarchy that matches the types of resources. For instance, in several paths, the last segment's label is plural while the operations do not exchange a scalar data object. This was also perceived in operation using the `GET` HTTP method.

   This type of unexpected design complicates the learnability and usability of Web APIs [23].

3. **What types of changes are introduced in the evolution of Web APIs, and what is their impact on API clients?**
This work categorizes common API changes, such as endpoint additions, parameter modifications, and data model updates. A detailed analysis reveals that breaking changes occur 2.44 times more frequently than non-breaking ones. Although many breaking changes are fine-grained and may initially seem insignificant, their cumulative impact on client systems is substantial. The analysis quantifies this impact, emphasizing the critical need for effective change communication and mitigation strategies to support seamless client integration.

   We provide a list of about 200observed changes that happen in Web API histories classified based on our estimate of their impacts.

   Based on this understanding of Web API changes, we proposed how to visualize API histories from two perspectives. One focuses on the overall changes that targeted API element during a time window (API CHANGES). The other focuses on locating the changes in time and highlighting the compatibility of changes with the version upgrade/downgrade (API VERSION CLOCK). The overall compliance of web API versioning with changes is empirically analyzed and is discussed in Question 5.

4. **What versioning practices are commonly applied in Web APIs, and how do they impact usability, reliability, and client awareness?**
The thesis examines real-world versioning practices, including semantic versioning, path-based, and header-based approaches, uncovering significant inconsistencies in API versioning methods. Over 50 distinct version identifier formats were identified in API metadata. Among path-based versioning, the most common format was the use of the major version number (e.g., v1, v2). A key observation is a temporal shift in versioning scheme preferences: in 2015, Semantic Versioning (SemVer) was employed by 59% of APIs, while the Major Version Number format accounted for 38.11%. By 2023, SemVer's adoption surged to 87.60%, whereas the Major Version Number format declined to just 5.70%.

5. **To what extent do real-world APIs correctly follow established versioning schemes, such as semantic versioning, and what variations exist?**
Real-world APIs show a notable adherence to established versioning schemes like semantic versioning, but

there are significant variations in practice. According to the research findings, semantic versioning (SemVer) is widely adopted, with its usage peaking at 87% in average.

However, deviations from the expected *semantic* of semantic versioning are common. Many APIs do not consistently follow the semantic versioning rules, particularly regarding breaking changes. For example, the study found that a considerable number of APIs (2,282) released breaking changes as minor or patch-level updates, contrary to SemVer guidelines, which suggest that such changes should trigger a major release. Additionally, there are instances where major version updates do not introduce any breaking changes, with only 12 distinct types of non-breaking changes identified in these releases.

11.2 Threats To Validity

11.2.1 Internal Validity

One potential threat to internal validity lies in the accuracy of the data collection and preprocessing pipeline. The dataset was primarily sourced from GitHub repositories and other public platforms, where irrelevant OpenAPI specifications of non-real APIs could have impacted the analysis. To address this, rigorous filtering was applied to include only valid specifications with consistent histories, ensuring that invalid data entries were excluded. And, also use maturity filtering criteria such as the number of commits which we limited to at least 10 commits that are introducing changes in the API specification.

11.2.2 External Validity

The dataset consisted predominantly of OpenAPI specifications from public repositories, which might not fully represent private APIs or those using other specification formats, such as RAML or WSDL. Although the focus on OpenAPI was justified by its widespread adoption, the findings may not generalize to APIs adhering to alternative paradigms. Expanding the analysis to include other specification formats in future research would help address this limitation.

Another potential limitation is the scope of API versioning practices studied, which centered on semantic versioning. While – in theory – this is a widely accepted practice, some providers or application domains might prioritize alternative strategies, potentially limiting the applicability of the insights to such contexts.

11.2.3 Construct Validity

The identification of structural patterns and smells relied on heuristics and tooling developed during the research. These tools might still miss edge cases or yield false positives.

11.2.4 Reliability

The reproducibility of the findings could be affected by the dynamic nature of the repositories, where changes or deletions might occur. To address this, a dataset archive has been made publicly available, ensuring that future researchers can replicate and extend the studies.

11.3 Retrospective

This thesis presents insights from large-scale empirical analysis of Web APIs, focusing on their structural patterns, evolution processes, and versioning practices. By leveraging a dataset of OpenAPI specifications collected from diverse sources, including GitHub and SwaggerHub, this work offers a data-driven exploration of real-world practices in API design and management. The findings contribute to addressing gaps between theoretical concepts and real world practiced and propose prototypes of tools for supporting web API design tasks and controlling their evolution.

11.3.1 Contributions and key findings

The analysis of structural patterns identified four recurring building blocks that support modularity and reusability: enumerable, appendable, mutable, and collective structures. These patterns serve as foundational elements in Web API design, promoting consistency and scalability across diverse applications. Furthermore, the investigation into API structural alignment revealed frequent deviations from established design principles, such as logical structuring and consistent naming conventions, highlighting the need for improved adherence to best practices.

The study of API evolution categorized over 200 types of changes, revealing that breaking changes occur 2.44 times more frequently than non-breaking changes. Although these breaking changes are often minor in isolation, their cumulative impact on client applications can be significant. This underscores the necessity of effective change management strategies and tools that mitigate disruption during API evolution.

In the context of versioning, the research uncovered a wide range of practices, from path-based and header-based approaches to inconsistent applications of semantic versioning. Despite the widespread claim of semantic versioning adherence, the analysis revealed that many APIs fail to meet its fundamental principles, resulting in potential client confusion and integration challenges. These insights emphasize the need for more rigorous guidelines and automated validation tools to support versioning consistency.

To address these challenges, the research introduced two prototype tools – OAS2Tree and APIcture – to assist developers in visualizing API structures and tracking changes over time. OAS2Tree offers a hierarchical view of API components, facilitating the detection of design inconsistencies and potential flaws. APIcture, on the other hand, provides an interactive timeline of API changes, enabling stakeholders to understand the evolution process and its impact on clients. These tools, though preliminary, illustrate the potential of integrating visualization and analysis techniques into the API development lifecycle.

11.3.2 Research Limitations

Although a significant contribution of this thesis is the creation and analysis of a large dataset [1], one notable limitation is the lack of direct involvement of human participants in the evaluation process. The research relied exclusively on data-driven methodologies to derive insights and assess the effectiveness of proposed tools and prototypes. Although this approach ensures objectivity and scalability, it does not capture subjective perspectives, usability challenges, or real-world applicability experienced by developers who interact with these APIs. Incorporating human-centered evaluation, such as user studies, interviews, or surveys, could have provided deeper insights into, for instance, how the proposed research tool prototypes align with developer needs, enhance usability, and address practical challenges.

11.3.3 Future research directions

Building on the findings of this thesis, several opportunities for future research emerge. Expanding the dataset to include private APIs and industry-specific use cases would provide a more comprehensive understanding of API practices across different domains. The prototypes developed, OAS2Tree and APIcture, could be enhanced to support real-time analysis and integration into popular development environments, enabling developers to receive immediate feedback during API design and evolution.

[1] Publicly made accessible through an API: `http://openapi.inf.usi.ch/`

Moreover, conducting user studies to evaluate the usability and impact of these tools could inform their iterative improvement and adoption. Investigating the role of machine learning in predicting API evolution patterns and detecting design anomalies is another promising direction. By combining empirical evidence with predictive models, future research could provide proactive recommendations to improve API quality and usability.

The findings of this research underscore the importance of standardization in API versioning and design practices. Collaborative efforts between academia and industry could lead to the development of standardized frameworks and tools that support sustainable API design and management. Such efforts would not only bridge the gap between theory and practice but also contribute to a more robust and user-centered API ecosystem.

11.4 Research Publications

Most of the findings presented in this thesis have been published as research contributions. The following is a list of the research papers that I contributed to during my doctoral studies. The papers highlighted in **bold** represent the core contributions that form the basis for the discussions in this thesis. A brief summary of its motivation and key contributions is provided.

1. Diana Carolina Munõz Hurtado, Souhaila Serbout, Cesare Pautasso. Mining Security Documentation Practices in OpenAPI Descriptions. 22nd IEEE International Conference on Software Architecture (ICSA 2025). April 2025. Odense, Denmark.

2. Patric Genfer, Souhaila Serbout, Georg Simhandl, Uwe Zdun, Cesare Pautasso, "Understanding Security Tactics in Microservice APIs using Annotated Software Architecture Decomposition Models – A Controlled Experiment", Accepted to the Empirical Software Engineering Journal.

3. **[146] Souhaila Serbout, Cesare Pautasso, "OAS2Tree: Visual API-First Design", 18th European Conference on Software Architecture (ECSA). September 2024. Luxembourg.**

   → In this demo paper, we present a visualization tool that generates a Web API tree based on an OpenAPI specification, even if the specification is incomplete or invalid. The tool is designed for use during the modeling phase or when learning the functionalities of a Web API. It is available as both a web application for easy access and as a VSCode extension for seamless integration with development environments

4. Petr Pícha, Souhaila Serbout, "On the Adoption of Open Source Software Licensing – A Pattern Collection", 28th European Conference on Pattern Languages of Programs (EuroPLoP). July 2024. Kloster Irsee, Germany.

5. **[145] - Souhaila Serbout, Cesare Pautasso, "How Many Web APIs Evolve Following Semantic Versioning?", 24th International Conference on Web Engineering (ICWE). June 2024. Tampere, Finland.**

   → As the title suggests, this paper explores the adherence of Web APIs to semantic versioning principles. Our approach involves mining git histories of OpenAPI specifications to analyze API changes between consecutive versions. The findings underscore the need for support tools integrated into development environments to manage discrepancies between versioning and the evolution of Web APIs.

6. **[143] - Souhaila Serbout, Cesare Pautasso, "APIstic: A Large Collection of OpenAPI Metrics", 21st International Conference on Mining Software Repositories (MSR). April 2024. Lisbon, Portugal.**

   → The main contribution of this paper is a dataset of OpenAPI specification with computed metrics measuring all of the APIs structure, data models, security and natural language documentation. The specifications are collected from four sources and comprise over a million artifacts. We aim to help researchers select study samples based on metrics combinations from the metrics dataset.

7. **[144] - Souhaila Serbout, and Cesare Pautasso. "How Are Web APIs Versioned in Practice? A Large-Scale Empirical Study". Journal of Web Engineering, 23(4): 465–506, August 2024.**

   → This paper is a journal extension version of the conference paper titled "An Empirical Study of Web API Versioning Practices", presented at the International Conference on Web Engineering. In this extension, we investigate additional versioning practices that were not covered in the original paper, such as header-based versioning. Furthermore, we examine how our previous findings, based solely on data from GitHub, remain valid when expanding the study to include data from other repositories like SwaggerHub and APIs.guru.

8. **[141] - Souhaila Serbout, Diana Carolina M. Hurtado, and Cesare Pautasso, "Interactively exploring API changes and versioning consistency", 11th IEEE Working Conference on Software Visualization (VISSOFT). October 2023. Bogota, Colombia.**

   *→ In this paper, we propose two visualizations that depict Web API evolution from two different perspectives. **API Changes** uses the sunburst metaphor to represent API elements that have been added, removed, or modified. **API Versions Clock** employs the clock metaphor to display the chronological order of changes and their impact (breaking/nonbreaking), alongside the API versioning. These visualizations are generated through a CLI tool, which can be installed in the development environment to easily create interactive HTML visualizations that can be integrated into any form of evolution documentation.*

9. [140] - Souhaila Serbout, Amine El Malki, Cesare Pautasso, and Uwe Zdun, "API Rate Limit - A Pattern Collection", 28th European Conference on Pattern Languages of Programs (EuroPLoP). July 2023. Kloster Irsee, Germany.

10. **[142] - Souhaila Serbout, and Cesare Pautasso. "An empirical study of Web API versioning practices" International Conference on Web Engineering (ICWE). June 2023. Alicante, Spain.**

    → While Semantic Versioning is the standard versioning scheme adopted by most well-known package managers (e.g., Maven, NPM), this paper investigates whether the same applies to Web APIs. We analyzed the version identifiers of a large number of APIs and identified 55 different versioning schemes, including various calendar-based formats, labels, and combinations of formats. A key finding is that semantic versioning emerged as the predominant scheme, with noticeable shifts toward it over time in a few cases.

11. [147] - Souhaila Serbout, Cesare Pautasso, and Uwe Zdun. "How Composable is the Web? An Empirical Study on OpenAPI Data Model Compatibility." 2022 IEEE International Conference on Web Services (ICWS). July 2022. Barcelona, Spain.

12. [43] - Di Lauro Fabio, Souhaila Serbout, and Cesare Pautasso. "A large-scale empirical assessment of Web API size evolution." Journal of Web Engineering. 21(6): 1937-1979, November 2022.

13. [73] - Ivanchikj Ana, Souhaila Serbout, and Cesare Pautasso. "Live process modeling with the BPMN Sketch Miner." Journal of Software and Systems Modeling. 21(5), 1877-1906, October 2022.

14. [149] - Souhaila Serbout, Alessandro Romanelli, and Cesare Pautasso. "ExpressO: From Express.js Implementation Code to OpenAPI Interface Descriptions." 16th European Conference on Software Architecture (ECSA). September 2022. Prague, Czech Republic.

15. [44] - Di Lauro Fabio, Souhaila Serbout, and Cesare Pautasso. "To Deprecate or to Simply Drop Operations? An Empirical Study on the Evolution of a Large OpenAPI Collection." 16th European Conference on Software Architecture (ECSA). September 2022. Prague, Czech Republic.

16. **[139] - Souhaila Serbout, Fabio Di Lauro, and Cesare Pautasso. "Web APIs structures and data models analysis." 19th International Conference on Software Architecture (ICSA). March 2022. Honolulu, Hawaii.**

    → In this paper, we analyze the structures and data models of over 40,000 Web APIs, defining metrics to measure quantifiable aspects of the APIs. Our findings reveal that approximately 30% of the analyzed Web APIs are read-only, though this percentage decreases when focusing on APIs with more than three paths. Additionally, we found that OpenAPI users effectively exploit the language's modularity by reusing predefined schemas. In one case we found that 151 out of the defined schemas were reused.

17. **[148] - Souhaila Serbout, Pautasso Cesare, Uwe Zdun, Olaf Zimmerman. "From OpenAPI fragments to API pattern primitives and design smells." 26th European Conference on Pattern Languages of Programs (EuroPLoP). July 2021. Kloster Irsee, Germany.**

    → In this paper, we explore how recurring structural patterns in Web APIs can be identified and categorized as API pattern primitives or design smells. Our approach involves extracting the API tree structure from OpenAPI specifications and isolating API fragments as subtrees. We then apply clustering techniques to detect recurring fragments, allowing us to assess the occurrence of API pattern primitives and a few structural smell fragments.

18. [42] - Di Lauro Fabio, Souhaila Serbout, and Cesare Pautasso. "Towards a large-scale empirical assessment of Web APIs evolution." 21st International Conference on Web Engineering (ICWE). June 2021. Biarritz, France.

19. [72] - Ivanchikj Ana, Souhaila Serbout, and Cesare Pautasso. "From text to visual BPMN process models: Design and evaluation." 23rd ACM/IEEE international conference on model-driven engineering languages and systems (MODELS). October 2020. Montreal, Canada.

Bibliography

[1] [2022]. OAS2Tree VSCode. https://marketplace.visualstudio.com/items?itemName=oas2tree.oas2tree. xviii, 55

[2] [2023]. APIcture. https://www.npmjs.com/package/apict. 175

[3] [n.d.a]. RAML. https://raml.org/. Accessed: 2021-06-01. 4, 10

[4] [n.d.b]. I/O Docs. https://support.mashery.com/docs/read/IO_Docs. Accessed: 2021-06-01. 10

[5] [n.d.c]. Google BigQuery API. https://cloud.google.com/bigquery/docs/reference/rest/. 31

[6] [n.d.d]. GitHub Archive. http://www.gharchive.org/. 31

[7] [n.d.e]. Bmore Responsive API. https://codeforbaltimore.github.io/Bmore-Responsive/. 157

[8] [n.d.f]. Semantic Versioning. https://semver.org/. 157, 183, 186, 222

[9] [n.d.g]. https://docs.npmjs.com/about-semantic-versioning. 186

[10] Aghajani, E., Nagy, C., Vega-Márquez, O. L., Linares-Vásquez, M., Moreno, L., Bavota, G. and Lanza, M. [2019]. Software documentation issues unveiled, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, pp. 1199–1210. 33

[11] Allamanis, M., Barr, E. T., Devanbu, P. and Sutton, C. [2018]. A survey of machine learning for big code and naturalness, *ACM Comp. Surv.* **51**(4): 1–37. 11

[12] Allamaraju, S. [2010]. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*, O'Reilly. 88

[13] Almeida, R. B., Junes, V. R. C., da Silva Machado, R., da Rosa, D. Y. L., Donato, L. M., Yamin, A. C. and Pernas, A. M. [2019]. A distributed event-driven architectural model based on situational awareness applied on internet of things, *Information and software technology* **111**: 144–158. 3

[14] Alon, U., Zilberstein, M., Levy, O. and Yahav, E. [2019]. Code2vec: Learning distributed representations of code, *Symposium on Principles of Programming Languages* **3**(POPL): 1–29. 11

[15] Babur, Ö. and Cleophas, L. [2017]. Using n-grams for the automated clustering of structural models, *International Conference on Current Trends in Theory and Practice of Informatics*, Vol. 10139 of *LNCS*, pp. 510–524.
URL: *http://link.springer.com/10.1007/978-3-319-51963-0%5F40* 11

[16] Babur, Ö., Cleophas, L. and van den Brand, M. [2019]. Metamodel clone detection with samos, *Journal of Comp. Lang.* **51**: 57–74. 11

[17] Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L. and Pierantonio, A. [2016]. Automated clustering of metamodel repositories, *Int. Conf. on Advanced Information Systems Engineering*, pp. 342–358. 11

[18] Bass, L., Weber, I. and Zhu, L. [2015]. *DevOps – A Software Architect's Perspective*, Addison-Wesley. 156

[19] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R. and Strollo, O. [2012]. When does a refactoring induce bugs? an empirical study, *12th International Working Conference on Source Code Analysis and Manipulation*, IEEE, pp. 104–113. 14

[20] Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C. E., Di Penta, M., Oliveto, R. and Poshyvanyk, D. [2014]. The impact of api change-and fault-proneness on the user ratings of android apps, *IEEE Transactions on Software Engineering* **41**(4): 384–407. 7

[21] Beurer-Kellner, L., Von Pilgrim, J., Tsigkanos, C. and Kehrer, T. [2022]. A transformational approach to managing data model evolution of web services, *IEEE Transactions on Services Computing* . 6

[22] Bogart, C., Kästner, C., Herbsleb, J. and Thung, F. [2016]. How to break an API: cost negotiation and community values in three software ecosystems, *24th International Symposium on Foundations of Software Engineering*, pp. 109–120. 183

[23] Bogner, J., Kotstein, S. and Pfaff, T. [2023]. Do RESTful API design rules have an impact on the understandability of web apis?, *Empirical software engineering* **28**(6): 132. 5, 17, 51, 68, 70, 226

[24] Bogner, J., Wagner, S. and Zimmermann, A. [2020]. Collecting service-based maintainability metrics from restful api descriptions: static analysis and threshold derivation, *European Conference on Software Architecture*, Springer, pp. 215–227. 32

[25] Bojinov, V. [2016]. *RESTful Web API Design with Node. js*, Packt Publishing Ltd. 51

[26] Bonorden, L. and Riebisch, M. [2022]. API deprecation: A systematic mapping study, *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, pp. 451–458. 14

[27] Brandner, M., Craes, M., Oellermann, F. and Zimmermann, O. [2004]. Web services-oriented architecture in production in the finance industry, *Informatik-Spektrum* **27**(2): 136–145. 3

[28] Brito, G., Mombach, T. and Valente, M. T. [2019]. Migrating to graphql: A practical assessment, *Proc. 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 140–150. 169

[29] Brito, G. and Valente, M. T. [2020]. REST vs GraphQL: A controlled experiment, *International Conference on Software Architecture (ICSA)*, IEEE, pp. 81–91. 5

[30] Cheh, C. and Chen, B. [2021]. Analyzing openapi specifications for security design issues, *Secure Development Conference (SecDev)*, IEEE, pp. 15–22. 32

[31] Choi, M. [2012]. A performance analysis of restful open api information system, *International Conference on Future Generation Information Technology*, Springer, pp. 59–64. 12

[32] Christensen, E. [2001]. Web services description language (wsdl) 1.1.
URL: *http://www.w3.org/TR/2001/NOTE-wsdl-20010315* 10

[33] Clarisó, R. and Cabot, J. [2018]. Applying graph kernels to model-driven engineering problems, *Int. Workshop on Machine Learning and Software Engineering in Symbiosis*, pp. 1–5. 11

[34] Coleman, M. and Liau, T. L. [1975]. A computer readability formula designed for machine scoring., *Journal of Applied Psychology* **60**(2): 283. 33, 34

[35] Cossette, B. E. and Walker, R. J. [2012]. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11. 15

[36] Daigneau, R. [2011]. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, Addison-Wesley Professional. 10

[37] Daigneau, R. [2012]. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*, Addison-Wesley. 139, 211, 222

[38] Davis, A. M. [1995]. *201 Principles of Software Development*, McGraw-Hill. 10

[39] De Renzis, A., Garriga, M., Flores, A., Cechich, A., Mateos, C. and Zunino, A. [2017]. A domain independent readability metric for web service descriptions, *Computer Standards & Interfaces* **50**: 124–141. 33

[40] De Souza, C. R. and Bentolila, D. L. [2009]. Automatic evaluation of api usability using complexity metrics and visualizations, *2009 31st International Conference on Software Engineering-Companion Volume*, IEEE, pp. 299–302. 69

[41] Decan, A. and Mens, T. [2019]. What do package dependencies tell us about semantic versioning?, *IEEE Transactions on Software Engineering* **47**(6): 1226–1240. 6

[42] Di Lauro, F., Serbout, S. and Pautasso, C. [2021]. Towards large-scale empirical assessment of web apis evolution, *International Conference on Web Engineering*, Springer, pp. 124–138.
URL: *https://link.springer.com/10.1007/978-3-030-74296-6%5F10* 231

[43] Di Lauro, F., Serbout, S. and Pautasso, C. [2022a]. A large-scale empirical assessment of web api size evolution, *Journal of Web Engineering* **21**(6): 1937–1980. 187, 230

[44] Di Lauro, F., Serbout, S. and Pautasso, C. [2022b]. To deprecate or to simply drop operations? an empirical study on the evolution of a large openapi collection, *in* I. Gerostathopoulos, G. Lewis, T. Batista and T. Bureš (eds), *Software Architecture*, Springer, pp. 38–46. 130, 230

[45] Dietrich, J., Pearce, D., Stringer, J., Tahir, A. and Blincoe, K. [2019]. Dependency versioning in the wild, *Proc. 16th International Conference on Mining Software Repositories (MSR)*, pp. 349–359. 6, 186

[46] Domingo, Á., Echeverría, J., Pastor, Ó. and Cetina, C. [2020]. Evaluating the benefits of model-driven development, *in* S. Dustdar, E. Yu, C. Salinesi, D. Rieu and V. Pant (eds), *Advanced Information Systems Engineering*, Vol. 12127 of *Lecture Notes in Computer Science*, Springer, pp. 353–367.
URL: *http://link.springer.com/10.1007/978-3-030-49435-3%5F22* 11

[47] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. [2017]. Microservices: yesterday, today, and tomorrow, *Present and ulterior software engineering* pp. 195–216. 3

[48] Ed-Douibi, H., Canovas Izquierdo, J. L. and Cabot, J. [2018a]. Automatic generation of test cases for rest apis: A specification-based approach, *Proceedings - 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference, EDOC 2018* pp. 181–190. 10

[49] Ed-Douibi, H., Cánovas Izquierdo, J. L. and Cabot, J. [2018b]. OpenAPItoUML: a tool to generate UML models from OpenAPIdefinitions, *in* T. Mikkonen, R. Klamma and J. Hernández (eds), *International Conference on Web Engineering*, Springer, Springer, pp. 487–491. 84

[50] Effendi, S. D. B., Çirisci, B., Mukherjee, R., Nguyen, H. A. and Tripp, O. [2023]. A language-agnostic framework for mining static analysis rules from code changes, *45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE/ACM, pp. 327–339. 7

[51] Erder, M., Pureur, P. and Woods, E. [2021]. *Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps*, Addison-Wesley. 157

[52] Espinha, T., Zaidman, A. and Gross, H.-G. [2014]. Web api growing pains: Stories from client developers and their code, *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, IEEE, pp. 84–93. 3, 13, 155

[53] Espinha, T., Zaidman, A. and Gross, H.-G. [2015]. Web api growing pains: Loosely coupled yet strongly tied, *Journal of Systems and Software* **100**: 27–43.
URL: *https://www.sciencedirect.com/science/article/pii/S0164121214002180* 3, 13

[54] Fernandes, S. and Bernardino, J. [2015]. What is bigquery?, *Proceedings of the 19th International Database Engineering & Applications Symposium*, pp. 202–203. 31

[55] Fielding, R. T. [2000]. *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis, University of California, Irvine. Doctoral dissertation. 9, 12

[56] Fowler, M. [2010]. Richardson maturity model: steps toward the glory of rest.
URL: *https://www.martinfowler.com/articles/richardsonMaturityModel.html* 91

[57] Giretti, A. [2022]. Api versioning, *Beginning gRPC with ASP.NET Core 6*, pp. 223–237. 183, 212

[58] Grent, H., Akimov, A. and Aniche, M. [2021]. Automatically identifying parameter constraints in complex web apis: A case study at adyen, *arXiv preprint arXiv:2102.00871* p. 71–80.
URL: *https://doi.org/10.1109/ICSE-SEIP52600.2021.00016* 75

[59] Grill, T., Polacek, O. and Tscheligi, M. [2012]. Methods towards api usability: A structural analysis of usability problem categories, *International conference on human-centred software engineering*, Springer, pp. 164–180. 12

[60] Grünewald, E., Wille, P., Pallas, F., Borges, M. C. and Ulbricht, M.-R. [2021]. Tira: an openapi extension and toolbox for gdpr transparency in restful architectures, *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, pp. 312–319. 186

[61] Guerrero-Garcia, J., Gonzalez-Calleros, J. M., Vanderdonckt, J. and Munoz-Arteaga, J. [2009]. A theoretical survey of user interface description languages: Preliminary results, *2009 Latin American Web Congress*, IEEE, pp. 36–43. 10

[62] Hadley, M. J. [2006]. Web application description language (wadl), *Technical report*, USA. 10

[63] Haupt, F., Leymann, F., Scherer, A. and Vukojevic-Haupt, K. [2017]. A framework for the structural analysis of rest apis, *2017 IEEE International Conference on Software Architecture (ICSA)*, IEEE, pp. 55–58. 11, 12, 32

[64] Haupt, F., Leymann, F. and Vukojevic-Haupt, K. [2018a]. Api governance support through the structural analysis of rest apis, *Computer Science-Research and Development* **33**(3): 291–303. 11, 12

[65] Haupt, F., Leymann, F. and Vukojevic-Haupt, K. [2018b]. Api governance support through the structural analysis of rest apis, *Comput. Sci.* **33**(3–4): 291–303. 93

[66] Henning, M. [2007]. API: Design matters: Why changing APIs might become a criminal offense., *Queue* **5**(4): 24–36. 51

[67] Hentrich, C. and Zdun, U. [2011]. *Process-Driven SOA: Patterns for Aligning Business and IT*, Auerbach Publications. 10

[68] Higginbotham, J. [2021]. *Principles of Web API Design: Delivering Value with APIs and Microservices*, Addison-Wesley Signature Series, Addison-Wesley. 157

[69] Hohpe, G. and Woolf, B. [2003]. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley. 10

[70] Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A. and Ducasse, S. [2018]. How do developers react to api evolution? a large-scale empirical study, *Software Quality Journal* **26**: 161–191. 13, 15, 155

[71] Humble, J. and Farley, D. [2010]. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, Addison-Wesley. 175

[72] Ivanchikj, A., Serbout, S. and Pautasso, C. [2020]. From text to visual bpmn process models: design and evaluation, *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, ACM, p. 229–239.
URL: *https://doi.org/10.1145/3365438.3410990* 231

[73] Ivanchikj, A., Serbout, S. and Pautasso, C. [2022]. Live process modeling with the bpmn sketch miner, *Softw. Syst. Model.* **21**(5): 1877–1906.
URL: *https://doi.org/10.1007/s10270-022-01009-w* 230

[74] Javan Jafari, A., Costa, D. E., Shihab, E. and Abdalkareem, R. [2023]. Dependency update strategies and package characteristics, *ACM Transactions on Software Engineering and Methodology* **32**(6): 1–29. 14

[75] Jin, B., Sahni, S. and Shevat, A. [2018]. *Designing Web APIs: Building APIs That Developers Love*, O'Reilly. 5, 6, 51, 68

[76] Johnson, P. [2002]. *Enterprise software system integration: An architectural perspective*, PhD thesis, KTH. 14

[77] Jonnada, S. and Joy, J. K. [2019]. Measure your api complexity and reliability, *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*, IEEE, pp. 104–109. 68

[78] Kagdi, H. H. [2008]. *Mining software repositories to support software evolution*, PhD thesis, Kent State University. 7

[79] Knoche, H. and Hasselbring, W. [2021]. Continuous api evolution in heterogenous enterprise software systems, *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, IEEE, pp. 58–68.
URL: *http://arxiv.org/abs/2103.11397* *http://dx.doi.org/10.1109/ICSA51549.2021.00014* *https://ieeexplore.ieee.org/document/9426799/* 14

[80] Koçi, R., Franch, X., Jovanovic, P. and Abelló, A. [2019]. Classification of changes in api evolution, *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, pp. 243–249. 161

[81] Koçi, R., Franch, X., Jovanovic, P. and Abelló, A. [2023]. Web api evolution patterns: A usage-driven approach, *Journal of Systems and Software* **198**: 111609.
URL: *https://www.sciencedirect.com/science/article/pii/S0164121223000043* 133, 137, 155

[82] Koci, R., Franch, X., Jovanovic, P. and Abelló, A. [2024]. Web api change-proneness prediction, *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 429–434. 5, 12

[83] Kopecky, J., Fremantle, P. and Boakes, R. [2014]. A history and future of web apis, *Information Technology* **56**(3): 90–97. 3, 10

[84] Koren, I. and Klamma, R. [2018]. The exploitation of openapi documentation for the generation of web frontends, *Companion Proceedings of the The Web Conference 2018*, ACM Press, New York, New York, USA, pp. 781–787.
URL: *http://dl.acm.org/citation.cfm?doid=3184558.3188740* 10, 88

[85] Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., Arruda, D., Lifschitz, S. and Zhou, Y. [2020]. From a monolithic big data system to a microservices event-driven architecture, *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)*, IEEE, pp. 213–220. 3

[86] Lam, P., Dietrich, J. and Pearce, D. J. [2020]. Putting the semantics into semantic versioning, *Proc. of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, p. 157–179.
URL: *https://doi.org/10.1145/3426428.3426922* 222

[87] Lamothe, M., Guéhéneuc, Y.-G. and Shang, W. [2021]. A systematic review of api evolution literature, *ACM Computing Surveys (CSUR)* **54**(8): 1–36.
URL: *https://doi.org/10.1145/3470133* 15

[88] Lanza, M. and Marinescu, R. [2007]. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*, Springer. 71

[89] Lauret, A. [2019]. *The design of web APIs*, Simon and Schuster. 5, 6, 10, 51, 68

[90] Lercher, A., Glock, J., Macho, C. and Pinzger, M. [2024]. Microservice api evolution in practice: A study on strategies and challenges, *Journal of Systems and Software* **215**: 112110. 3, 5, 14, 15

[91] Lewis, J. and Fowler, M. [2014]. Microservices: a definition of this new architectural term, `https://martinfowler.com/articles/microservices.html`.
URL: *https://martinfowler.com/articles/microservices.html* 14

[92] Li, D., Mei, H., Shen, Y., Su, S., Zhang, W., Wang, J., Zu, M. and Chen, W. [2018]. Echarts: a declarative framework for rapid construction of web-based visualization, *Visual Informatics* **2**(2): 136–146. 160, 163, 167

[93] Li, J., Xiong, Y., Liu, X. and Zhang, L. [2013]. How does web service api evolution affect clients?, *20th International Conference on Web Services*, IEEE, pp. 300–307. xix, 5, 13, 132, 133, 155

[94] Li, L. and Chou, W. [2011]. Design and describe rest api without violating rest: A petri net based approach, *2011 IEEE International Conference on Web Services*, IEEE, pp. 508–515. 51, 68

[95] Li, W., Wu, F., Fu, C. and Zhou, F. [2023]. A large-scale empirical study on semantic versioning in golang ecosystem, *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 1604–1614. 13

[96] Li, Y., Liu, S., Zhao, L. and Pan, G. [2017]. Self-adapted restful web api evolution model, *Jisuanji Jicheng Zhizao Xitong/Computer Integrated Manufacturing Systems, CIMS* **23**: 1020–1030. 6

[97] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R. and Poshyvanyk, D. [2013]. Api change and fault proneness: A threat to the success of android apps, *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pp. 477–487. 6

[98] Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U. and Stocker, M. [2019]. Interface evolution patterns: balancing compatibility and extensibility across service life cycles, *Proc. 24th EuroPLoP*, EuroPLoP '19, ACM, pp. 1–24.
URL: *https://doi.org/10.1145/3361149.3361164* 14, 129, 130, 155, 166, 183, 185, 201

[99] Maleshkova, M., Pedrinaci, C. and Domingue, J. [2010]. Investigating web APIs on the world wide web, *8th European Conference on Web Services*, IEEE, pp. 107–114. 11, 12

[100] Mark, M. [2011]. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*, O'Reilly. 5, 51

[101] Marquardt, K. [2010]. Patterns for software release versioning, *Proc. of the 15th European Conference on Pattern Languages of Programs (EuroPLoP)*. 6, 187

[102] McCabe, T. J. [1976]. A complexity measure, *IEEE Transactions on software Engineering* (4). 70

[103] McGovern, J., Sims, O., Jain, A. and Little, M. [2006]. Event-driven architecture, *Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations* pp. 317–355. 3

[104] Medjaoui, M., Wilde, E., Mitra, R. and Amundsen, M. [2021]. *Continuous API management*, O'Reilly. 10, 17, 157

[105] Mens, T., Demeyer, S., D'Ambros, M., Gall, H., Lanza, M. and Pinzger, M. [2008]. Analysing software repositories to understand software evolution, *Software evolution* pp. 37–67. 7

[106] Meshram, S. U. [2021]. Evolution of modern web services–rest api with its architecture and design, *International Journal of Research in Engineering, Science and Management* **4**(7): 83–86. 5, 6

[107] Mujahid, S., Abdalkareem, R., Shihab, E. and McIntosh, S. [2020]. Using others' tests to identify breaking updates, *17th international conference on mining software repositories*, ACM, pp. 466–476.
URL: *https://doi.org/10.1145/3379597.3387476* 14

[108] Murer, S. and Bonati, B. [2010]. *Managed evolution: a strategy for very large information systems*, Springer. 14

[109] Murphy, L., Kery, M. B., Alliyu, O., Macvean, A. and Myers, B. A. [2018]. Api designers in the field: Design practices and challenges for creating usable apis, *2018 ieee symposium on visual languages and human-centric computing (vl/hcc)*, IEEE, pp. 249–258. 51

[110] Neumann, A., Laranjeiro, N. and Bernardino, J. [2018]. An analysis of public rest web service apis, *IEEE Transactions on Services Computing* **14**(4): 957–970. 10, 12

[111] Newman, S. [2015]. *Building Microservices: Designing Fine-Grained Systems*, O'Reilly. 3

[112] Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Pierantonio, A. and Iovino, L. [2019]. Automated classification of metamodel repositories: A machine learning approach, *Int. Conf. on Model Driven Engineering Languages and Systems*, pp. 272–282. 11

[113] *OAS2Tree* [n.d.a].
URL: *http://api-ace.inf.usi.ch/openapi-to-tree/* 81

[114] *OAS2Tree* [n.d.b].
URL: *https://marketplace.visualstudio.com/items?itemName=oas2tree.oas2tree* 81

[115] OASDiff [n.d.]. oasdiff tool. `https://github.com/Tufin/oasdiff`. 160, 213

[116] Ochoa, L., Degueule, T., Falleri, J.-R. and Vinju, J. [2022]. Breaking bad? semantic versioning and impact of breaking changes in maven central, *Empirical Software Engineering* **27**(3): 1–42. 6, 13, 14

[117] Ofoeda, J., Boateng, R. and Effah, J. [2023]. An institutional perspective on application programming interface development and integration, *Information Technology & People* . 10

[118] OpenAPI Initiative [n.d.]. OpenAPI Initiative. `https://www.openapis.org/`. Accessed: 2021-06-01. 4, 10, 81, 155, 186, 188, 211

[119] *OpenAPI Generator* [n.d.]. `https://github.com/OpenAPITools/openapi-generator`.
URL: *https://github.com/OpenAPITools/openapi-generator* 88

[120] Palma, F., Dubois, J., Moha, N. and Guéhéneuc, Y.-G. [2014]. Detection of REST patterns and antipatterns: a heuristics-based approach, *International Conference on Service-Oriented Computing*, Springer, pp. 230–244. 12

[121] Patni, S. [2017]. *Pro RESTful APIs*, Springer. 88

[122] Pautasso, C., Ivanchikj, A. and Schreier, S. [2016]. A pattern language for restful conversations, *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee, Germany. 118

[123] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J. and Josuttis, N. M. [2017]. Microservices in practice, part 1: Reality check and service design, *IEEE Software* **34**(1): 91–98.
URL: *https://doi.org/10.1109/MS.2017.24* 14

[124] Petrillo, F., Merle, P., Moha, N. and Guéhéneuc, Y.-G. [2016]. Are rest apis for cloud computing well-designed? an exploratory study, *Proc. ICSOC*, Springer, pp. 157–170. 5

[125] Polák, M. and Holubová, I. [2015]. Rest api management and evolution using mda, *Proceedings of the Eighth International C$*$Conference on Computer Science & Software Engineering*, pp. 102–109. 5

[126] Postman [2021]. Postman, Postman. `https://www.postman.com/`. Accessed: 2024-06-01.
URL: *https://www.postman.com/* 76

[127] Raatikainen, M., Kettunen, E., Salonen, A., Komssi, M., Mikkonen, T. and Lehtonen, T. [2021]. State of the practice in application programming interfaces (APIs): A case study, *Proc. 15th European Conference on Software Architecture (ECSA)*, pp. 191–206. 183

[128] Raemaekers, S., van Deursen, A. and Visser, J. [2017]. Semantic versioning and impact of breaking changes in the maven repository, *Journal of Systems and Software* **129**: 140–158. 13, 14

[129] Rahmatulloh, A., Nugraha, F., Gunawan, R., Darmawan, I., Haerani, E. and Rizal, R. [2024]. Event-driven architecture (eda) vs api-driven architecture (ada): Which performs better in microservices?, *2024 International Conference on Artificial Intelligence, Blockchain, Cloud Computing, and Data Analytics (ICoABCD)*, IEEE, pp. 31–36. 4

[130] Richards, M. [2015]. *Microservices vs. service-oriented architecture*, O'Reilly Media Sebastopol. 3

[131] Richardson, C. [2018]. *Microservices Patterns*, Manning. 3

[132] Richardson, L., Amundsen, M. and Ruby, S. [2013]. *RESTful Web APIs*, O'Reilly. 88

[133] Richardson, L. and Ruby, S. [2007]. *RESTful Web Services*, O'Reilly. 88

[134] Roche, J. [2013]. Adopting devops practices in quality assurance, *Communications of the ACM* **56**(11): 38–43. 175

[135] Saaty, T. L. [2008]. Decision making with the analytic hierarchy process, *International journal of services sciences* **1**(1): 83–98. 12

[136] Schmidt, D. C. [2006]. Model-driven engineering, *Computer-IEEE Computer Society-* **39**(2): 25. 11

[137] Schmiedmayer, P., Bauer, A. and Bruegge, B. [2023]. Reducing the impact of breaking changes to web service clients during web api evolution, *10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 1–11. 14

[138] Schreibmann, V. and Braun, P. [2015]. Model-driven development of restful apis, *International Conference on Web Information Systems and Technologies*, Vol. 2, SCITEPRESS, pp. 5–14. 4

[139] Serbout, S., Di Lauro, F. and Pautasso, C. [2022]. Web apis structures and data models analysis, *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, IEEE, pp. 84–91. 46, 60, 75, 231

[140] Serbout, S., Malki, A. E., Pautasso, C. and Zdun, U. [2023]. Api rate limit adoption - a pattern collection, *Proc. 28th European Conference on Pattern Languages of Programs (EuroPLoP 2023)*, EuroPLoP '23, ACM, Kloster Irsee, Germany.
URL: *https://doi.org/10.1145/3628034.3628039* 230

[141] Serbout, S., Muñoz Hurtado, D. C. and Pautasso, C. [2023]. Interactively exploring api changes and versioning consistency, *Working Conference on Software Visualization (VISSOFT)*, IEEE, Bogota, Colombia, pp. 28–39. 230

[142] Serbout, S. and Pautasso, C. [2023]. An empirical study of web api versioning practices, *in* I. Garrigós, J. M. Murillo Rodríguez and M. Wimmer (eds), *International Conference on Web Engineering*, Springer, pp. 303–318. 46, 161, 167, 230

[143] Serbout, S. and Pautasso, C. [2024a]. APIstic: a large collection of OpenAPI metrics, *21st International Conference on Mining Software Repositories (MSR)*, MSR '24, IEEE, ACM, pp. 265–277.
URL: *https://doi.org/10.1145/3643991.3644932* 46, 229

[144] Serbout, S. and Pautasso, C. [2024b]. How are web apis versioned in practice? a large-scale empirical study, *Journal of Web Engineering* **23**: 465–506. 46, 230

[145] Serbout, S. and Pautasso, C. [2024c]. How many web apis evolve following semantic versioning?, *in* K. Stefanidis, K. Systä, M. Matera, S. Heil, H. Kondylakis and E. Quintarelli (eds), *International Conference on Web Engineering*, Springer, Springer, pp. 344–359. 46, 229

[146] Serbout, S. and Pautasso, C. [2024d]. Oas2tree: Visual api-first design, *18th European Conference on Software Architecture*, Springer, pp. 29–44. 229

[147] Serbout, S., Pautasso, C. and Zdun, U. [2022]. How composable is the web? an empirical study on openapi data model compatibility, *2022 IEEE International Conference on Web Services (ICWS)*, pp. 415–424. 230

[148] Serbout, S., Pautasso, C., Zdun, U. and Zimmermann, O. [2021]. From openapi fragments to api pattern primitives and design smells, *Proceedings of the 26th European Conference on Pattern Languages of Programs*, EuroPLoP '21, ACM, pp. 1–35.
URL: *https://doi.org/10.1145/3489449.3489998* 46, 81, 231

[149] Serbout, S., Romanelli, A. and Pautasso, C. [2022]. Expresso: From express. js implementation code to openapi interface descriptions, *in* T. Batista, T. Bureš, C. Raibulet and H. Muccini (eds), *European Conference on Software Architecture*, Springer, Springer, pp. 29–44. 174, 230

[150] Shafiq, S., Mashkoor, A., Mayr-Dorn, C. and Egyed, A. [2020]. Machine learning for software engineering: A systematic mapping, *arXiv preprint arXiv:2005.13299* . 11

[151] Shepperd, M. [1988]. A critique of cyclomatic complexity as a software metric, *Software Engineering Journal* **3**(2): 30–36. 70

[152] Singh, A., Singh, V., Aggarwal, A. and Aggarwal, S. [2022]. Event driven architecture for message streaming data-driven microservices systems residing in distributed version control system, *2022 International Conference on Innovations in Science and Technology for Sustainable Development (ICISTSD)*, IEEE, pp. 308–312. 3

[153] Smith, E. A. and Senter, R. [1967]. *Automated readability index*, number AD0667273, Aerospace Medical Research Laboratories. `https://apps.dtic.mil/sti/citations/AD0667273`. 33, 34

[154] Sohan, S., Anslow, C. and Maurer, F. [2015]. A case study of web api evolution, *2015 IEEE World Congress on Services*, IEEE, pp. 245–252. 5, 13, 131, 132, 133

[155] Spacy [n.d.]. Models documentation.
URL: *https://spacy.io/models/en* 95

[156] Stocker, M. and Zimmermann, O. [2021]. From code refactoring to api refactoring: Agile service design and evolution, *Service-Oriented Computing (SummerSOC 2021)*, Vol. 1429 of *Communications in Computer and Information Science*, Springer, pp. 174–193. 155

[157] Stocker, M. and Zimmermann, O. [2023]. Api refactoring to patterns: catalog, template and tools for remote interface evolution, *Proceedings of the 28th European Conference on Pattern Languages of Programs*, EuroPLoP '23, ACM, pp. 1–32.
URL: *https://doi.org/10.1145/3628034.3628073* 14, 133

[158] Sturgeon, P. [2016]. *Build APIs you won't hate*, LeanPub.
URL: *https://leanpub.com/build-apis-you-wont-hate* 88

[159] Subramanian, H. and Raj, P. [2019]. *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs*, Packt Publishing Ltd. 51

[160] Suter, P. and Wittern, E. [2015]. Inferring web api descriptions from usage data, *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, IEEE, IEEE, pp. 7–12.
URL: *http://ieeexplore.ieee.org/document/7372275/* 10

[161] *Swagger Codegen* [n.d.]. `https://swagger.io/tools/swagger-codegen/`.
URL: *https://swagger.io/tools/swagger-codegen/* 88

[162] Team, H. F. [2023]. Hugging face api: Pre-trained models for nlp and ai.
URL: *https://huggingface.co/docs* 10

[163] Team, O. [2020]. Gpt-3: Openai's language model api.
URL: *https://openai.com/api/* 10

[164] The Open API Initiative [n.d.]. Oai, https://openapis.org.
URL: *https://openapis.org/* 88

[165] Thönes, J. [2015]. Microservices, *IEEE software* **32**(1): 116–116. 3

[166] Tian, Y., Kochhar, P. S. and Lo, D. [2017]. An exploratory study of functionality and learning resources of web apis on programmableweb, *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE '17, ACM, pp. 202–207.
URL: *https://doi.org/10.1145/3084226.3084286* 12

[167] Tzavaras, A., Mainas, N., Bouraimis, F. and Petrakis, E. G. [2021]. Openapi thing descriptions for the web of things, *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE, pp. 1384–1391. 186

[168] Tzavaras, A., Mainas, N. and Petrakis, E. G. [2023]. Openapi framework for the web of things, *Internet of Things* **21**: 100675. 186

[169] van de Wetering, H., Klaassen, N. and Burch, M. [2020]. Space-reclaiming icicle plots, *2020 IEEE Pacific Visualization Symposium (PacificVis)*, IEEE, pp. 121–130. 166

[170] Varga, E. [2016]. *Versioning REST APIs*, Apress, Berkeley, CA, pp. 109–118.
URL: *https://doi.org/10.1007/978-1-4842-2196-9*$_6$6

[171] Wang, S., Keivanloo, I. and Zou, Y. [2014]. How do developers react to restful api evolution?, *Service-Oriented Computing: 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings 12*, Springer, pp. 245–259. 13

[172] Webber, J., Parastatidis, S. and Robinson, I. [2010]. *REST in Practice: Hypermedia and Systems Architecture*, 1st edn, O'Reilly Media, Inc. 88

[173] Welker, K. D. [2001]. The software maintainability index revisited, *CrossTalk* **14**: 18–21. 70

[174] Wittern, E. [2018]. Web apis-challenges, design points, and research opportunities: Invited talk at the 2nd international workshop on api usage and evolution (wapi'18), *2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution (WAPI)*, WAPI '18, IEEE, ACM, pp. 18–18.
URL: *https://doi.org/10.1145/3194793.3194801* 12

[175] Wittern, E., Ying, A. T., Zheng, Y., Dolby, J. and Laredo, J. A. [2017]. Statically checking web api requests in javascript, *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, ICSE '17, IEEE, IEEE Press, pp. 244–254.
URL: *https://doi.org/10.1109/ICSE.2017.30* 12

[176] Xavier, L., Brito, A., Hora, A. and Valente, M. T. [2017]. Historical and impact analysis of api breaking changes: A large-scale study, *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 138–147. 155

[177] Xavier, L., Hora, A. and Valente, M. T. [2017]. Why do we break apis? first answers from developers, *Proc. IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 392–396. 160

[178] Yang, J., Wittern, E., Ying, A. T., Dolby, J. and Tan, L. [2018]. Towards extracting web api specifications from documentation, *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, MSR '18, ACM, pp. 454–464.
URL: *https://dl.acm.org/doi/10.1145/3196398.3196411* 186

[179] Yasmin, J., Tian, Y. and Yang, J. [2020]. A first look at the deprecation of restful apis: An empirical study, *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, pp. 151–161. 14

[180] Zdun, U. and Avgeriou, P. [2008]. A catalog of architectural primitives for modeling architectural patterns, *Information and Software Technology* **50**(9): 1003–1034.
URL: *https://www.sciencedirect.com/science/article/pii/S0950584907001073* 88

[181] Zhang, D. and Tsai, J. J. [2003]. Machine learning and software engineering, *Softw. Qual.* **11**(2): 87–119. 11

[182] Zhang, L., Liu, C., Xu, Z., Chen, S., Fan, L., Chen, B. and Liu, Y. [2022]. Has my release disobeyed semantic versioning? static detection based on semantic differencing, *arXiv preprint arXiv:2209.00393* pp. 1–12. 13, 14, 186, 222

[183] Zhou, S., Jeong, H. and Green, P. A. [2017]. How consistent are the best-known readability equations in estimating the readability of design standards?, *IEEE Transactions on Professional Communication* **60**(1): 97–111. 34

[184] Zhou, X.-Y., Chen, W., Wu, G.-Q. and Jun, W. [2021]. REST API design analysis and empirical study, *Journal of Software* **33**(9): 3271–3296. 5

[185] Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C. and Stocker, M. [2020]. Interface responsibility patterns: Processing resources and operation responsibilities, *Proc. of the European Conference on Pattern Languages of Programs*, EuroPLoP '20, ACM.
URL: *https://doi.org/10.1145/3424771.3424822* 124, 126

[186] Zimmermann, O., Milinski, S., Craes, M. and Oellermann, F. [2004]. Second generation web services-oriented architecture in production in the finance industry, *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 283–289. 3

[187] Zimmermann, O., Pautasso, Cesare Lübke, D., Zdun, U., and Stocker, M. [2019]. Data-oriented interface responsibility patterns: Types of information holder resources, *Proc. of the European Conference on Pattern Languages of Programs*, EuroPLoP '19, ACM.
URL: *https://doi.org/10.1145/3424771.3424821* 125, 126

[188] Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C. and Zdun, U. [2020]. Introduction to microservice api patterns (map), *in* L. Cruz-Filipe, S. Giallorenzo, F. Montesi, M. Peressotti, F. Rademacher and S. Sachweh (eds), *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*, Vol. 78 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 4:1–4:17.
URL: *https://drops.dagstuhl.de/opus/volltexte/2020/11826* 88

[189] Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C. and Zdun, U. [2021]. Microservice api patterns, https://microservice-api-patterns.org/. 10, 75, 124

[190] Zimmermann, O., Stocker, M., Lübke, D. and Zdun, U. [2017]. Interface representation patterns: Crafting and consuming message-based remote apis, *Proc. of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, ACM, pp. 27:1–27:36.
URL: *http://doi.acm.org/10.1145/3147704.3147734* 124

[191] Zimmermann, O., Stocker, M., Lubke, D., Zdun, U. and Pautasso, C. [2022]. *Patterns for API design: simplifying integration with loosely coupled message exchanges*, Addison-Wesley Professional. 3, 17

[192] Zuse, H. [2019]. *Software complexity: measures and methods*, Vol. 4, Walter de Gruyter GmbH & Co KG. 69