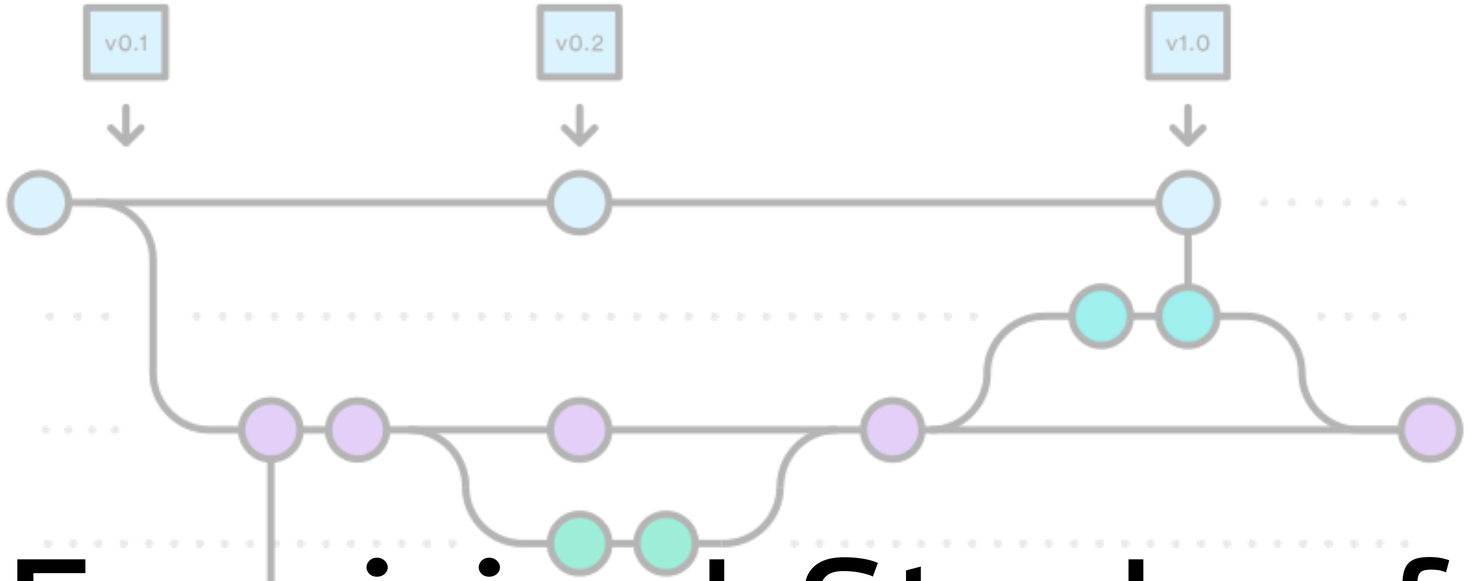
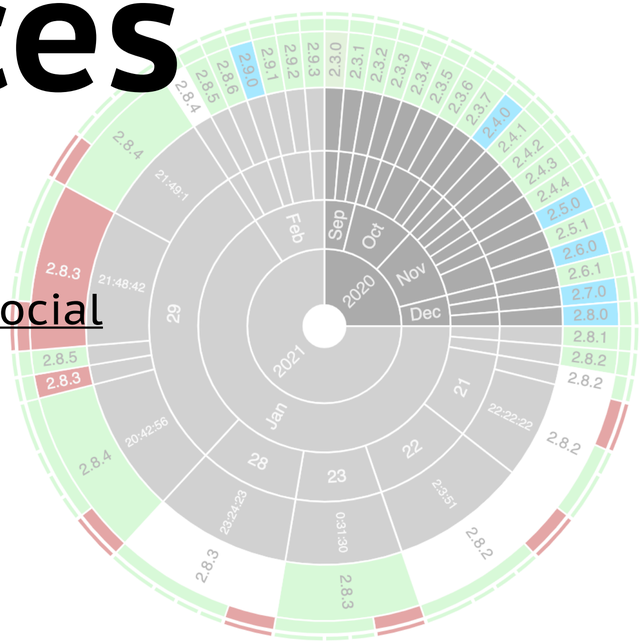


ICWE 2023 ALICANTE

An Empirical Study of **Web API Versioning Practices**

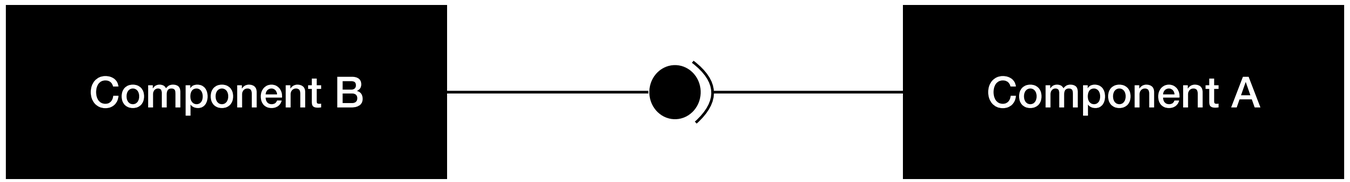


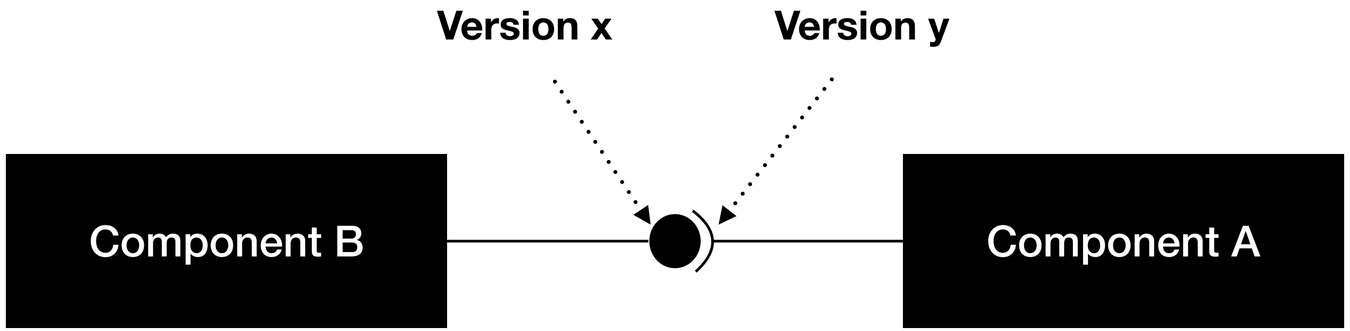
Università della Svizzera italiana

Software Institute

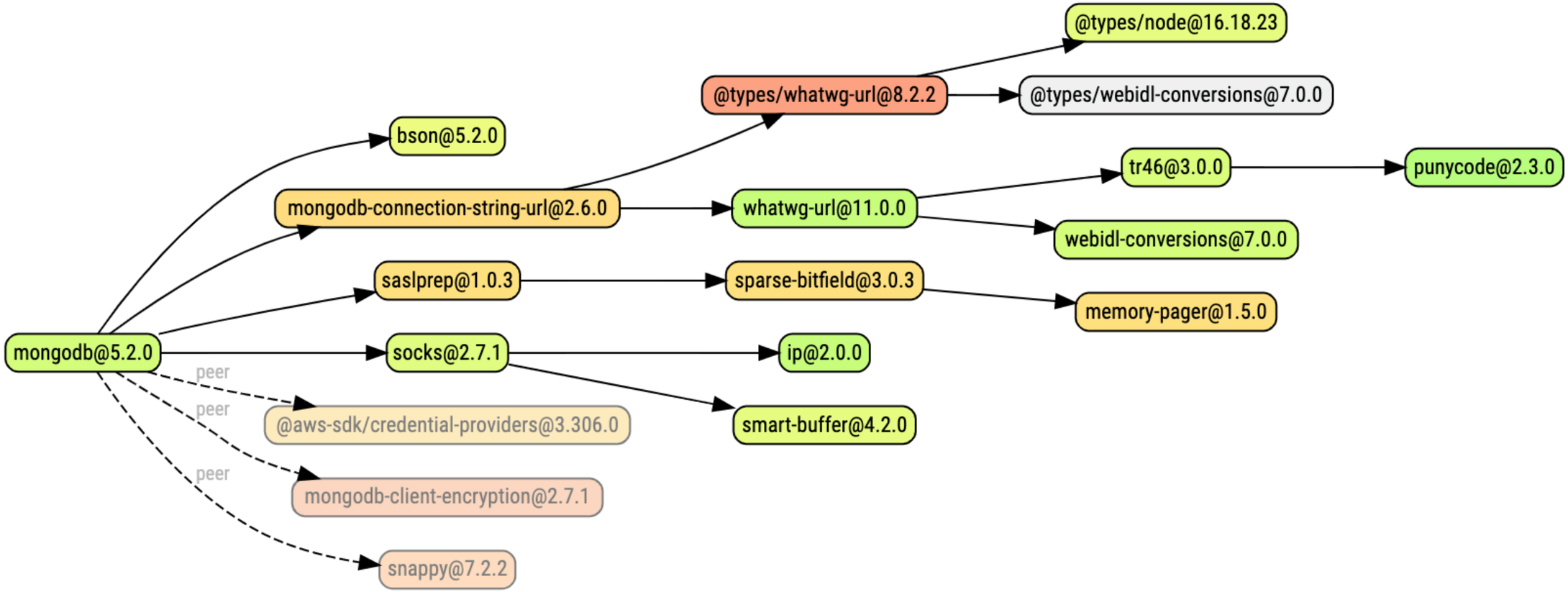
Souhaila Serbout
@SouhailaSerbout
souhaila.serbout@usi.ch
(Speaker)

Cesare Pautasso
@pautasso@scholar.social
cesare.pautasso@usi.ch


Component B
Component A

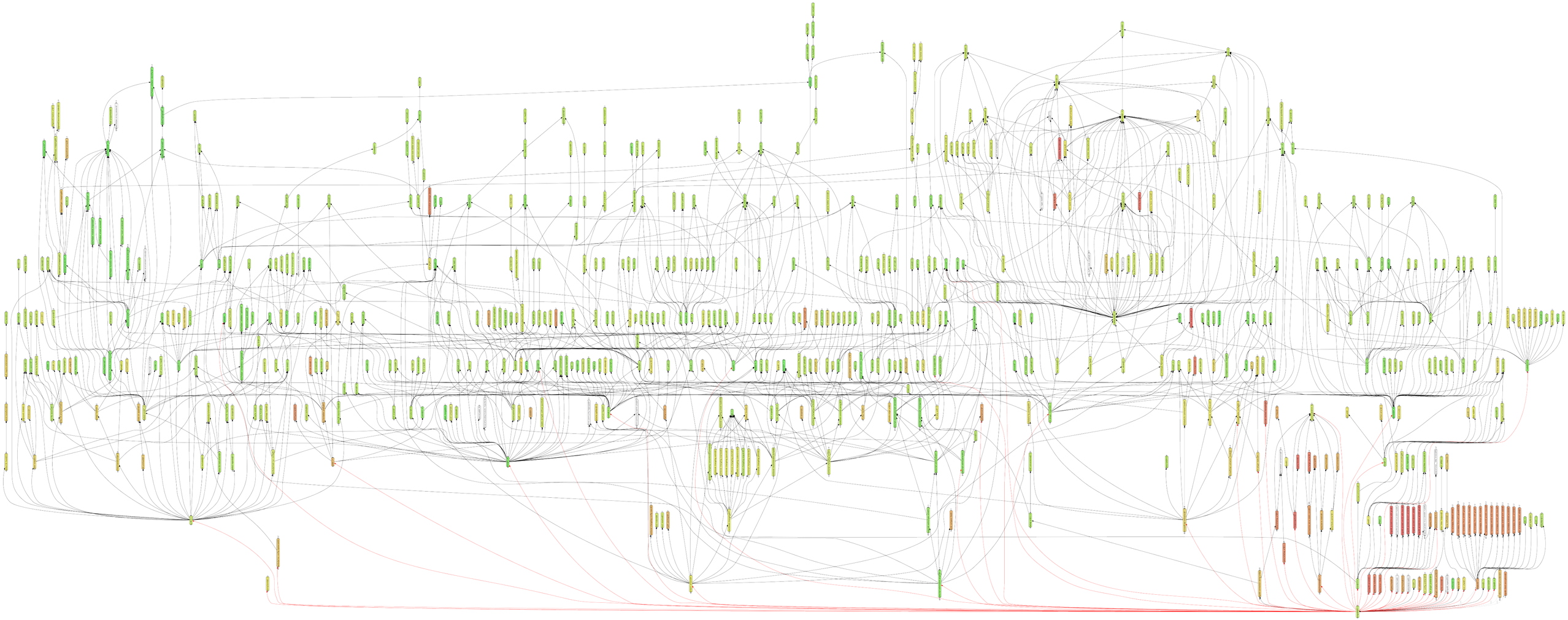

Version x
Version y
Component B
Component A

MongoDB module dependencies



Popularity score: 🔴 = 0%, 🟡 = 50%, 🟢 = 100%

(Dev dependencies are excluded from the graph)



Popularity score: = 0%, = 50%, = 100%

(Dev dependencies are included in the graph)

Semantic versioning

Incremented when **backward incompatible** changes are inctroduced

Incremented **if only backwards compatible bug fixes** are introduced

```
Major.Minor.Patch [-Pre]
```

Incremented if new, **backwards compatible functionality** is introduced

[1] https://semver.org/

Incrementing semantic versions in published packages

To help developers who rely on your code, we recommend starting your package version at `1.0.0` and incrementing as follows:

Code status	Stage	Rule	Example version
First release	New product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

[2] https://docs.npmjs.com/about-semantic-versioning

Incrementing semantic versions in published packages

To help developers who rely on your code, we recommend starting your package version at `1.0.0` and incrementing as follows:

Code status	Stage	Rule	Example version
First release	New product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

[2] https://docs.npmjs.com/about-semantic-versioning

Fix the versioning #1805

Closed **danielchatfield** opened this issue on Aug 27, 2014 · 68 comments

Repository
github.com/jashkenas/underscore

Homepage
underscorejs.org

Weekly Downloads
10,323,984

danielchatfield commented on Aug 27, 2014

1.7.0 introduced **loads** of breaking changes.

The number of dependant modules which are now broken as a result is huge, personally I think that 1.7.0 should be killed (removed from npm) and 2.0 released - the longer the delay the harder it will be to do this.

underscore.js is solely consumed via package managers that mandate the use of semver, you may personally not like semver but that is what is used by the installers to determine compatibility. Last time this was brought up you stated that if you used semver then we would be on underscore version 47 now - well that is much better than having broken code everywhere and lodash has managed to keep the version number below 4.0.0 without breaking everyone's code.

12

[3] https://github.com/jashkenas/underscore

The Journal of Systems and Software 129 (2017) 140–158

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Semantic versioning and impact of breaking changes in the Maven repository

S. Raemaekers[a,b,*], A. van Deursen[b], J. Visser[c]

[a] ING, Haarlemmerweg, Amsterdam, The Netherlands
[b] Technical University Delft, Delft, The Netherlands
[c] Software Improvement Group, Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 16 February 2015
Revised 20 February 2016
Accepted 6 April 2016
Available online 22 April 2016

Keywords:
Semantic versioning
Breaking changes
Software libraries

ABSTRACT

Systems that depend on third-party libraries may have to be updated when updates to these libraries become available in order to benefit from new functionality, security patches, bug fixes, or API improvements. However, often such changes come with changes to the existing interfaces of these libraries, possibly causing rework on the client system. In this paper, we investigate versioning practices in a set of more than 100,000 jar files from Maven Central, spanning over 7 years of history of more than 22,000 different libraries. We investigate to what degree versioning conventions are followed in this repository. Semantic versioning provides strict rules regarding major (breaking changes allowed), minor (no breaking changes allowed), and patch releases (only backward-compatible bug fixes allowed). We find that around one third of all releases introduce at least one breaking change. We perform an empirical study on potential rework caused by breaking changes in library releases and find that breaking changes have a significant impact on client libraries using the changed functionality. We find out that minor releases generally have larger release intervals than major releases. We also investigate the use of deprecation tags and find out that these tags are applied improperly in our dataset.

1. Introduction

For users of software libraries or application programming interfaces (APIs), backward compatibility is a desirable trait. Without backward compatibility, library users will face increased risk and cost when upgrading their dependencies. In spite of these costs and risks, library upgrades may be desirable or even necessary, for example if the newer version contains required additional functionality or critical security fixes. To conduct the upgrade, the library user will need to know whether there are incompatibilities, and, if so, which ones.

Determining whether there are incompatibilities, however, is hard to do for the library user (it is, in fact, undecidable in general). Therefore, it is the library creator's responsibility to indicate the level of compatibility of a library update. One way to inform library users about incompatibilities is through version numbers. As an example, *semantic versioning*[1] (`semver`) suggests a versioning scheme in which three digit version numbers MAJOR.MINOR.PATCH have the following semantics:

- *MAJOR:* This number should be incremented when incompatible API changes are made;
- *MINOR:* This number should be incremented when functionality is added in a backward-compatible manner;
- *PATCH:* This number should be incremented when backward-compatible bug fixes are made.

As an approximation of the (undecidable) notion of backward compatibility, we use the concept of a *binary compatibility* as defined in the Java language specification. The Java Language Specification[2] states that *a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.* This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities as well. For the purpose of this paper, we define any change that does not maintain binary compatibility between releases to be a *breaking change*.

* Corresponding author at: Technical University Delft, Delft, The Netherlands. Tel.: +31626966234.
E-mail addresses: stevenraemaekers@gmail.com (S. Raemaekers), arie.vandeursen@tudelft.nl (A. van Deursen), j.visser@sig.eu (J. Visser).

[1] http://semver.org.

[2] http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html.

http://dx.doi.org/10.1016/j.jss.2016.04.008
0164-1212/

The Journal of Systems and Software 129 (2017) 140–158

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Semantic versioning and impact of breaking changes in the Maven repository

S. Raemaekers[a,b,*], A. van Deursen[b], J. Visser[c]

[a] ING, Haarlemmerweg, Amsterdam, The Netherlands
[b] Technical University Delft, Delft, The Netherlands
[c] Software Improvement Group, Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 16 February 2015
Revised 20 February 2016
Accepted 6 April 2016
Available online 22 April 2016

Keywords:
Semantic versioning
Breaking changes
Software libraries

ABSTRACT

Systems that depend on third-party libraries may have to be updated when updates to these libraries become available in order to benefit from new functionality, security patches, bug fixes, or API improvements. However, often such changes come with changes to the existing interfaces of these libraries, possibly causing rework on the client system. In this paper, we investigate versioning practices in a set of more than 100,000 jar files from Maven Central, spanning over 7 years of history of more than 22,000 different libraries. We investigate to what degree versioning conventions are followed in this repository. Semantic versioning provides strict rules regarding major (breaking changes allowed), minor (no breaking changes allowed), and patch releases (only backward-compatible bug fixes allowed). We find that around one third of all releases introduce at least one breaking change. We perform an empirical study on potential rework caused by breaking changes in library releases and find that breaking changes have a significant impact on client libraries using the changed functionality. We find out that minor releases generally have larger release intervals than major releases. We also investigate the use of deprecation tags and find out that these tags are applied improperly in our dataset.

1. Introduction

For users of software libraries or application programming interfaces (APIs), backward compatibility is a desirable trait. Without backward compatibility, library users will face increased risk and cost when upgrading their dependencies. In spite of these costs and risks, library upgrades may be desirable or even necessary, for example if the newer version contains required additional functionality or critical security fixes. To conduct the upgrade, the library user will need to know whether there are incompatibilities, and, if so, which ones.

Determining whether there are incompatibilities, however, is hard to do for the library user (it is, in fact, undecidable in general). Therefore, it is the library creator's responsibility to indicate the level of compatibility of a library update. One way to inform library users about incompatibilities is through version numbers. As an example, *semantic versioning*[1] (`semver`) suggests a versioning scheme in which three digit version numbers MAJOR.MINOR.PATCH have the following semantics:

- *MAJOR:* This number should be incremented when incompatible API changes are made;
- *MINOR:* This number should be incremented when functionality is added in a backward-compatible manner;
- *PATCH:* This number should be incremented when backward-compatible bug fixes are made.

As an approximation of the (undecidable) notion of backward compatibility, we use the concept of a *binary compatibility* as defined in the Java language specification. The Java Language Specification[2] states that *a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.* This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities as well. For the purpose of this paper, we define any change that does not maintain binary compatibility between releases to be a *breaking change.*

* Corresponding author at: Technical University Delft, Delft, The Netherlands. Tel.: +31626966234.
E-mail addresses: stevenraemaekers@gmail.com (S. Raemaekers), arie.vandeursen@tudelft.nl (A. van Deursen), j.visser@sig.eu (J. Visser).

[1] http://semver.org.

[2] http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html.

http://dx.doi.org/10.1016/j.jss.2016.04.008
0164-1212/

Maven in 2017

Breaking changes (BCs) were spread over all the software releases: Major (35.8%), Minor(35.7%), Patch (23.8%)

The Journal of Systems and Software 129 (2017) 140–158

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Semantic versioning and impact of breaking changes in the Maven repository

S. Raemaekers[a,b,*], A. van Deursen[b], J. Visser[c]

[a] ING, Haarlemmerweg, Amsterdam, The Netherlands
[b] Technical University Delft, Delft, The Netherlands
[c] Software Improvement Group, Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 16 February 2015
Revised 20 February 2016
Accepted 6 April 2016
Available online 22 April 2016

Keywords:
Semantic versioning
Breaking changes
Software libraries

ABSTRACT

Systems that depend on third-party libraries may have to be updated when updates to these libraries become available in order to benefit from new functionality, security patches, bug fixes, or API improvements. However, often such changes come with changes to the existing interfaces of these libraries, possibly causing rework on the client system. In this paper, we investigate versioning practices in a set of more than 100,000 jar files from Maven Central, spanning over 7 years of history of more than 22,000 different libraries. We investigate to what degree versioning conventions are followed in this repository. Semantic versioning provides strict rules regarding major (breaking changes allowed), minor (no breaking changes allowed), and patch releases (only backward-compatible bug fixes allowed). We find that around one third of all releases introduce at least one breaking change. We perform an empirical study on potential rework caused by breaking changes in library releases and find that breaking changes have a significant impact on client libraries using the changed functionality. We find out that minor releases generally have larger release intervals than major releases. We also investigate the use of deprecation tags and find out that these tags are applied improperly in our dataset.

1. Introduction

For users of software libraries or application programming interfaces (APIs), backward compatibility is a desirable trait. Without backward compatibility, library users will face increased risk and cost when upgrading their dependencies. In spite of these costs and risks, library upgrades may be desirable or even necessary, for example if the newer version contains required additional functionality or critical security fixes. To conduct the upgrade, the library user will need to know whether there are incompatibilities, and, if so, which ones.

Determining whether there are incompatibilities, however, is hard to do for the library user (it is, in fact, undecidable in general). Therefore, it is the library creator's responsibility to indicate the level of compatibility of a library update. One way to inform library users about incompatibilities is through version numbers. As an example, *semantic versioning*[1] (`semver`) suggests a versioning scheme in which three digit version numbers MAJOR.MINOR.PATCH have the following semantics:

MAJOR: This number should be incremented when incompatible API changes are made;
MINOR: This number should be incremented when functionality is added in a backward-compatible manner;
PATCH: This number should be incremented when backward-compatible bug fixes are made.

As an approximation of the (undecidable) notion of backward compatibility, we use the concept of a *binary compatibility* as defined in the Java language specification. The Java Language Specification[2] states that *a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.* This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities as well. For the purpose of this paper, we define any change that does not maintain binary compatibility between releases to be a *breaking change.*

* Corresponding author at: Technical University Delft, Delft, The Netherlands. Tel.: +31626966234.
E-mail addresses: stevenraemaekers@gmail.com (S. Raemaekers), arie.vandeursen@tudelft.nl (A. van Deursen), j.visser@sig.eu (J. Visser).

[1] http://semver.org.

[2] http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html.

http://dx.doi.org/10.1016/j.jss.2016.04.008
0164-1212/

Maven in 2017

Breaking changes (BCs) were spread over all the software releases: Major (35.8%), Minor(35.7%), Patch (23.8%).

Slight increased adherance to SemVer in Maven Repositories Over the years [4] .

Empirical Software Engineering (2022) 27: 61
https://doi.org/10.1007/s10664-021-10052-y

Breaking bad? Semantic versioning and impact of breaking changes in Maven Central

An external and differentiated replication study

Lina Ochoa[1] · Thomas Degueule[2] · Jean-Rémy Falleri[2,3] · Jurgen Vinju[1,4]

Accepted: 30 August 2021 / Published online: 17 March 2022

Abstract

Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. However, when a library evolves, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its API. These changes might trigger compile-time, link-time, or run-time errors in client code. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult. Understanding how libraries evolve helps client developers to know which changes to expect and where to expect them, and library developers to understand how they might impact their clients. In the most extensive study to date, Raemaekers et al. investigate to what extent developers of Java libraries hosted on the Maven Central Repository (MCR) follow semantic versioning conventions to signal the introduction of BCs and how these changes impact client projects. Their results suggest that BCs are widespread without regard for semantic versioning, with a significant impact on clients. In this paper, we conduct an external and differentiated replication study of their work. We identify and address some limitations of the original protocol and expand the analysis to a new corpus spanning seven more years of the MCR. We also present a novel static analysis tool for Java bytecode, `Maracas`, which provides us with: (i) the set of all BCs between two versions of a library, and; (ii) the set of locations in client code impacted by individual BCs. Our key findings, derived from the analysis of 119,879 library upgrades and 293,817 clients, contrast with the original study and show that 83.4% of these upgrades do comply with semantic versioning. Furthermore, we observe that the tendency to comply with semantic versioning has significantly increased over time. Finally, we find that most BCs affect code that is not used by any client, and that only 7.9% of all clients are affected by BCs. These findings should help (i) library developers to understand and anticipate the impact of their changes; (ii) library users to estimate library upgrading effort and to pick libraries that are less likely to break, and; (iii) researchers to better understand the dynamics of library-client co-evolution in Java.

Communicated by: Gabriele Bavota

✉ Lina Ochoa
l.m.ochoa.venegas@tue.nl

Empirical Software Engineering (2022) 27: 61
https://doi.org/10.1007/s10664-021-10052-y

Breaking bad? Semantic versioning and impact of breaking changes in Maven Central

An external and differentiated replication study

Lina Ochoa[1] · Thomas Degueule[2] · Jean-Rémy Falleri[2,3] · Jurgen Vinju[1,4]

Accepted: 30 August 2021 / Published online: 17 March 2022

Abstract

Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. However, when a library evolves, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its API. These changes might trigger compile-time, link-time, or run-time errors in client code. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult. Understanding how libraries evolve helps client developers to know which changes to expect and where to expect them, and library developers to understand how they might impact their clients. In the most extensive study to date, Raemaekers et al. investigate to what extent developers of Java libraries hosted on the Maven Central Repository (MCR) follow semantic versioning conventions to signal the introduction of BCs and how these changes impact client projects. Their results suggest that BCs are widespread without regard for semantic versioning, with a significant impact on clients. In this paper, we conduct an external and differentiated replication study of their work. We identify and address some limitations of the original protocol and expand the analysis to a new corpus spanning seven more years of the MCR. We also present a novel static analysis tool for Java bytecode, `Maracas`, which provides us with: (i) the set of all BCs between two versions of a library, and; (ii) the set of locations in client code impacted by individual BCs. Our key findings, derived from the analysis of 119,879 library upgrades and 293,817 clients, contrast with the original study and show that 83.4% of these upgrades do comply with semantic versioning. Furthermore, we observe that the tendency to comply with semantic versioning has significantly increased over time. Finally, we find that most BCs affect code that is not used by any client, and that only 7.9% of all clients are affected by BCs. These findings should help (i) library developers to understand and anticipate the impact of their changes; (ii) library users to estimate library upgrading effort and to pick libraries that are less likely to break, and; (iii) researchers to better understand the dynamics of library-client co-evolution in Java.

Communicated by: Gabriele Bavota

✉ Lina Ochoa
l.m.ochoa.venegas@tue.nl

Maven in 2022

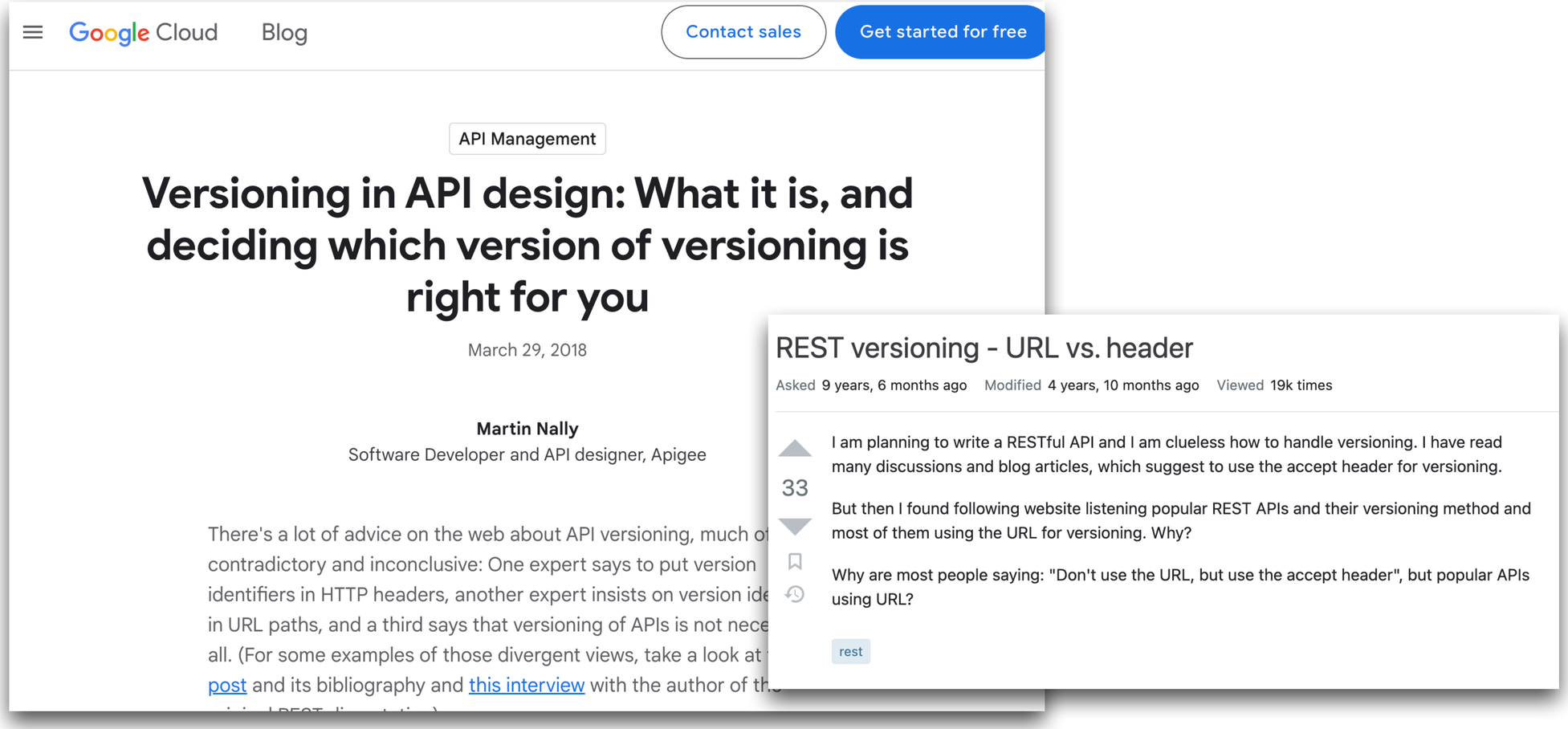
83.4% of all library upgrades comply with SemVer principles ; Breaking changes were introduced only when they are expected [5].

What about Web APIs ?

What about Web APIs ?

🥺 APIs are not centrally deployed

How to version web APIs ?


Google Cloud
Blog
Contact sales
Get started for free
API Management
Versioning in API design: What it is, and deciding which version of versioning is right for you
March 29, 2018
Martin Nally
Software Developer and API designer, Apigee
There's a lot of advice on the web about API versioning, much o
contradictory and inconclusive: One expert says to put version
identifiers in HTTP headers, another expert insists on version ide
in URL paths, and a third says that versioning of APIs is not nece
all. (For some examples of those divergent views, take a look at
post and its bibliography and this interview with the author of th

REST versioning - URL vs. header
Asked 9 years, 6 months ago
Modified 4 years, 10 months ago
Viewed 19k times
33
I am planning to write a RESTful API and I am clueless how to handle versioning. I have read many discussions and blog articles, which suggest to use the accept header for versioning.
But then I found following website listening popular REST APIs and their versioning method and most of them using the URL for versioning. Why?
Why are most people saying: "Don't use the URL, but use the accept header", but popular APIs using URL?
rest

Google Cloud Blog

Contact sales | Get started for free

API Management

Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Martin Nally
Software Developer and API designer, Apigee

REST versioning - URL vs. header

Asked 9 years, 6 months ago Modified 4 years, 10 months ago Viewed 19k times

I am planning to write a RESTful API and I am clueless how to handle versioning. I have read many discussions and blog articles, which suggest to use the accept header for versioning.

...en I found following website listening popular REST APIs and their versioning method and ...of them using the URL for versioning. Why?

...are most people saying: "Don't use the URL, but use the accept header", but popular APIs ... URL?

Search Medium

Xeno Fox
May 13, 2019 · 11 min read · Listen

You're thinking about API versioning in the wrong way.

And the way you have implemented versioning is not correct.

Google Cloud Blog

Contact sales Get started for free

API Management

Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Software De

REST versioning -

Now brace yourself, here is what you need to take away from reading this article.

> "You should not be versioning your API at all."

Now before you Alt+Tab away, you might as well hear me out, since you've read this far into the article. Versioning your API is not the correct way to resolve the problem you are facing. Instead you should be versioning your

Search Medium

Xeno Fox

May 13, 2019 · 11 min read · Listen

You're thinking about API wrong way.

And the way you have implemented versioning is not correct.

allegro Tech Blog

about us

Jan 29 2015

Content headers or how to version your API?

how to version your API?

easy to upgrade. If you forget about it, you your API will force you to contact all your t, both you and your clients will be very ns of your resources. But there is no single different ways. Below you find three most

ber?

Version in URL

The easiest way to handle multiple versions is to put the version number into the URL. You can find this approach for example in Twitter API.

```
http://myapplication.com/api/v1/user/1
http://myapplication.com/api/v2/user/1
```

Google Cloud Blog

Contact sales | Get started for free

API Management

Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Software De

REST versioning -

Now brace yourself, here is what you need to take away from reading this article.

"You should not be versioning your API at all."

ght as well hear me out, since you've
g your API is not the correct way to
stead you should be versioning your

Search Medium

Your API versioning is wrong, which is why I decided to do it 3 different wrong ways

10 FEBRUARY 2014

In the end, I decided the fairest, most balanced way was to piss everyone off equally. Of course I'm talking about API versioning and not since the great "tabs versus spaces" debate have I seen so many strong beliefs in entirely different camps.

allegro Tech Blog

about us

Jan 29 2015

Content headers or how to version your API?

how to version your API?

easy to upgrade. If you forget about it, you
your API will force you to contact all your
t, both you and your clients will be very
ns of your resources. But there is no single
different ways. Below you find three most

ber?

Version in URL

The easiest way to handle multiple versions is to put the version number into the URL. You can find this approach for example in Twitter API.

```
http://myapplication.com/api/v1/user/1
http://myapplication.com/api/v2/user/1
```

Google Cloud Blog

Contact sales

Get started for free

API Management

Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Software De

REST versioning -

Now brace yourself, here is what you need to
article.

"You should not be versioning

Search Medium

Your API versioning is wrong, which is why I decided to do it 3 different wrong ways

10 FEBRUARY 2014

In the end, I decided the fairest, most balanced way was to piss everyone off equally. Of course I'm talking about API versioning and not since the great "tabs versus spaces" debate have I seen so many strong beliefs in entirely different camps.

allegro Tech Blog

about us

Google Cloud Blog

Contact sales

Get started for free

API Management

API versioning best practices: When you need versioning and when you don't

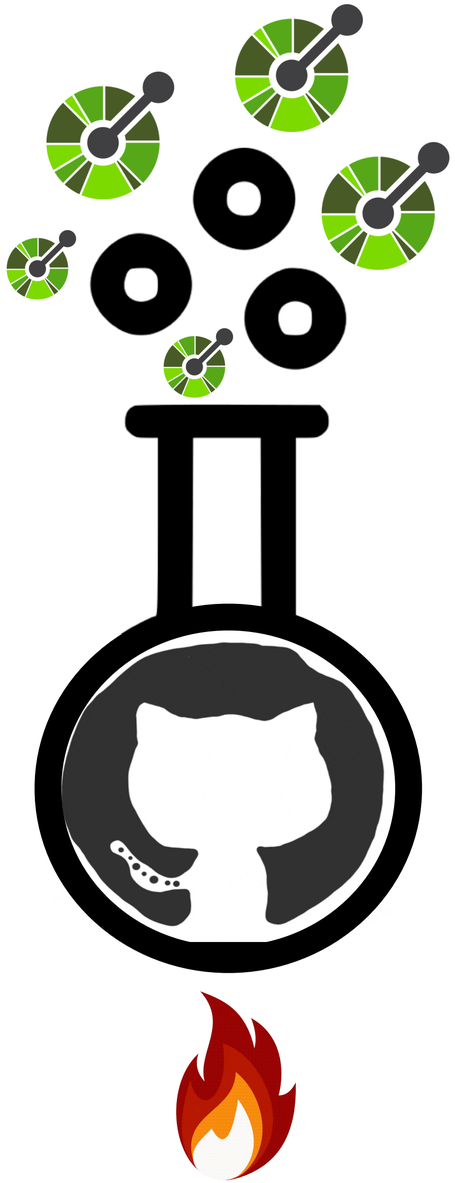
May 15, 2017

Martin Nally
Software Developer and API designer,
Apigee

When versioning makes sense—and when it doesn't

How to version web APIs ?

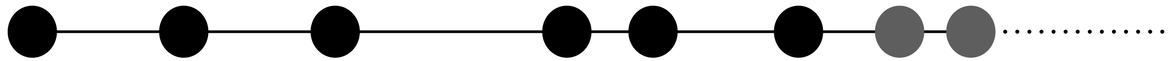
How do developers version web APIs ?

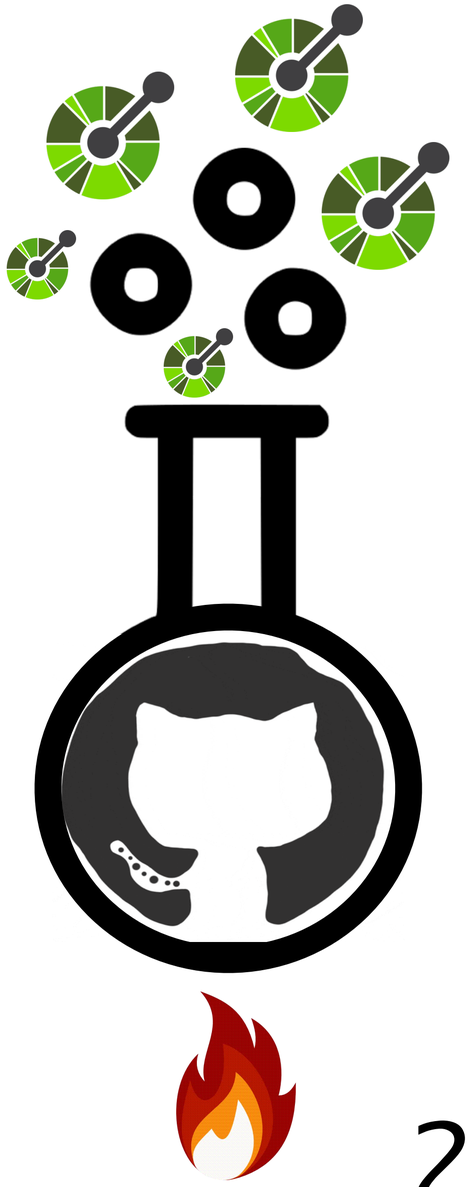


7,114 Web APIs

APIs with more than 10 commits

186,259 Commits

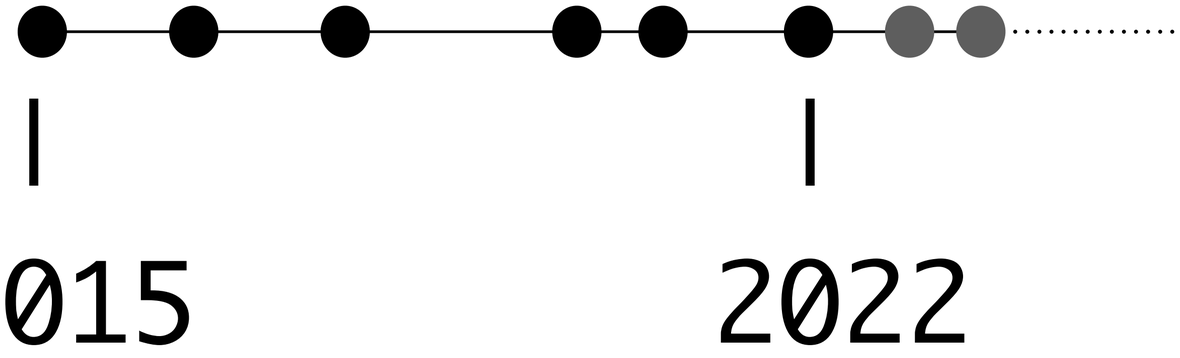


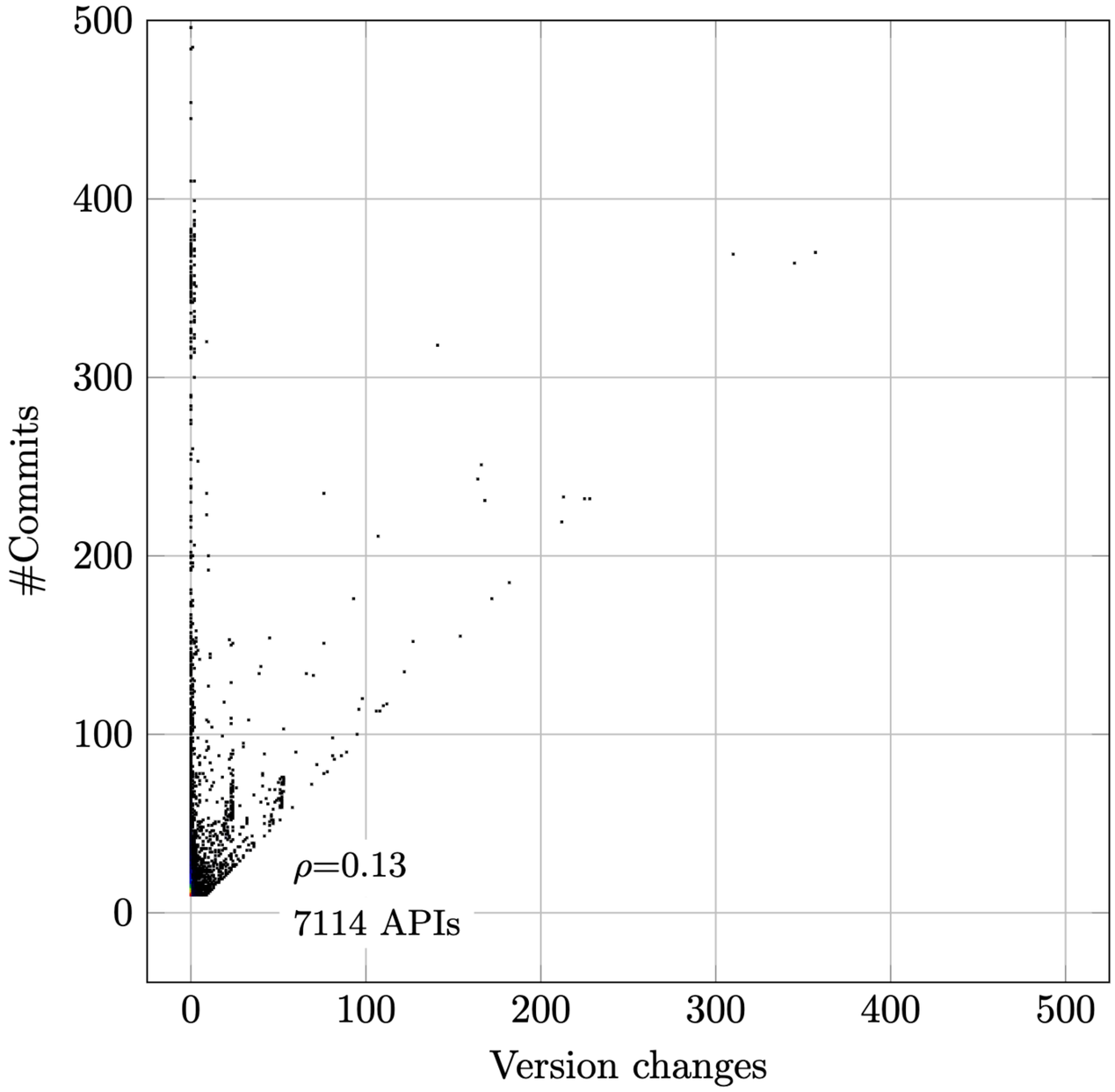


7,114 Web APIs

APIs with more than 10 commits

186,259 Commits



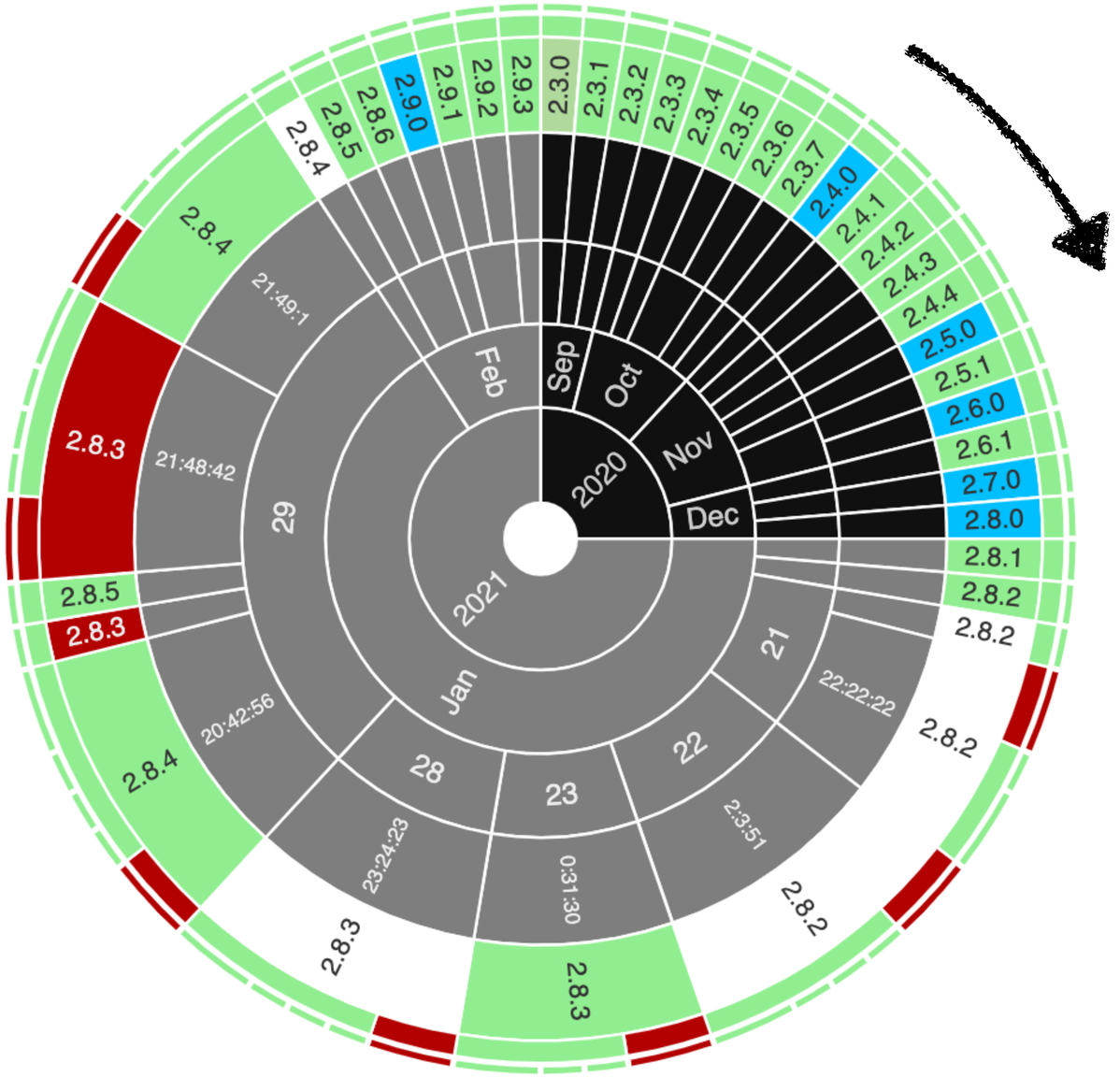

500
400
300
200
100
0
#Commits
ρ=0.13
7114 APIs
0
100
200
300
400
500
Version changes

Xero OAuth 2 Identity Service API

`https://api.xero.com`

33 version changes in 154 days

First commit : September 2020

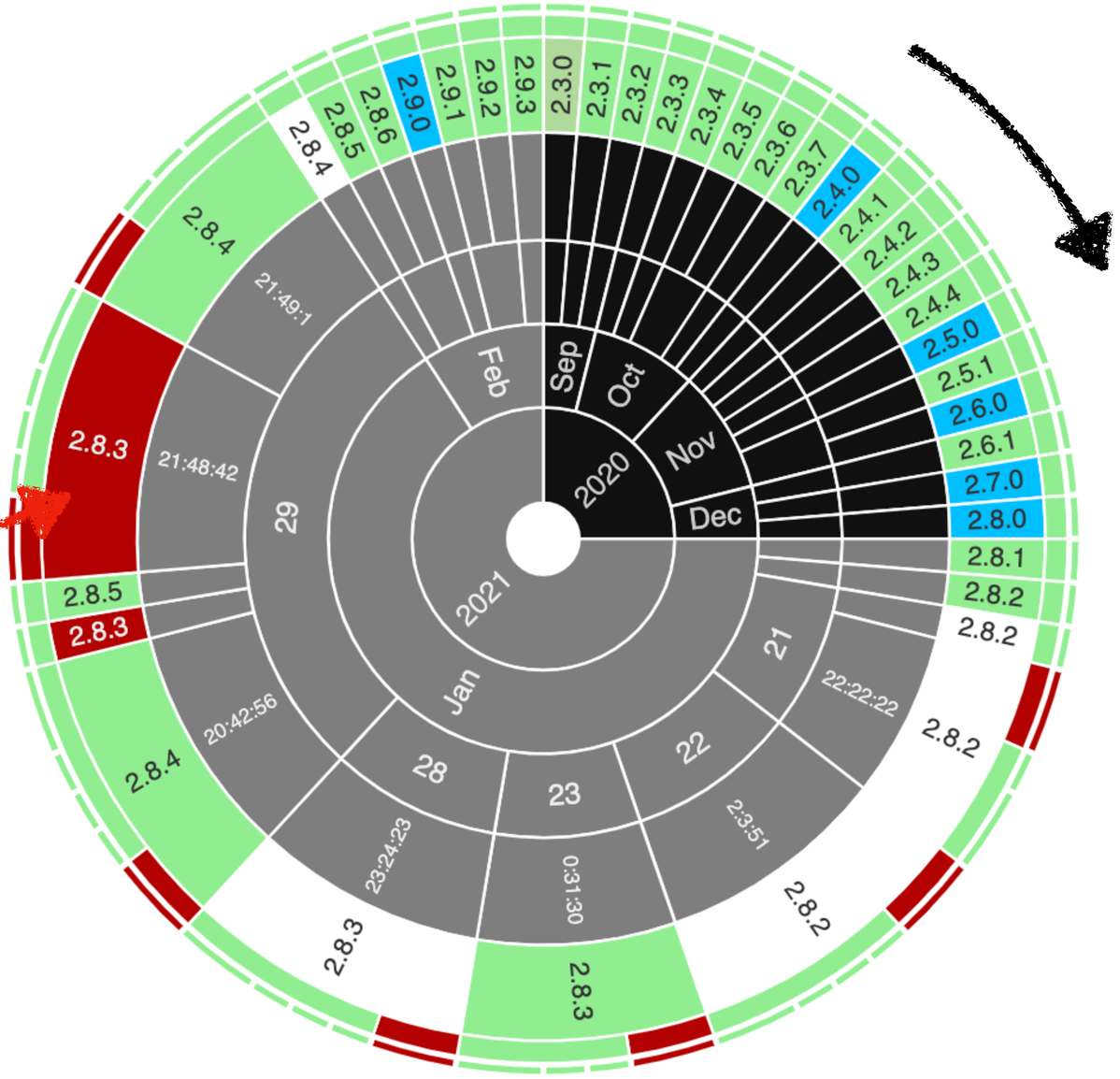


- **Patch**
- **Minor**
- **No version change**
- **Version decrease !**

Xero OAuth 2 Identity Service API

`https://api.xero.com`

First commit : September 2020

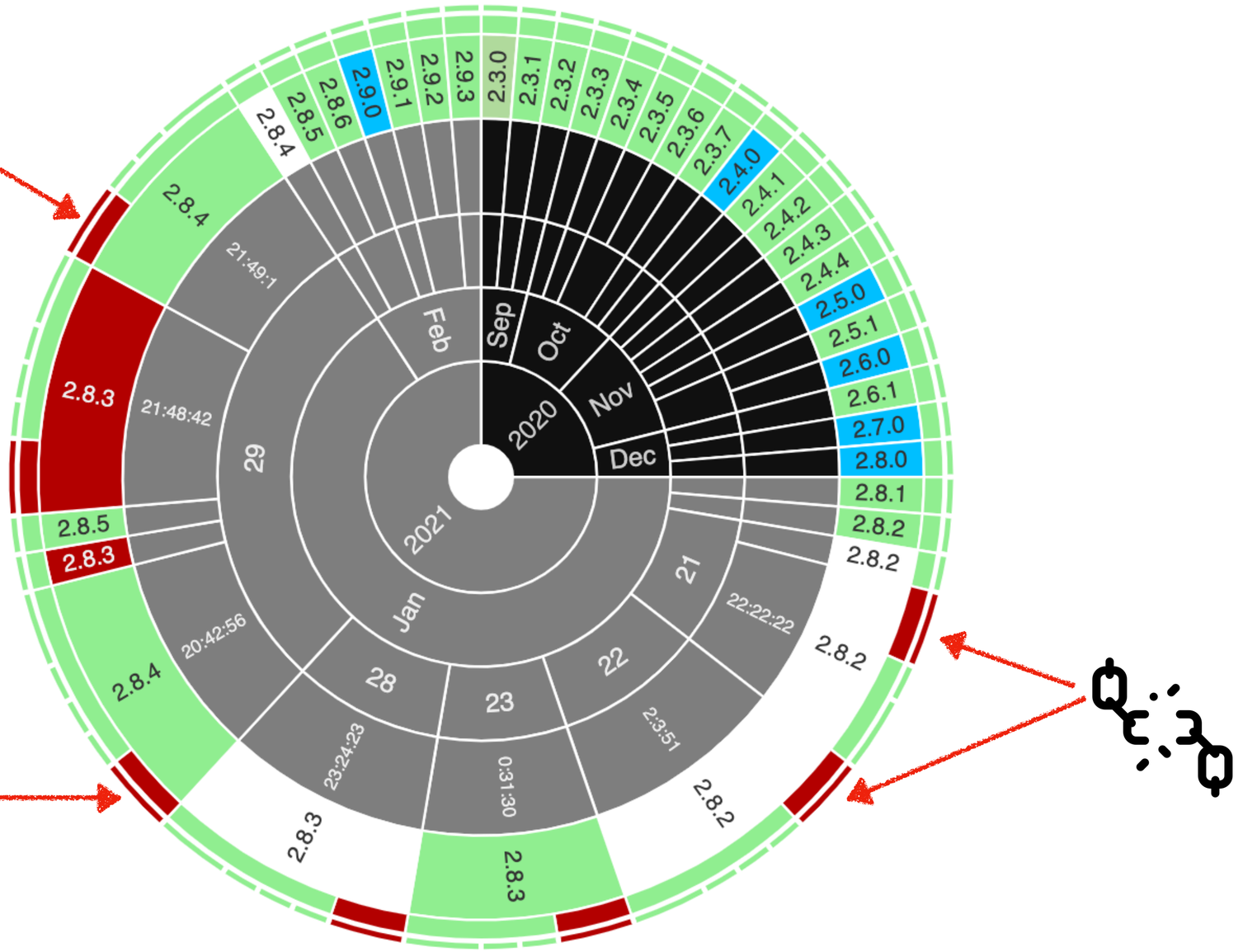


- Patch
- Minor
- No version change
- Version decrease !

Xero OAuth 2 Identity Service API

`https://api.xero.com`

First commit : September 2020



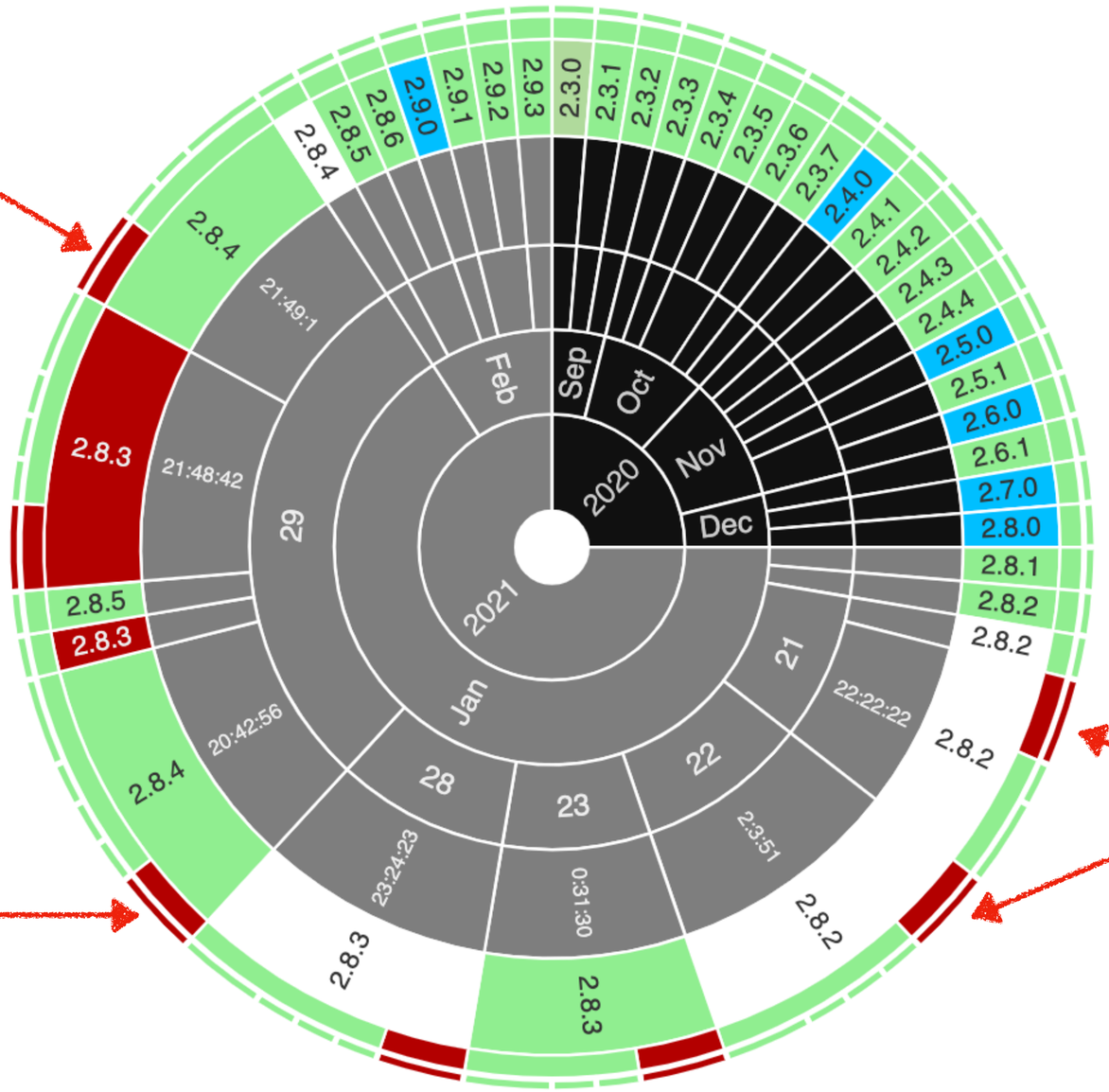
- Patch
- Minor
- No version change
- Version decrease !

No major releases

Xero OAuth 2 Identity Service API

`https://api.xero.com`

First commit : September 2020



- Breaking changes introduced in patches.

- Breaking changes introduced without version change.

Patch

Minor

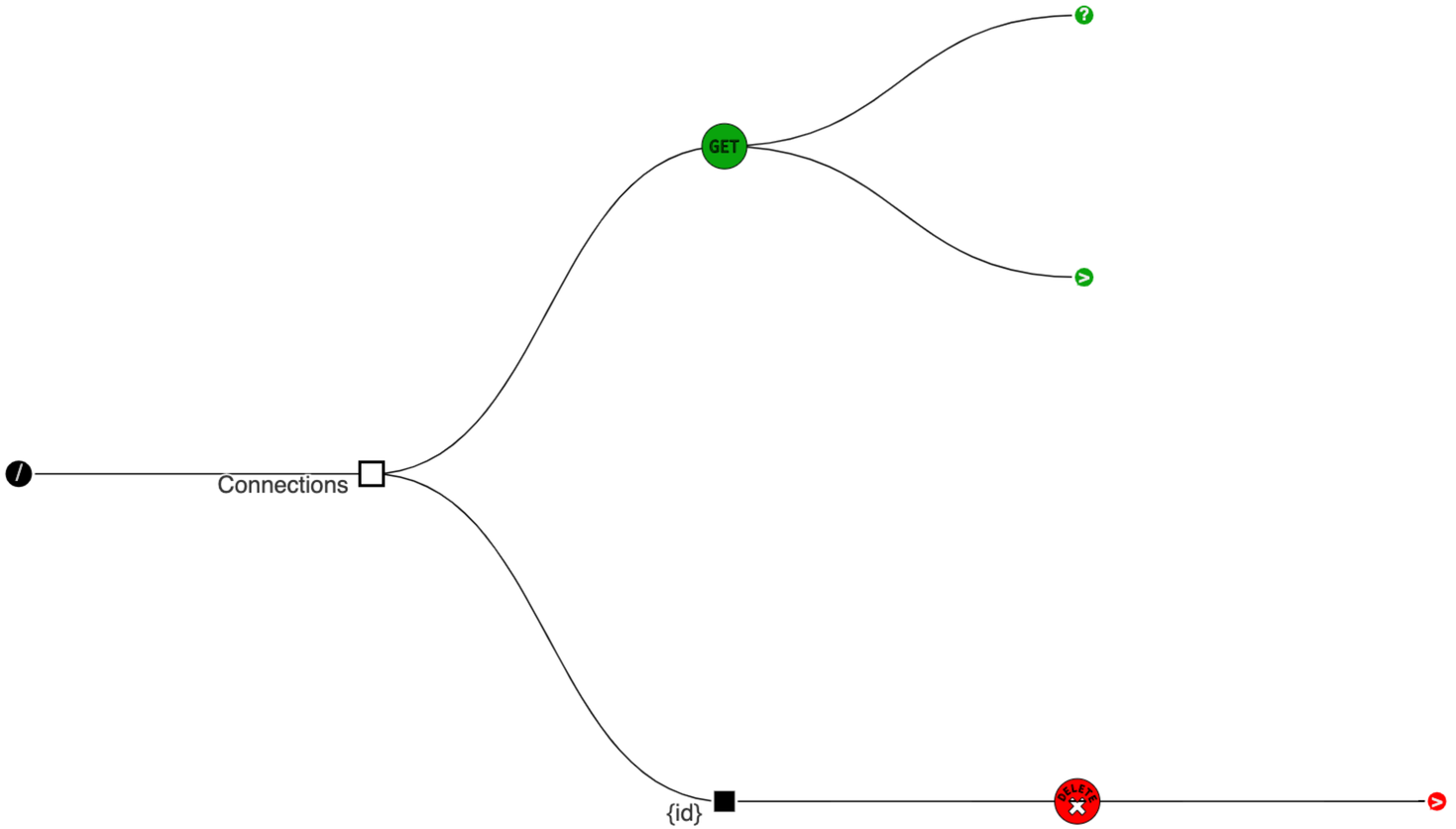
No version change

Version decrease !

No major releases

Xero OAuth 2 Identity Service API

Version: 2.9.4
Description: These endpoints are related to managing authentication tokens and identity for Xero API



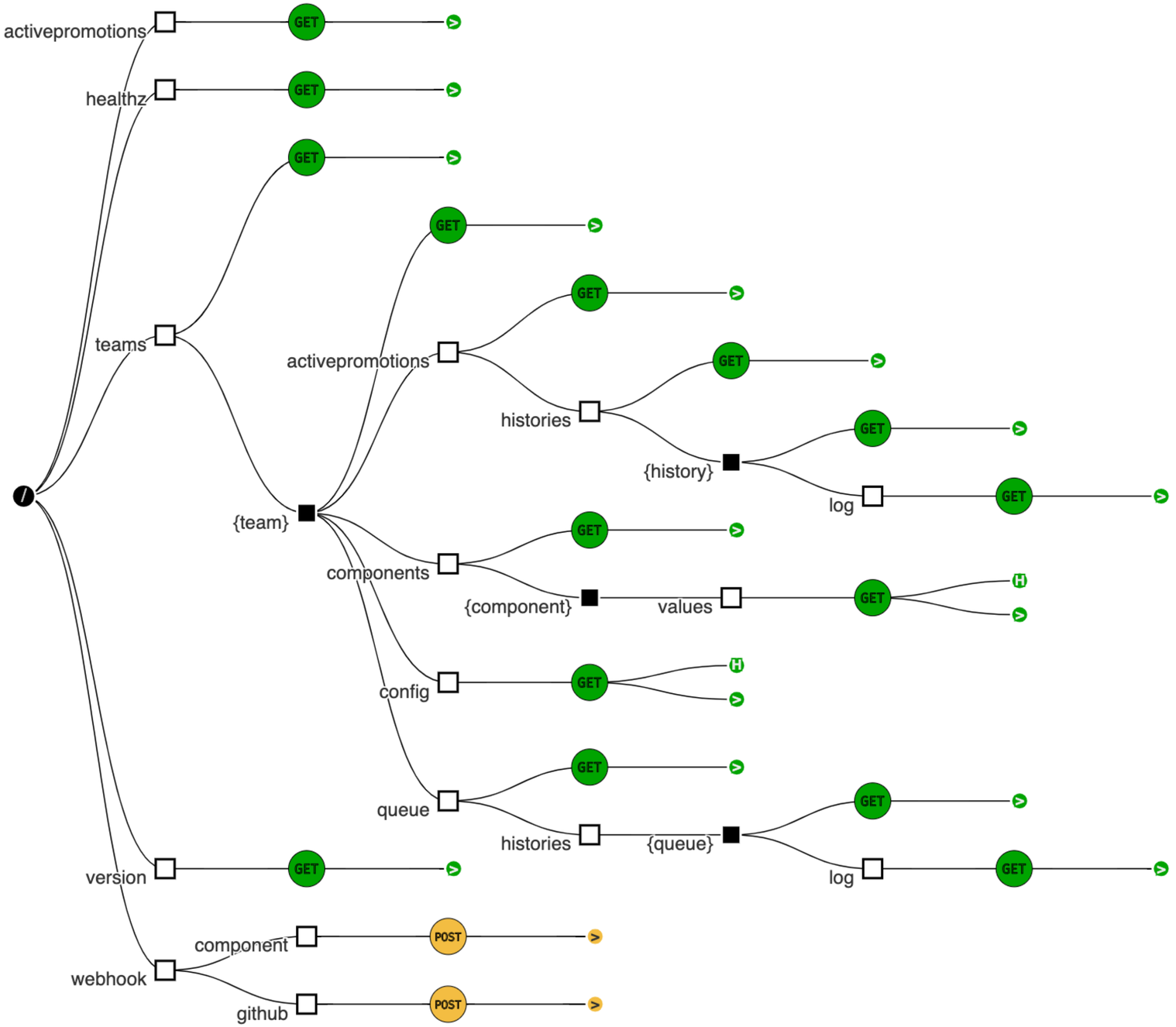
```yaml
openapi: 3.0.0
info:
  version: "2.3.0"
  title: Xero oAuth 2 identity service
  description: This specifing endpoints related to managing authentication tokens
    and identity for Xero API
  termsOfService:
  "https://developer.xero.com/xero-developer-platform-terms-conditions/"
  contact:
    name: "Xero Platform Team"
    email: "api@xero.com"
    url: "https://developer.xero.com"
  license:
    name: MIT
    url: 'https://github.com/XeroAPI/Xero-OpenAPI/blob/master/LICENSE'
servers:
  - description: Xero Identity service API
    url: 'https://api.xero.com'
```

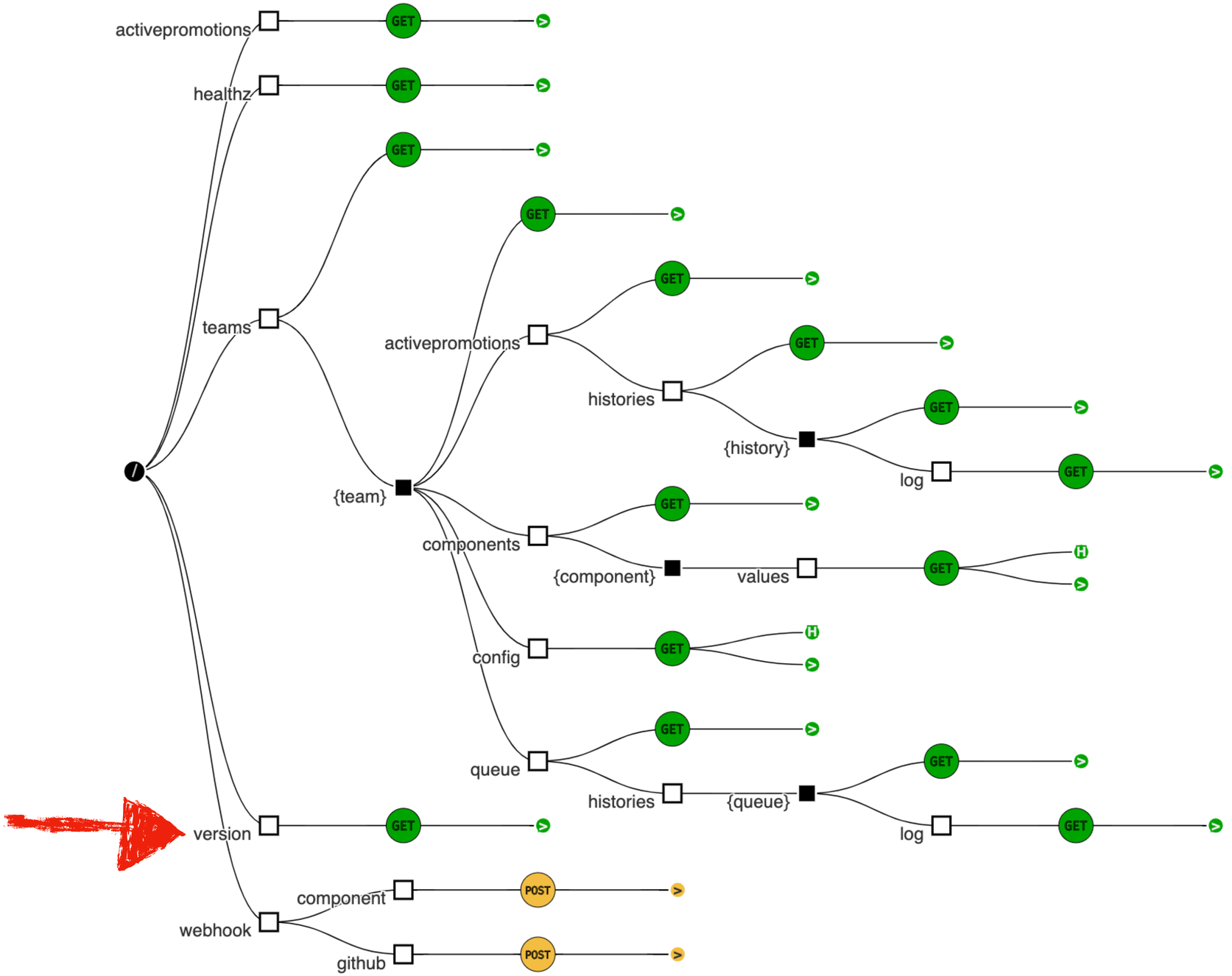
Metadata-based versioning

```yaml
openapi: 3.0.0
info:
  version: "2.3.0"
  title: Xero oAuth 2 identity service
  description: This specifing endpoints related to managing authentication tokens
    and identity for Xero API
  termsOfService:
  "https://developer.xero.com/xero-developer-platform-terms-conditions/"
  contact:
    name: "Xero Platform Team"
    email: "api@xero.com"
    url: "https://developer.xero.com"
  license:
    name: MIT
    url: 'https://github.com/XeroAPI/Xero-OpenAPI/blob/master/LICENSE'
servers:
  - description: Xero Identity service API
    url: 'https://api.xero.com'
```

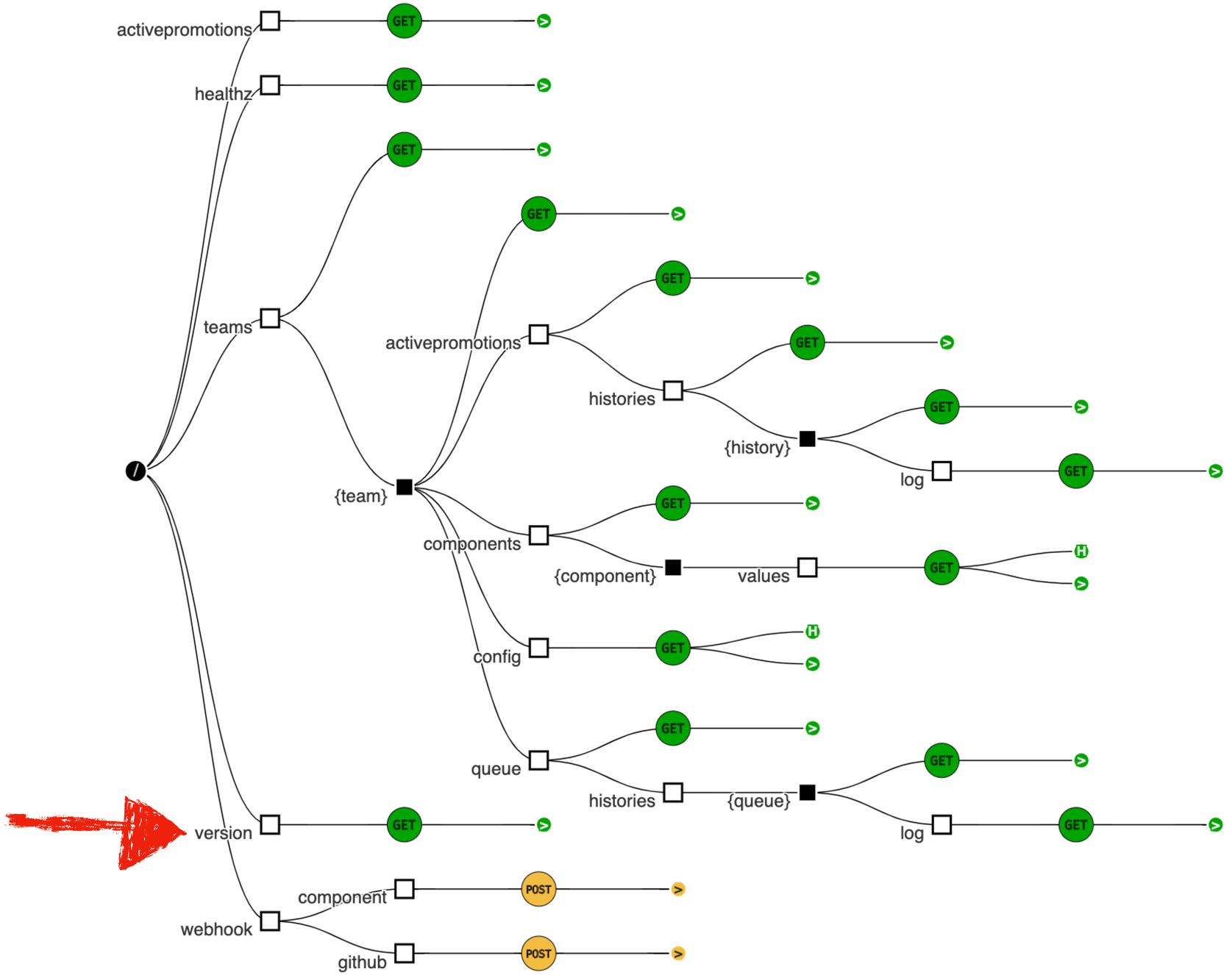
Metadata-based versioning

```yaml
openapi: 3.0.0
info:
  version: "2.3.0"
  title: Xero oAuth 2 identity service
  description: This specifing endpoints related to managing authentication tokens
    and identity for Xero API
  termsOfService:
  "https://developer.xero.com/xero-developer-platform-terms-conditions/"
  contact:
    name: "Xero Platform Team"
    email: "api@xero.com"
    url: "https://developer.xero.com"
  license:
    name: MIT
    url: 'https://github.com/XeroAPI/Xero-OpenAPI/blob/master/LICENSE'
servers:
  - description: Xero Identity service API
    url: 'https://api.xero.com'
```

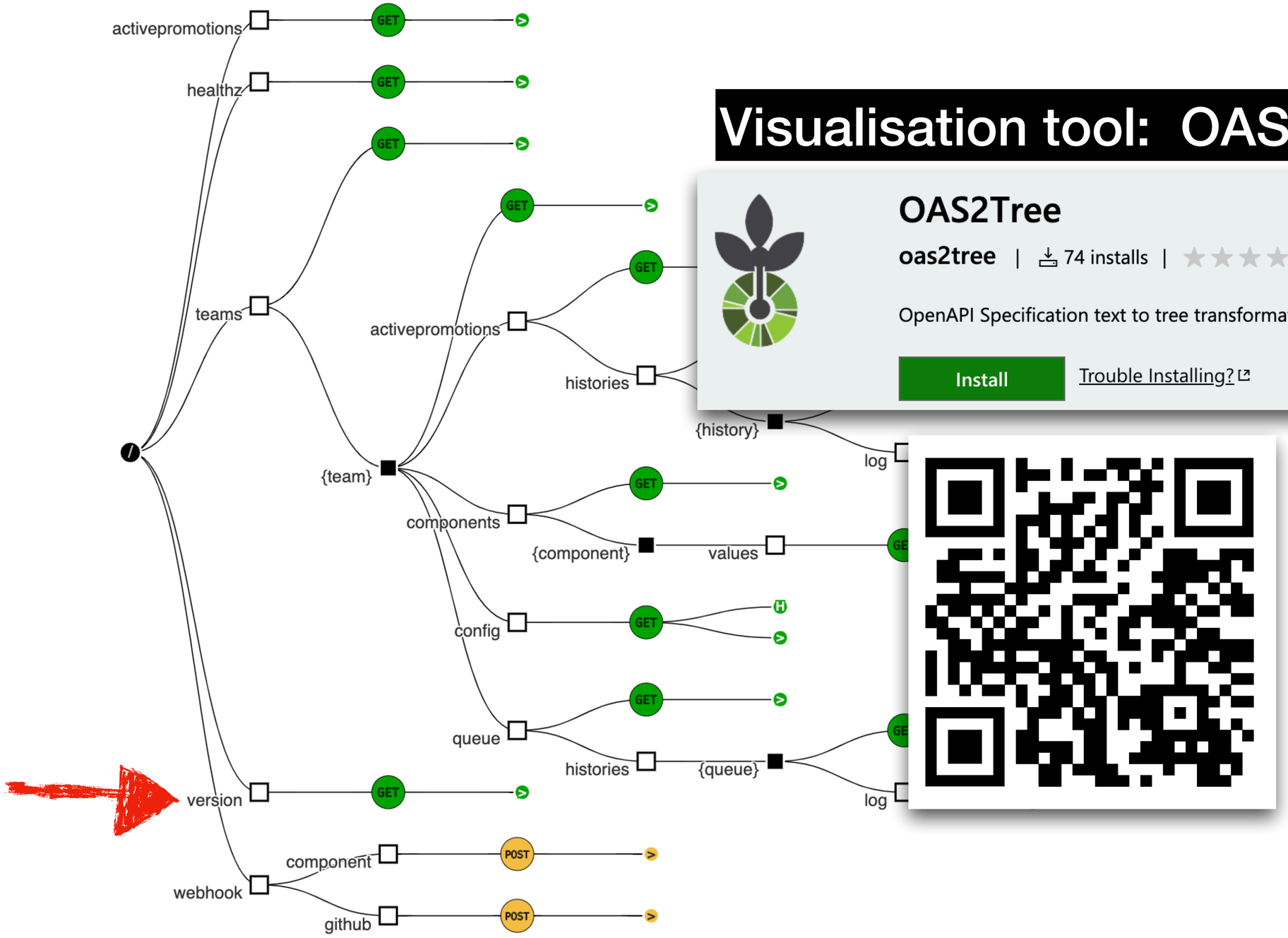

activepromotions
GET
healthz
GET
GET
teams
GET
activepromotions
GET
histories
GET
{history}
GET
log
GET
/
{team}
components
GET
{component}
values
GET
config
GET
queue
GET
histories
{queue}
GET
log
GET
version
GET
webhook
component
POST
github
POST


activepromotions
GET
healthz
GET
GET
teams
activepromotions
GET
GET
histories
GET
{history}
GET
log
GET
/
{team}
components
GET
{component}
values
GET
config
GET
queue
GET
histories
{queue}
GET
log
GET
version
GET
webhook
component
POST
github
POST

Dynamic Versioning

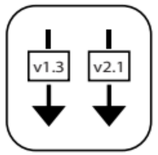


Dynamic Versioning



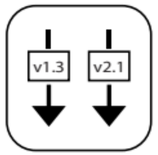
Visualisation tool: OAS2Tree


Um, dude?! There's a problem with the API!
Yeah, that makes sense...
It's because we deployed a new version, and all the endpoints have changed
Are you kidding? And when were you planning on telling us??
Uh... I mean I was planning to tell you...
Really? When??
Right now! I'm telling you about it right now! Now you know about the changes!
Now go fix your code



“All in production” interface evolution pattern

[7] "Interface evolution patterns: Balancing compatibility and extensibility across service life cycles." *Proceedings of the 24th European Conference on Pattern Languages of Programs*. 2019.

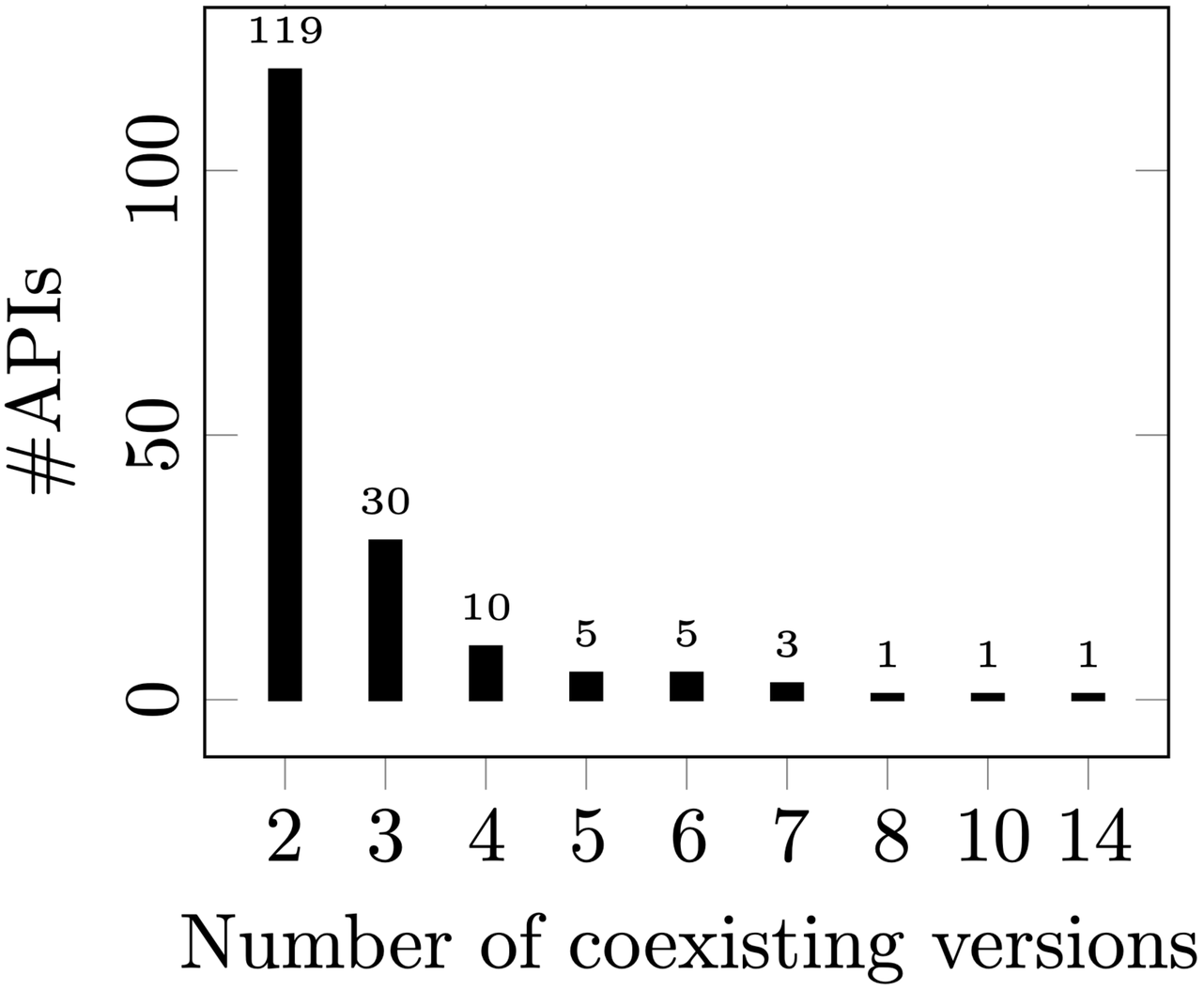


Two

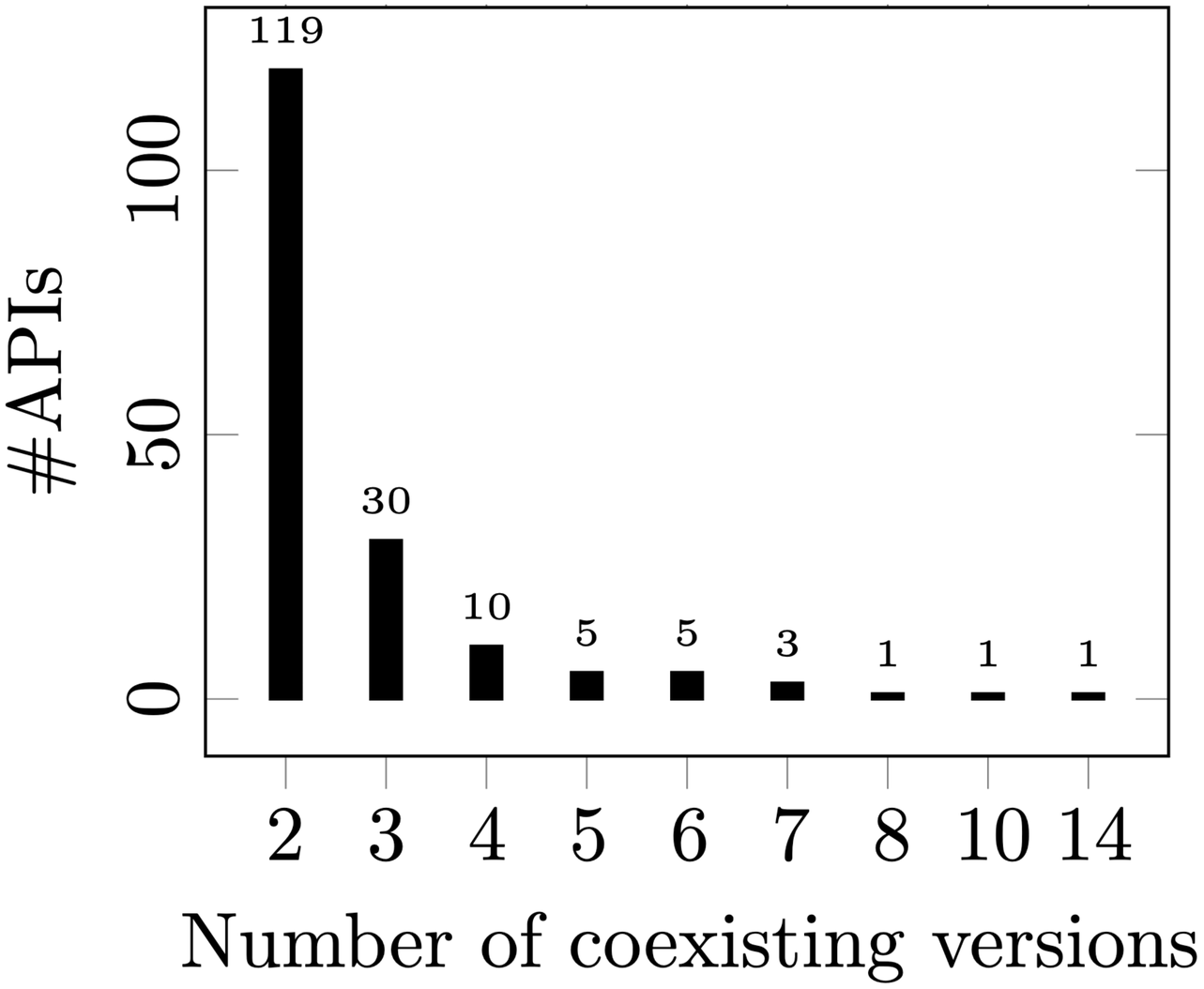
“~~All~~ in production” interface evolution pattern

[7] "Interface evolution patterns: Balancing compatibility and extensibility across service life cycles." *Proceedings of the 24th European Conference on Pattern Languages of Programs*. 2019.

APIs with multiple coexistent versions



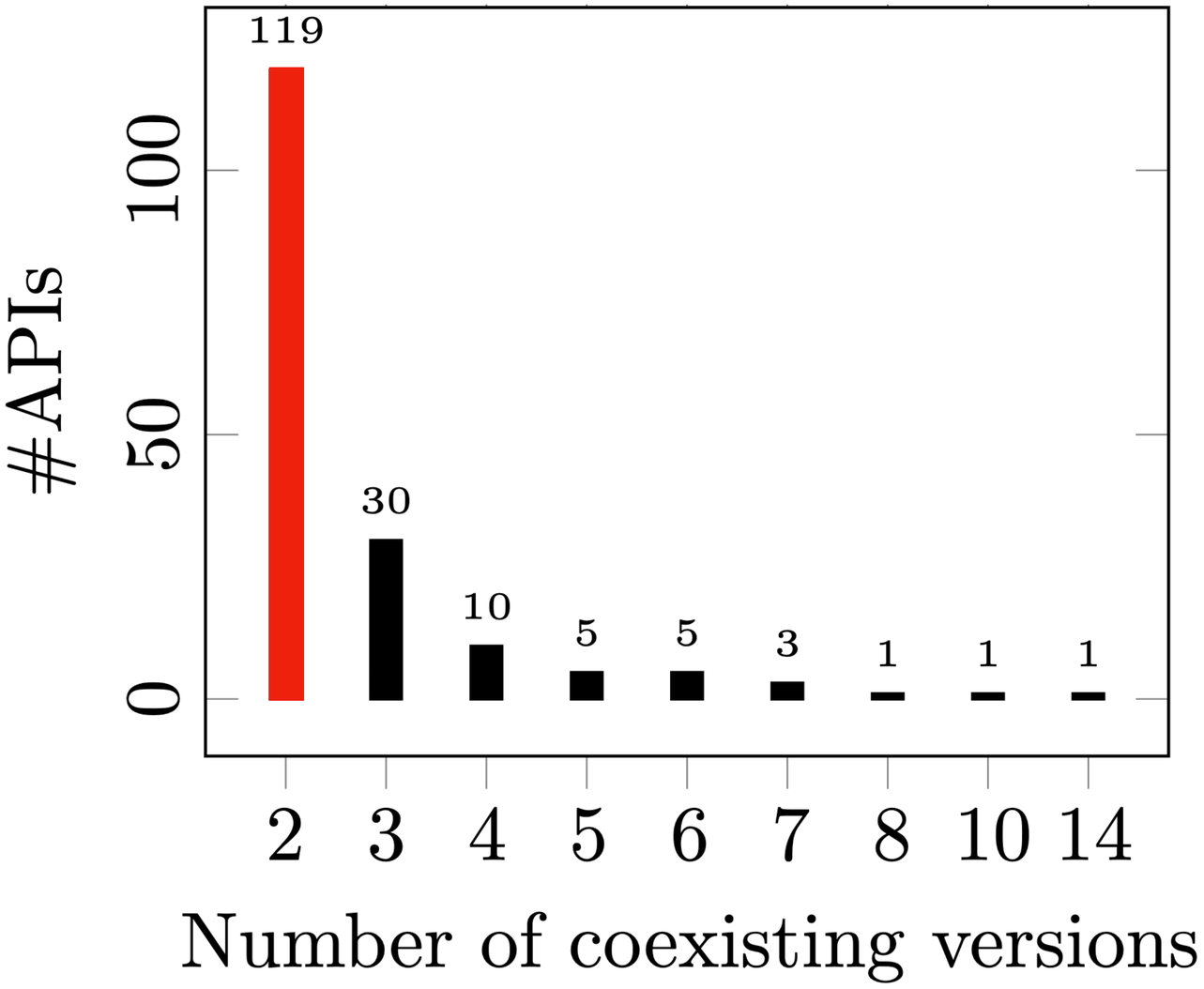
APIs with multiple coexistent versions



Among 7,114 Web APIs

175 Web API adopting the “Two in production” Interface evolution pattern

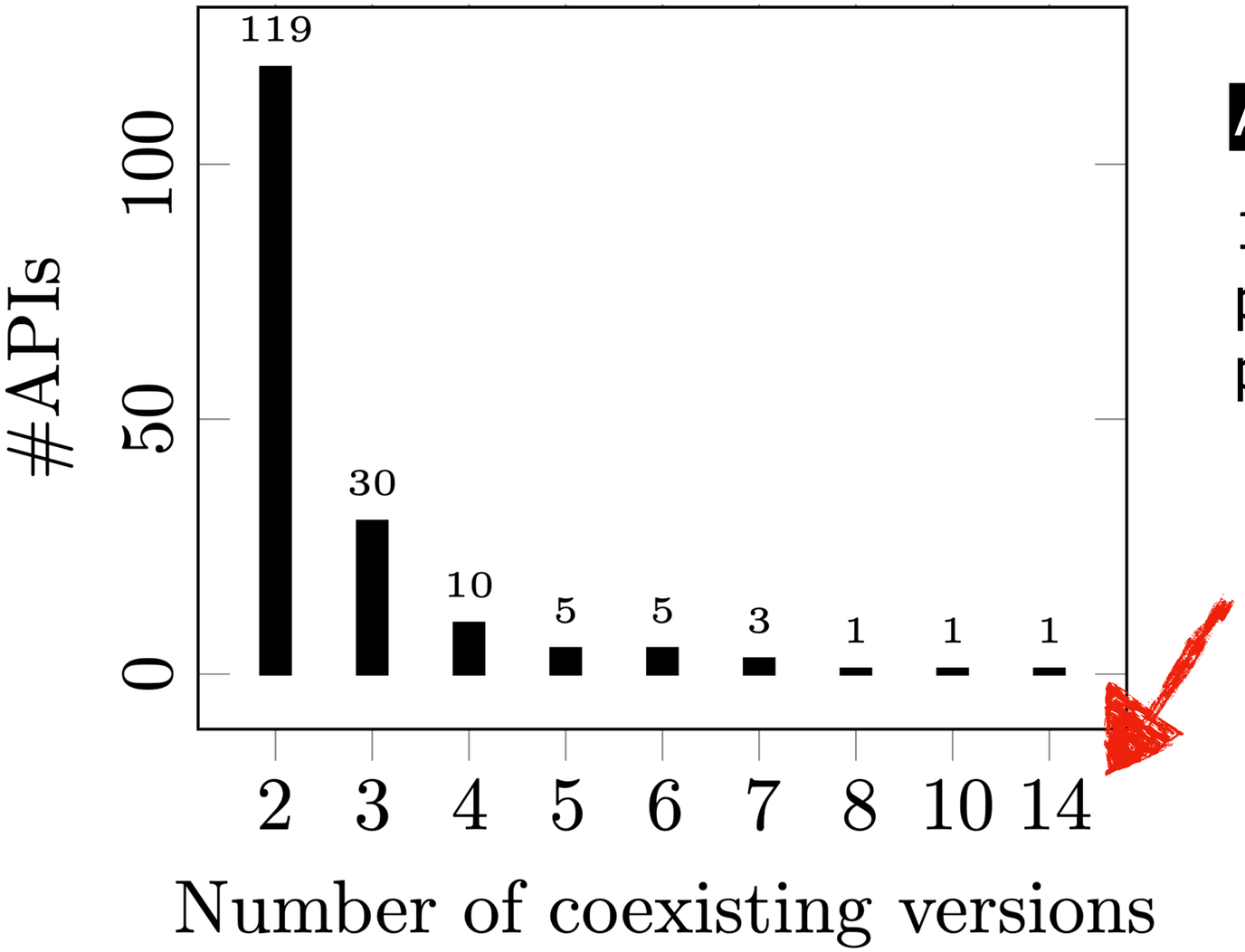
APIs with multiple coexistent versions



Among 7,114 Web APIs

175 Web API adopting the “Two in production” Interface evolution pattern

APIs with multiple coexistent versions



Among 7,114 Web APIs

175 Web API adopting the “Two in production” Interface evolution pattern

Path-based versioning

`https://{DomainName}/{basePath}`

Path-based versioning

`https://{DomainName}/{basePath}`

`http://myAPI.domain.com/v1/ressources`

Path-based versioning

```
https://{DomainName}/{basePath}
```

```
http://myAPI.domain.com/v1/ressources
```

```
http://myAPI.domain.com/v2/ressources
```

```
http://myAPI.domain.com/v3/ressources
```

.
.
.
.

Versioning strategy should be defined upfront

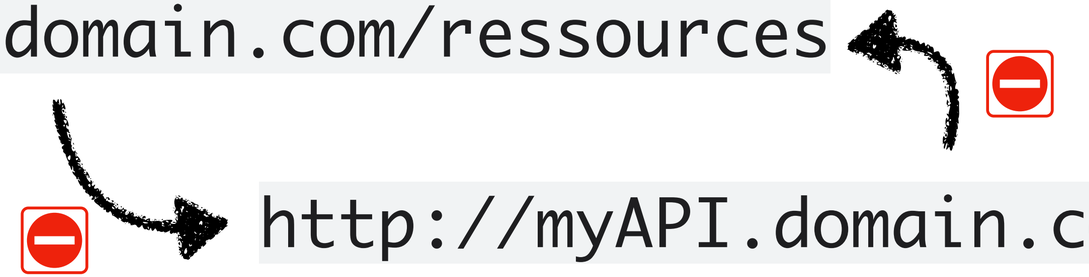
Versioning strategy should be defined upfront

`http://myAPI.domain.com/ressources`

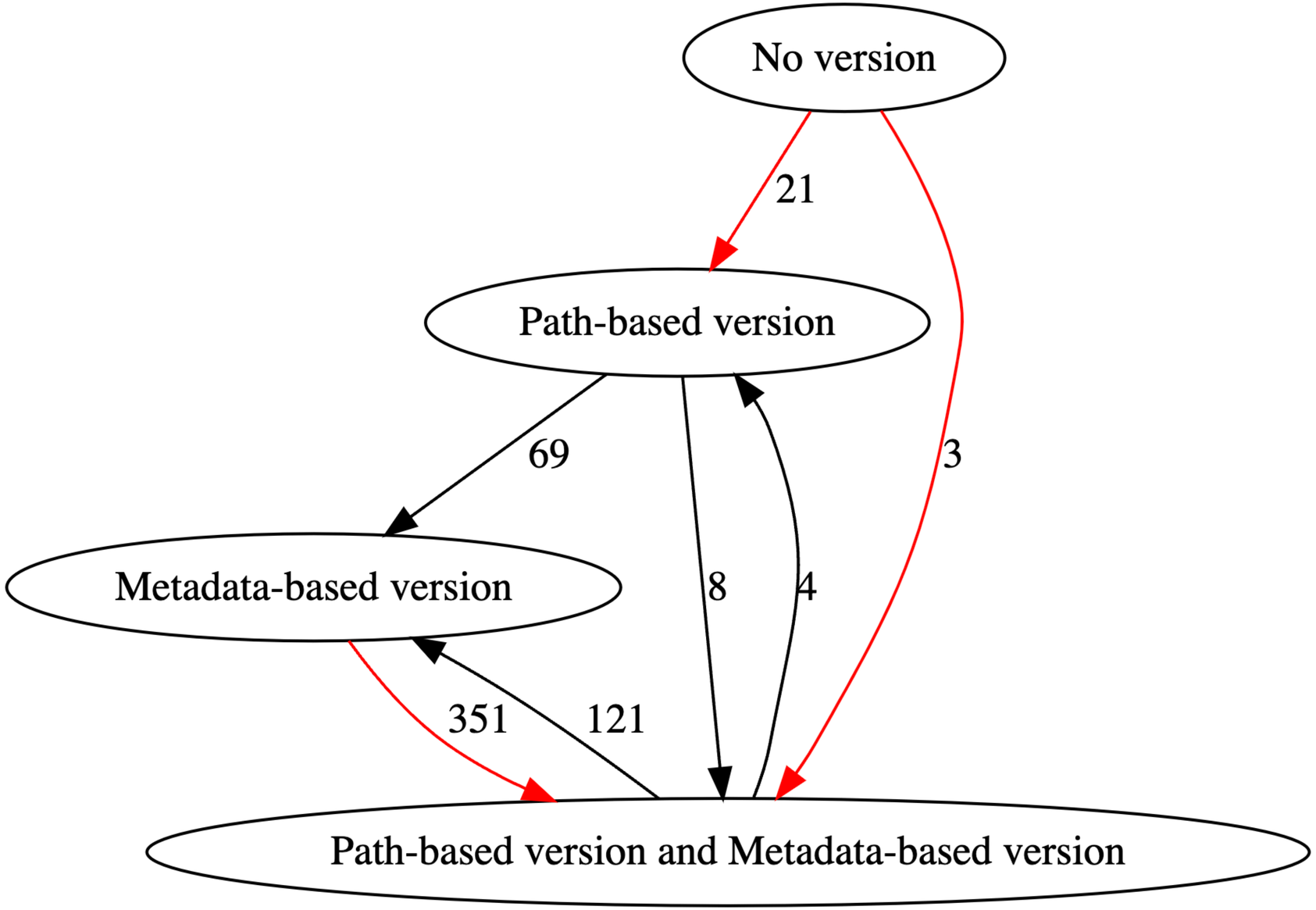
`http://myAPI.domain.com/v1/ressources`

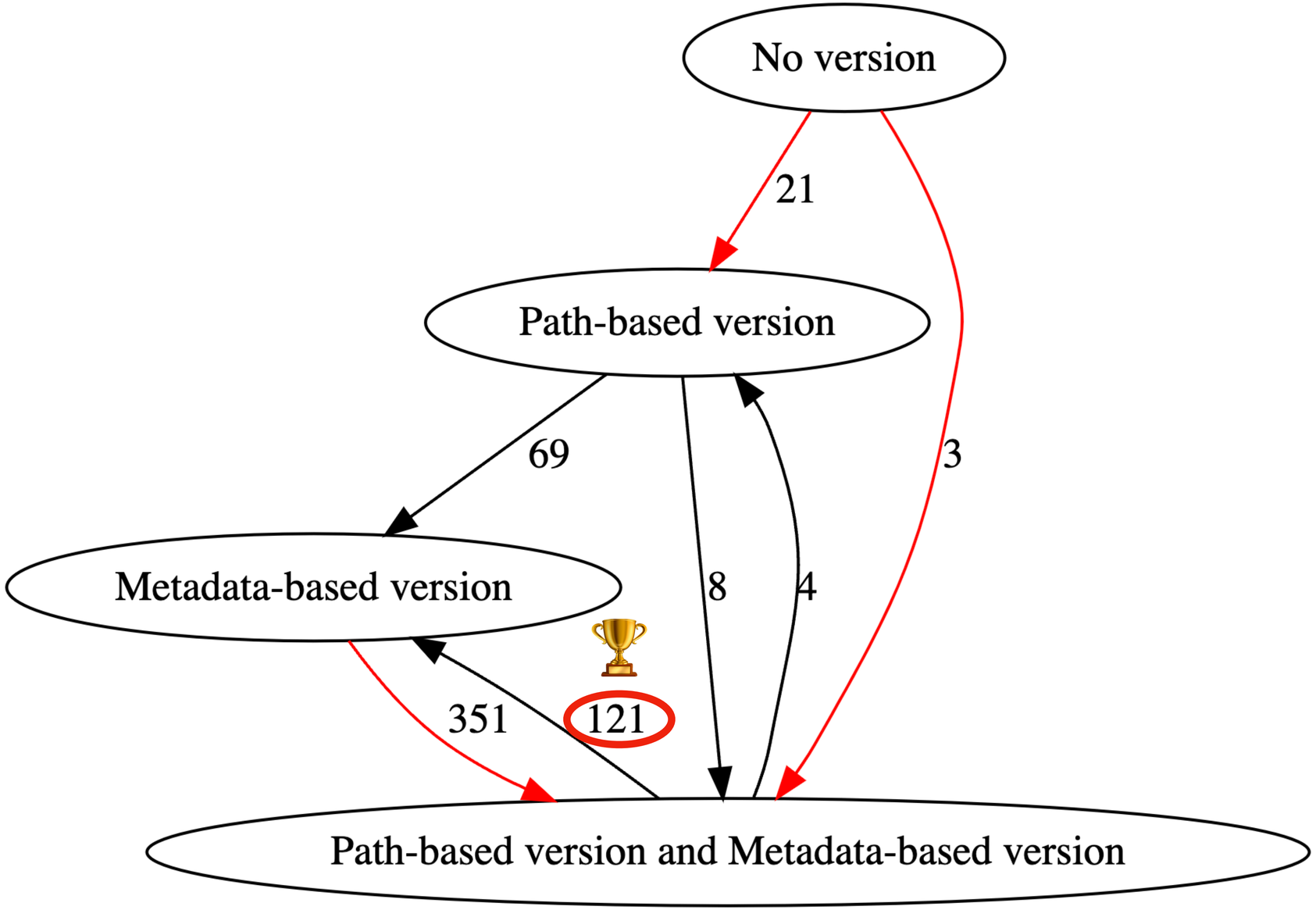
Versioning strategy should be defined upfront

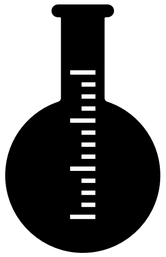
`http://myAPI.domain.com/ressources`



`http://myAPI.domain.com/v1/ressources`


No version
21
Path-based version
69
3
Metadata-based version
8
4
351
121
Path-based version and Metadata-based version


No version
21
Path-based version
69
3
Metadata-based version
8
4
351
121
Path-based version and Metadata-based version



7,114 Web APIs

186,259 Commits

Path-based versioning	Meta data-based versioning	Dynamic versioning
1%	**36%**	70%	3%

What are the **formats** of the version identifiers?

OpenAPI specification of

GitHub API

```yaml
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
    - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                format: date
                example: '2021-01-01'
            examples:
              default:
                value:
                - '2021-01-01'
                - '2021-06-01'
                - '2022-01-01'
```

OpenAPI specification of

GitHub API

```yaml
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
    - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                format: date
                example: '2021-01-01'
            examples:
              default:
                value:
                - '2021-01-01'
                - '2021-06-01'
                - '2022-01-01'
```

OpenAPI specification of

GitHub API

```yaml
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
    - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                format: date
                example: '2021-01-01'
            examples:
              default:
                value:
                - '2021-01-01'
                - '2021-06-01'
                - '2022-01-01'
```

OpenAPI specification of

GitHub API

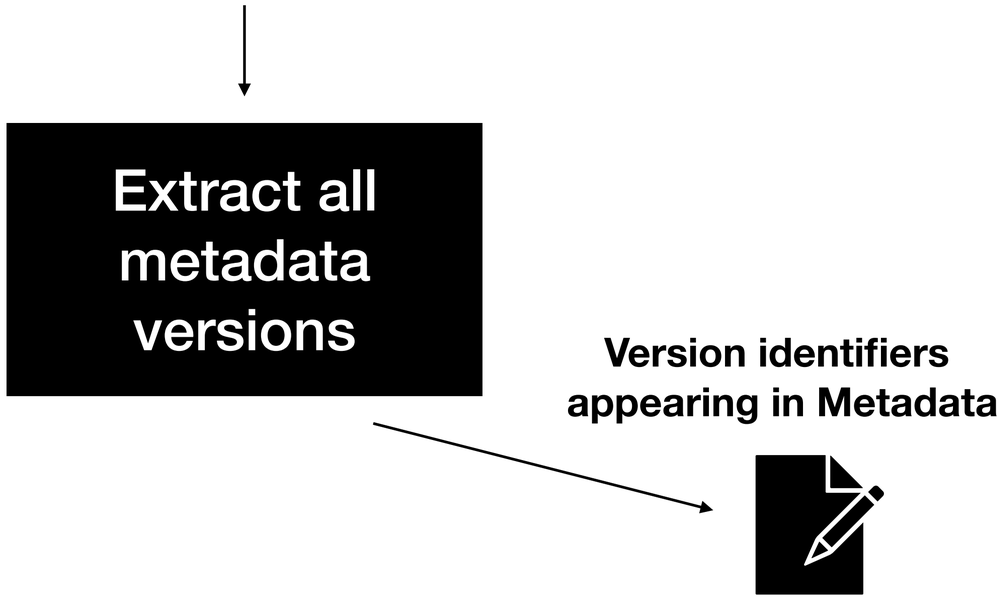
```yaml
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
    - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                format: date
                example: '2021-01-01'
            examples:
              default:
                value:
                - '2021-01-01'
                - '2021-06-01'
                - '2022-01-01'
```

Calendar Versioning (CalVer)

186,259 API Specs

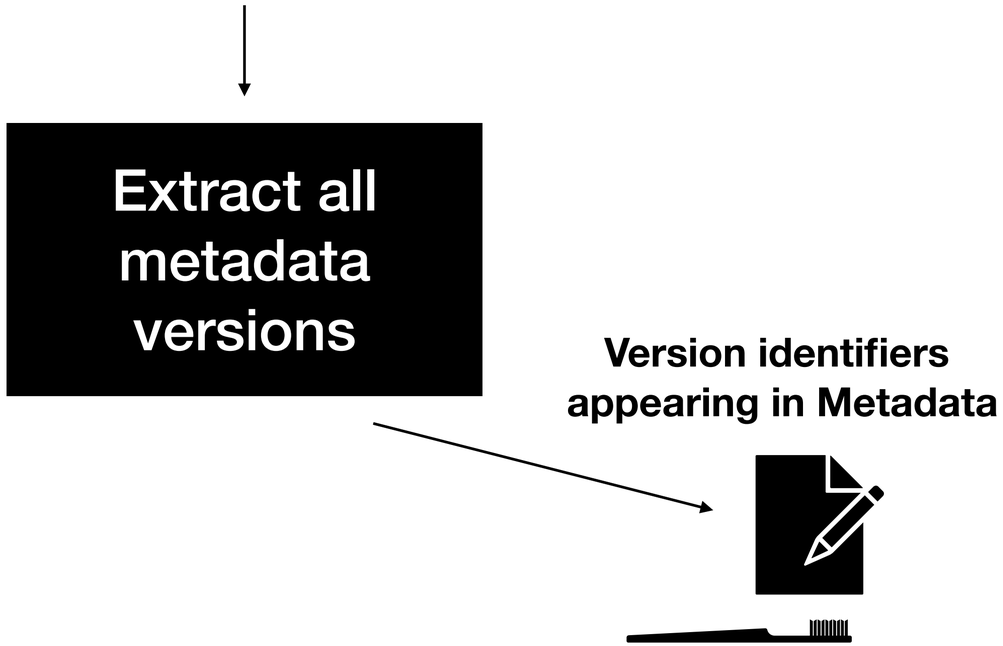
Extract all metadata versions

186,259 API Specs

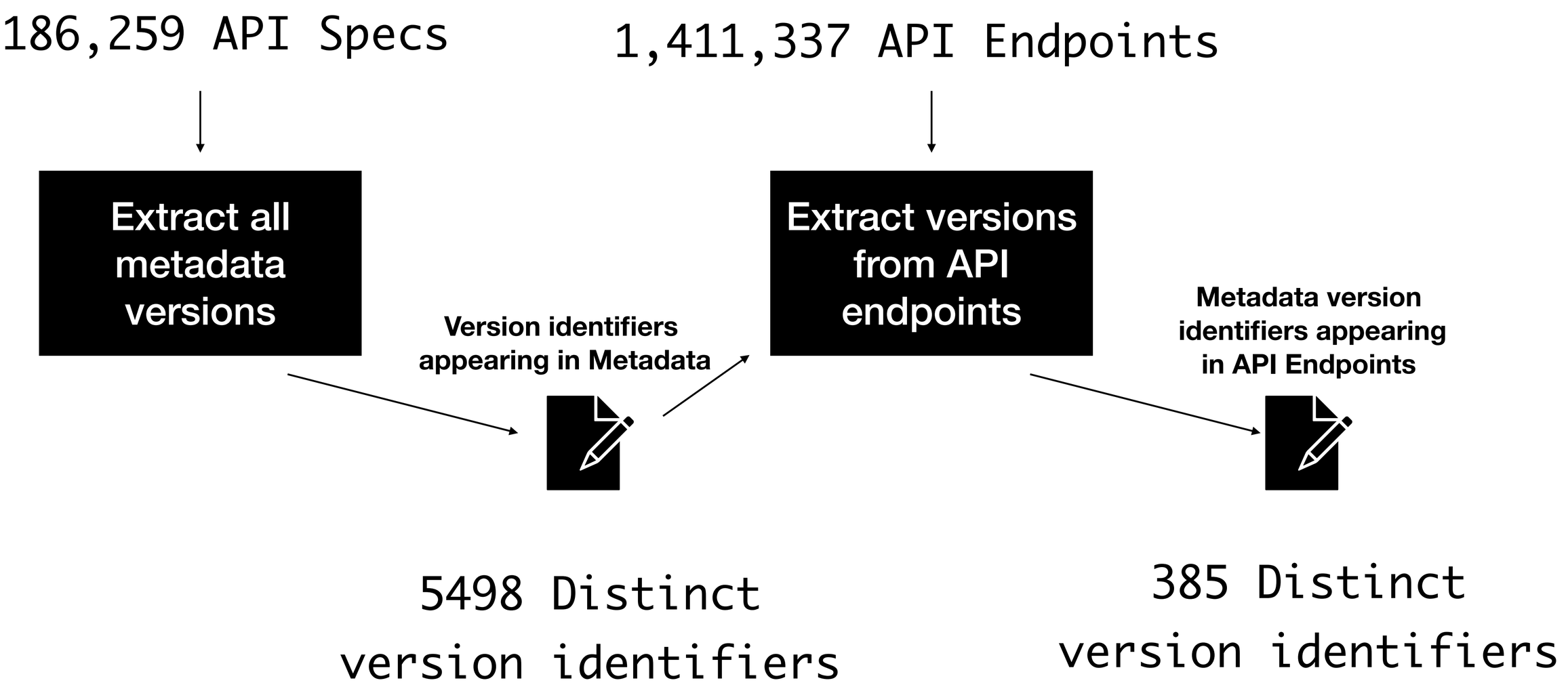


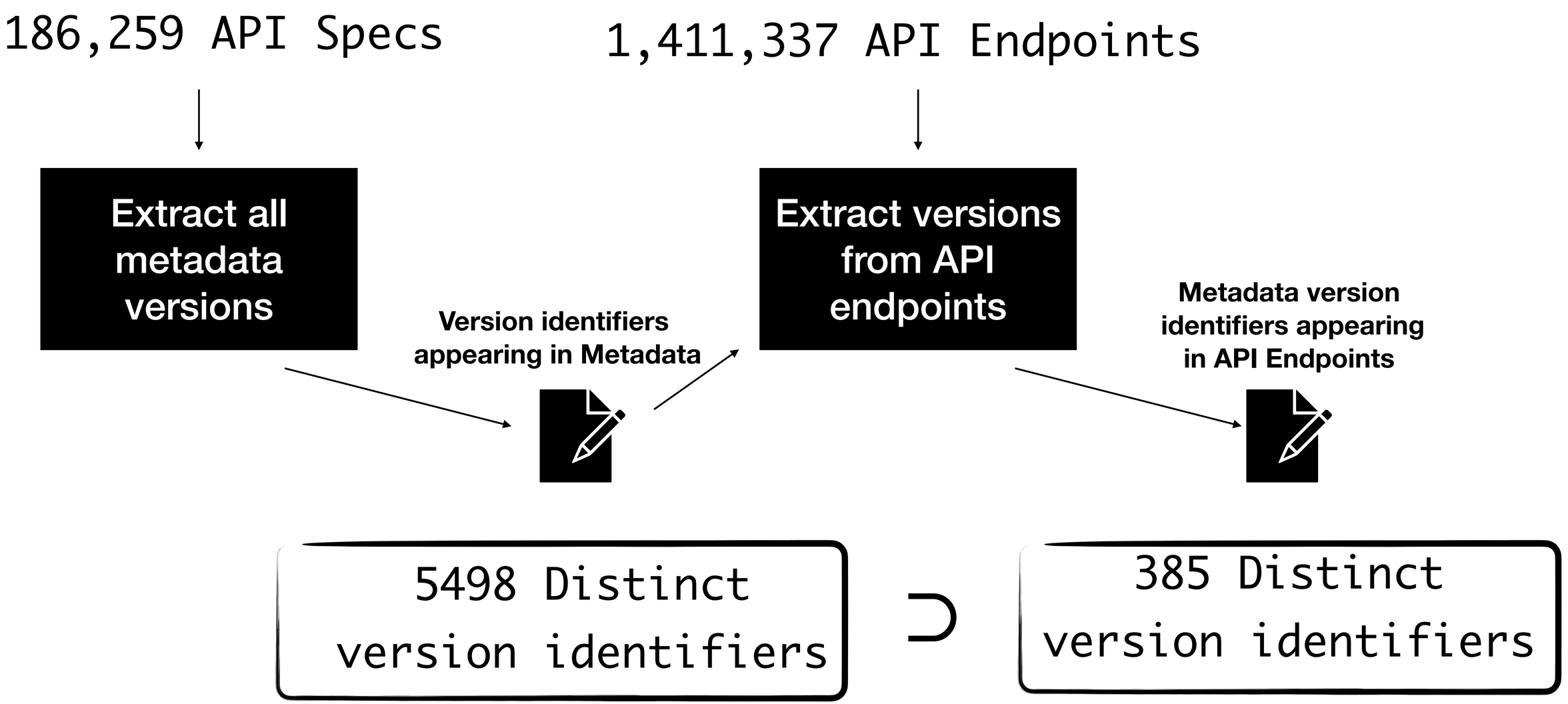
5511 Distinct version identifiers

186,259 API Specs



5498 Distinct version identifiers

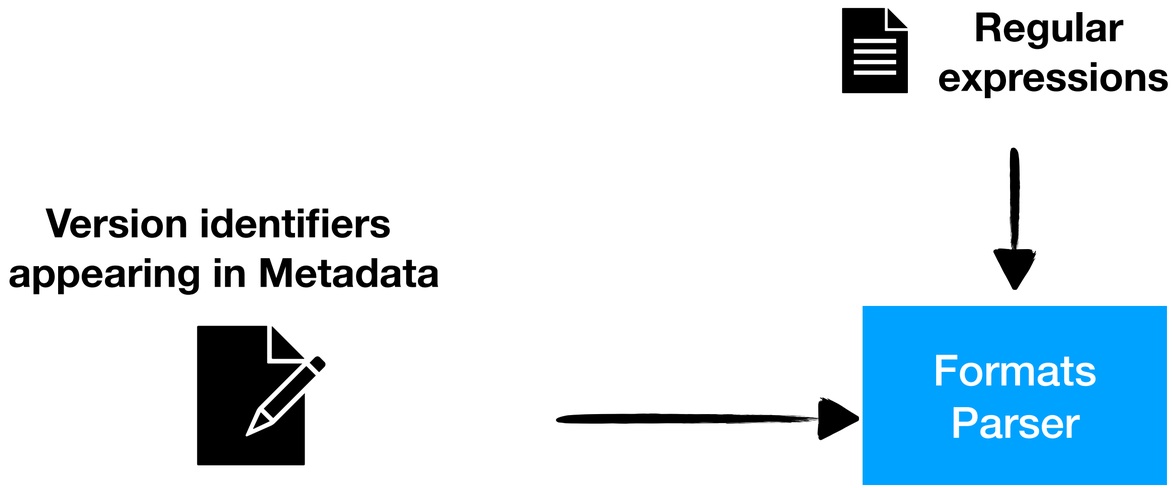

186,259 API Specs
1,411,337 API Endpoints
Extract all metadata versions
Version identifiers appearing in Metadata
Extract versions from API endpoints
Metadata version identifiers appearing in API Endpoints
5498 Distinct version identifiers
385 Distinct version identifiers


186,259 API Specs
1,411,337 API Endpoints
Extract all metadata versions
Version identifiers appearing in Metadata
Extract versions from API endpoints
Metadata version identifiers appearing in API Endpoints
5498 Distinct version identifiers
⊃
385 Distinct version identifiers

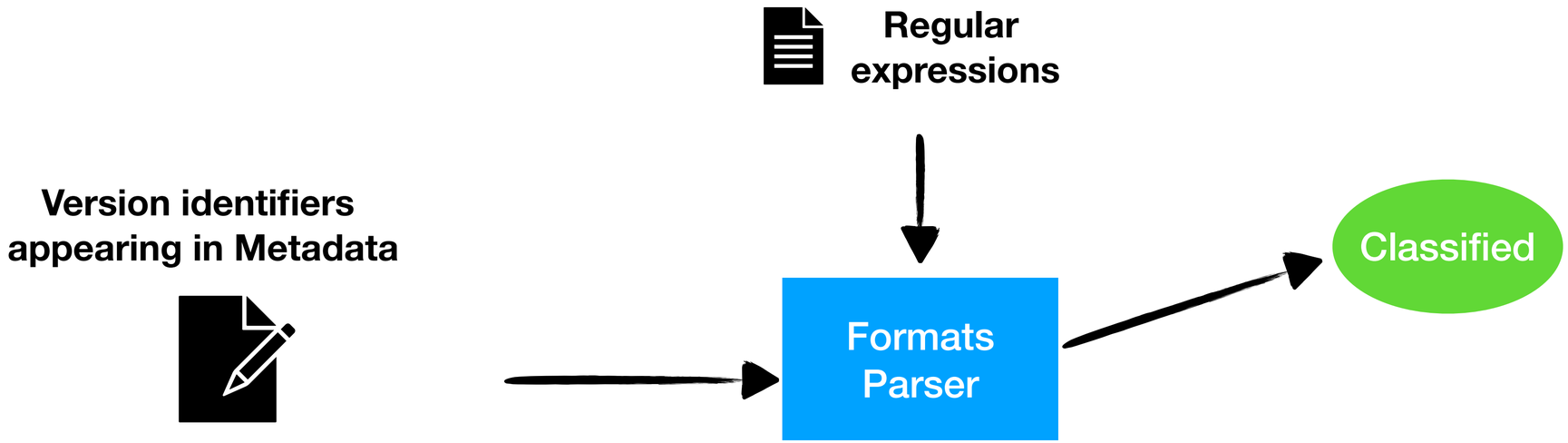
Version identifiers appearing in Metadata

5511 Distinct version identifiers

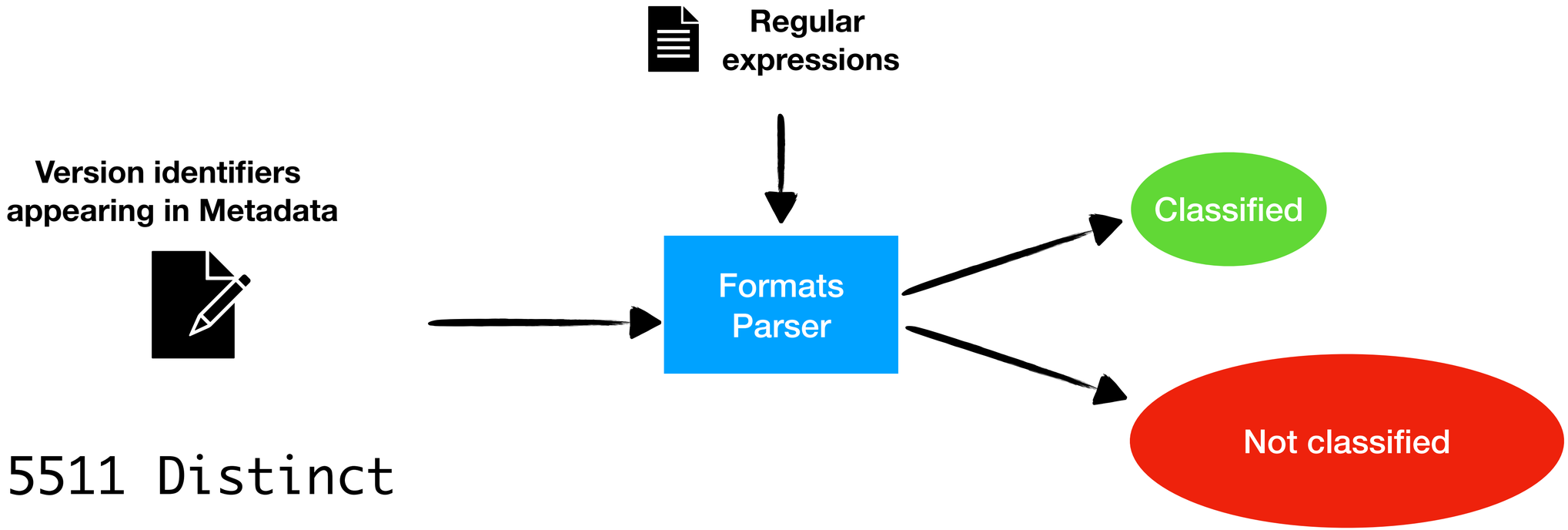
5511 Distinct
version identifiers



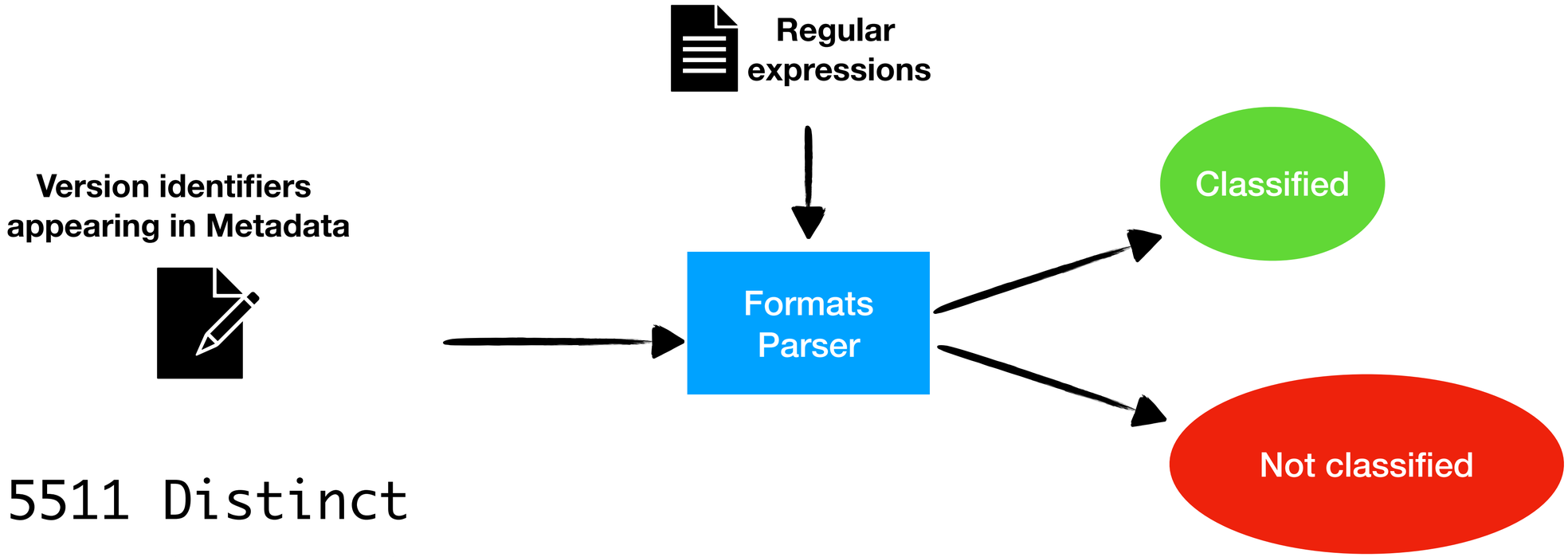
5511 Distinct
version identifiers



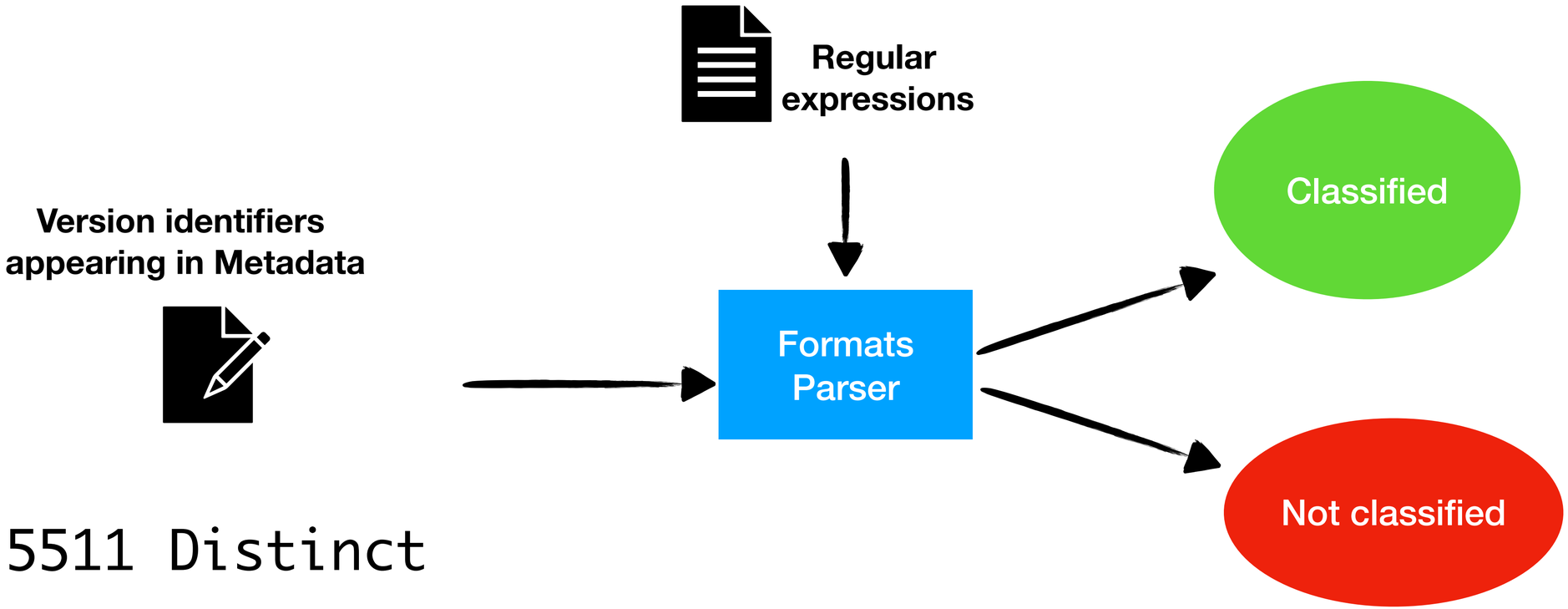
5511 Distinct
version identifiers



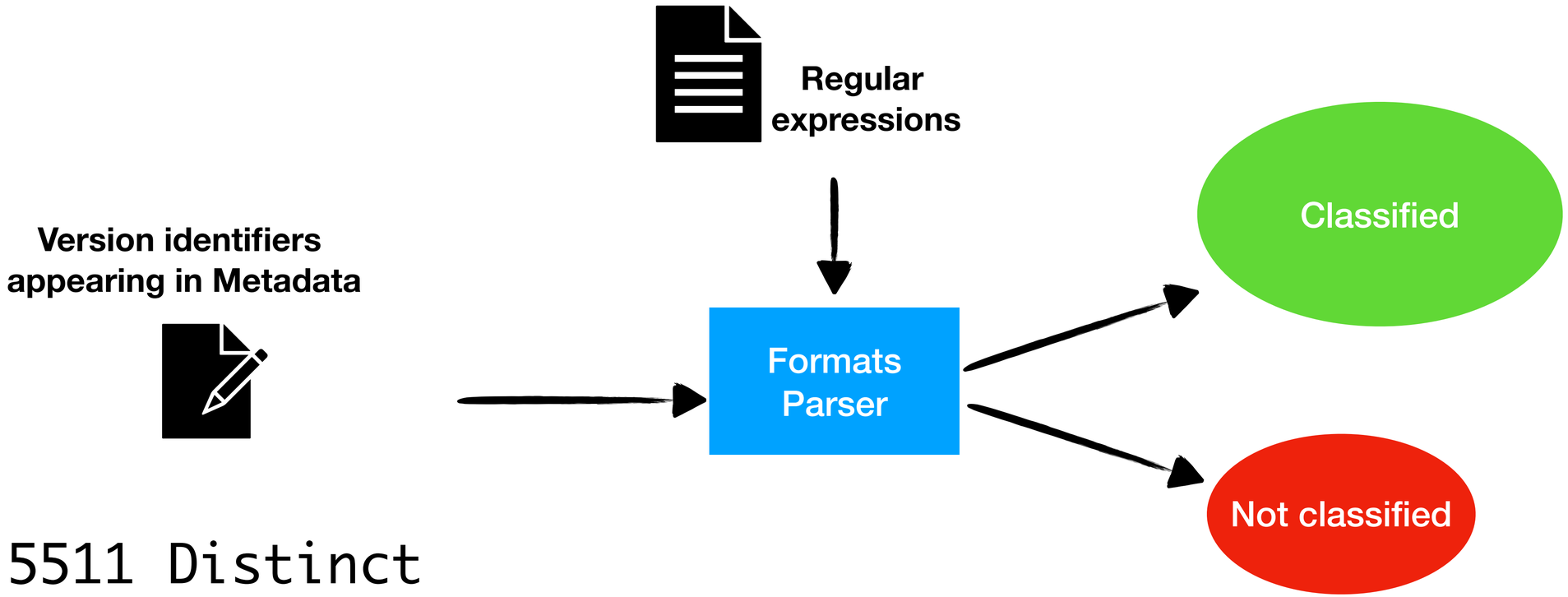
5511 Distinct version identifiers



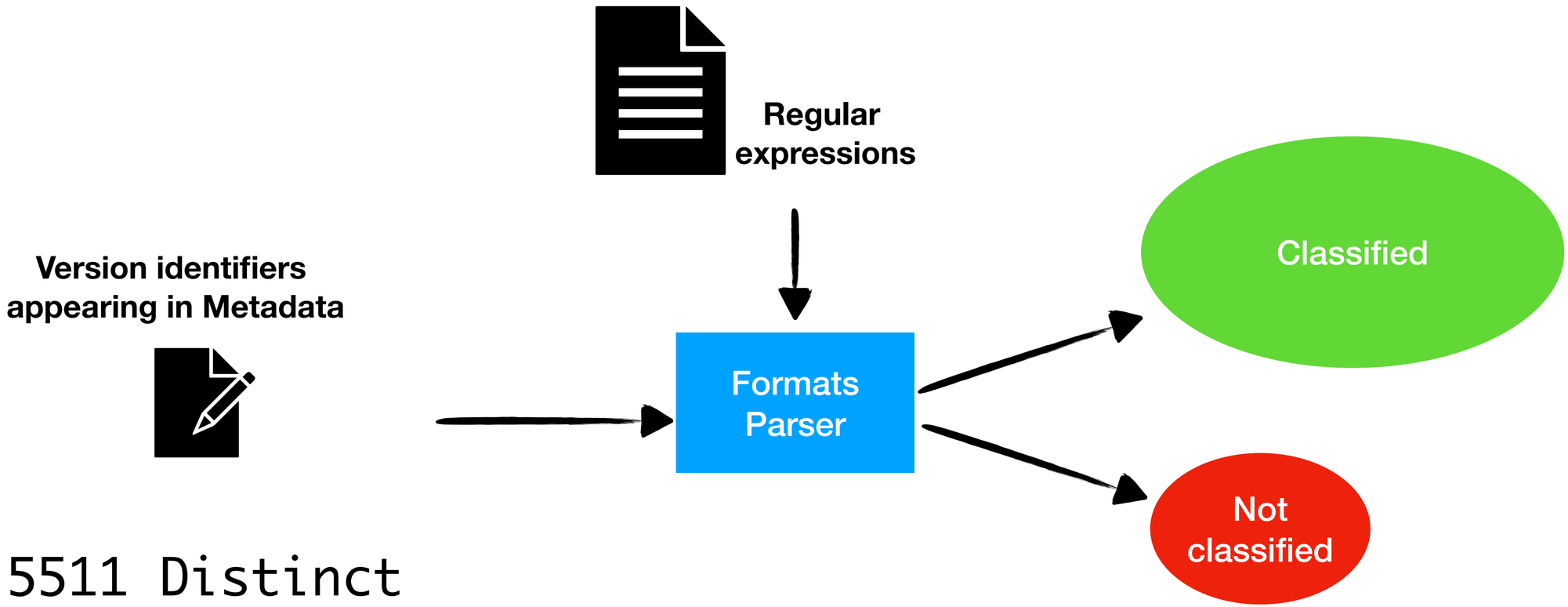
5511 Distinct version identifiers



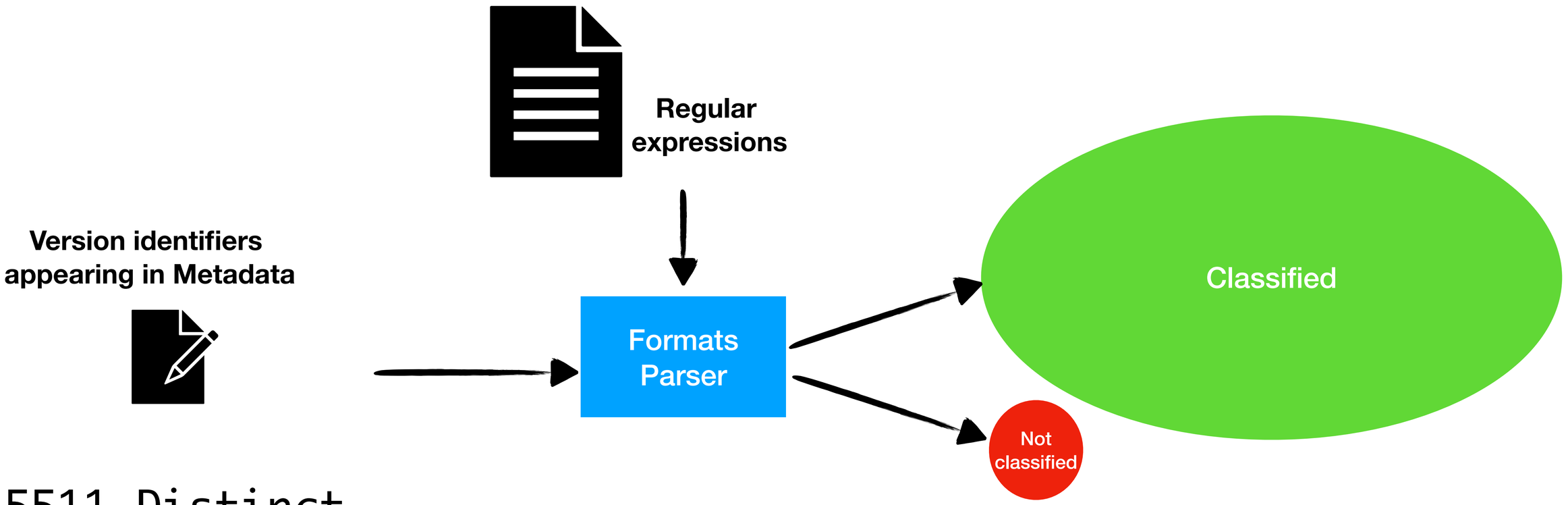
5511 Distinct
version identifiers



5511 Distinct
version identifiers



5511 Distinct
version identifiers

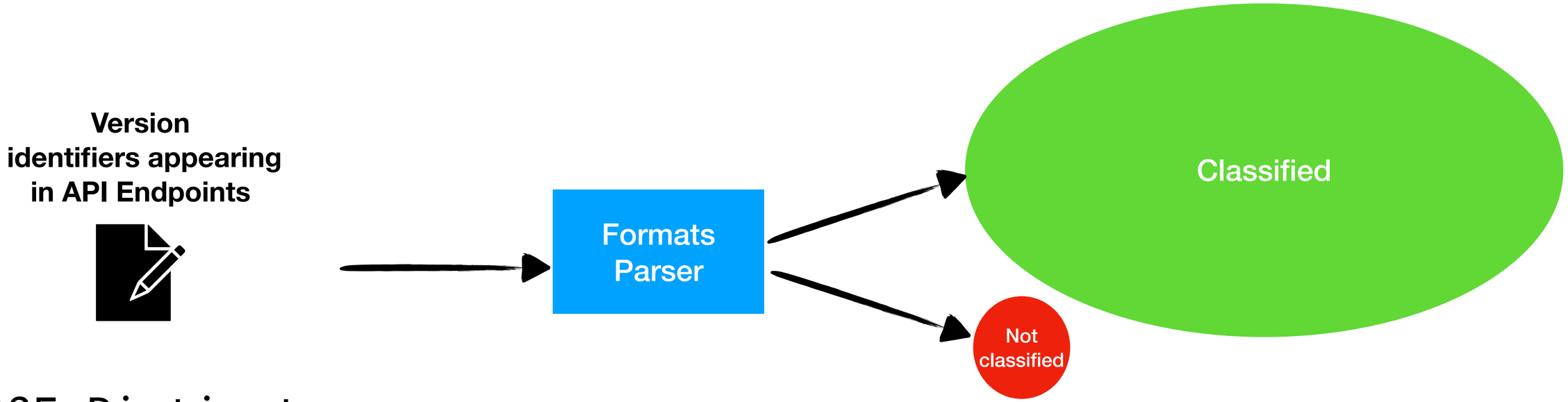


5511 Distinct
version identifiers

R4.2a
auto
Master
バージョン

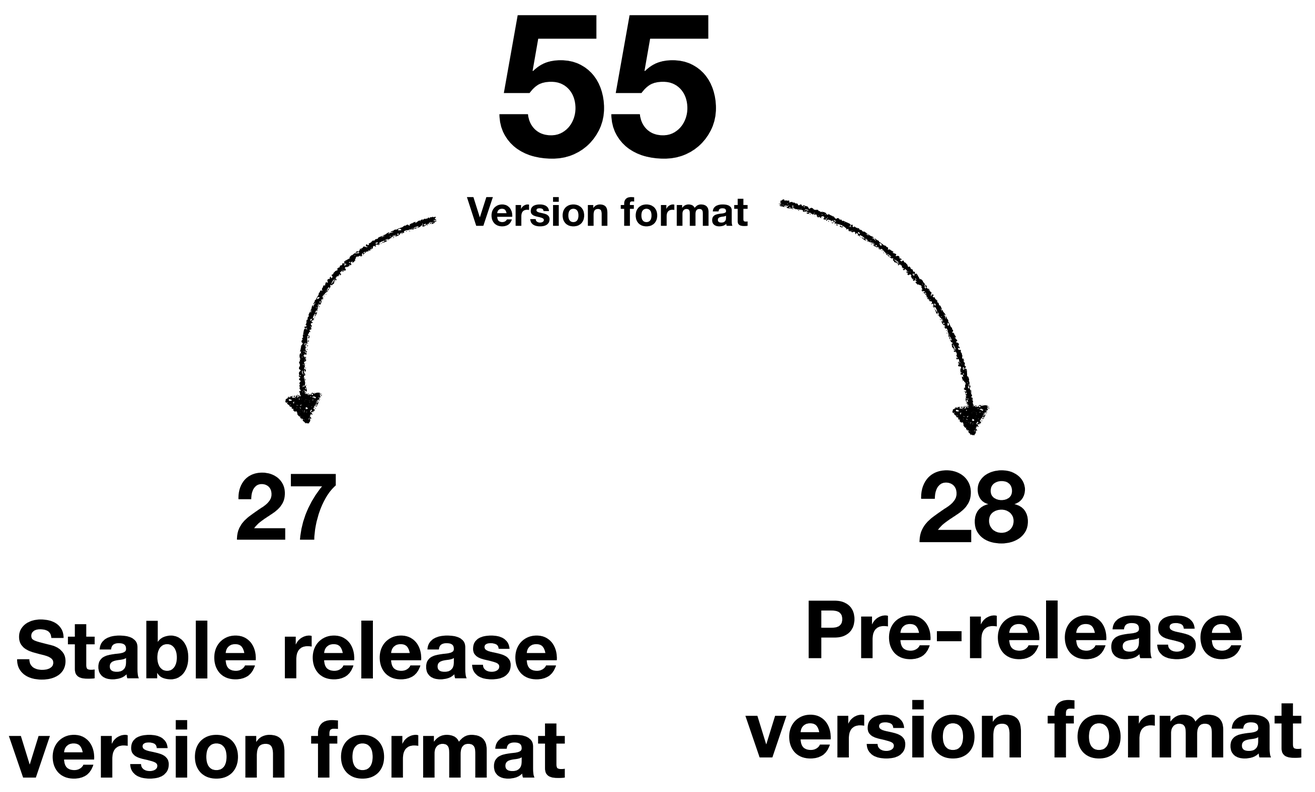
Version identifiers appearing in API Endpoints

385 Distinct version identifiers



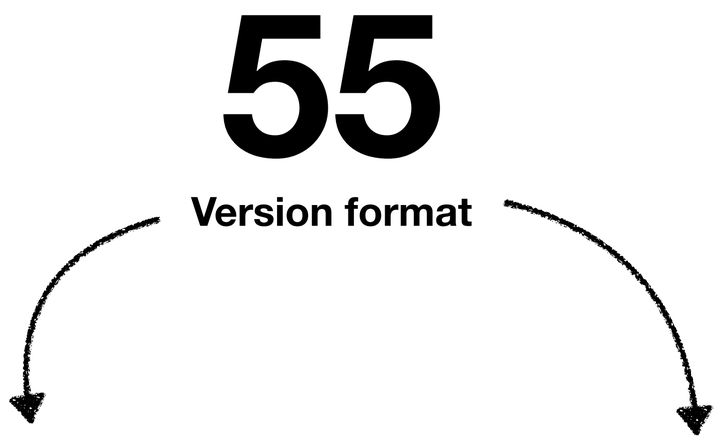
385 Distinct
version identifiers

https://github.com/USI-INF-Software/API-Versioning-practices-detection


55
Version format
27
Stable release version format
28
Pre-release version format

Format	Most Frequent Version Identifier		#APIs	#Commits				*VC*			
				max	*avg*	*mdn*	*stdev*	*max*	*avg*	*mdn*	*stdev*
semver-3	1.0.0	40.45%	3531	1031	28	17	37	496	4	0	17
semver-2	1.0	64.92%	1093	3585	30	15	116	77	1	0	4
v*	v1	80.32%	489	692	42	20	74	4	0	0	0
date(yyyy-mm-dd)	2017-03-01	4.87%	327	52	14	12	4	52	0	0	3
other	v1b3	7.23%	213	222	29	18	32	33	1	0	3
integer	1	36.30%	48	143	27	17	24	113	5	0	20
v*beta*	v1beta1	60.10%	115	360	136	35	146	3	0	0	1
date-preview*	2015-10-01-preview	11.93%	72	47	13	12	5	2	0	0	0
semver-3#	1.0.0-oas3	27.62%	33	215	32	15	41	18	2	0	4
v*beta*.*	v2beta1.1	19.44%	26	30	24	24	4	12	3	3	4
latest*	latest	52.75%	25	137	27	15	28	2	0	0	0
v*alpha*	v1alpha	51.34%	18	339	56	24	91	3	0	0	1
semver-SNAPSHOT*	1.0.0-SNAPSHOT	31.61%	18	172	32	16	38	36	5	0	9
semver-beta*	v1.0-beta	28.37%	17	113	40	29	29	9	1	0	2
v*p*beta*	v1p3beta1	23.45%	9	347	162	35	153	3	1	0	1
beta	1beta1	100.00%	7	37	15	11	9	0	0	0	0
beta*	beta	65.49%	7	47	26	26	12	0	0	0	0
semver-alpha*	1.0.0-alpha	28.04%	7	48	23	15	15	2	0	0	1
semver-2#	1.3-DUMMY	12.26%	6	24	16	15	5	3	2	2	1
semver (beta*)	1.0 (beta)	29.89%	6	58	39	46	13	46	18	26	18
date(yyyy.mm.dd)	2019.10.15	10.45%	6	24	22	24	4	24	20	24	9
#semver-3	2019.0.0	29.73%	5	37	22	17	11	3	1	2	1
semver-rc*	1.0.0-rc1	38.14%	4	190	60	20	75	8	4	5	3
semver-4	6.4.3.0	3.31%	4	23	16	17	5	9	2	0	4
semver-rc*.*	2.0.0-RC1.0	41.69%	4	85	54	63	26	0	0	0	0
v*alpha*.*	v2alpha2.6	61.76%	3	26	23	22	2	4	1	0	2
alpha*	alpha	73.85%	2	35	26	35	9	0	0	0	0
dev*	dev	98.38%	2	172	91	172	81	0	0	0	0
date(yyyy-mm)	2021-10	67.44%	2	14	13	14	1	2	1	2	1
semver-pre*.*	3.5.0-pre.0	100.00%	1	10	10	10	0	0	0	0	0
date(yyyymmdd)	20190111	29.63%	1	13	13	13	0	0	0	0	0
semver-dev*	0.7.0.dev20191230	15.52%	1	40	40	40	0	0	0	0	0
v*-date	v1-20160622	57.14%	1	18	18	18	0	2	2	2	0
semver-alpha*.*	1.1.0-alpha.1	4.94%	1	146	146	146	0	0	0	0	0

Format	Most Frequent Version Identifier		#APIs	#Commits				*VC*			
				max	*avg*	*mdn*	*stdev*	*max*	*avg*	*mdn*	*stdev*
semver-3	1.0.0	40.45%	3531	1031	28	17	37	496	4	0	17
semver-2	1.0	64.92%	1093	3585	30	15	116	77	1	0	4
v*	v1	80.32%	489	692	42	20	74	4	0	0	0
date(yyyy-mm-dd)	2017-03-01	4.87%	327	52	14	12	4	52	0	0	3
other	v1b3	7.23%	213	222	29	18	32	33	1	0	3
integer	1	36.30%	48	143	27	17	24	113	5	0	20
v*beta*	v1beta1	60.10%	115	360	136	35	146	3	0	0	1
date-preview*	2015-10-01-preview	11.93%	72	47	13	12	5	2	0	0	0
semver-3#	1.0.0-oas3	27.62%	33	215	32	15	41	18	2	0	4
v*beta*.*	v2beta1.1	19.44%	26	30	24	24	4	12	3	3	4
latest*	latest	52.75%	25	137	27	15	28	2	0	0	0
v*alpha*	v1alpha	51.34%	18	339	56	24	91	3	0	0	1
semver-SNAPSHOT*	1.0.0-SNAPSHOT	31.61%	18	172	32	16	38	36	5	0	9
semver-beta*	v1.0-beta	28.37%	17	113	40	29	29	9	1	0	2
v*p*beta*	v1p3beta1	23.45%	9	347	162	35	153	3	1	0	1
beta	1beta1	100.00%	7	37	15	11	9	0	0	0	0
beta*	beta	65.49%	7	47	26	26	12	0	0	0	0
semver-alpha*	1.0.0-alpha	28.04%	7	48	23	15	15	2	0	0	1
semver-2#	1.3-DUMMY	12.26%	6	24	16	15	5	3	2	2	1
semver (beta*)	1.0 (beta)	29.89%	6	58	39	46	13	46	18	26	18
date(yyyy.mm.dd)	2019.10.15	10.45%	6	24	22	24	4	24	20	24	9
#semver-3	2019.0.0	29.73%	5	37	22	17	11	3	1	2	1
semver-rc*	1.0.0-rc1	38.14%	4	190	60	20	75	8	4	5	3
semver-4	6.4.3.0	3.31%	4	23	16	17	5	9	2	0	4
semver-rc*.*	2.0.0-RC1.0	41.69%	4	85	54	63	26	0	0	0	0
v*alpha*.*	v2alpha2.6	61.76%	3	26	23	22	2	4	1	0	2
alpha*	alpha	73.85%	2	35	26	35	9	0	0	0	0
dev*	dev	98.38%	2	172	91	172	81	0	0	0	0
date(yyyy-mm)	2021-10	67.44%	2	14	13	14	1	2	1	2	1
semver-pre*.*	3.5.0-pre.0	100.00%	1	10	10	10	0	0	0	0	0
date(yyyymmdd)	20190111	29.63%	1	13	13	13	0	0	0	0	0
semver-dev*	0.7.0.dev20191230	15.52%	1	40	40	40	0	0	0	0	0
v*-date	v1-20160622	57.14%	1	18	18	18	0	2	2	2	0
semver-alpha*.*	1.1.0-alpha.1	4.94%	1	146	146	146	0	0	0	0	0



27

Stable release version format

Major version number
SemVer
Tag
Date
Other

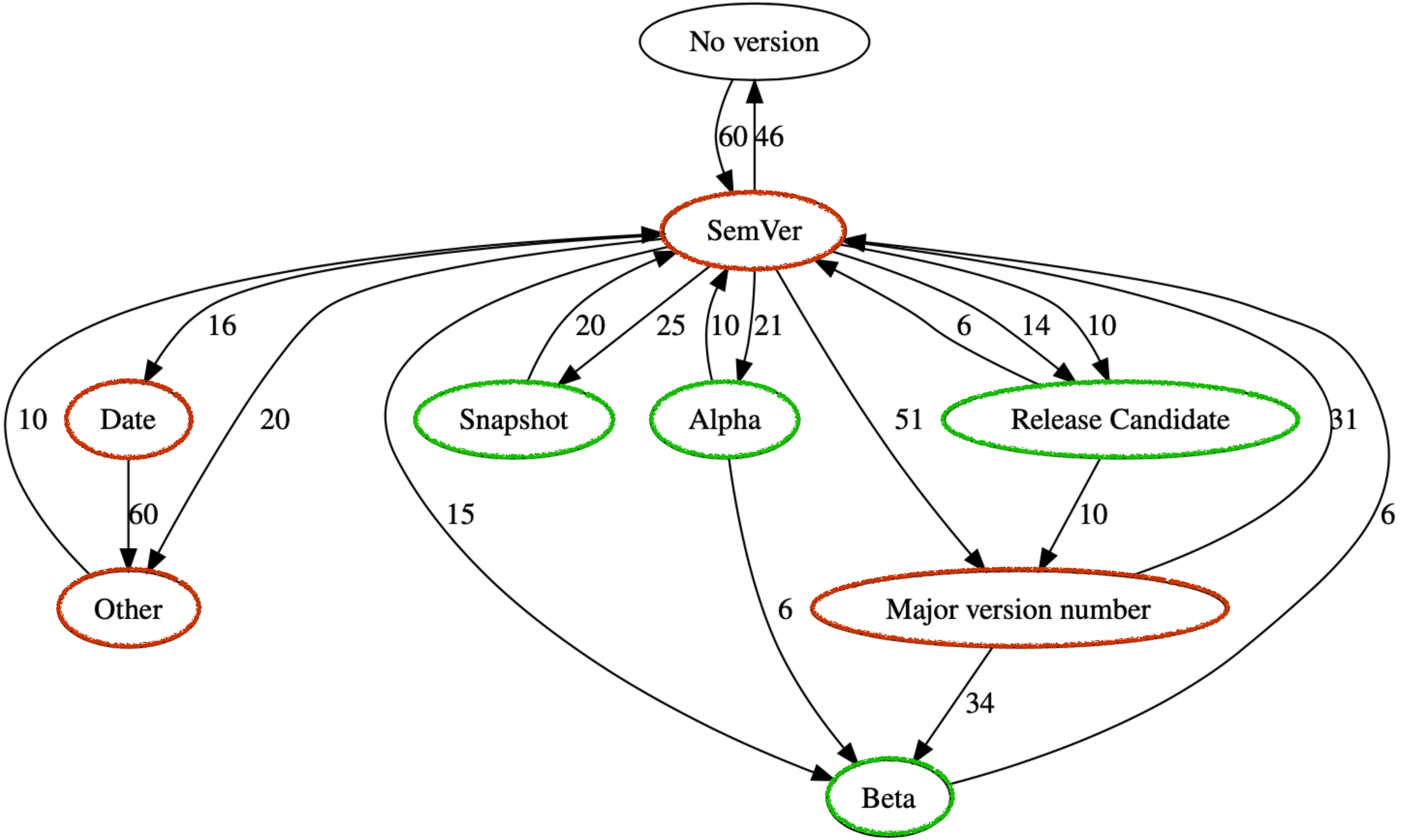
28

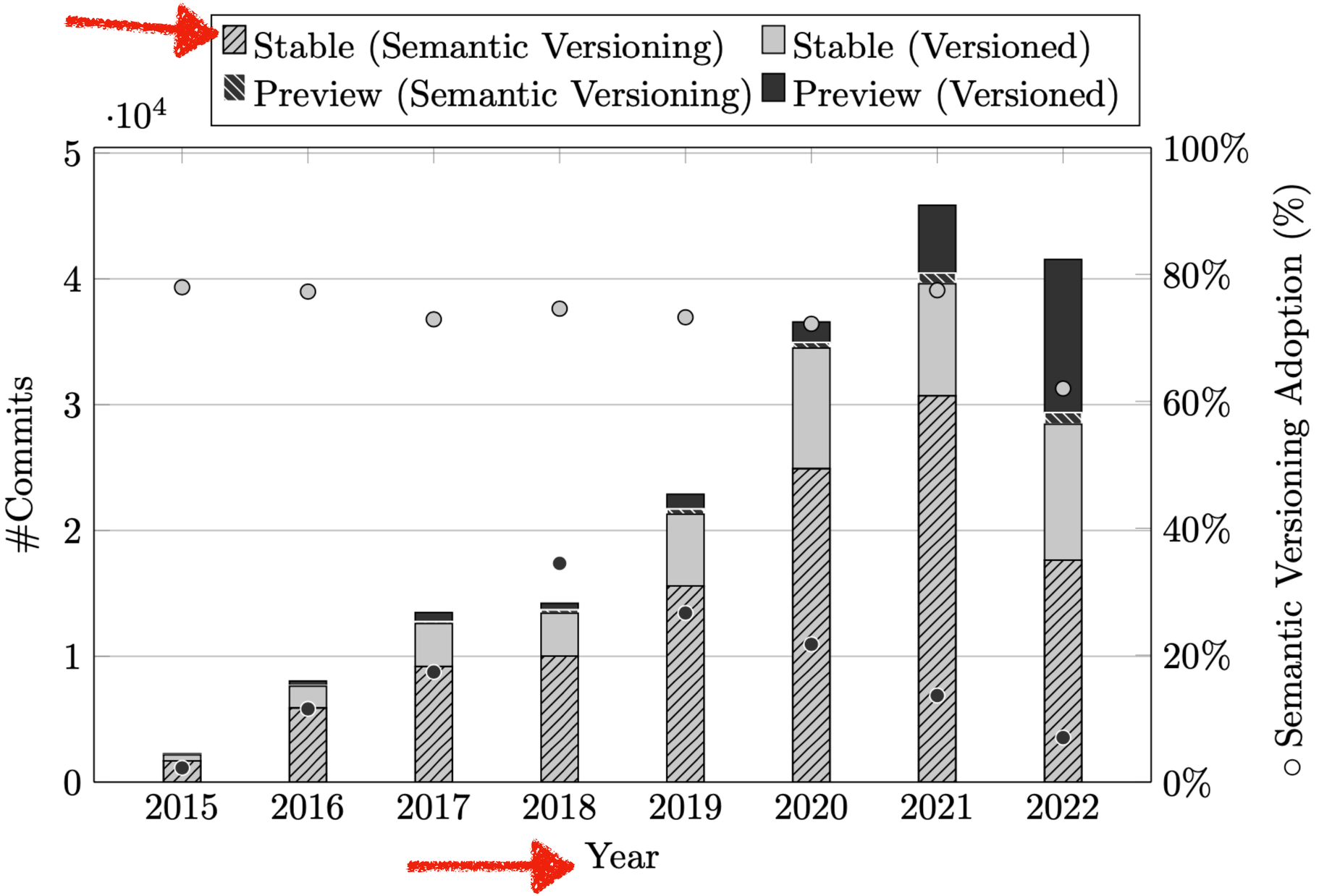
Pre-release version format

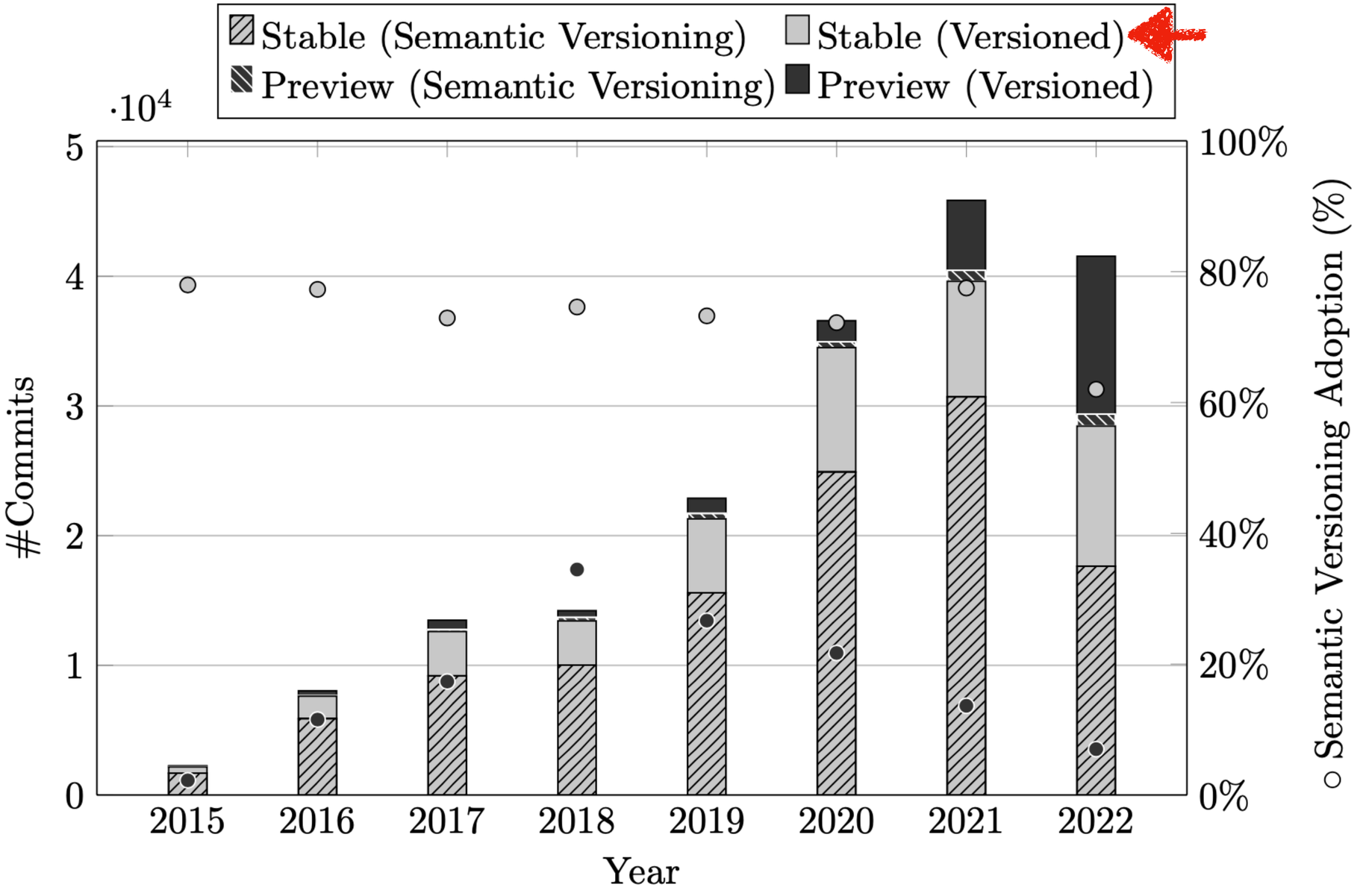
Develop
Snapshot
Preview
Alpha
Beta
Release Candidate

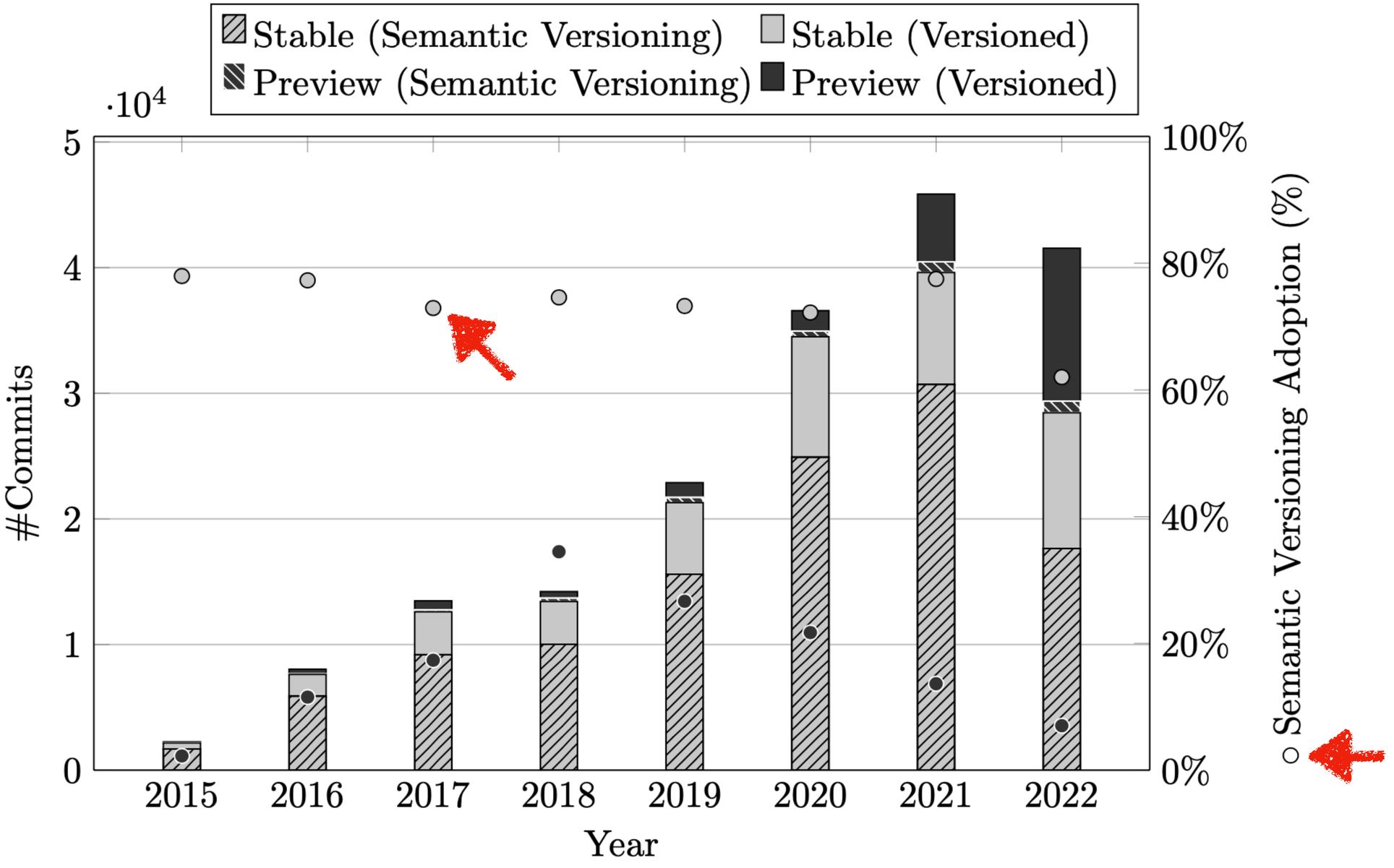
APIs with stable formats	
SemVer	4941
Major version number	804
Date	336
No version	268
Preview	73
Other	61
Beta	37
Tag	25
Snapshot	17
Alpha	10
Release Candidate	8

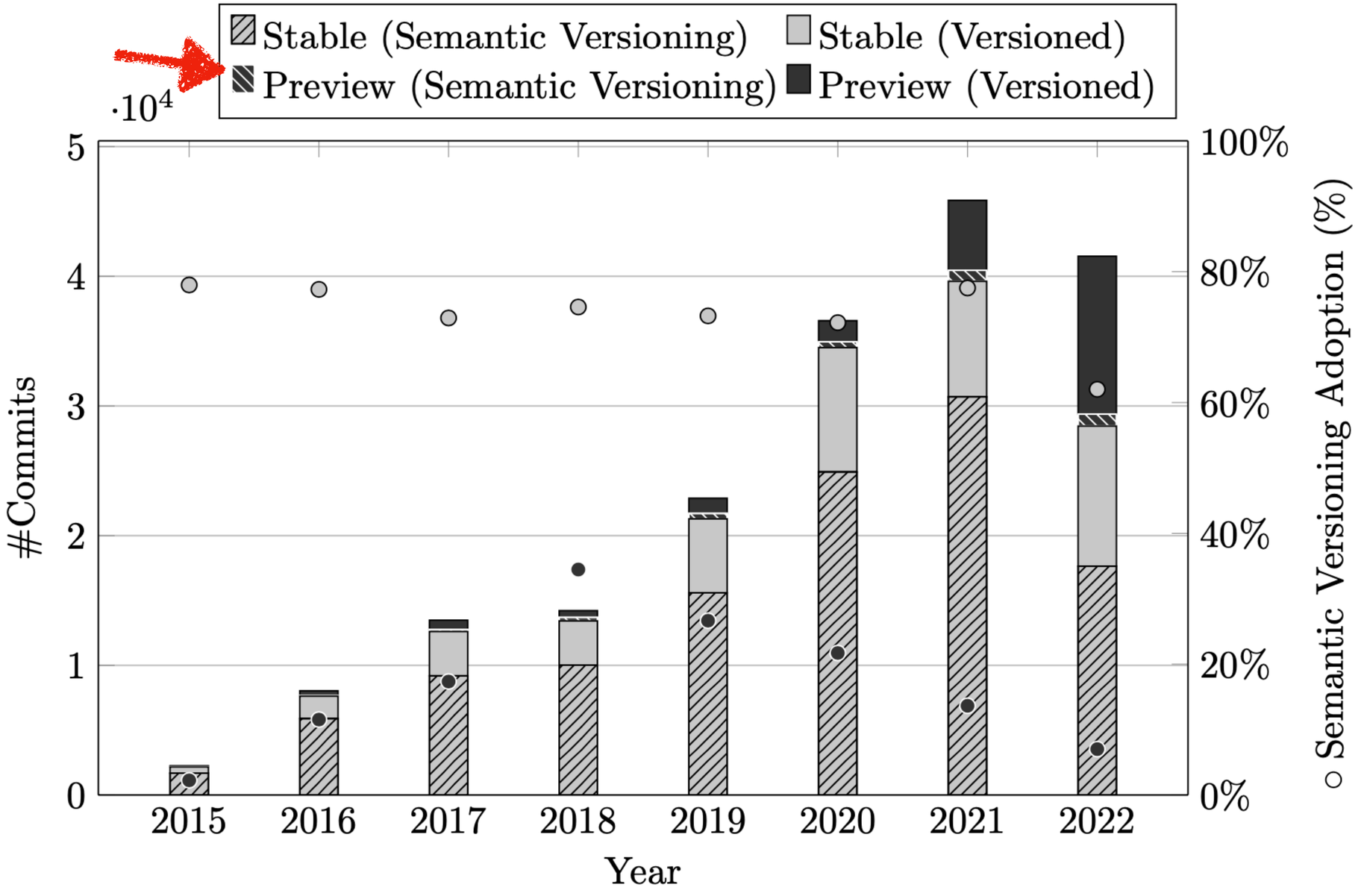
APIs with stable formats	
SemVer	4941
Major version number	804
Date	336
No version	268
Preview	73
Other	61
Beta	37
Tag	25
Snapshot	17
Alpha	10
Release Candidate	8

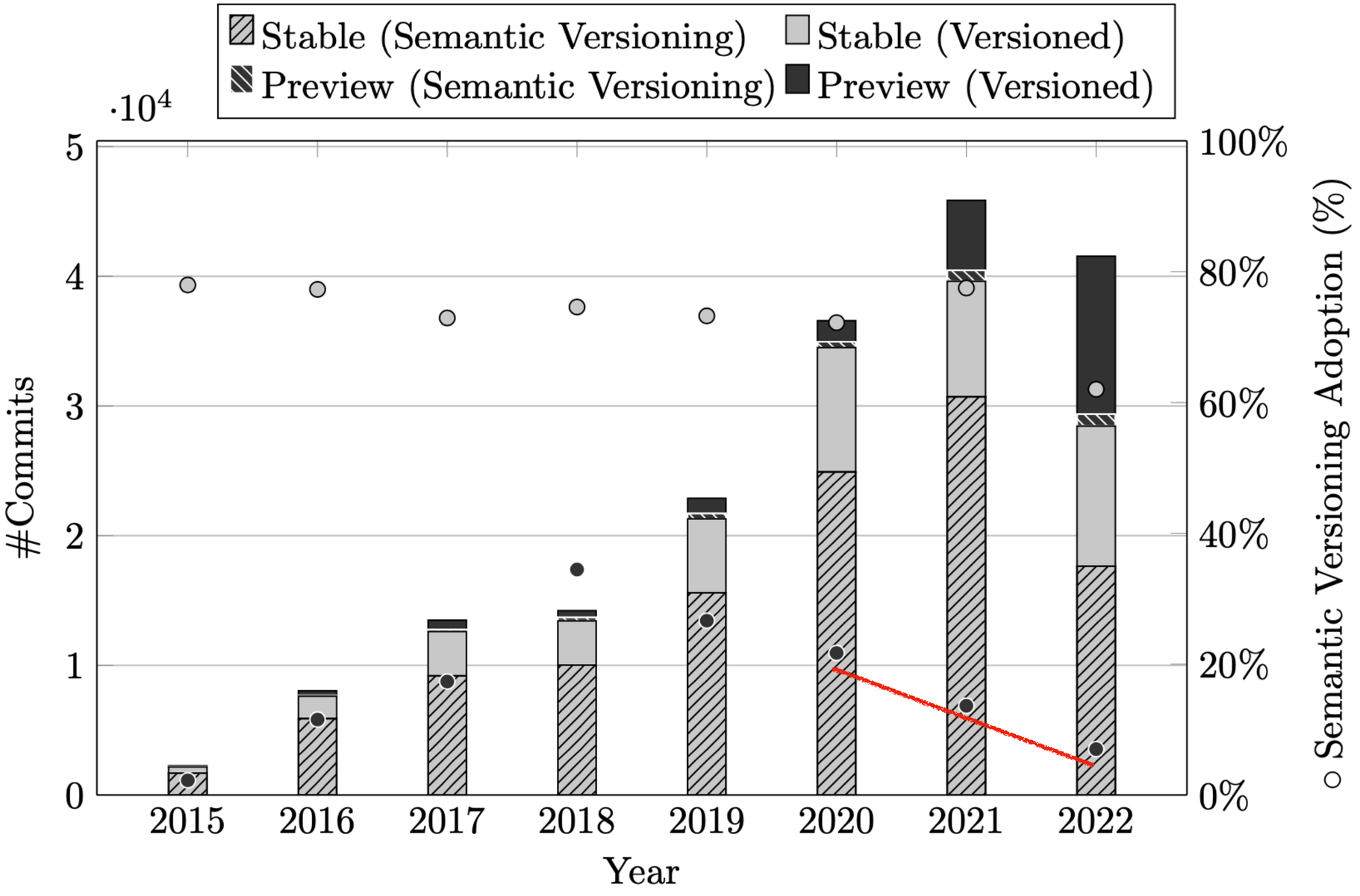

No version
60
46
SemVer
16
10
Date
20
60
Other
20
25
10
21
Snapshot
Alpha
15
6
14
10
51
Release Candidate
31
10
6
Major version number
6
34
Beta

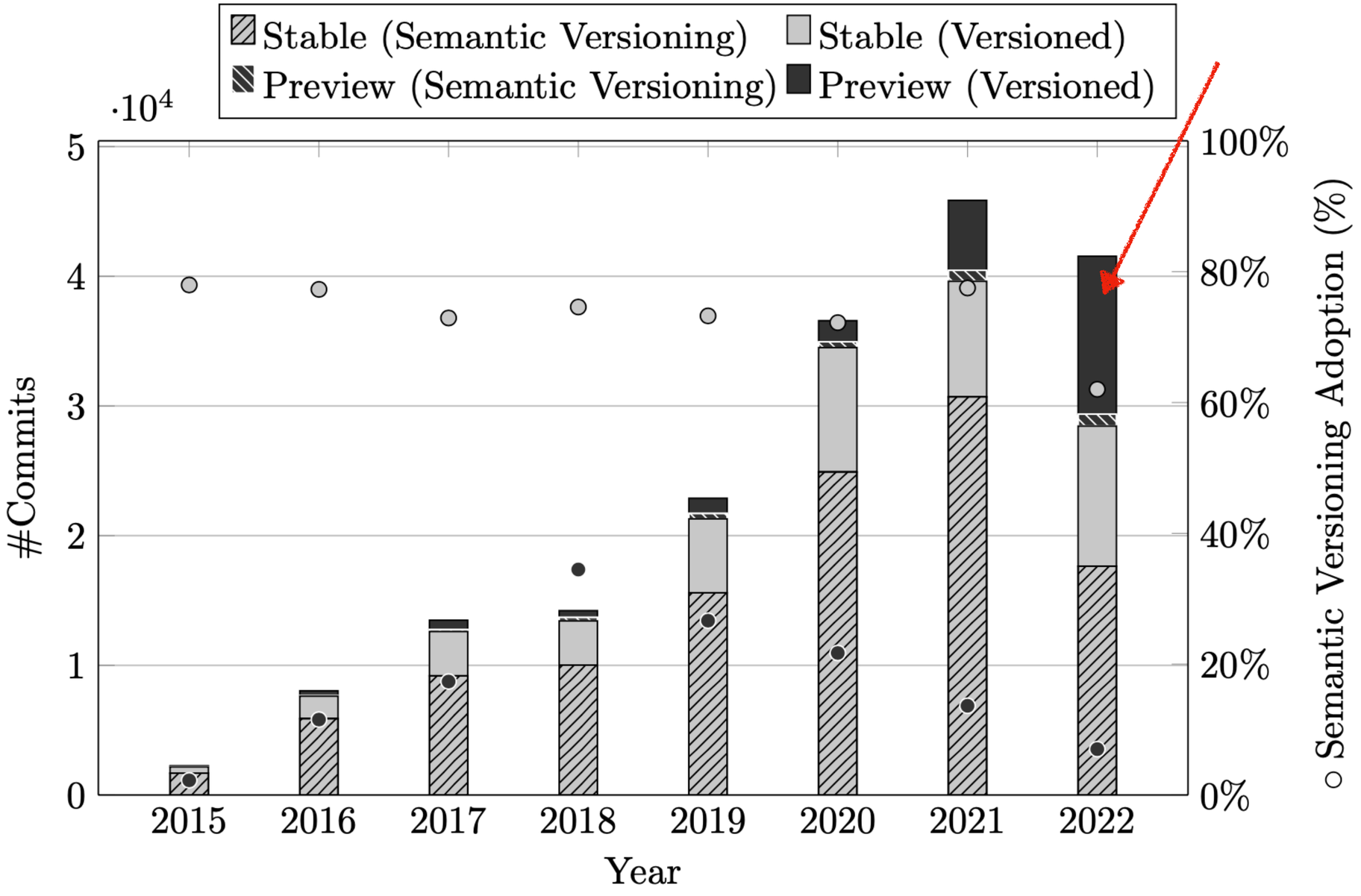

Stable (Semantic Versioning)
Stable (Versioned)
Preview (Semantic Versioning)
Preview (Versioned)
·10^4
#Commits
5
4
3
2
1
0
100%
80%
60%
40%
20%
0%
○ Semantic Versioning Adoption (%)
2015
2016
2017
2018
2019
2020
2021
2022
Year

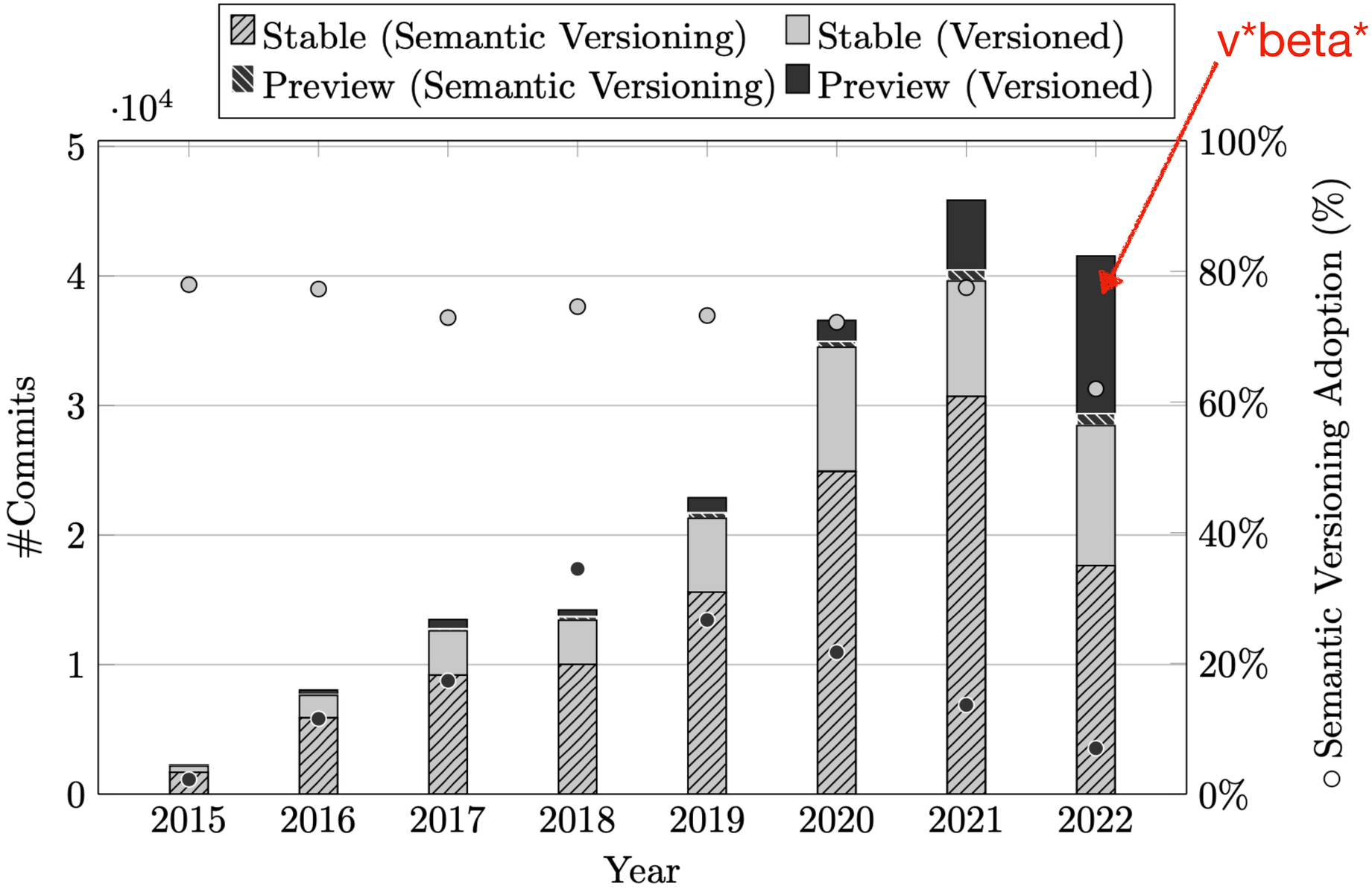

Stable (Semantic Versioning)
Stable (Versioned)
Preview (Semantic Versioning)
Preview (Versioned)
·10^4
#Commits
Year
○ Semantic Versioning Adoption (%)
0
1
2
3
4
5
0%
20%
40%
60%
80%
100%
2015
2016
2017
2018
2019
2020
2021
2022


Stable (Semantic Versioning)
Stable (Versioned)
Preview (Semantic Versioning)
Preview (Versioned)
·10^4
#Commits
Year
Semantic Versioning Adoption (%)
0
1
2
3
4
5
0%
20%
40%
60%
80%
100%
2015
2016
2017
2018
2019
2020
2021
2022


Stable (Semantic Versioning)
Stable (Versioned)
Preview (Semantic Versioning)
Preview (Versioned)
·10^4
5
4
3
2
1
0
#Commits
100%
80%
60%
40%
20%
0%
○ Semantic Versioning Adoption (%)
2015
2016
2017
2018
2019
2020
2021
2022
Year


Stable (Semantic Versioning)
Stable (Versioned)
Preview (Semantic Versioning)
Preview (Versioned)
·10^4
#Commits
5
4
3
2
1
0
100%
80%
60%
40%
20%
0%
○ Semantic Versioning Adoption (%)
2015
2016
2017
2018
2019
2020
2021
2022
Year

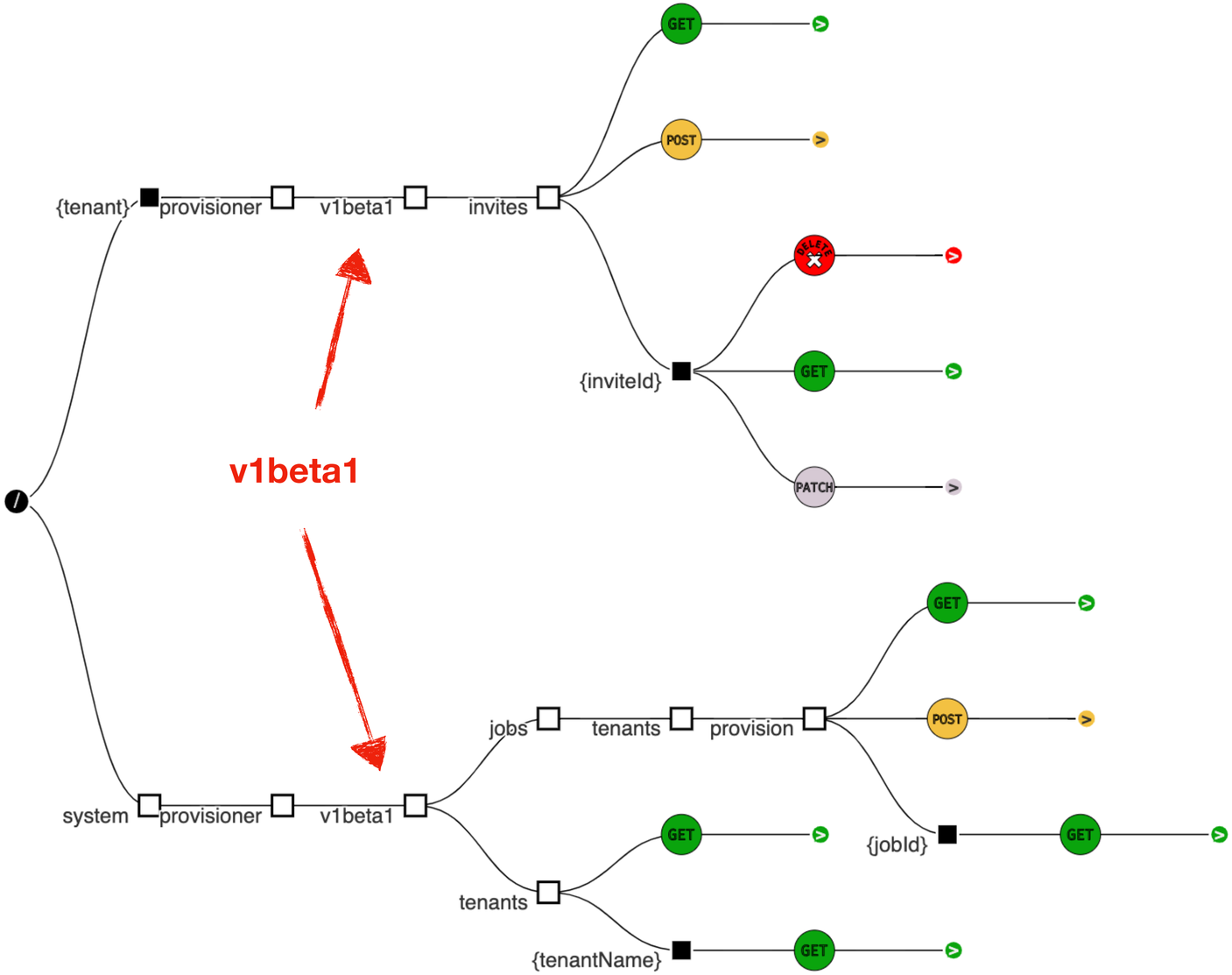

Stable (Semantic Versioning)
Stable (Versioned)
Preview (Semantic Versioning)
Preview (Versioned)
·10^4
#Commits
5
4
3
2
1
0
100%
80%
60%
40%
20%
0%
○ Semantic Versioning Adoption (%)
2015
2016
2017
2018
2019
2020
2021
2022
Year


Stable (Semantic Versioning)
Stable (Versioned)
Preview (Semantic Versioning)
Preview (Versioned)
v*beta* or v*alpha*
·10^4
#Commits
○ Semantic Versioning Adoption (%)
Year
0
1
2
3
4
5
0%
20%
40%
60%
80%
100%
2015
2016
2017
2018
2019
2020
2021
2022

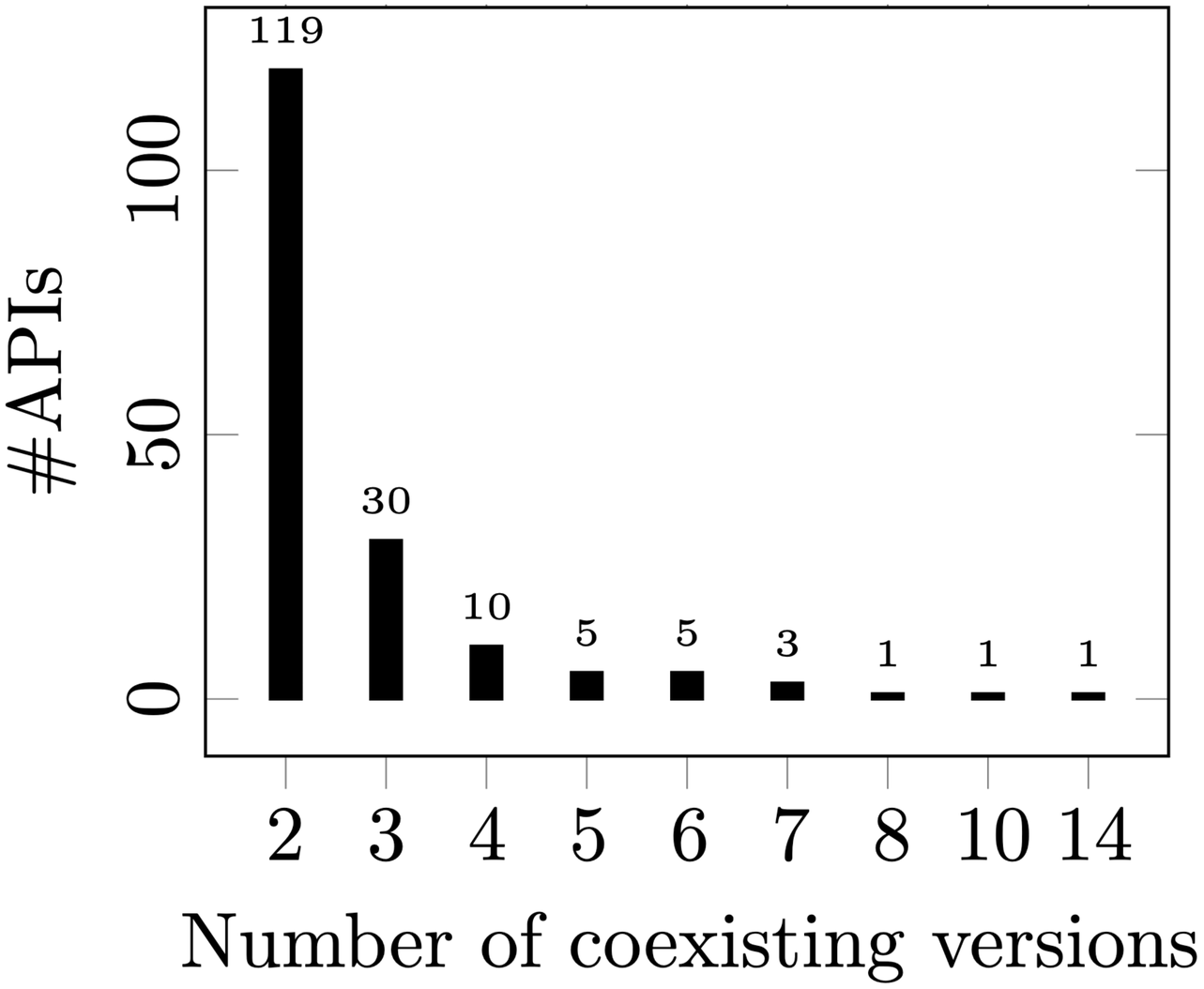
v1beta1.4

Provisioner

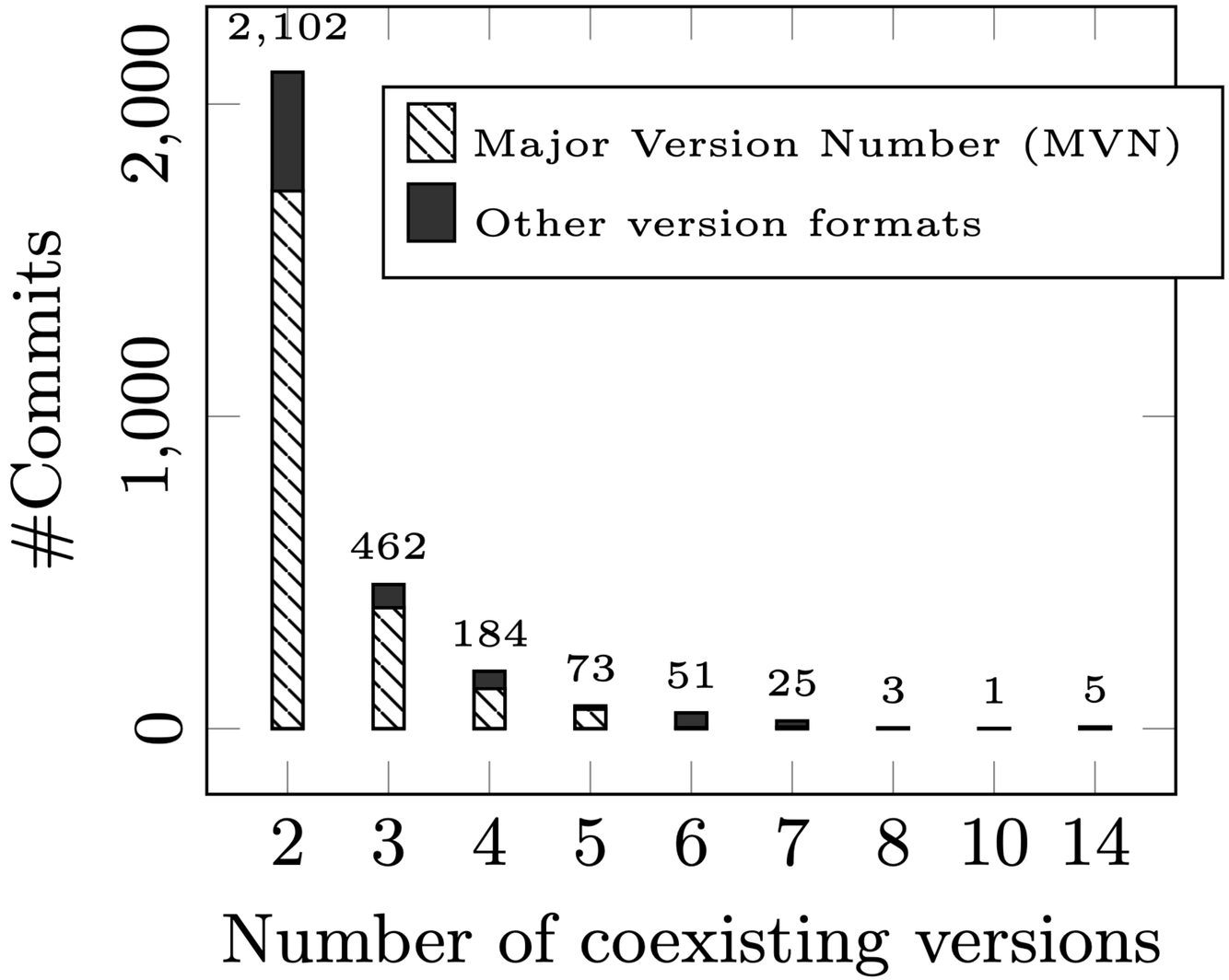
Version: v1beta1.4
Description: With the Provisioner service in Splunk Cloud Services, you can provision and manage tenants.



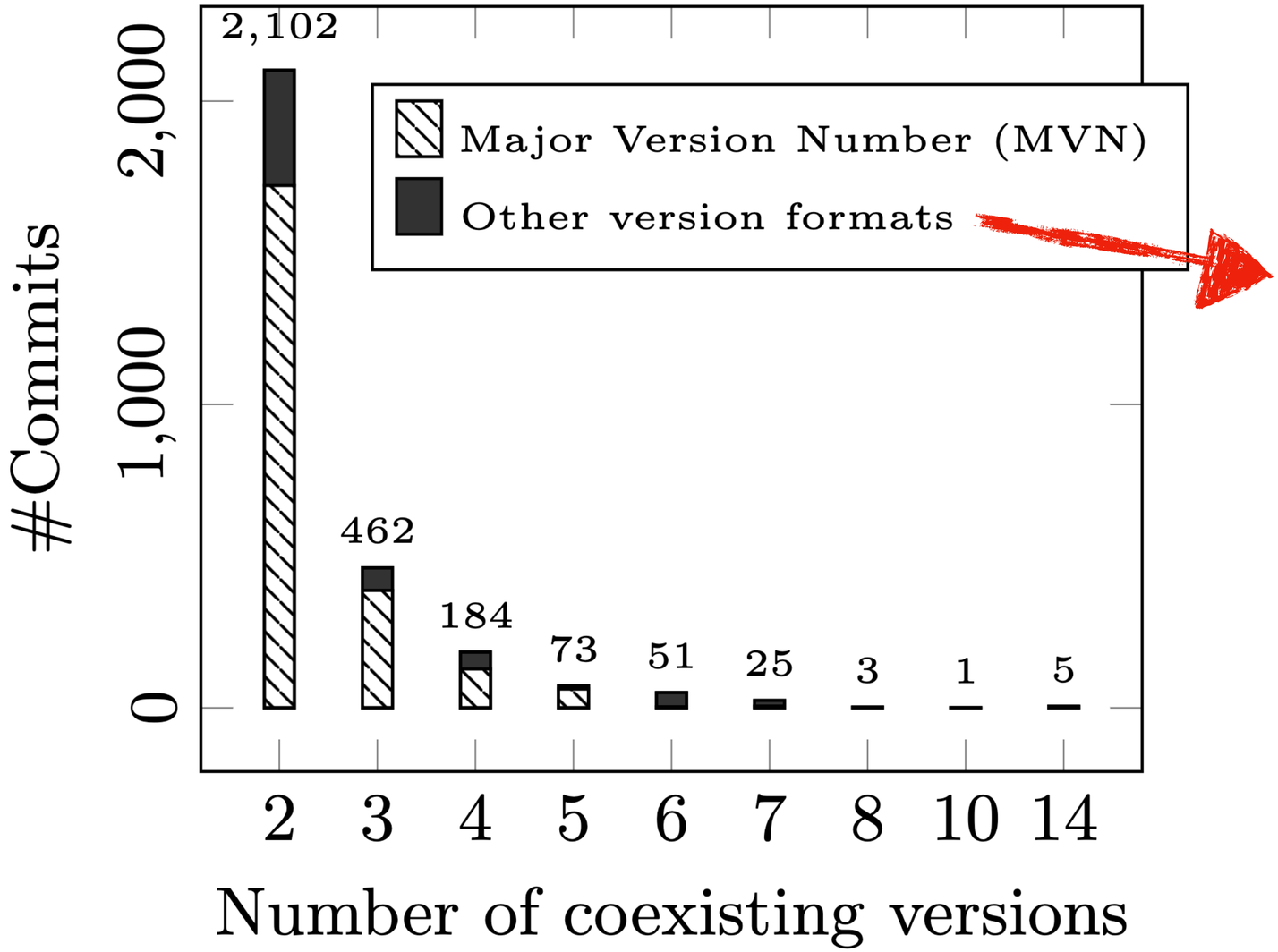
Version identifiers formats in APIs with multiple coexistent versions

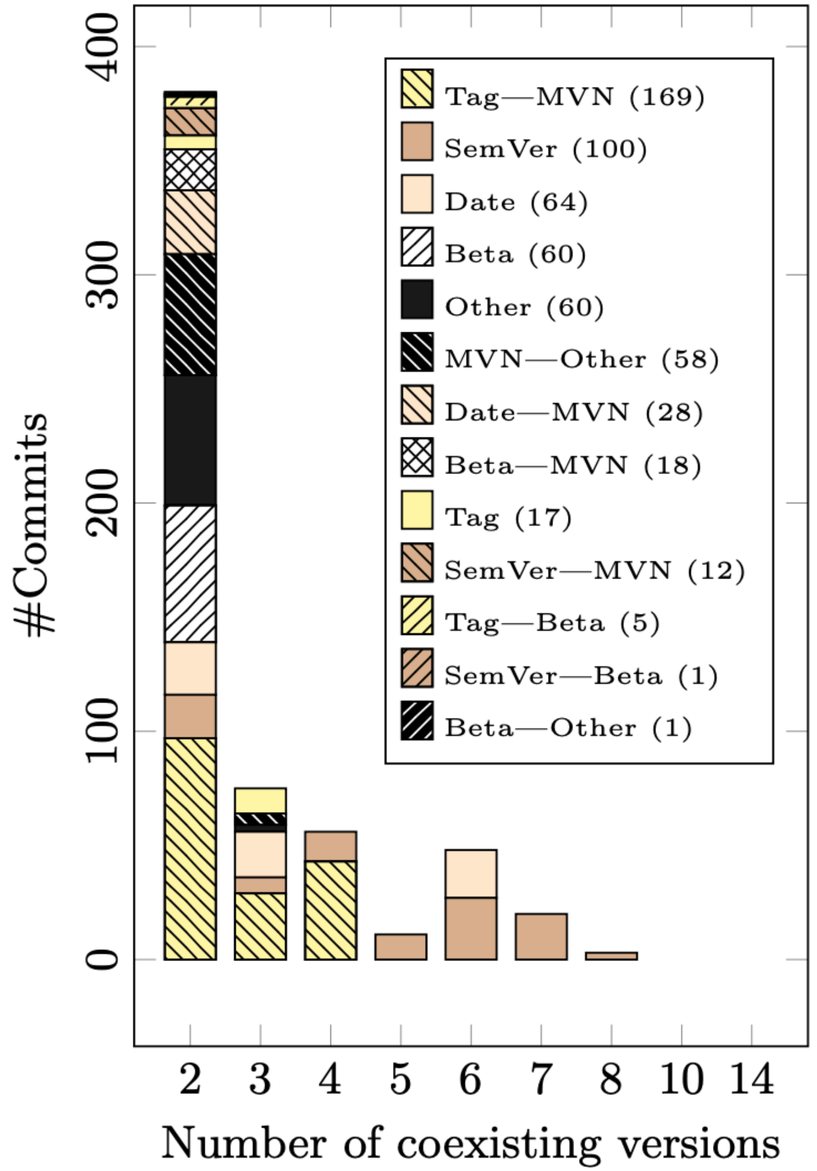


Version identifiers formats in APIs with multiple coexistent versions

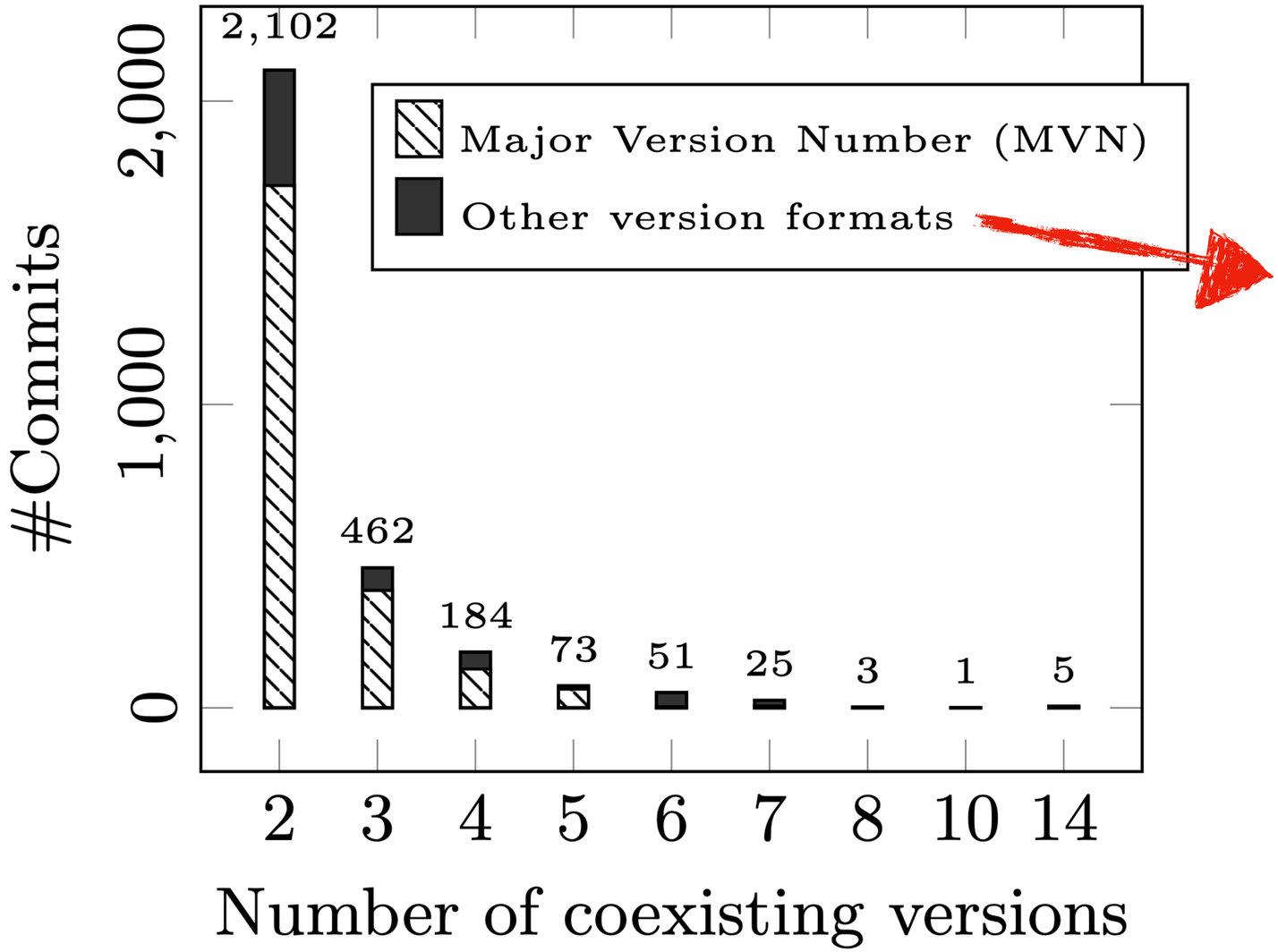


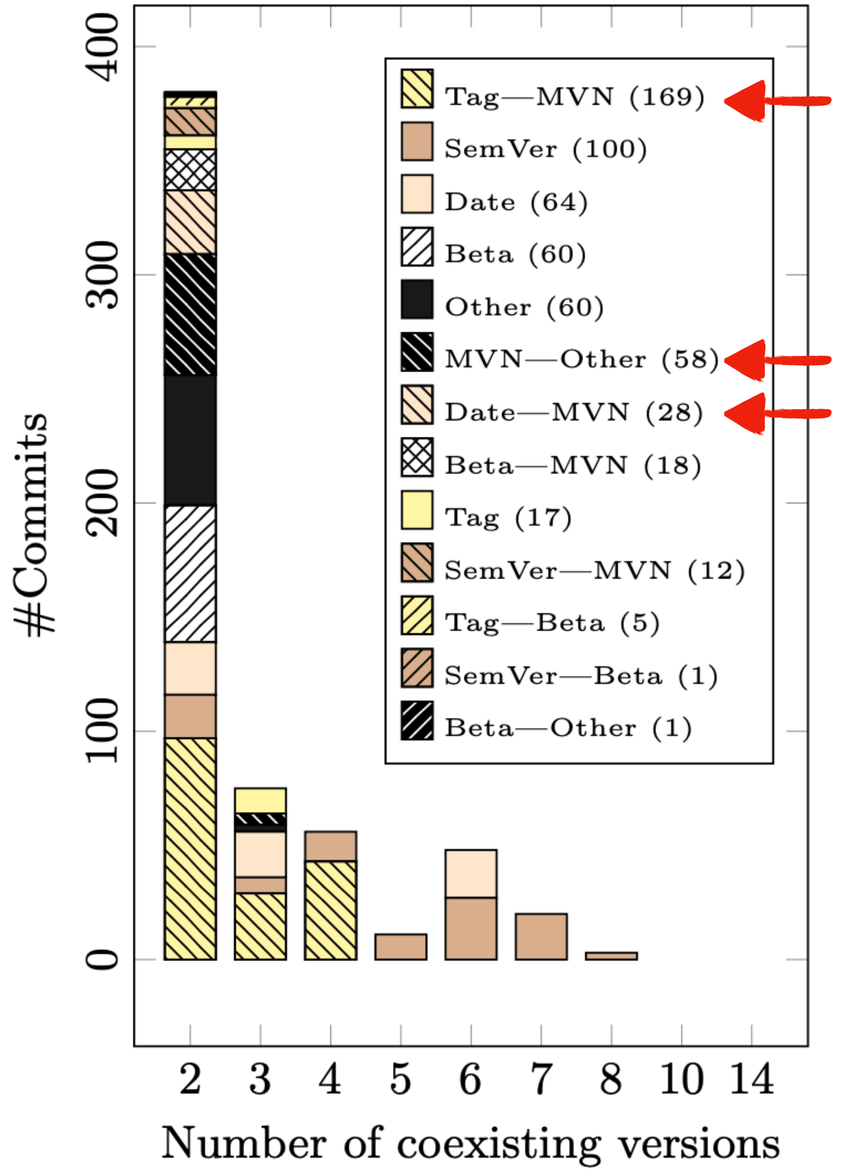
Version identifiers formats in APIs with multiple coexistent versions





Version identifiers formats in APIs with multiple coexistent versions





How do developers version Web API?

Approach

- **Usage of the two in-production patterns** in 175/7114 APIs. Up to 14 coexistent versions in the case of an API.
- **Usage of Path-based versioning**. 36% of the APIs used Path-based versioning.
  - ❌ 496 APIs switched to/from Path-based versioning in the middle of their history.
- **Usage of Metadata-based versioning.** 70% of the APIs use Metada-based versioning

How do developers version Web API?

Version identifiers formats

- Version identifiers are expressed in **55 different formats**
- Noticeable **switch to SemVer during histories of API** that change versrion identifies formats.
- **4941 APIs used only SemVer** during their whole history
- Significant **increase in the use of simpler pre-release** versioning formats.

Future Work

How do developers change the version identifier on each API change?

✓ **Focus on a subset of APIs with parsable version identifiers during all their history**

Future Work

How do developers change the version identifier on each API change?

- ✓ Focus on a subset of APIs with parsable version identifiers during all their history
- ✓ Analyse the version increase and corresponding APIs changes

References

[1] https://semver.org/

[2] https://docs.npmjs.com/about-semantic-versioning

[3] https://github.com/jashkenas/underscore

[4] 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, pp.140-158.

[5] Ochoa, L., Degueule, T., Falleri, J.R. and Vinju, J., 2022. Breaking bad? Semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study. Empirical Software Engineering, 27(3), p.61.

[6] Di Lauro, F., Serbout, S. and Pautasso, C., 2022. A Large-scale Empirical Assessment of Web API Size Evolution. Journal of Web Engineering, pp.1937-1980.

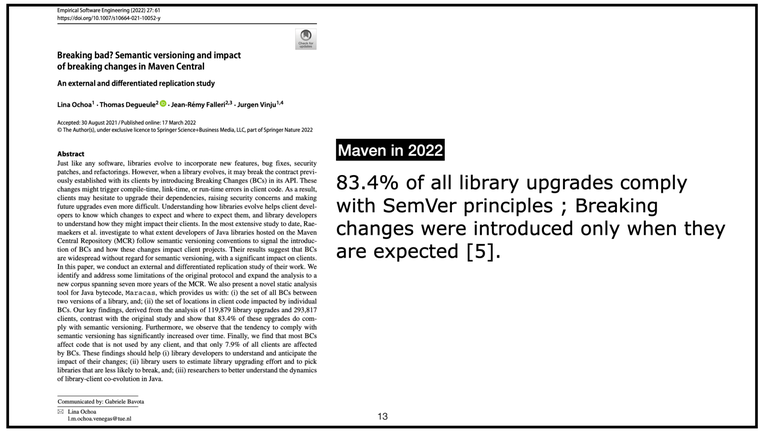
[7] Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U. and Stocker, M., 2019, July. Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (pp. 1-24)

[8] Serbout, S., and Pautasso, C., 2023, June. An Emperical Study of Web API Evolution. In Web Engineering: 23st International Conference, ICWE 2023. Alicante, Spain 6-9 June [To Appear]

[9] Visualisation tool: http://api-ace.inf.usi.ch/openapi-to-tree/

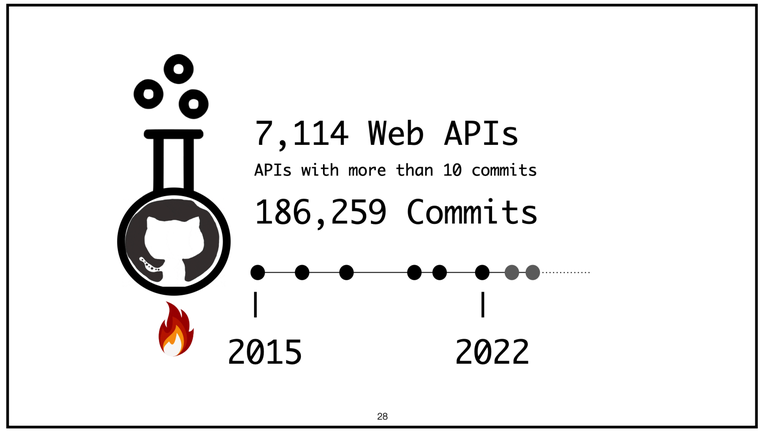
An Empirical Study of **Web API Versioning Practices**

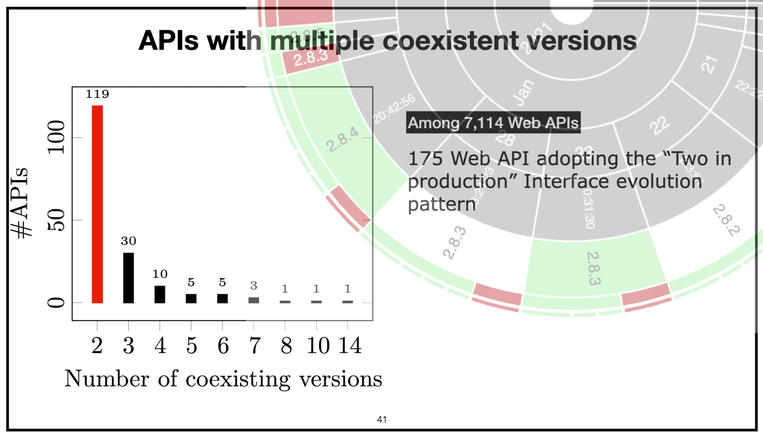
Souhaila Serbout (souhaila.serbout@usi.ch), Cesare Pautasso (cesare.pautasso@usi.ch)



Maven in 2022

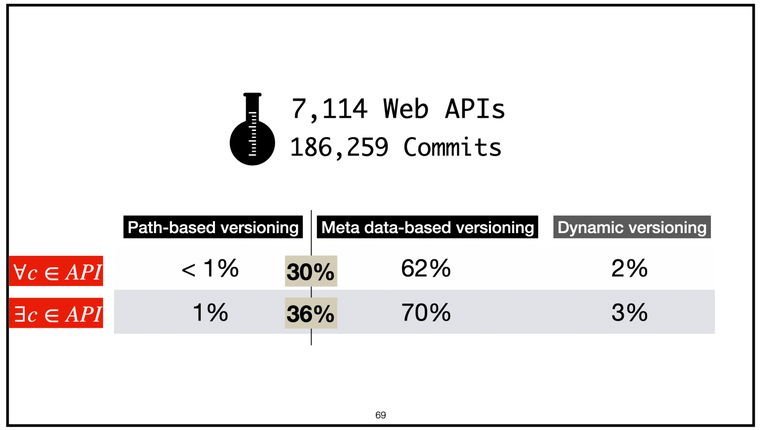
83.4% of all library upgrades comply with SemVer principles ; Breaking changes were introduced only when they are expected [5].

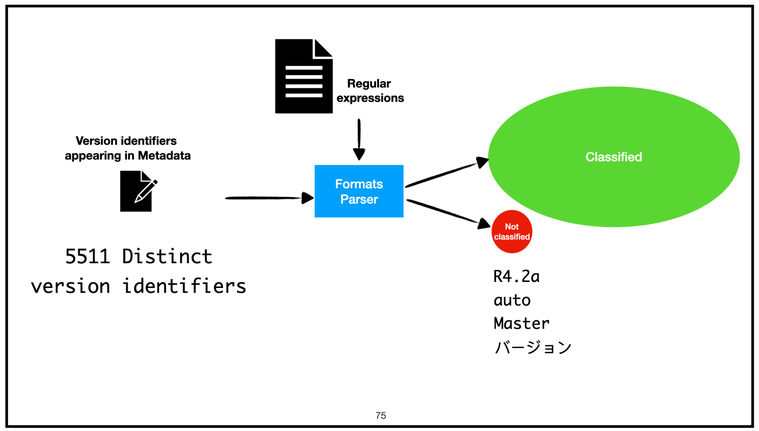


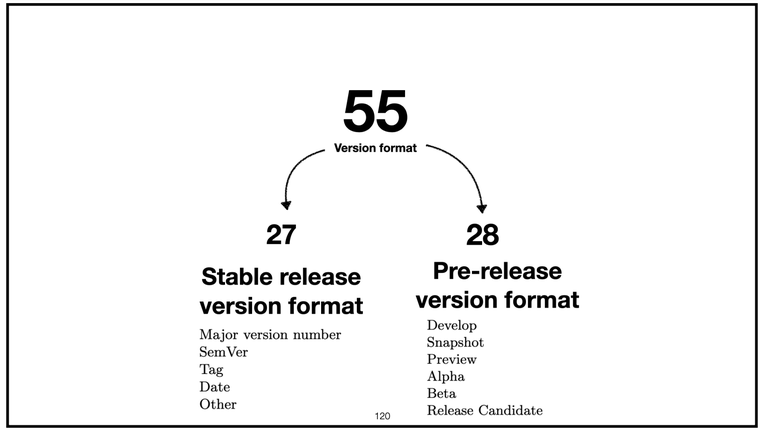


Among 7,114 Web APIs

175 Web API adopting the "Two in production" Interface evolution pattern







https://github.com/USI-INF-Software/API-Versioning-practices-detection

API Visualisation tool: http://api-ace.inf.usi.ch/openapi-to-tree/