Università della Svizzera italiana

Faculty of Informatics

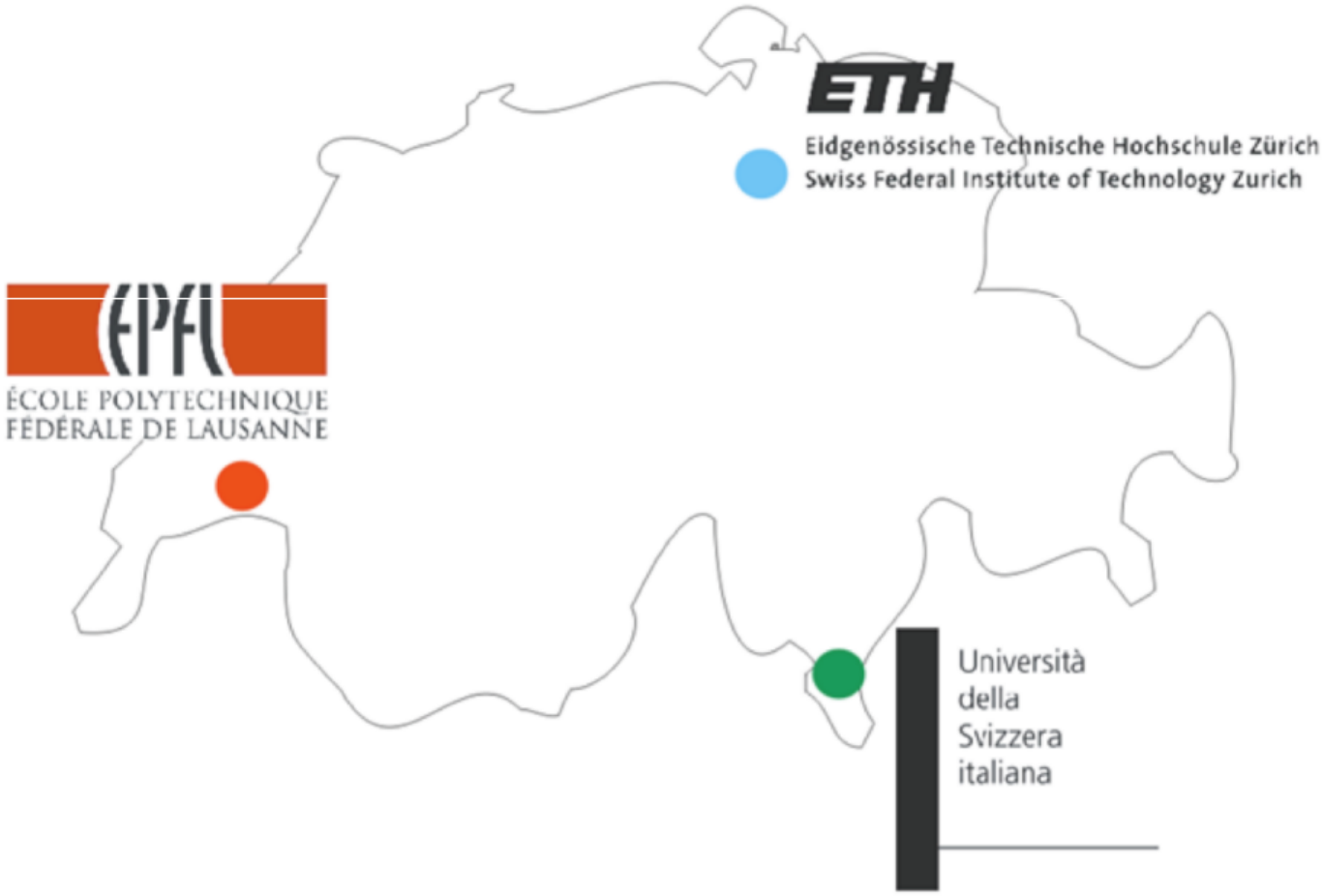
Disaster Recovery and Microservices: The BAC Theorem

Cesare Pautasso

http://www.pautasso.info

c.pautasso@ieee.org

@pautasso


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Università
della
Svizzera
italiana


Powered by
ASQ
University of Lugano (USI)
QUDOS2018


Powered by
ASQ
Faculty of Informatics
Opened 2004
30 Professors
150+ PhD/Postdoc researchers
New Software Institute
New Master in Software and Data Engineering
QUDOS2018

Architecture, Design and Web Information Systems Engineering

- RESTful Business Process Management
- BenchFlow - a benchmark for workflow engines
- Liquid.js - Liquid Software Architecture
- ASQ - Interactive Web Lectures
- Parallel JavaScript/Multicore Node.JS
- SAW - Collaborative Architectural Decision Making
- NaturalMash - Web Service Composition with Natural Language

http://design.inf.usi.ch

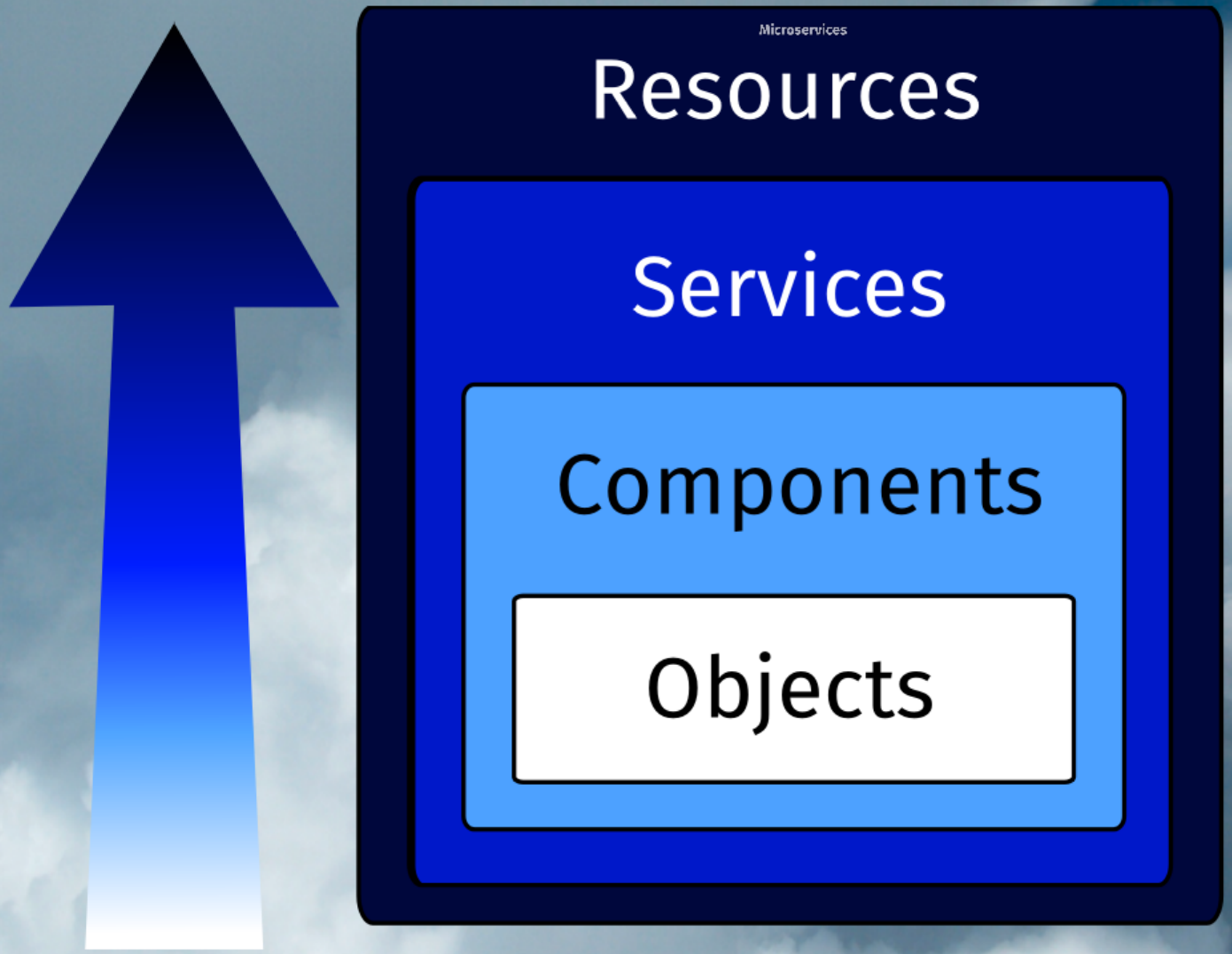
University of Lugano (USI)

Disaster Recovery and Microservices: The BAC Theorem

Microservices follow the polyglot persistence principle, where every microservice manages its own persistence independently. In this talk we illustrate the ultimate consequences of these assumptions, which can be summarized using the BAC theorem: only two are possible out of 1) a backed up microservice architecture; 2) full availability during normal operations; and 3) consistency after recovery. In other words, we will show that only Microservices Architecture running without a Backup can be both Available while remaining Consistent after disaster strikes. We will present and compare several coping strategies to deal with this limitation and discuss how it affects the monolith decomposition process at design time and the operational coupling between different microservices at run time.

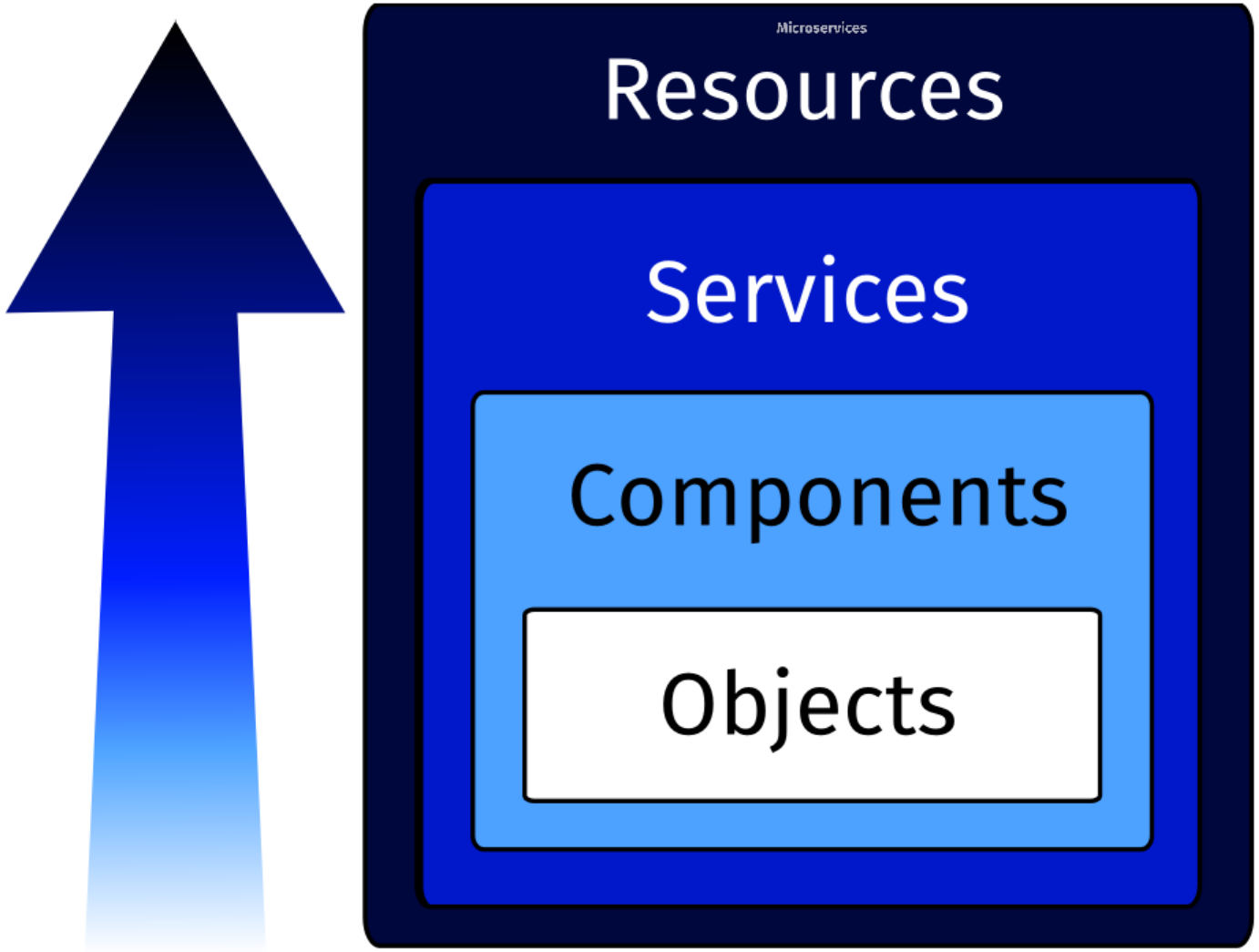

Powered by
A S Q
Microservices
QUDOS2018

Abstractions




Powered by
ASQ
Microservices
source
QUDOS2018

Abstractions



Will this component always terminate?

```
function f() {
    ...
    return 42;
}
```

Development

Will this service run forever?

```
while (true) {
      on f {
        return f()
      };
}
```

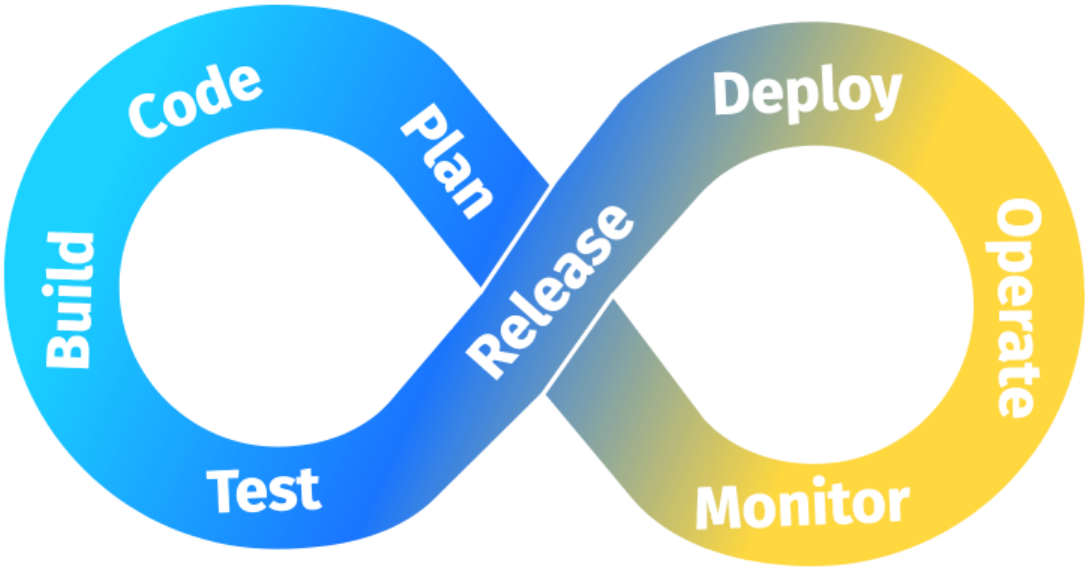
Operations

Will this microservice continuously change?

```
while (true) {
      on f {
-         return f()
+         //return f()
+         return f2()
      };
}
```

DevOps

DevOps



Microservices

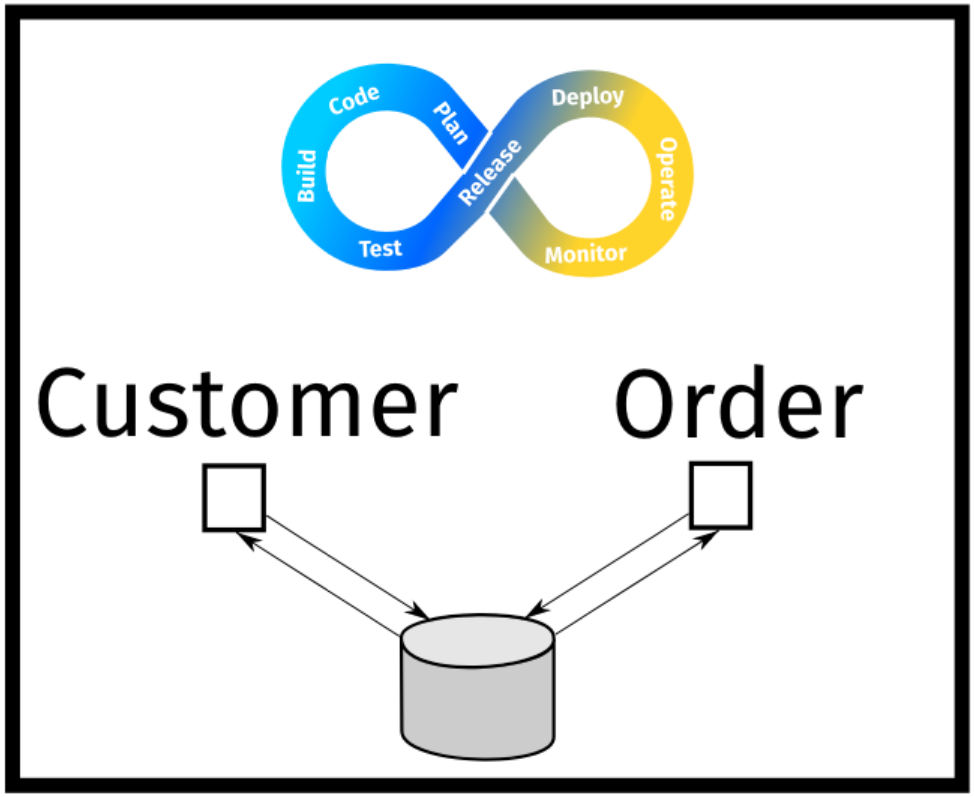
The microservice architectural style is an approach to developing a single application as a suite of **small** services, each running in its own **container** and communicating with **lightweight** mechanisms, often an **HTTP resource API**. These services are built around business capabilities and **independently deployable by fully automated deployment machinery**. There is a bare minimum of centralized management of these services, which may be written in **different programming languages** and use different data storage technologies.

Martin Fowler and James Lewis

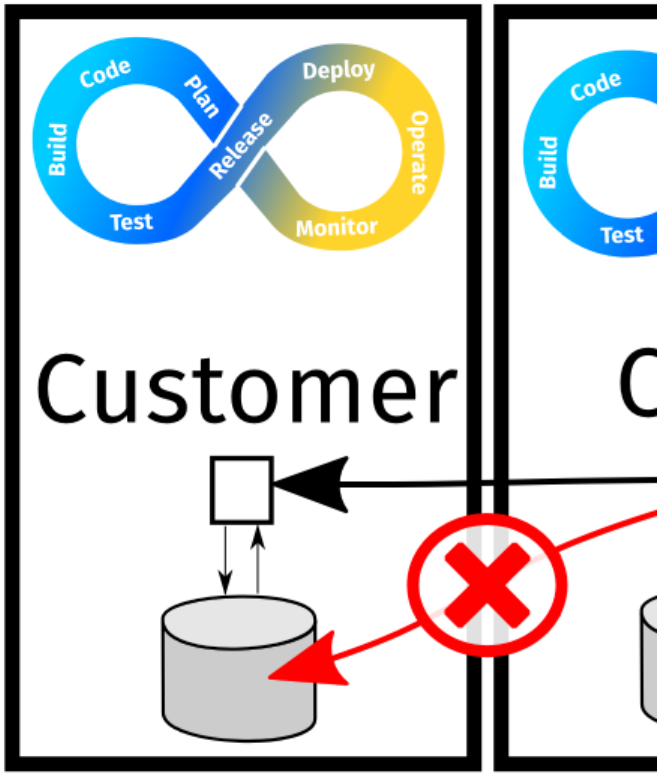
Microservices

The microservice architectural style is an approach to developing a single application as a suite of **small** services, each running in its own **container** and communicating with **lightweight** mechanisms, often an **HTTP resource API**. These services are built around business capabilities and **independently deployable by fully automated deployment machinery**. There is a bare minimum of centralized management of these services, which may be written in **different programming languages** and use **different data storage technologies**.

Martin Fowler and James Lewis



Monolith

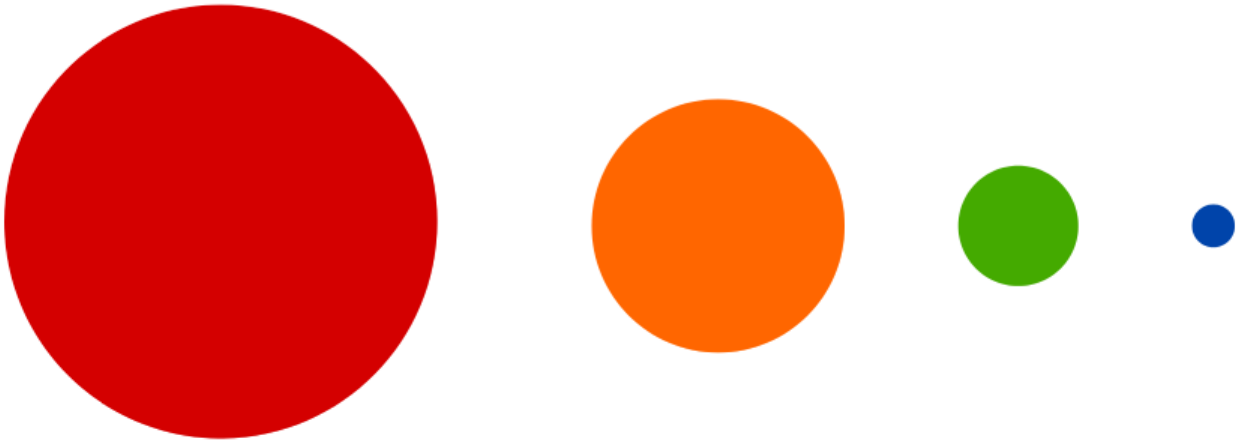


Microservi

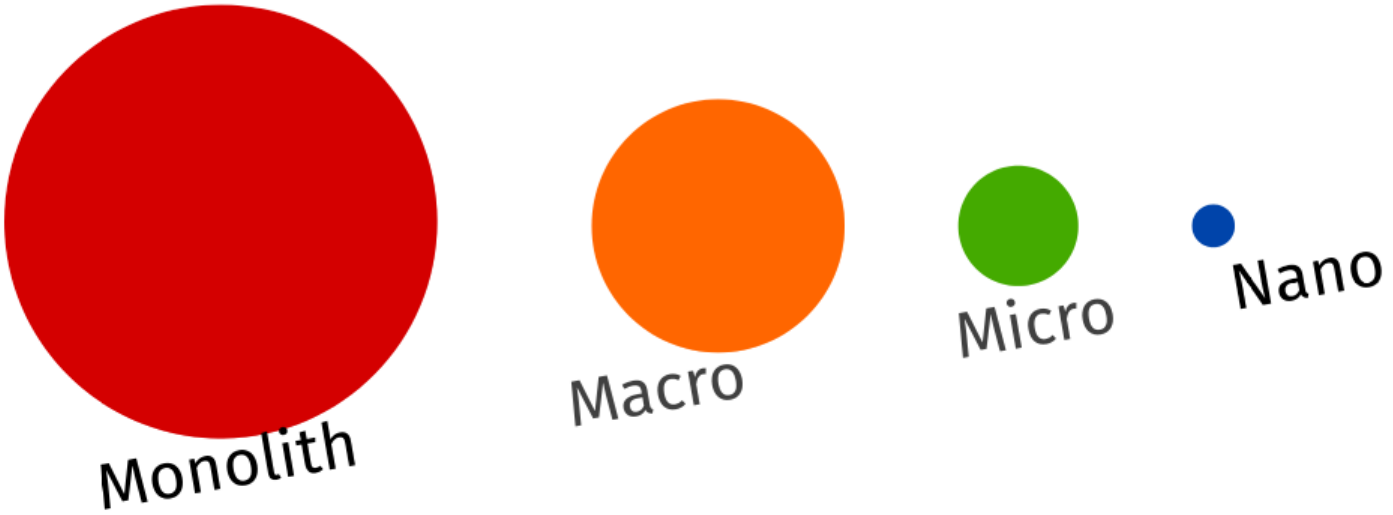
For us service orientation means encapsulating the data with the business logic that operates on the data, with the only access through a published service interface. No direct database access is allowed from outside the service, and there's **no data sharing among the services**.

Werner Vogels, Interviews Web Services: Learning from the Amazon technology platform , ACM Queue, 4(4), June 30, 2006

How small is a Microservice?



How small is a Microservice?

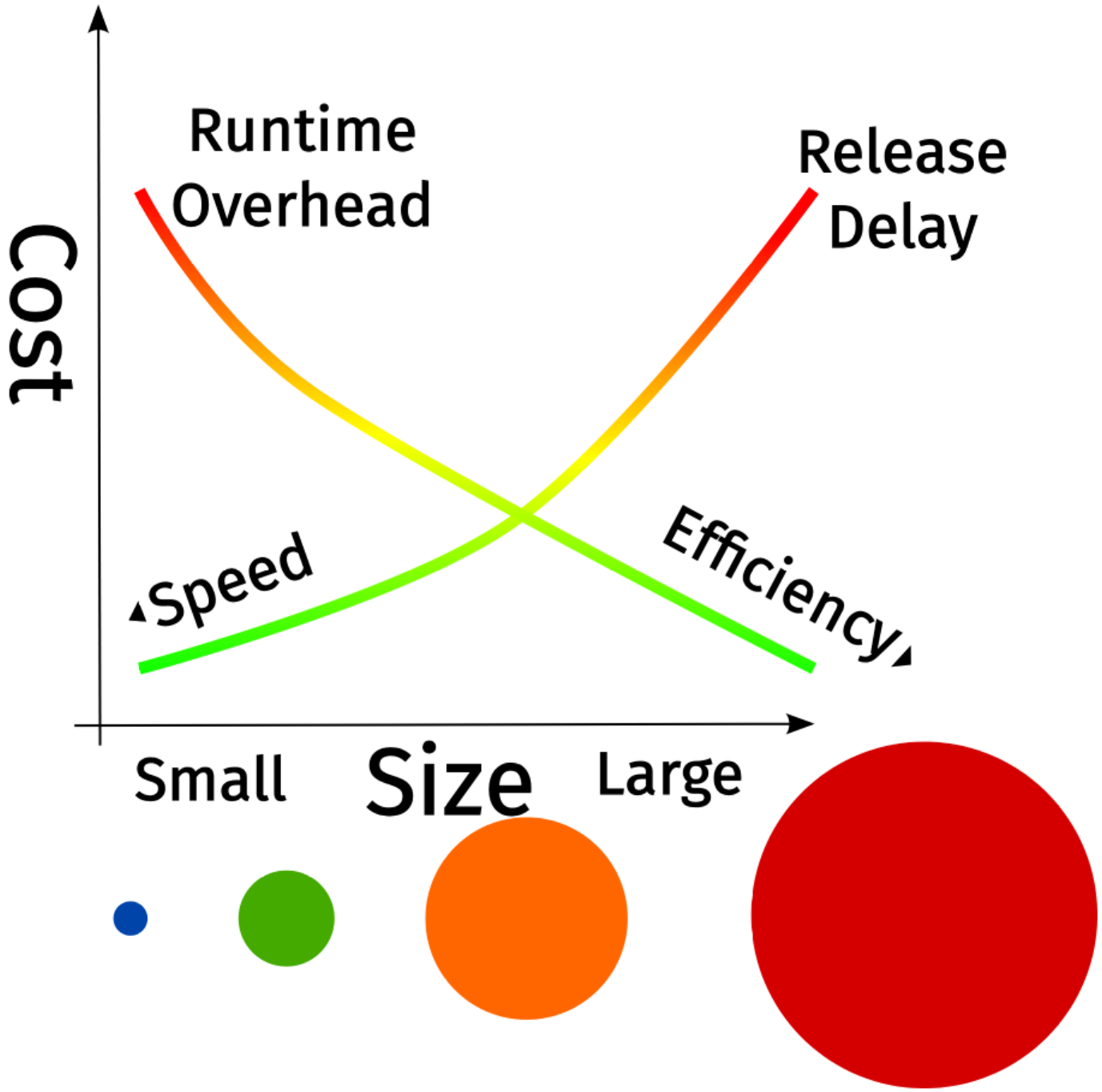


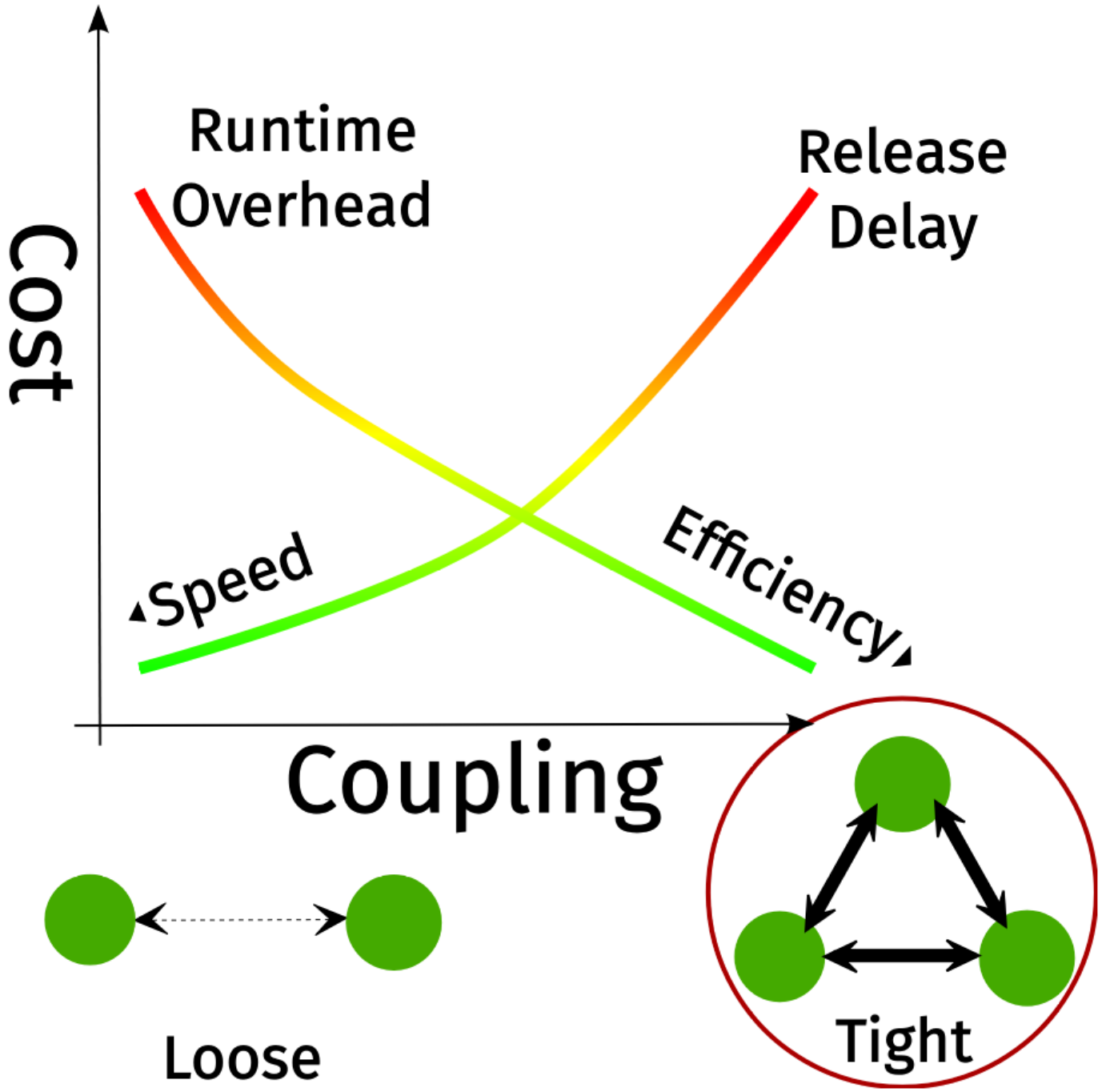

Powered by
ASQ
QUDOS2018

How small is a Microservice?

One team has full control of the entire devops (code, build, test, release, deploy and operate) cycle

Iterate fast: Many small frequent releases better than few large releases


Runtime
Overhead
Release
Delay
Cost
Speed
Efficiency
Small
Size
Large


Cost
Runtime Overhead
Release Delay
Speed
Efficiency
Coupling
Loose
Tight

Loosely Coupled Microservices

Avoid dependencies: If you have to hold a release until some other team is ready you do not have two separate microservices

Avoid cascading failures: A failed microservice should not bring down the whole system

Do you:

- ☐ Operate more than one microservice?
- ☐ Use polyglot persistence?
- ☐ Avoid storing everything in the same database?
- ☐ Assume eventual consistency?

SUBMIT

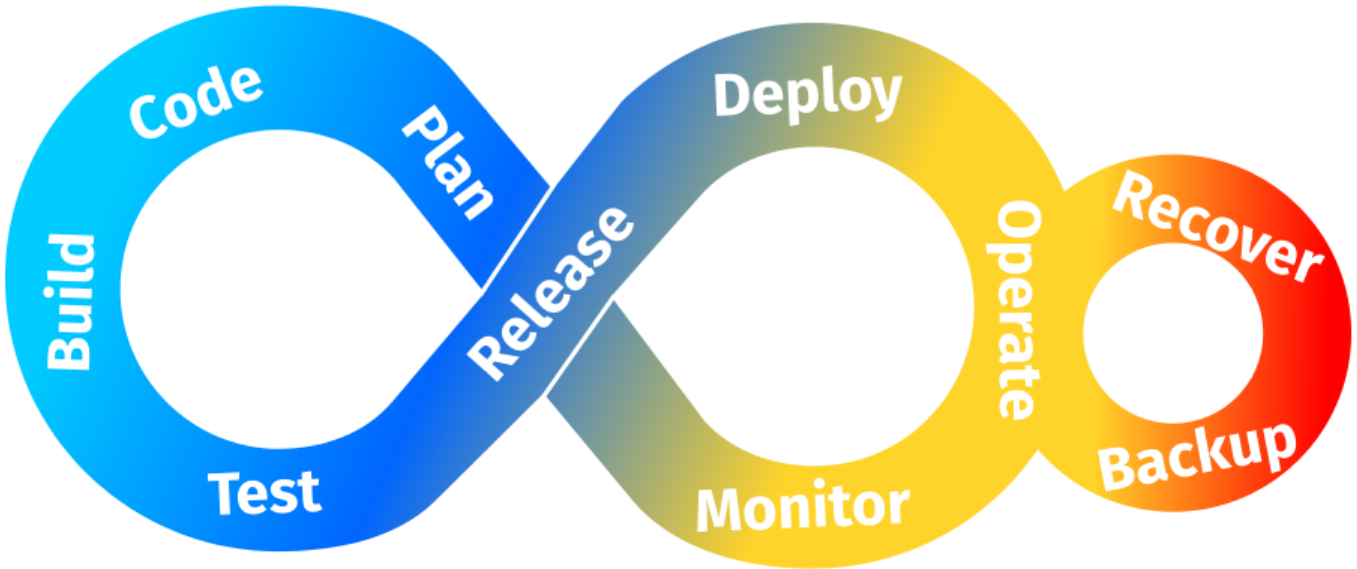
Microservices

Microservices prefer letting **each service manage its own database**, either different instances of the same database technology, or entirely different database systems - an approach called **Polyglot Persistence**.

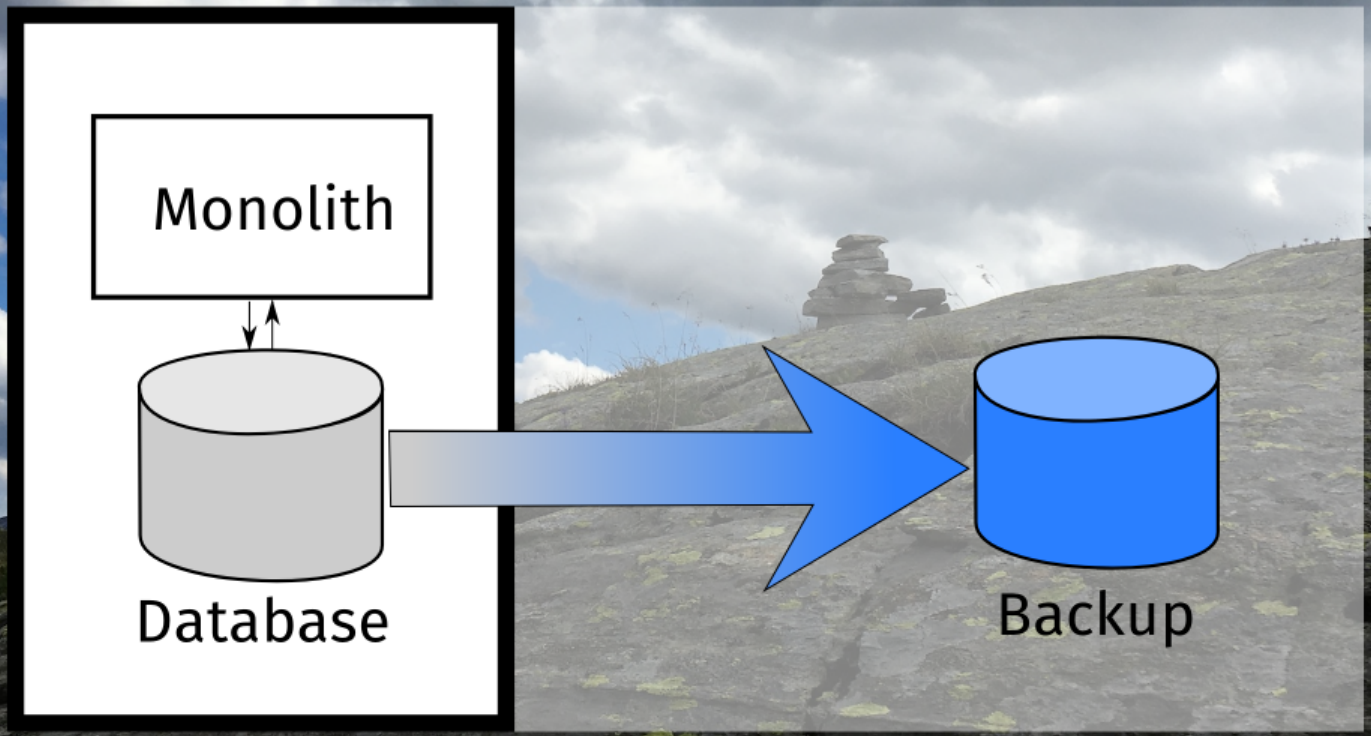
M. Fowler, J. Lewis https://www.martinfowler.com/articles/microservices.html


Powered by
A S Q
Eventual Inconsistency
Microservice architectures are doomed to become inconsistent after disaster strikes
Microservices
Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence.
M. Fowler, J. Lewis https://www.martinfowler.com/articles/microservices.html
@UDOS2018

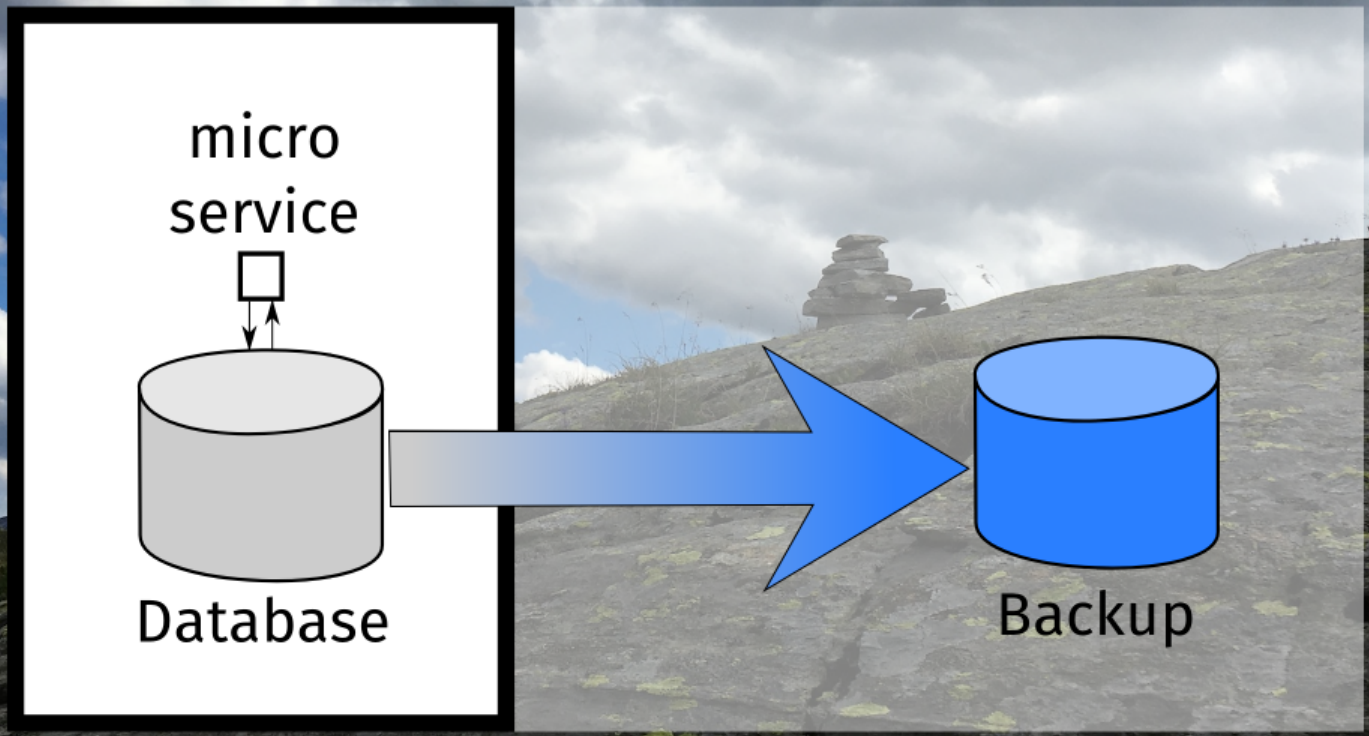
Devops meets Disaster Recovery



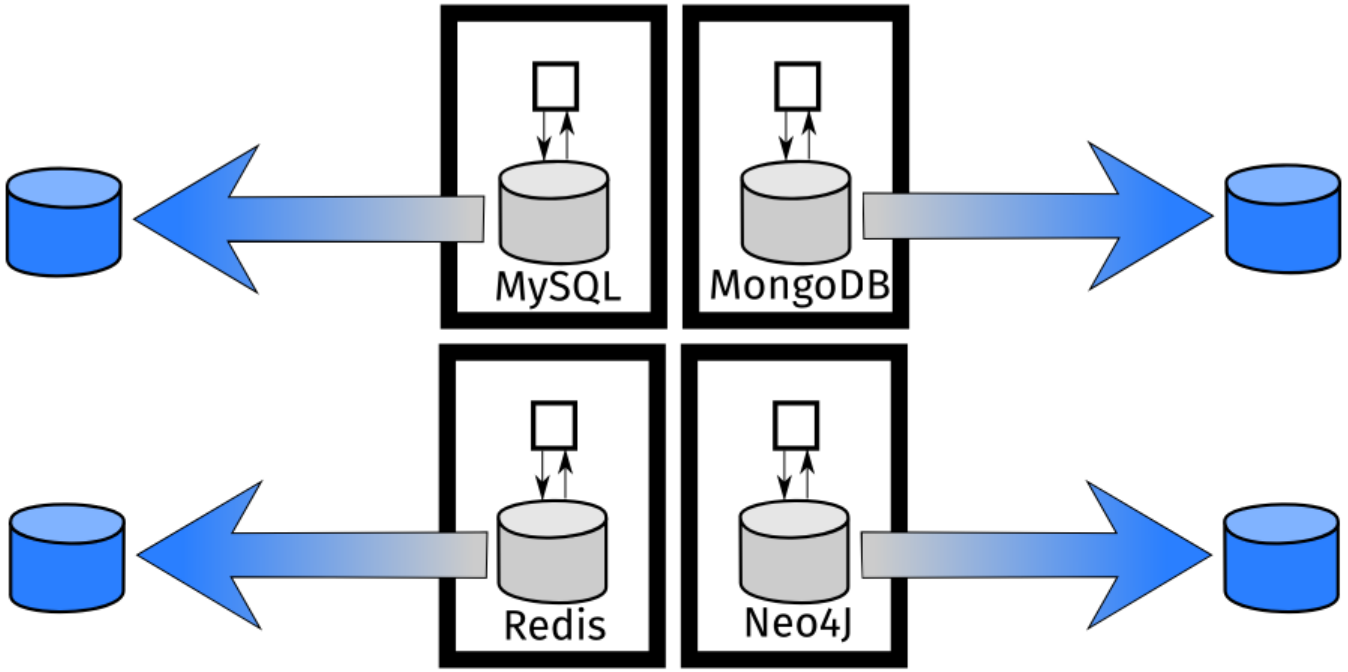
How do you back up a monolith?



How do you back up one microservice?

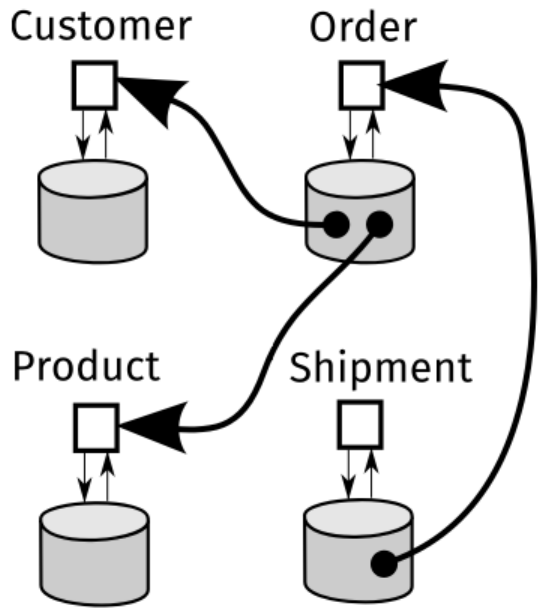


How do you back up an entire microservice architecture?



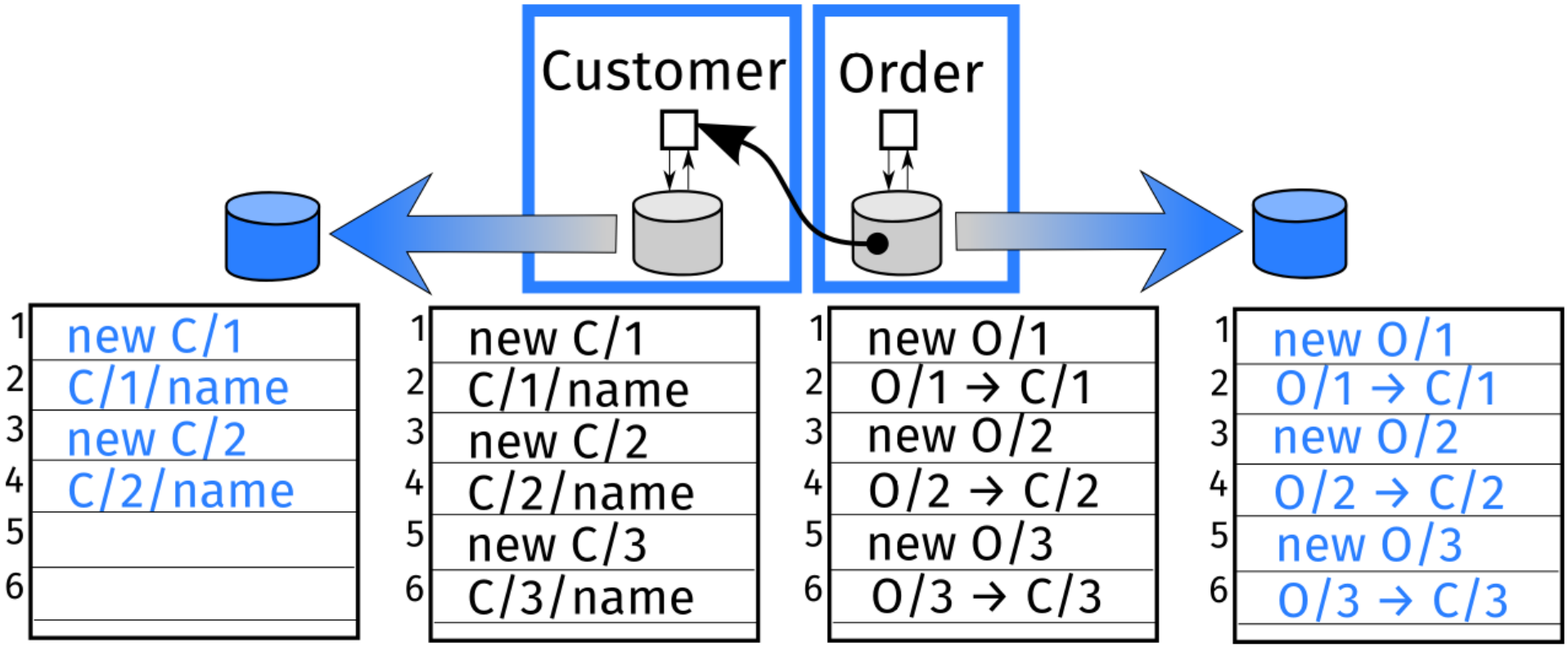
Are you sure?

Example



Data relationships across microservices =
Hypermedia

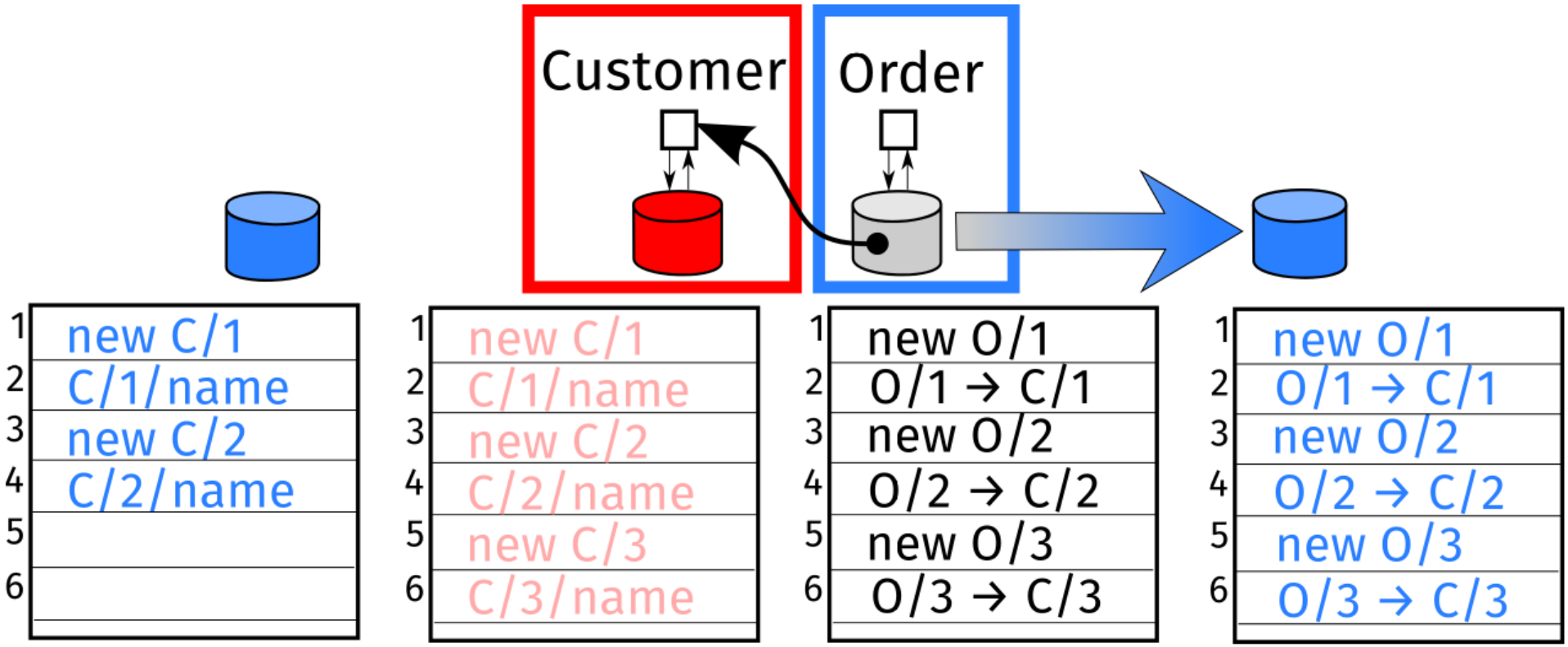
Independent Backup



Backups taken independently at different times

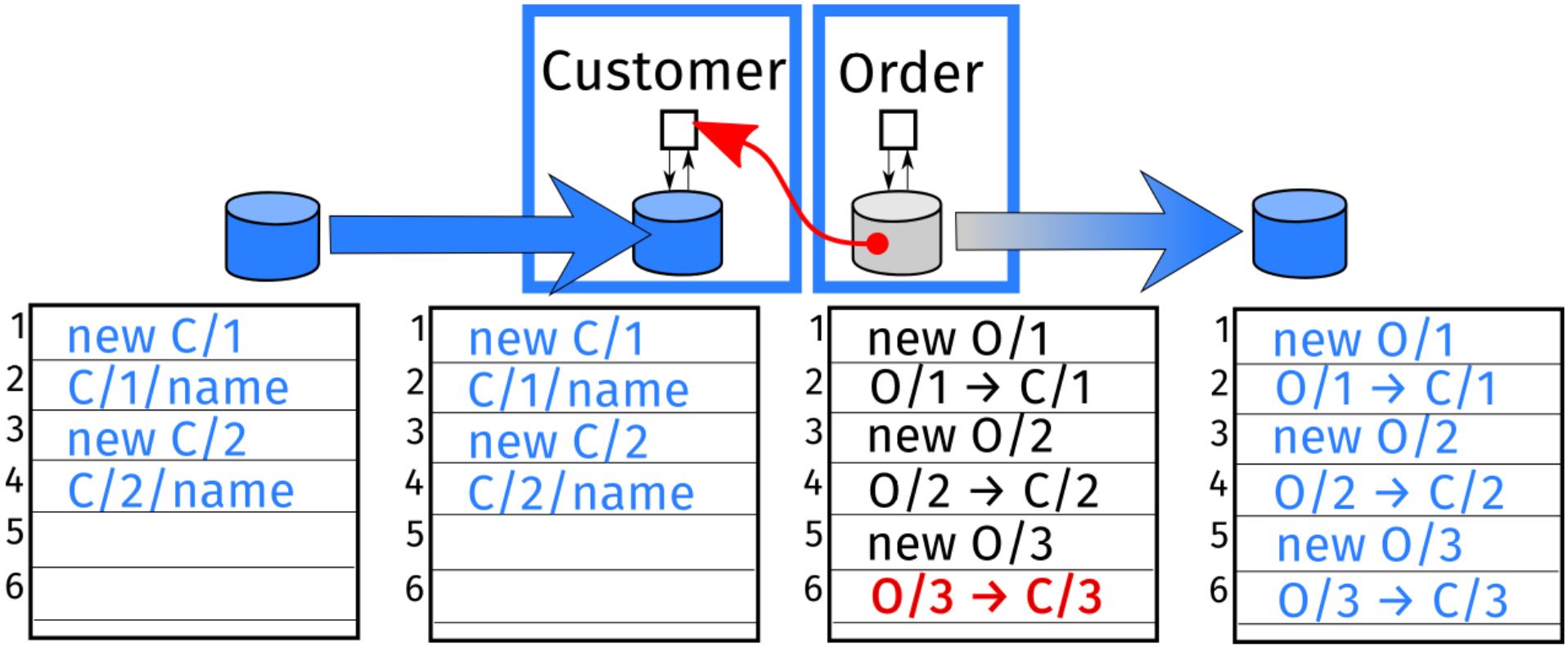

Powered by
ASQ
Disaster Strikes
CP306
GUARDIA COSTIERA
QUDOS2018

Disaster Strikes



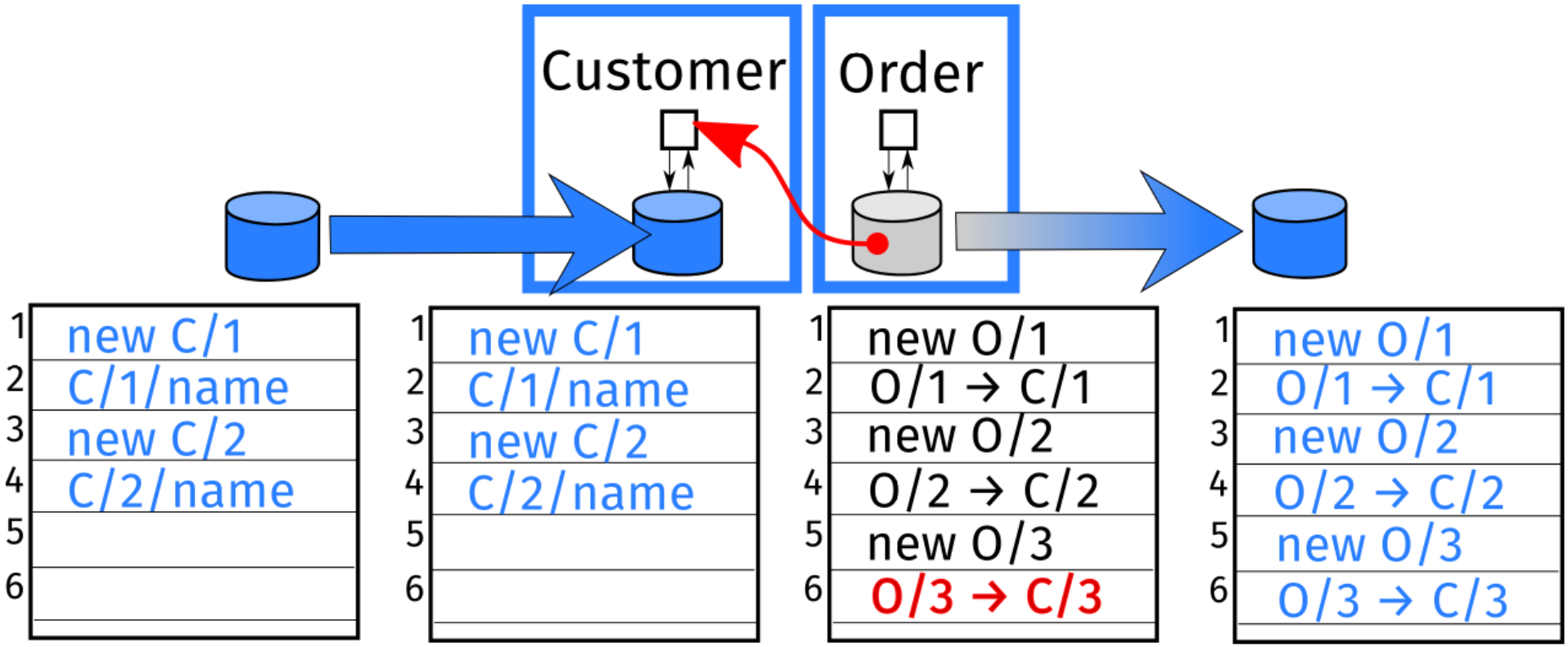
One microservice is lost

Recovery from Backup



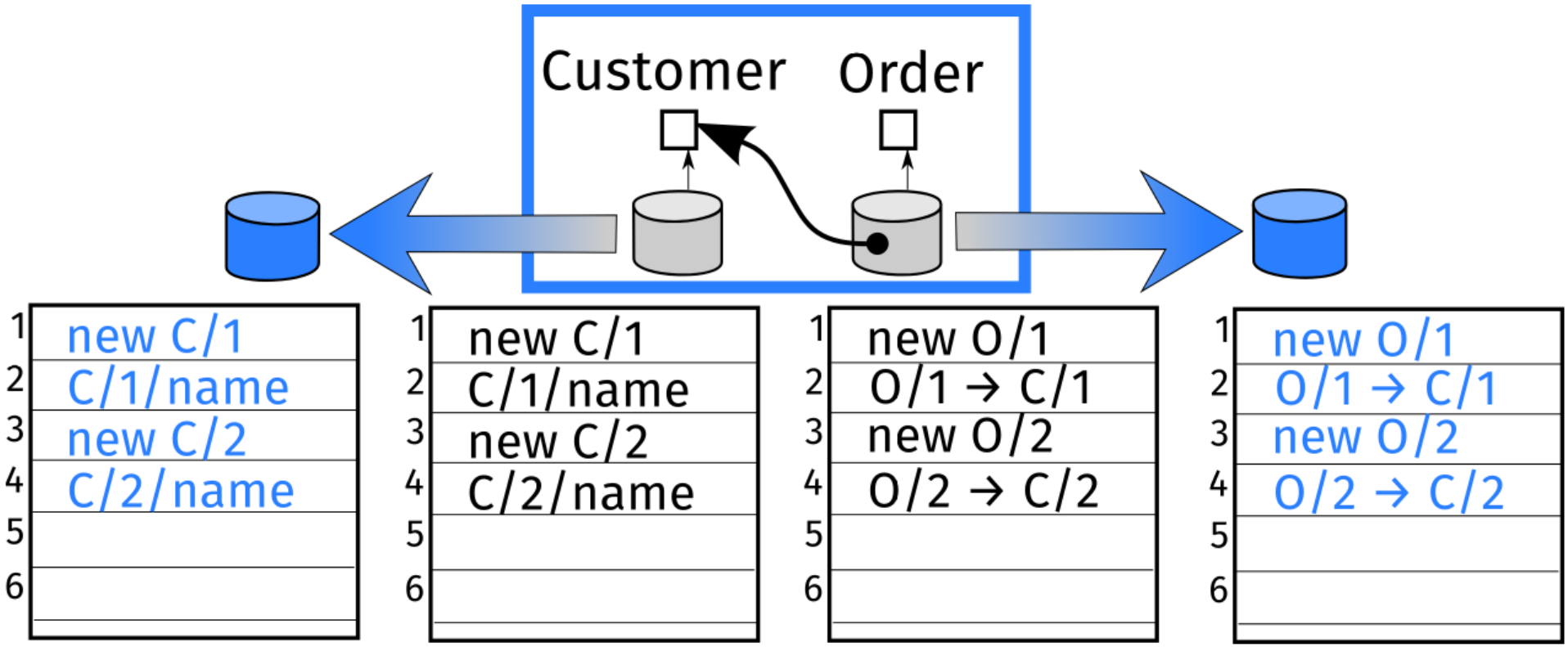
Broken link after recovery

Eventual Inconsistency



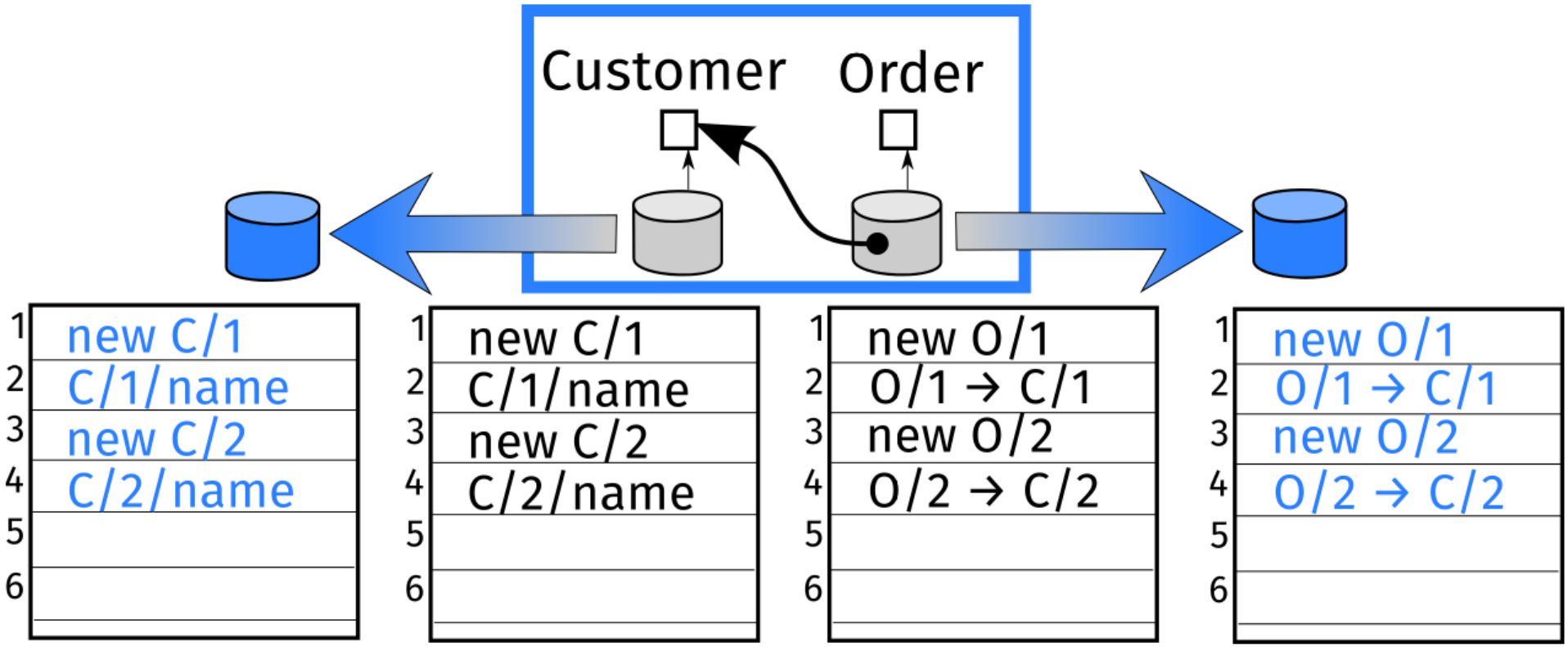
Broken link after recovery

Synchronized Backups



Backups of all microservices taken at the same time.

Limited Availability



No updates allowed anywhere while backing up the microservices

The BAC theorem

When **B**acking up a
microservice architecture,
it is not possible to have
both
Consistency and **A**vailability

Consistency

During normal operations, each microservice will eventually reach a consistent state

Referential integrity: links across microservice boundaries are guaranteed not to be broken

Availability

It is possible to both read **and update** the state of any microservice at any time

Backup

While backing up the system, is it possible to take a consistent snapshot of all microservices without affecting their availability?

No.

Backup + Availability

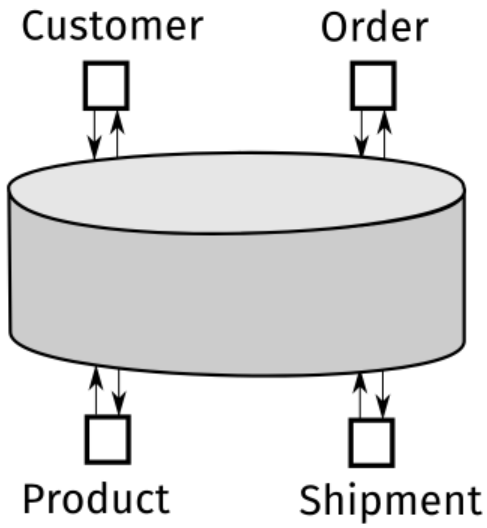
Backing up each microservice independently will eventually lead to inconsistency after recovering from backups taken at different times

Backup + Consistency

Taking a consistent backup requires to:

- disallow updates anywhere during the backup (limited availability)
- wait for the slowest microservice to complete the backup
- agree among all microservices on when to perform the backup (limited autonomy)

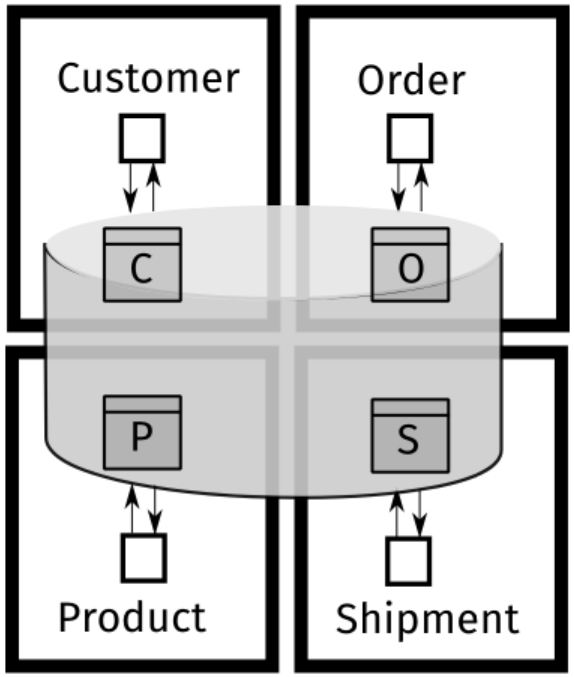
Shared Database



A centralized, shared database would require only one backup

Is this still a microservice architecture?

Shared Database, Split Schema

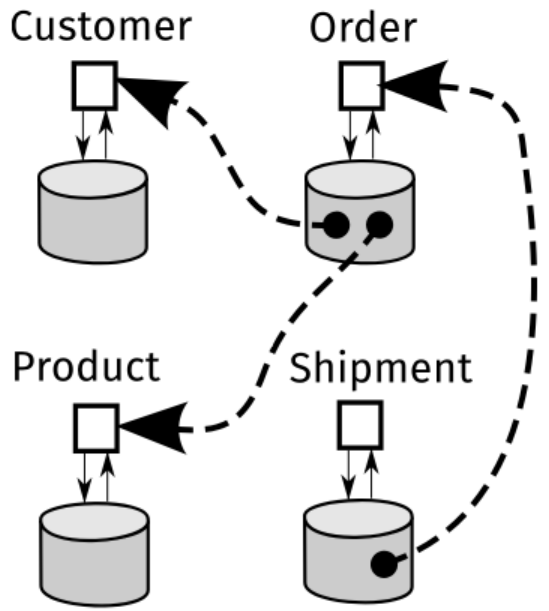


A centralized, shared database would require only one backup

Each microservice must use a logically separate schema

What happened to polyglot persistence?

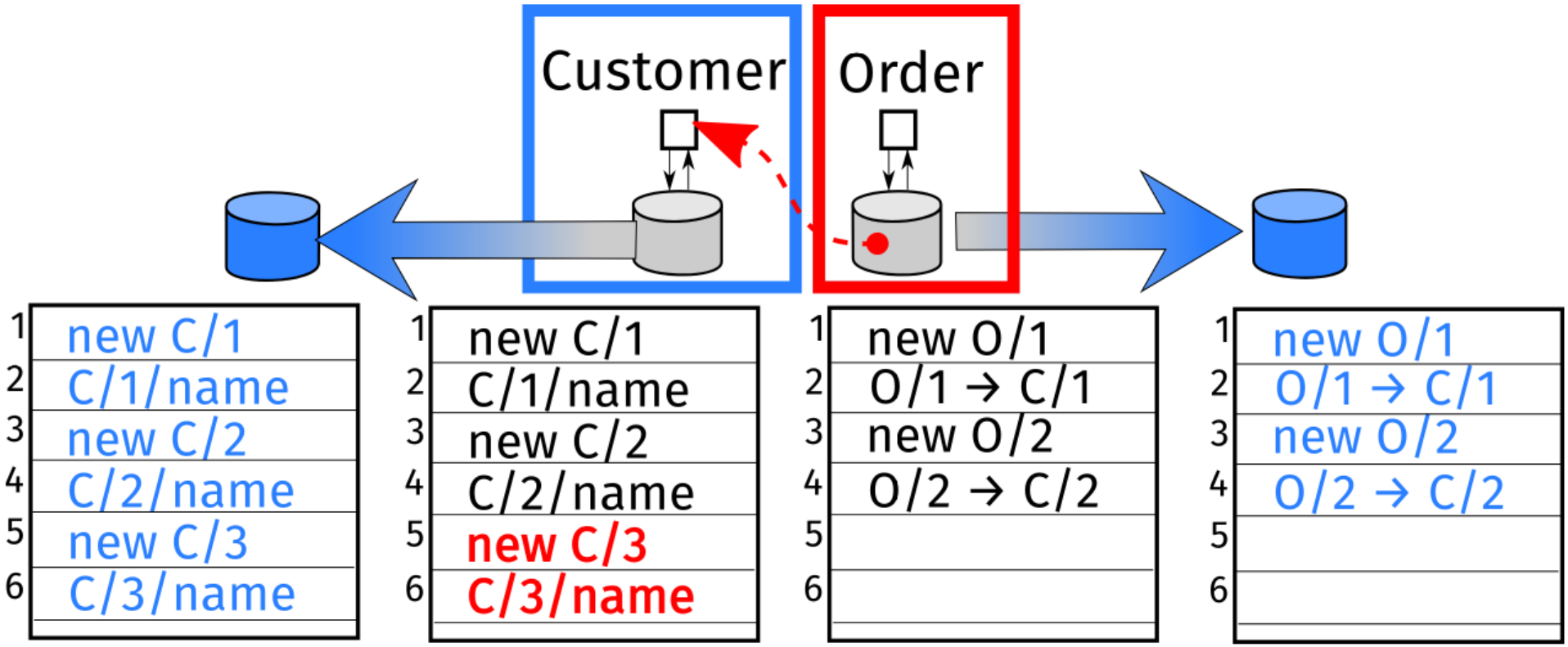
Links can break



No guarantees for references crossing microservice boundaries

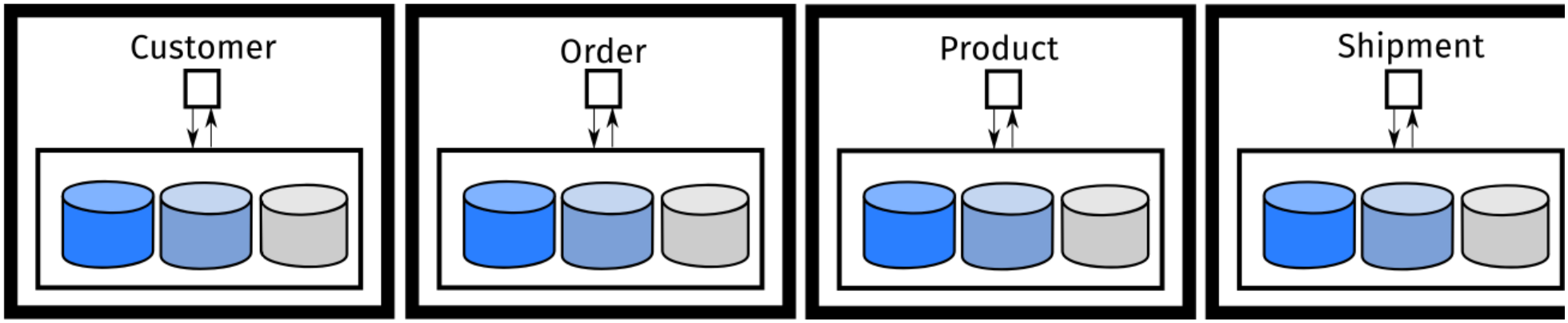
Microservices inherit a fundamental property of the Web

Orphan State



Orphan state is no longer referenced after recovery

Unstoppable System

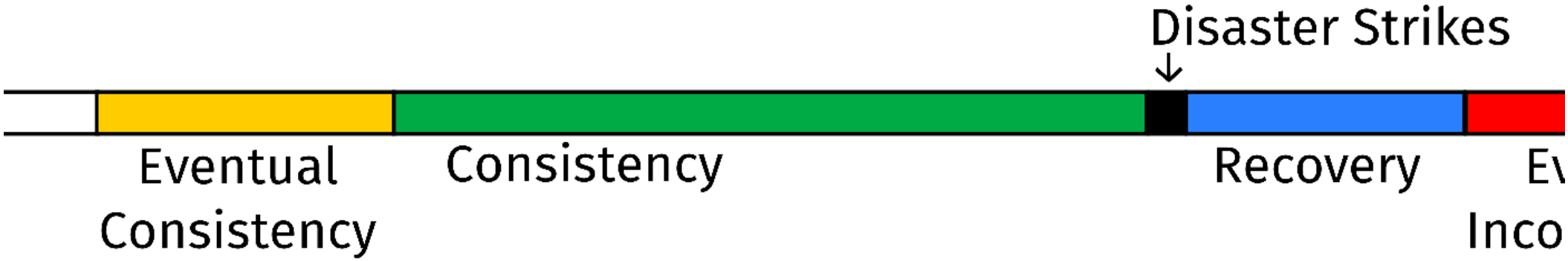


An expensive, replicated database with high-availability for every microservice


Powered by
A S Q
Unstoppable System
How do you restart an unstoppable system?
QUDOS2018

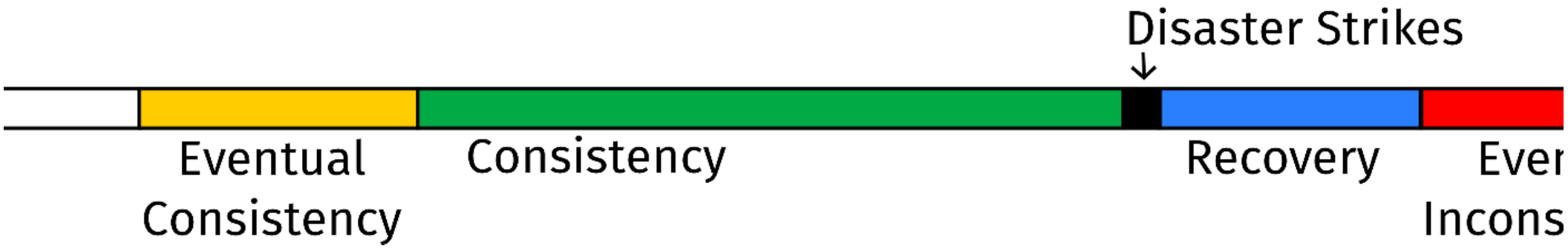
Eventual Consistency

Consistency


Disaster Strikes
Eventual Consistency
Consistency
Recovery
Ev
Inco

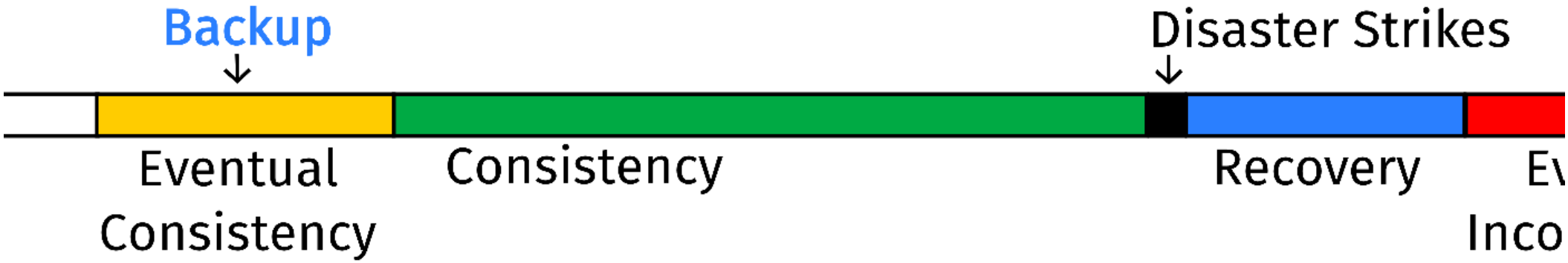
Eventual Consistency

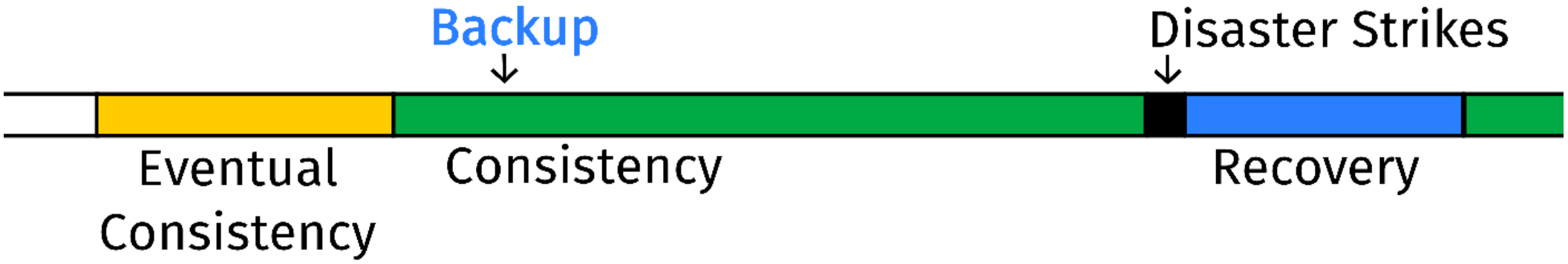
Retries are **enough** to deal with **temporary** failures of read operations, eventually the missing data will be found



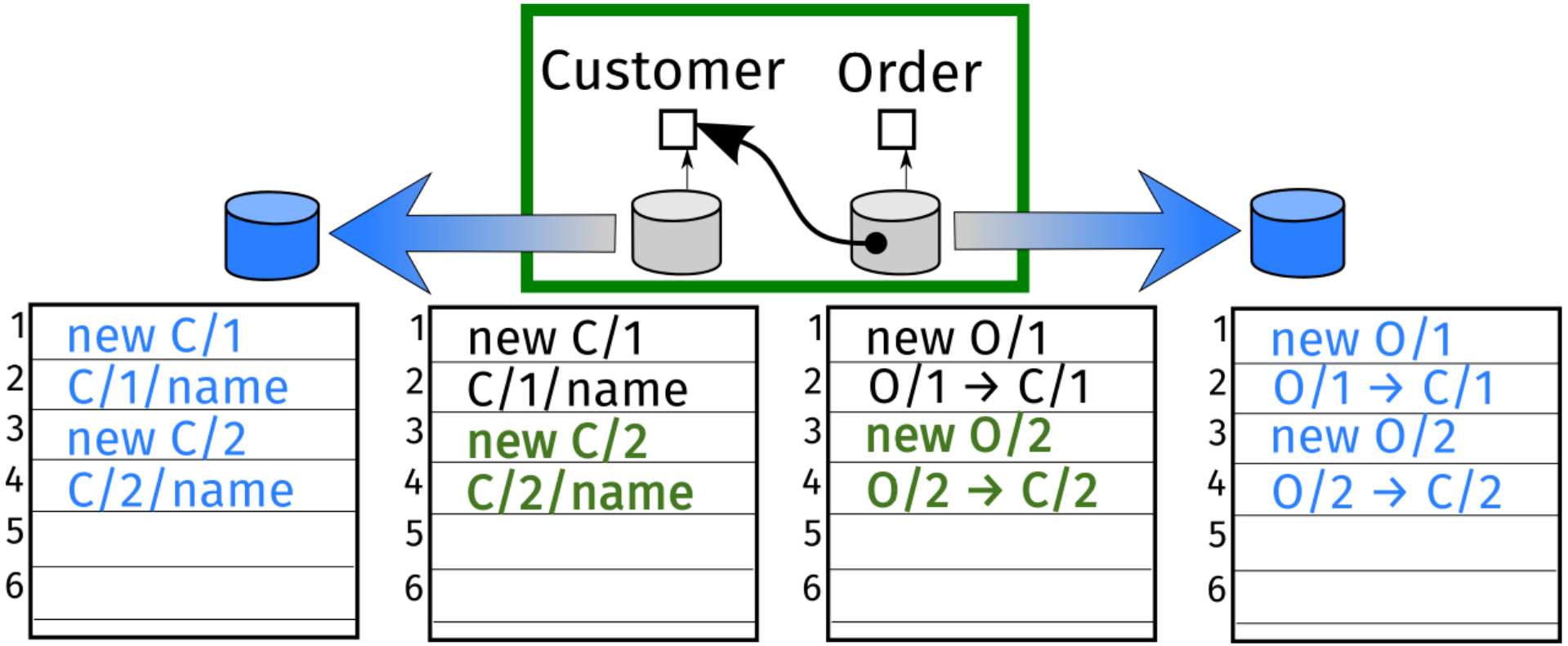
Eventual Inconsistency

Retries are **useless** to deal with **permanent** failures of read operations, which used to work just fine before disaster recovery


Backup
Disaster Strikes
Eventual
Consistency
Consistency
Recovery
Ev
Inco


Backup
Disaster Strikes
Eventual Consistency
Consistency
Recovery

Distributed Transactions



Take snapshots only when all microservices are consistent

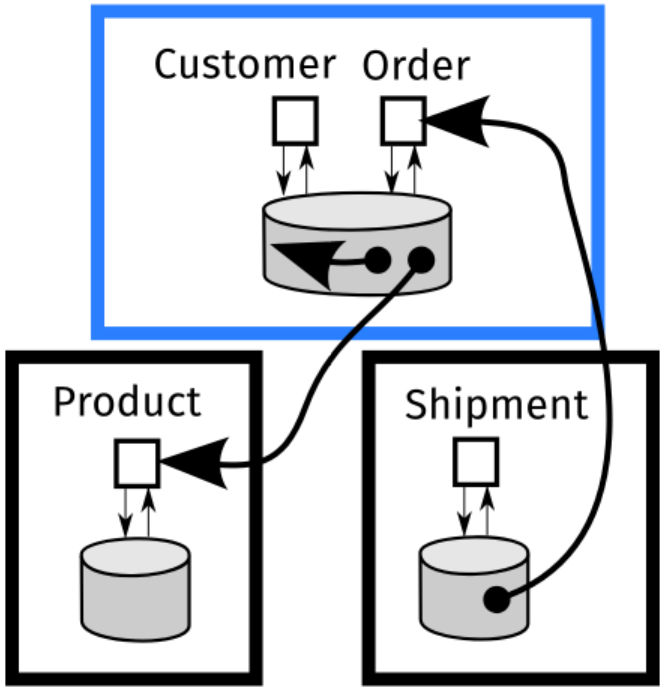
Avoid eventual consistency

Microservices

Distributed transactions are notoriously difficult to implement and as a consequence **microservice architectures emphasize transactionless coordination** between services, with explicit recognition that **consistency may only be eventual consistency** and problems are dealt with by compensating operations.

M. Fowler, J. Lewis https://www.martinfowler.com/articles/microservices.html

Splitting the Monolith



Keep data together for microservices that cannot tolerate eventual inconsistency

Does it apply to you?

- ☐ More than one stateful microservice
- ☐ Polyglot persistence
- ☐ Eventual Consistency
- ☐ (Cross-microservice references)
- ☐ Disaster recovery based on backup/restore
- ☐ **Independent** backups

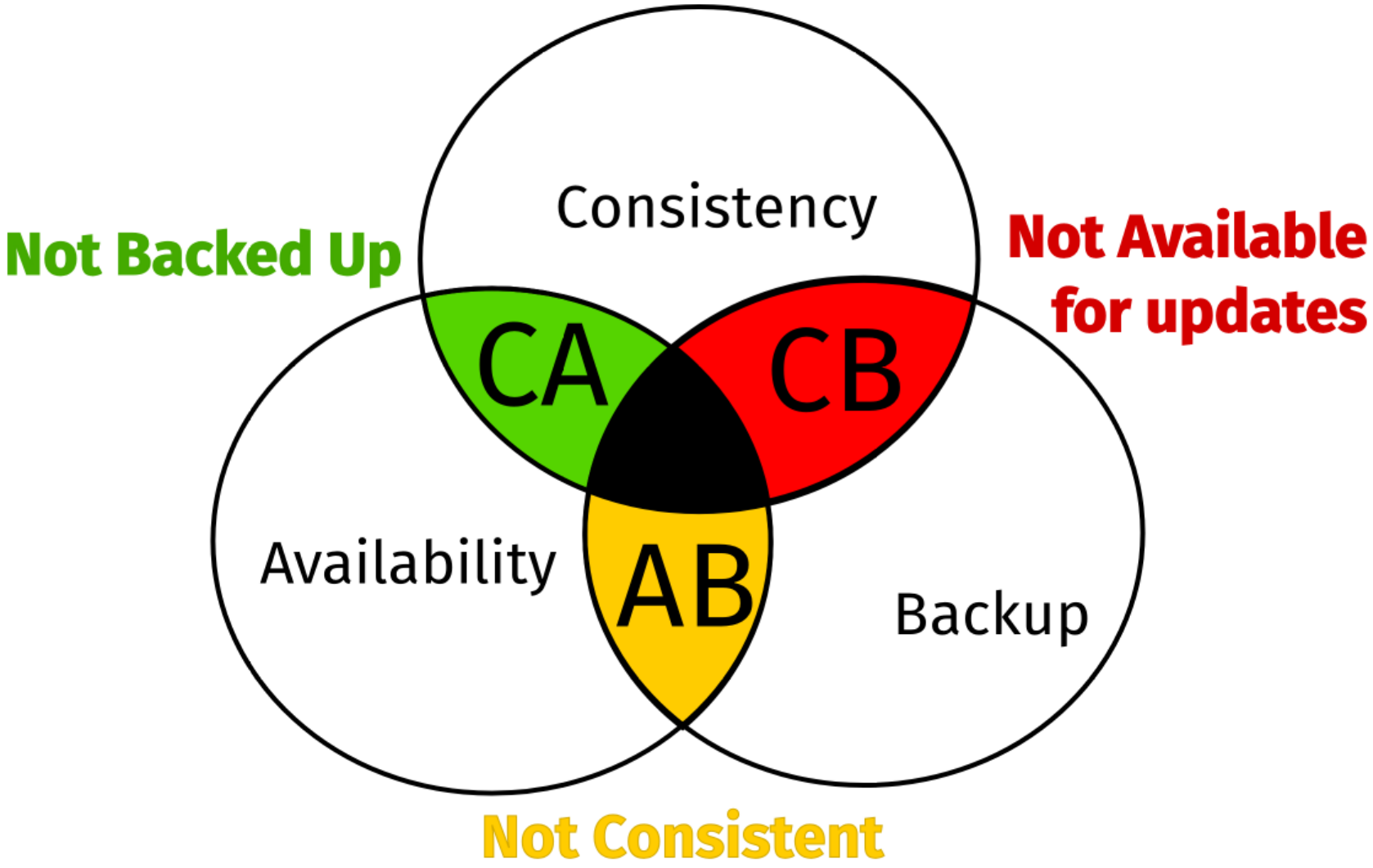
⇒ Eventual inconsistency (after disaster recovery)

Does it apply to you?

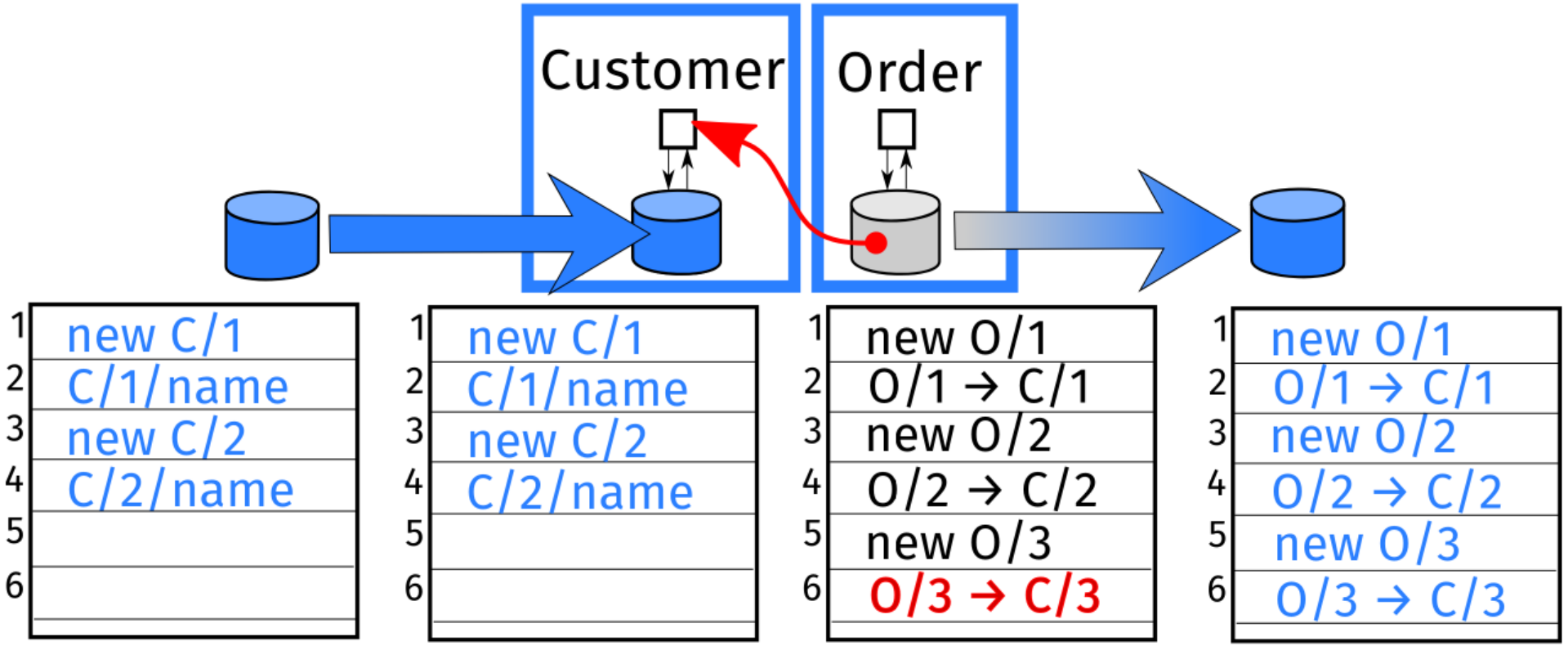
- ☐ More than one stateful microservice
- ☐ Polyglot persistence
- ☐ Eventual Consistency
- ☐ (Cross-microservice references)
- ☐ Disaster recovery based on backup/restore
- ☐ **Synchronized** backups (limited availability/autonomy)

⇒ Consistent Disaster Recovery

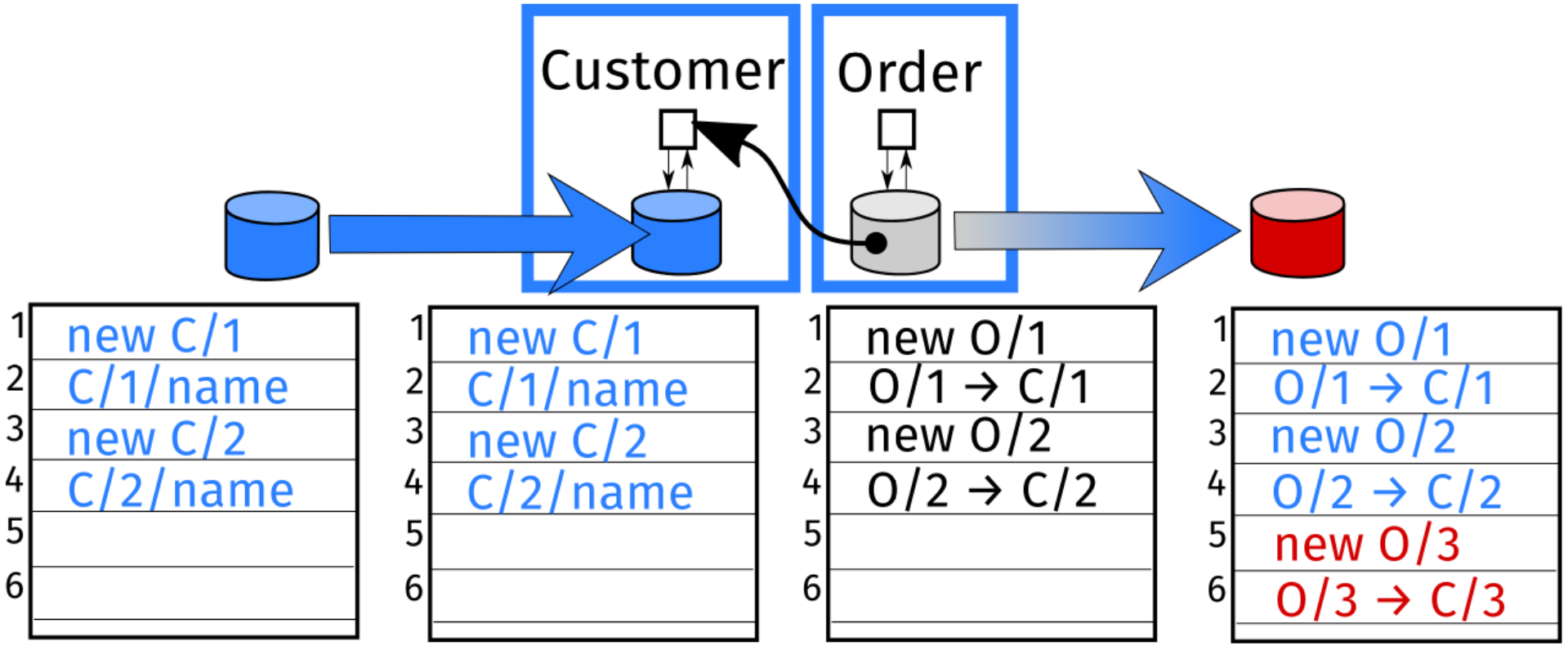
The BAC Theorem



No Backup


Customer
Order
1 new C/1
2 C/1/name
3 new C/2
4 C/2/name
5
6
1 new C/1
2 C/1/name
3 new C/2
4 C/2/name
5
6
1 new O/1
2 O/1 → C/1
3 new O/2
4 O/2 → C/2
5 new O/3
6 O/3 → C/3
1 new O/1
2 O/1 → C/1
3 new O/2
4 O/2 → C/2
5 new O/3
6 O/3 → C/3

No Backup



Trim to the oldest backup

Loose even more data!

The BAC Theorem

When Backing up a whole microservice architecture, it is not possible to have both Consistency and Availability

Corollaries

1. Microservice architectures eventually become inconsistent after disaster strikes when recovering from independent backups
2. Achieving consistent backups can be attempted by limiting the full availability of the system and synchronizing the backups


Powered by
ASQ
Dealing with the Consequences of BAC
1. Eventual Consistency breeds Eventual Inconsistency
2. Trade off: Cost of Recovery vs. Prevention
3. Cluster microservices to be backed up together
QUDOS2018

Consistent Disaster Recovery for Microservices: the BAC Theorem

Guy Pardon
Atomikos

Cesare Pautasso
Università della Svizzera Italiana, Lugano, Switzerland

Olaf Zimmermann
Hochschule für Technik Rapperswil (HSR FHO), Switzerland

How do you back up a microservice? You dump its database. But how do you back up an entire application decomposed into microservices? In this article, we discuss the tradeoff between the availability and consistency of a microservice-based architecture when a backup of the entire application is being performed. We demonstrate that service designers have to select two out of three qualities: backup, availability, and/or consistency (BAC). Service designers must also consider how to deal with consequences such as broken links, orphan state, and missing state.

Microservices are about the design of fine-grained services, which can be developed and operated by independent teams, ensuring that an architecture can organically grow and rapidly evolve.[1] By definition, each microservice is independently deployable and scalable; each stateful one relies on its own polyglot persistent storage mechanism. Integration at the database layer is not recommended, because it introduces coupling between the data representation internally used by multiple microservices. Instead, microservices should interact only through well-defined APIs, which—following the REST architectural style[2]—provide a clear mechanism for managing the state of the resources exposed by each microservice. Relationships between related entities are implemented using hypermedia,[3] so that representations retrieved from one microservice API can include links to other entities found on other microservice APIs. While there is no guarantee that a link retrieved from one microservice will point to a valid URL served by another, a basic notion of consistency can be introduced for the microservice-based application, requiring that such references can always be resolved, thus avoiding broken links. As the scale of the system grows, such a guarantee can be gradually weakened, as is currently the case for the World Wide Web.

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 ©2018 IEEE

References

- Guy Pardon, Cesare Pautasso, Olaf Zimmermann, Consistent Disaster Recovery for Microservices: the BAC Theorem , IEEE Cloud Computing, 5(1):49-59, January/February 2018
- Cesare Pautasso, Olaf Zimmermann, The Web as a Software Connector: Integration Resting on Linked Resources , IEEE Software, 35(1):93-98, January/February 2018
- Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier, A Pattern Language for RESTful Conversations , EuroPlop 2016
- Ana Ivanchikj, Cesare Pautasso, and Silvia Schreier, Visual modeling of RESTful conversations with RESTalk , Journal of Software & Systems Modeling, pp. 1-21, May, 2016. (Journal-First **Best Paper Award** at MODELS 2016)
- Florian Haupt, Frank Leymann, Cesare Pautasso: A conversation based approach for modeling REST APIs , 12th Working IEEE / IFIP Conference on Software Architecture (WICSA 2015), Montreal, CA, May 2015
- Gregor Hohpe, "Let's have a conversation", Internet Computing, IEEE 11.3 (2007): 78-81.
- Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures , University of California, Irvine, 2000
- Guy Pardon and Cesare Pautasso, Atomic Distributed Transactions: a RESTful Design , 5th International Workshop on Web APIs and RESTful Design (WS-REST), Seoul, Korea, ACM, April, 2014.

2016. (Journal-First **Best Paper Award** at MODELS 2016)

- Florian Haupt, Frank Leymann, Cesare Pautasso: A conversation based approach for modeling REST APIs , 12th Working IEEE / IFIP Conference on Software Architecture (WICSA 2015), Montreal, CA, May 2015
- Gregor Hohpe, "Let's have a conversation", Internet Computing, IEEE 11.3 (2007): 78-81.
- Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures , University of California, Irvine, 2000
- Guy Pardon and Cesare Pautasso, Atomic Distributed Transactions: a RESTful Design , 5th International Workshop on Web APIs and RESTful Design (WS-REST), Seoul, Korea, ACM, April, 2014.
- Jim Webber, Savas Parastatidis, Ian Robinson, REST in Practice: Hypermedia and Systems Architecture, O'Reilly, 2010
- Cesare Pautasso, Erik Wilde, Push-Enabling RESTful Business Processes , 9th International Conference on Service Oriented Computing (ICSOC 2011), Paphos, Cyprus, December 2011
- Cesare Pautasso, BPMN for REST , In Proceedings of the 3rd International Workshop on BPMN (BPMN 2011), Lucerne, Switzerland, November 2011
- Thomas Erl, Benjamin Carlyle, Cesare Pautasso, Raj Balasubramanian, SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST , Prentice Hall, 2012

Made with

http://asq.inf.usi.ch

Acknowledgements

Guy Pardon, Olaf Zimmermann, Florian Haupt,
Silvia Schreier, Ana Ivanchikj, Mathias Weske,
Adriatik Nikaj, Sankalita Mandal,
Hagen Overdick, Jesus Bellido, Rosa Alarcón,
Alessio Gambi, Daniele Bonetta,
Achille Peternier, Erik Wilde, Mike Amundsen,
Stefan Tilkov, James Lewis