Is it possible to consistently recover a microservice architecture?

Cesare Pautasso

Software Institute, USI, Lugano, Switzerland

http://www.pautasso.info/talks/2019/lamsade

@pautasso@scholar.social

powered by ASQ

Software Institute

- Opened July 2017
- 6 Professors
- 30+ PhD/Postdoc researchers
- Software Engineering Research (Analytics, Architecture, Education, Evolution, Testing, Verification)
- New Master Software and Data Engineering

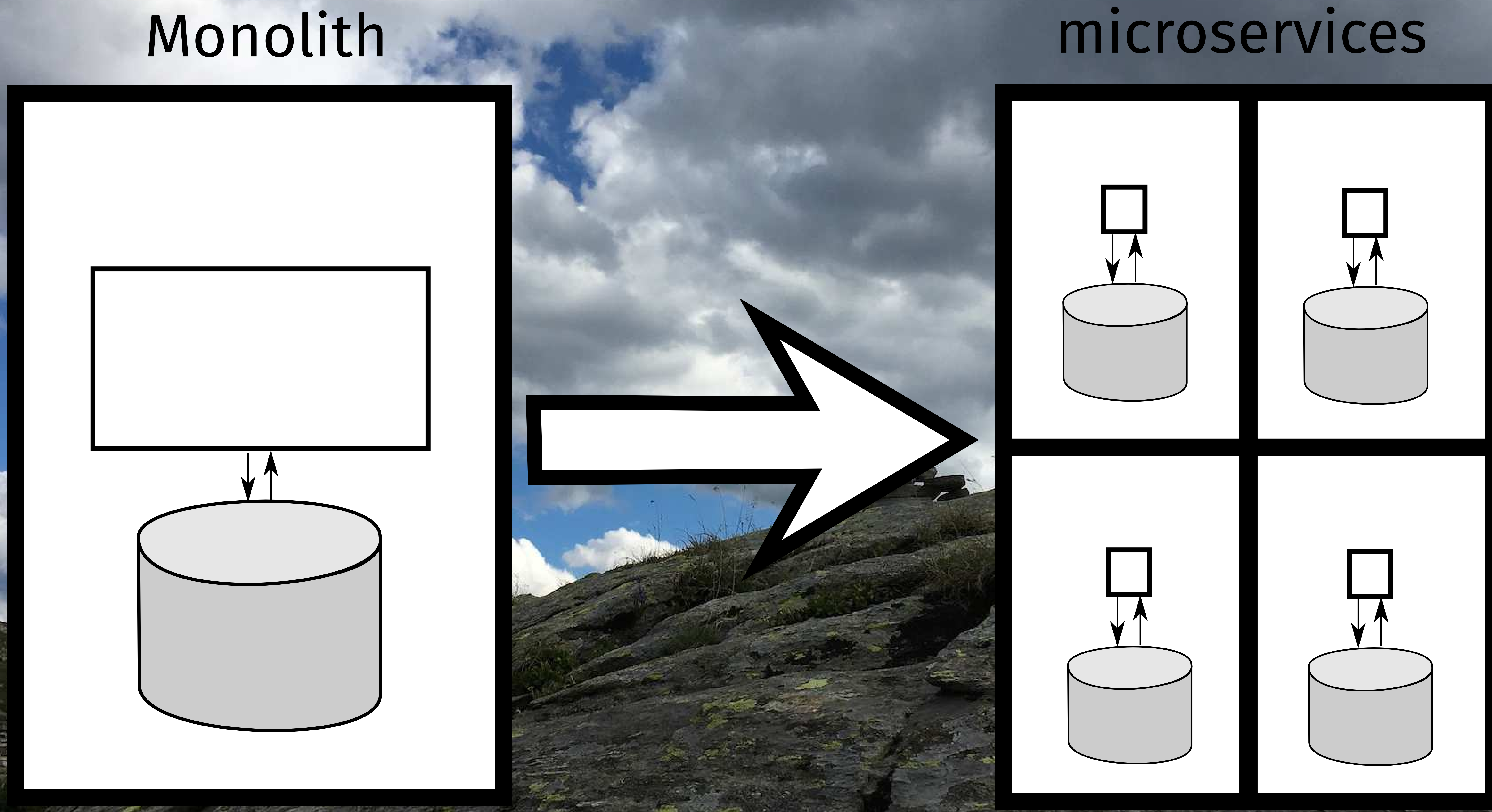
http://www.si.usi.ch/

Architecture, Design and Web Information Systems Engineering

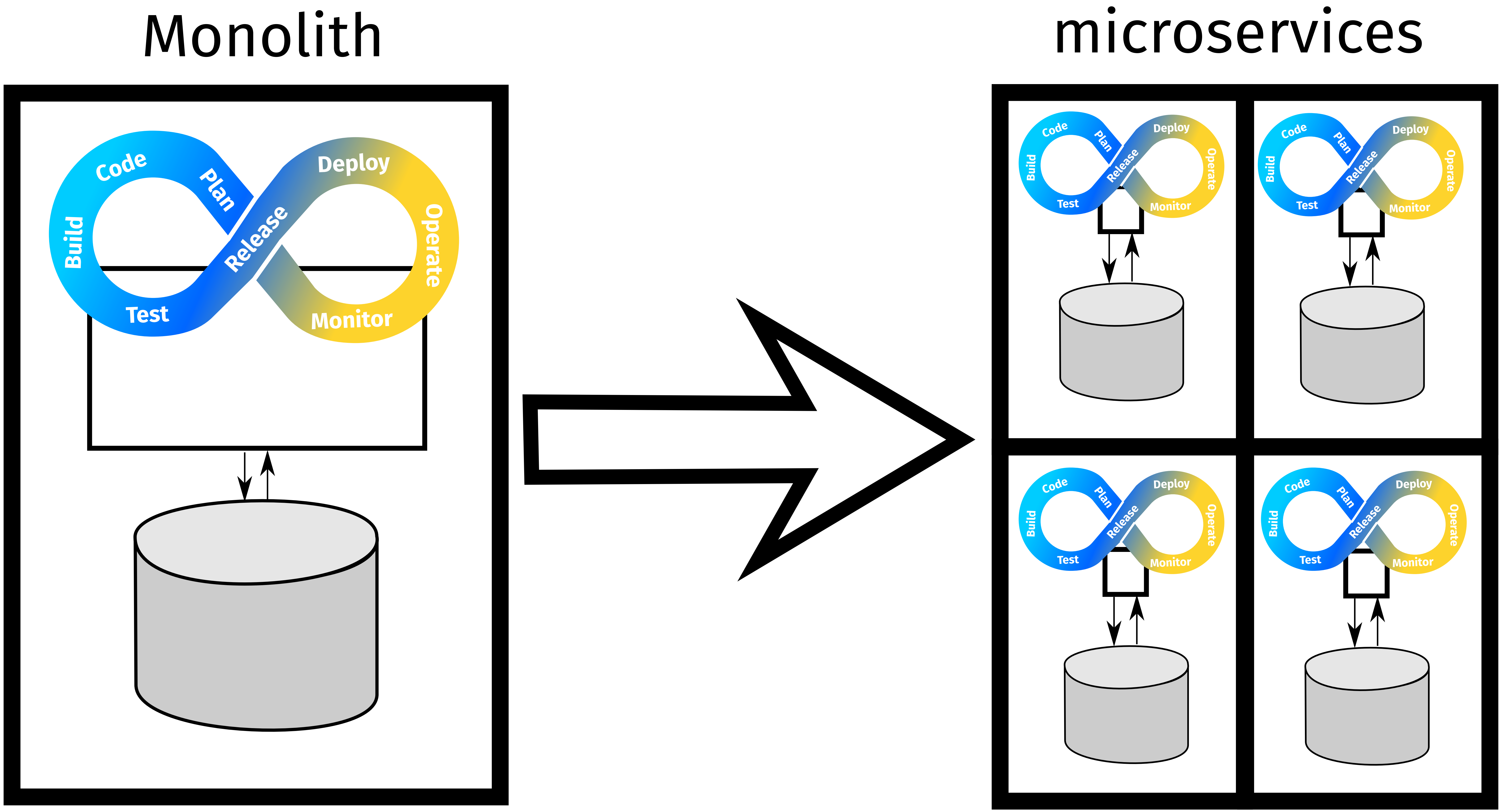
- RESTalk - API Conversation Modeling
- Liquid Software Architecture
- BenchFlow - a benchmark for workflow engines
- ASQ - Interactive Web Lectures, Classroom Analytics
- Parallel JavaScript/Multicore Node.JS
- SAW - Collaborative Architectural Decision Making
- NaturalMash - API Composition with Natural Language

http://design.inf.usi.ch

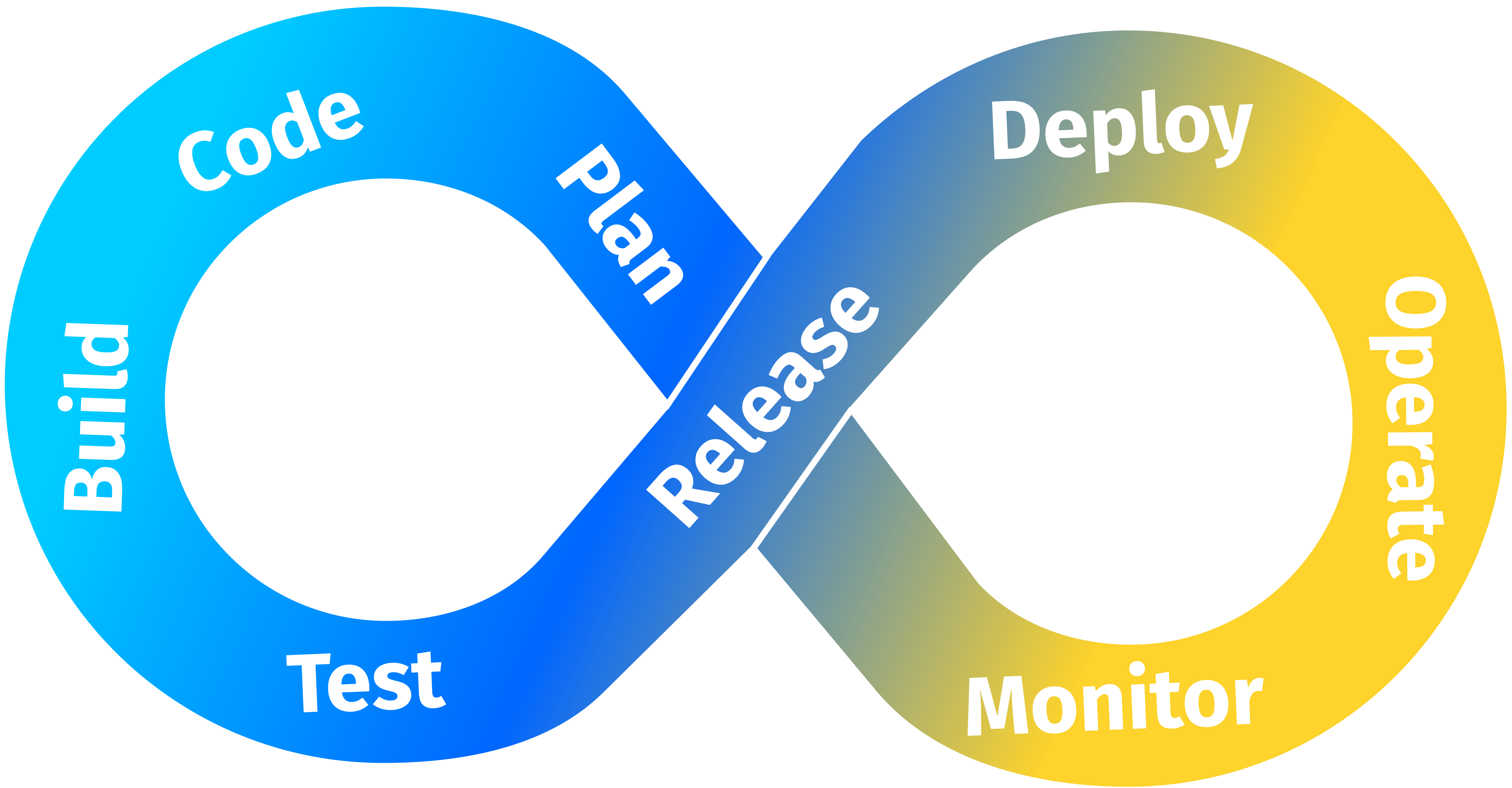
Decomposition



Independent DevOps Lifecycle



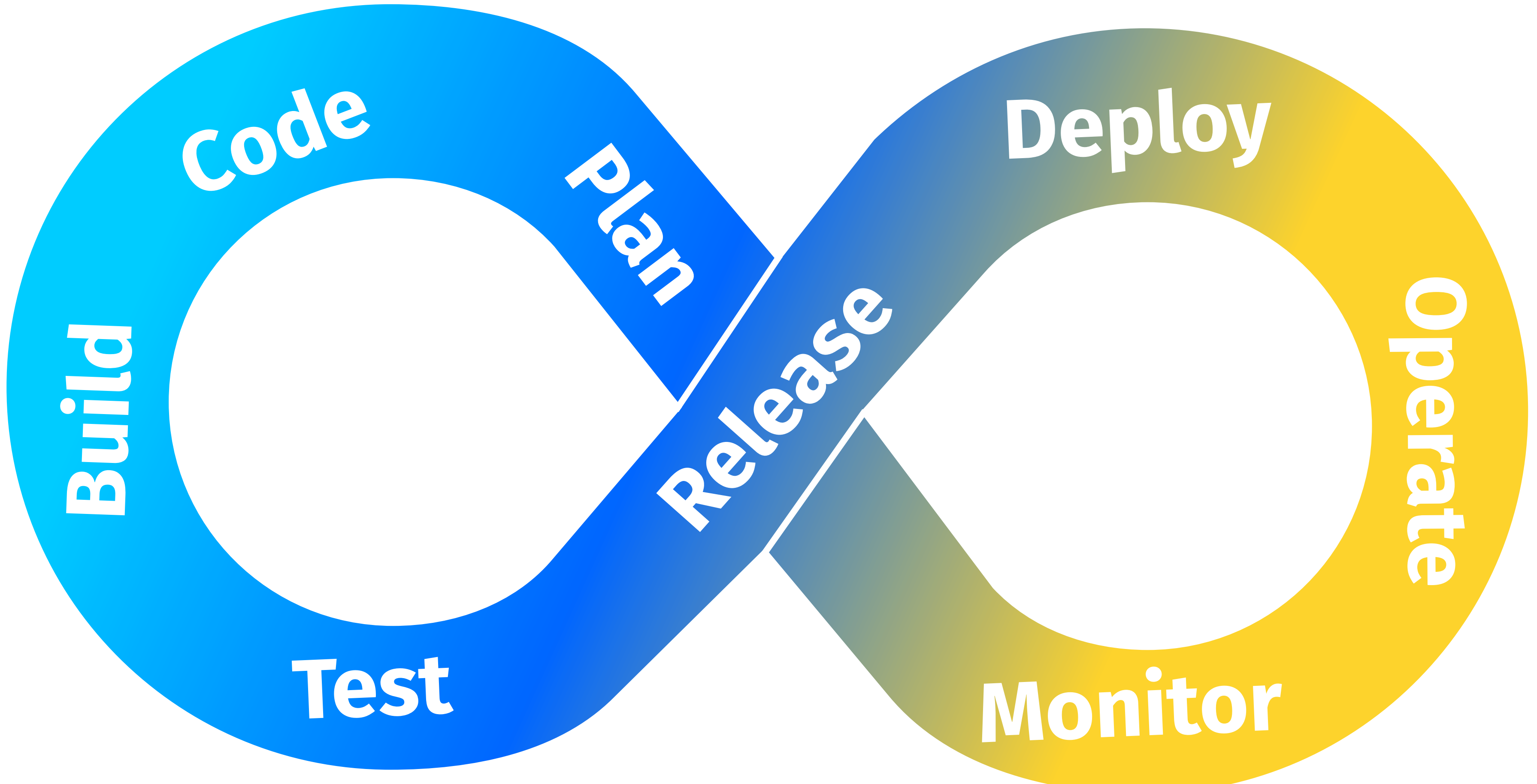
Autonomous Microservices



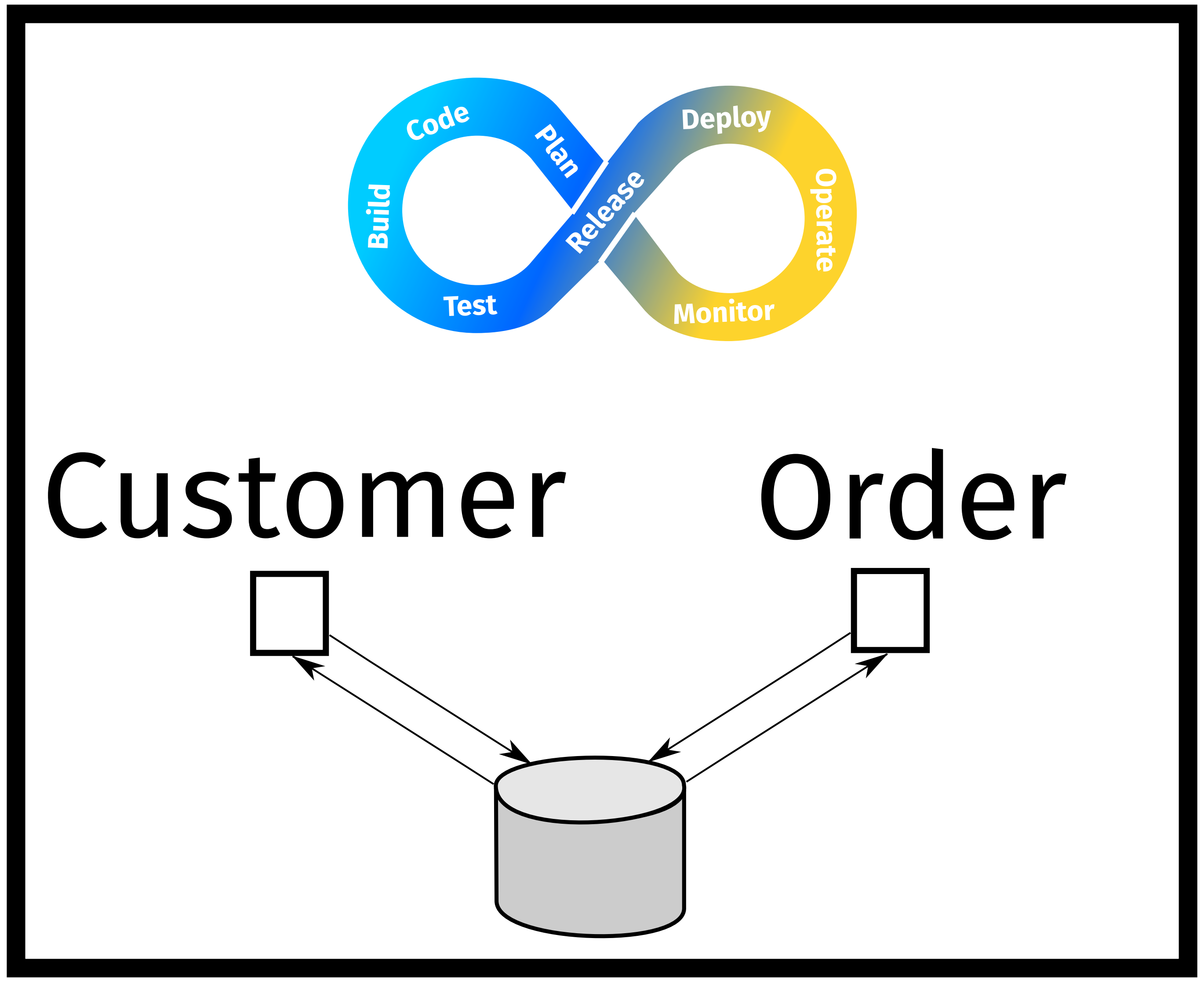
Autonomous Microservices

Rapid Evolution: 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

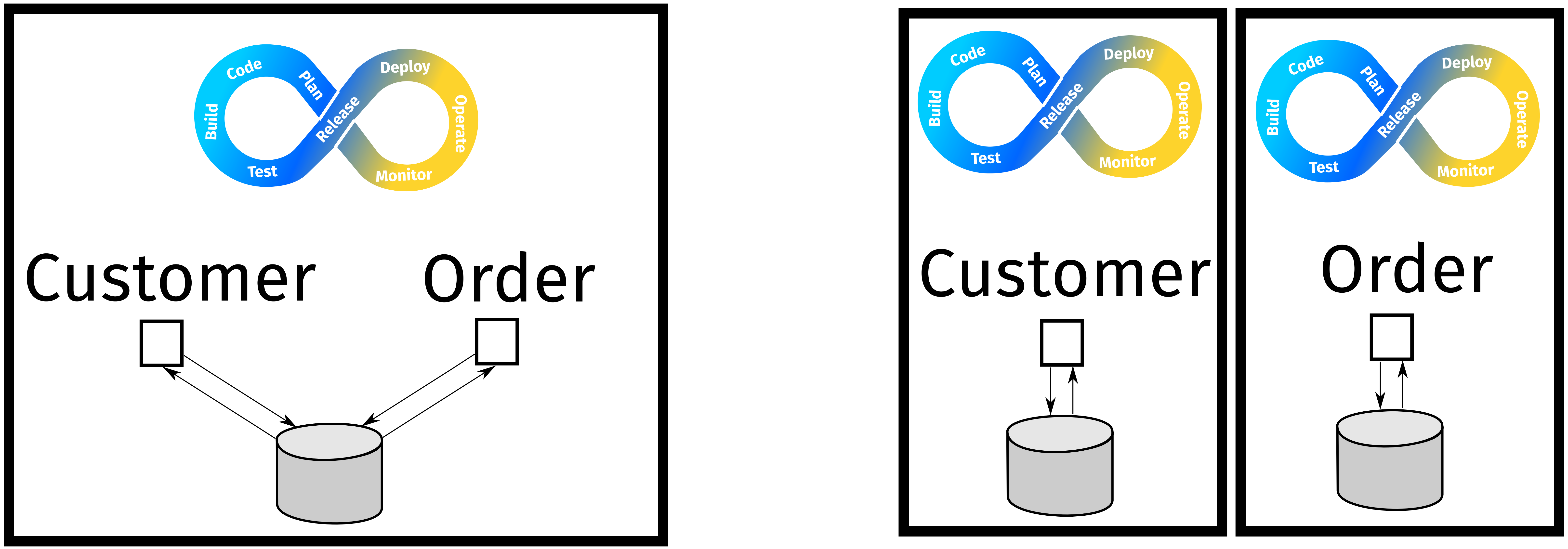


Isolated Microservices



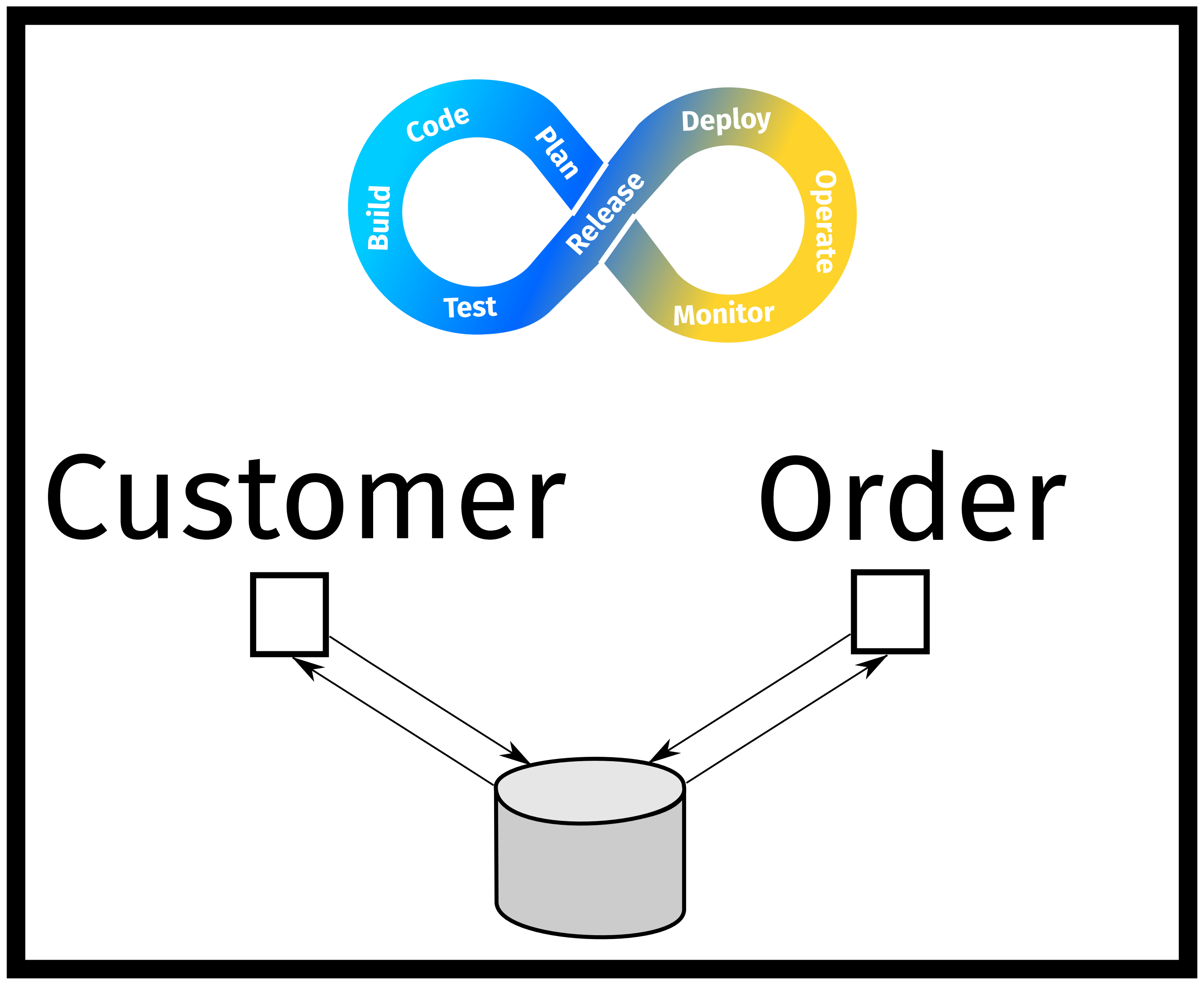
Monolith

Isolated Microservices

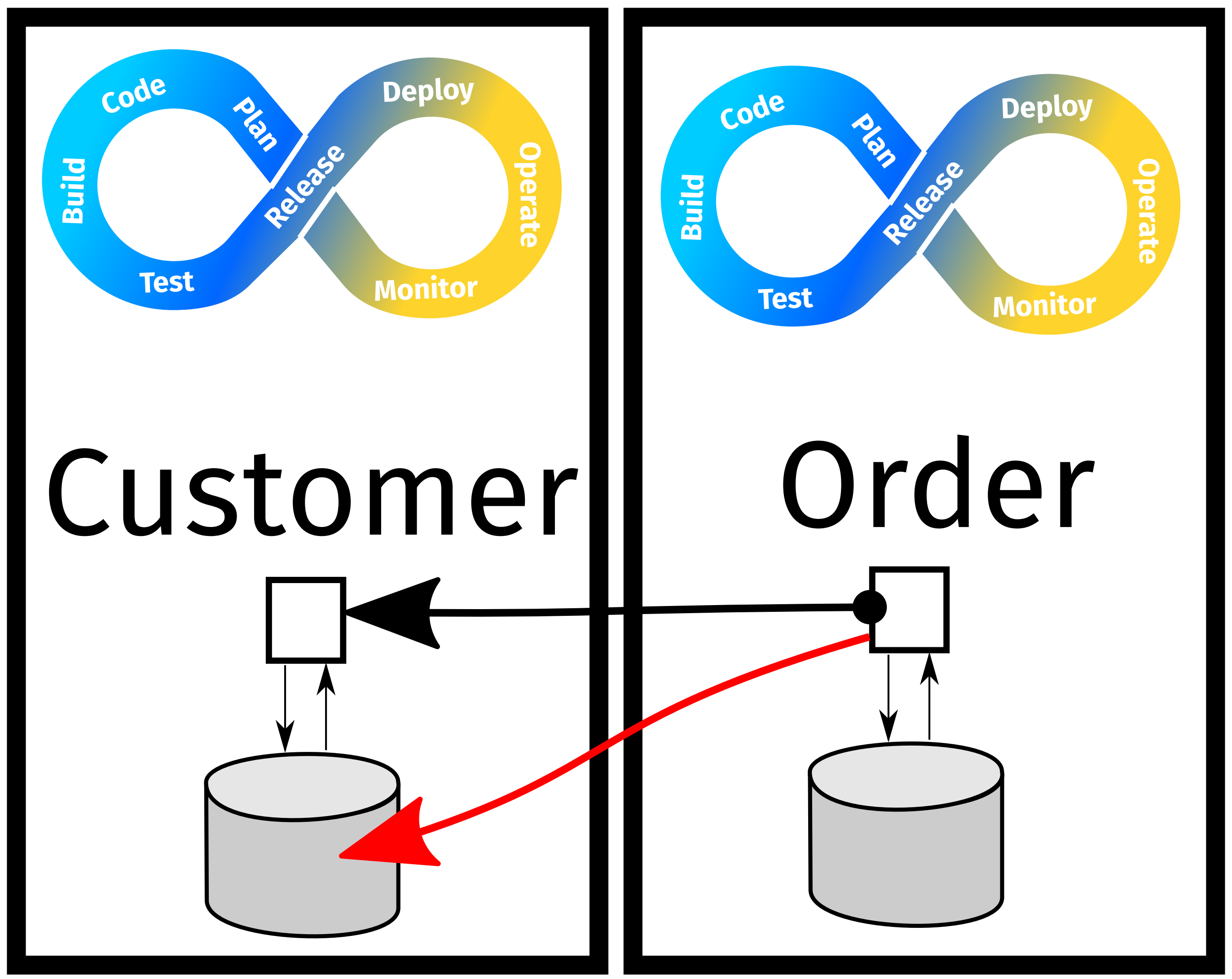


Monolith **Microservices**

Isolated Microservices



Monolith

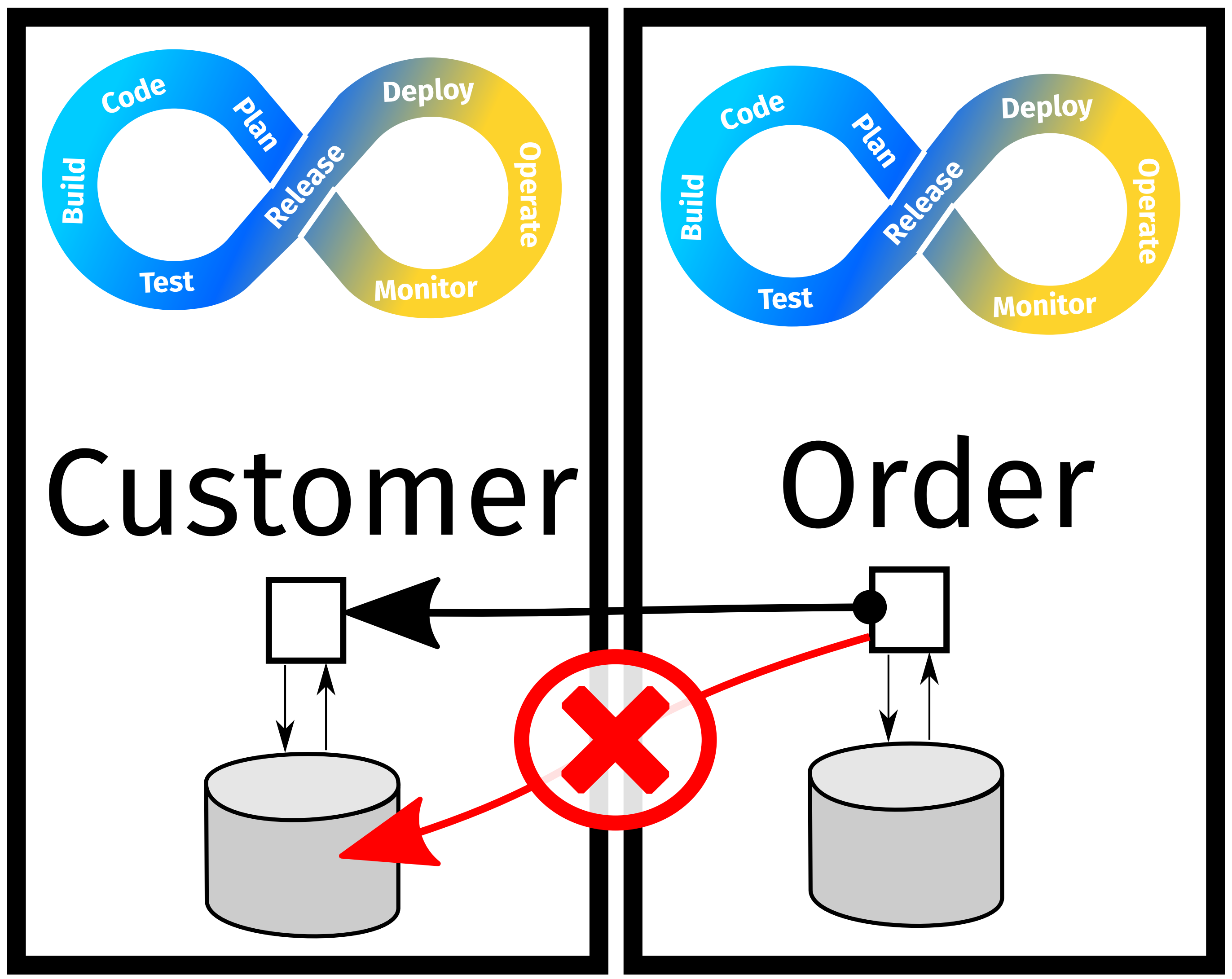


Microservices

Isolated Microservices

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



Microservices

Stateful 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

Stateful 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

Stateful Microservices

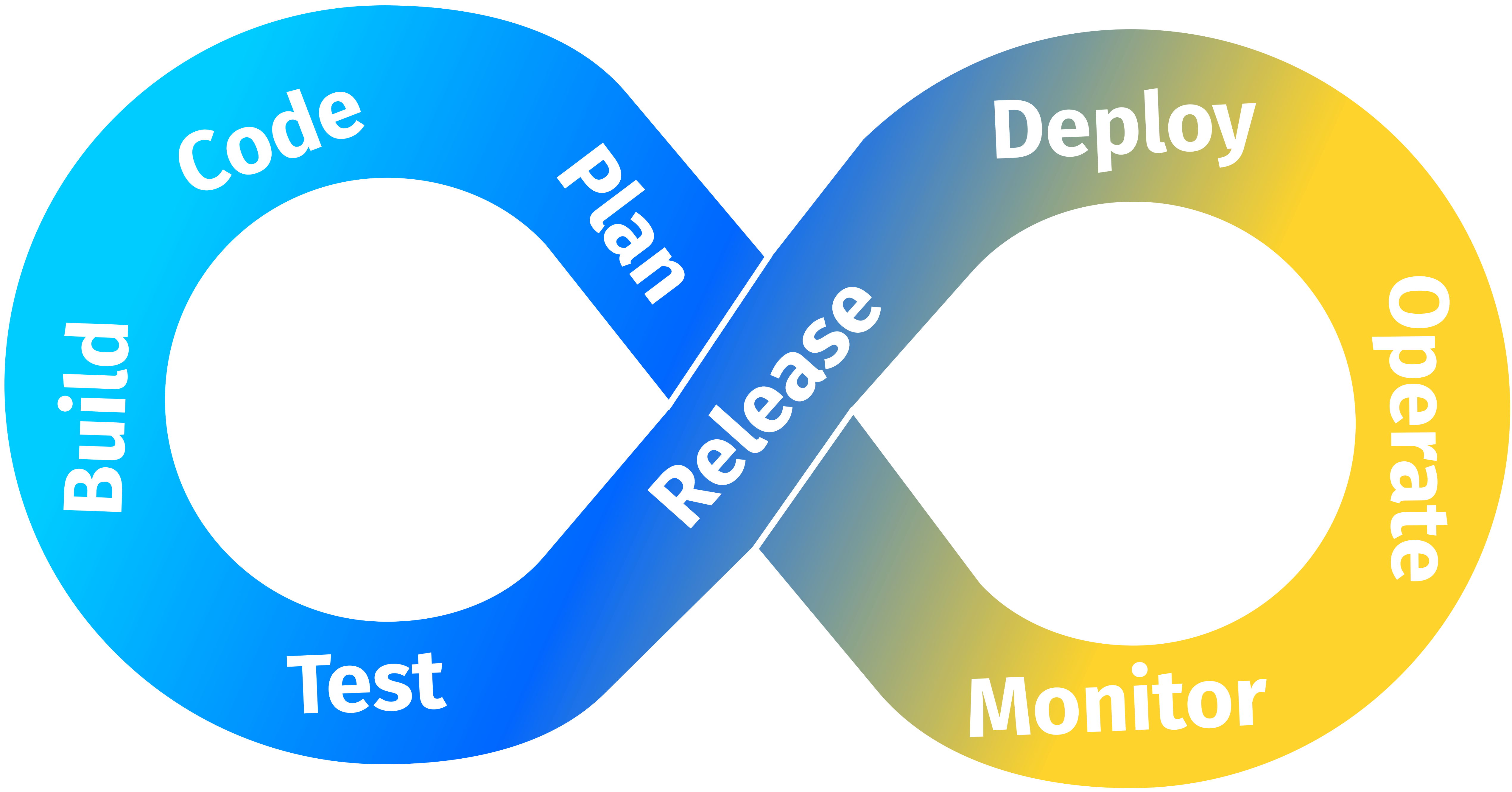
Microservices prefer letting **each service manage its own database**, either different instances of the same database technology, or entirely different database systems - an approach called **Polyglot Persistence**.

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

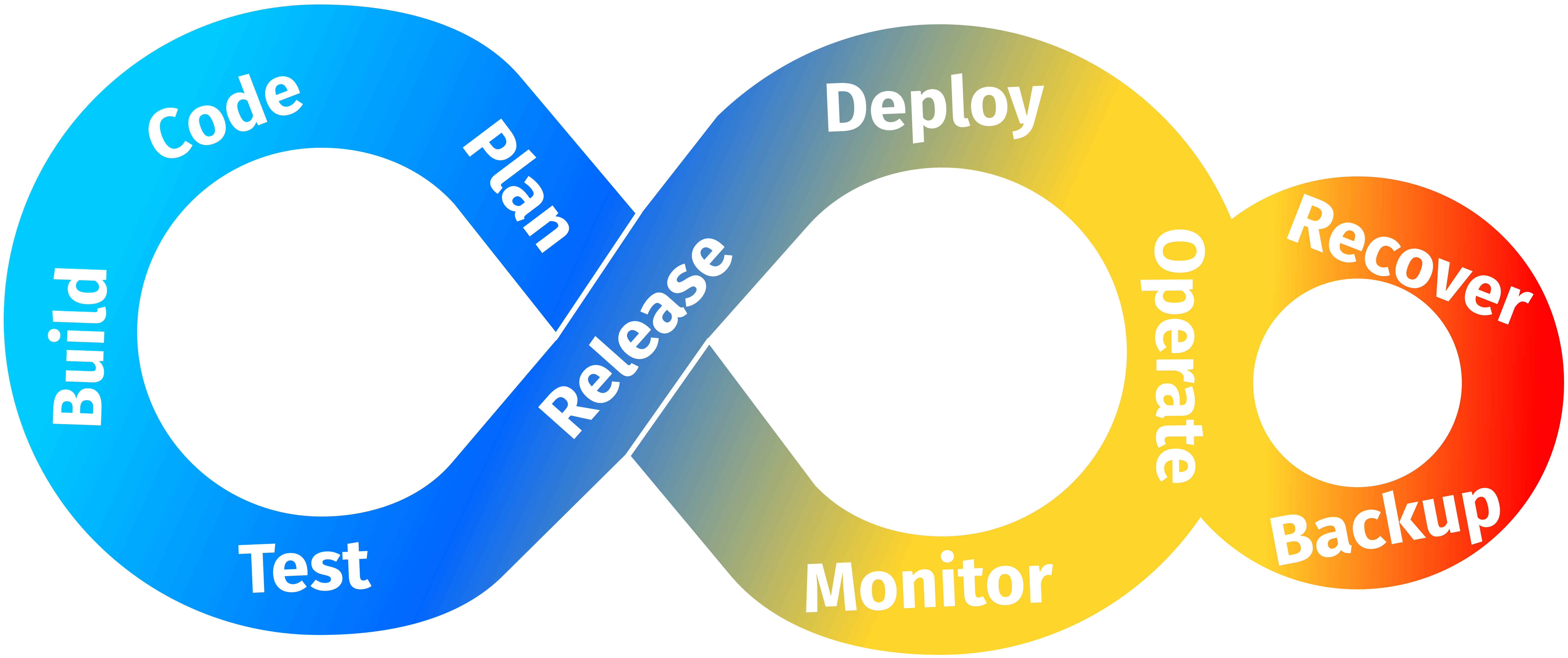
Eventual Inconsistency

Microservice architectures are doomed to become inconsistent after disaster strikes

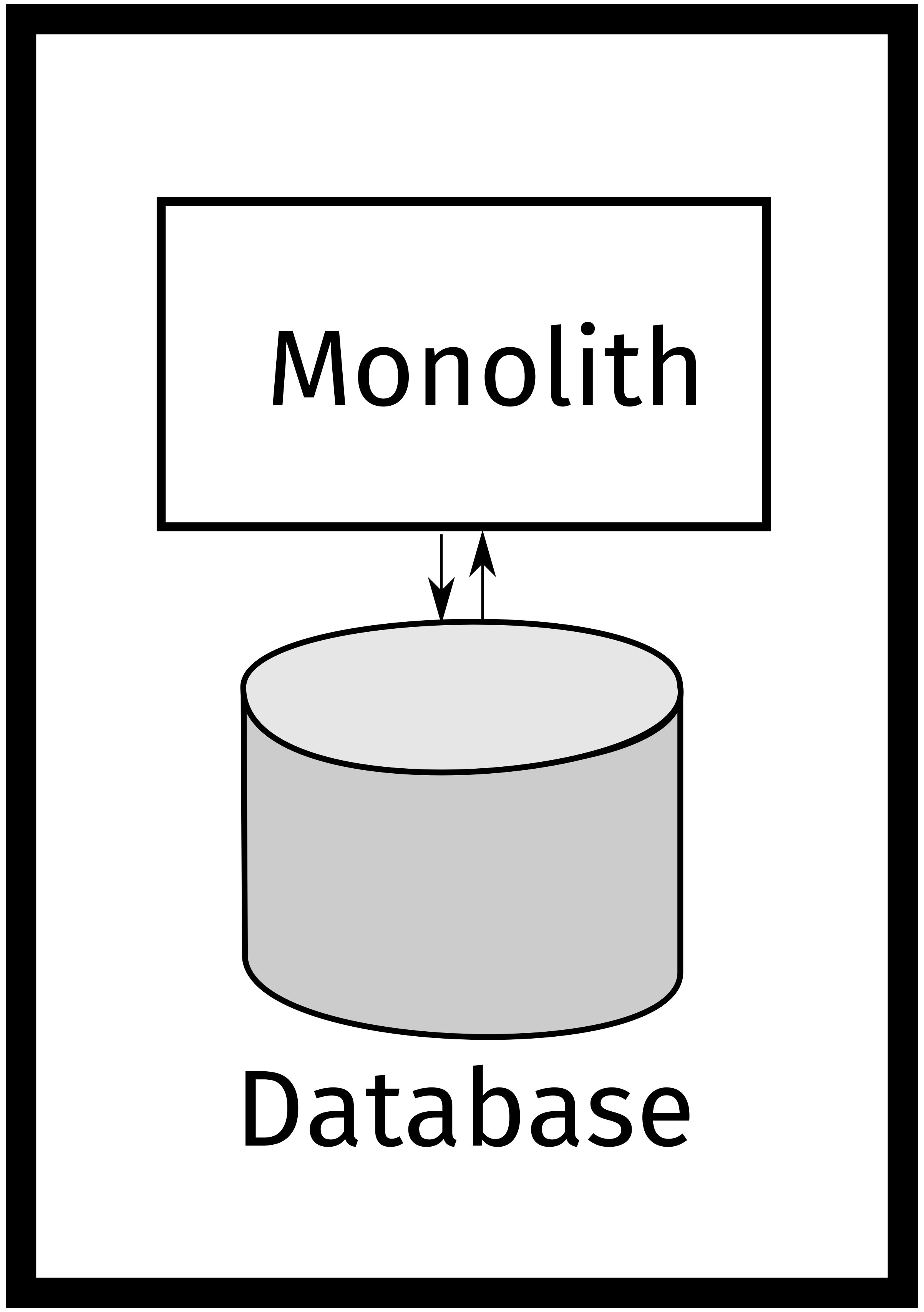
Devops meets Disaster Recovery



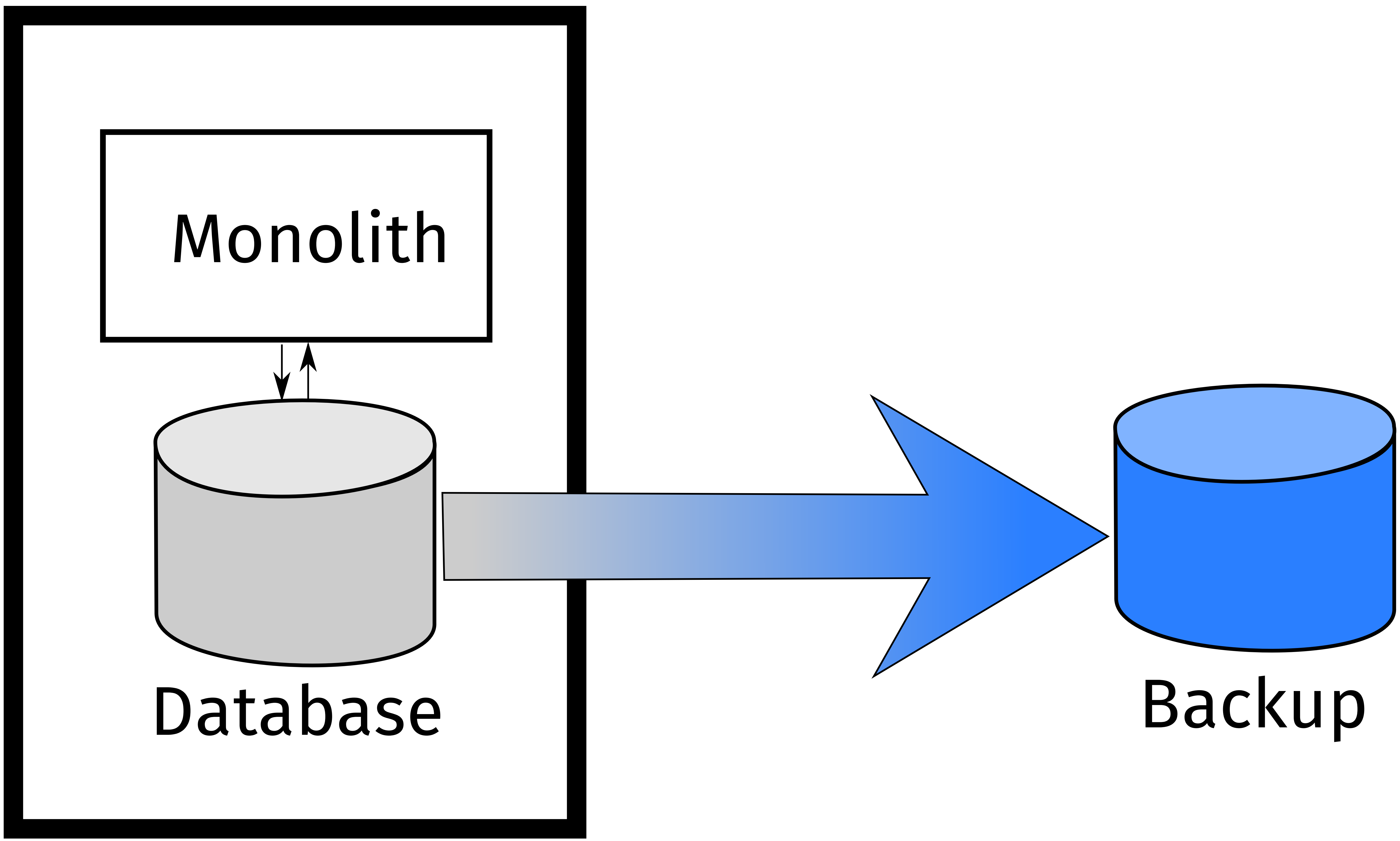
Devops meets Disaster Recovery



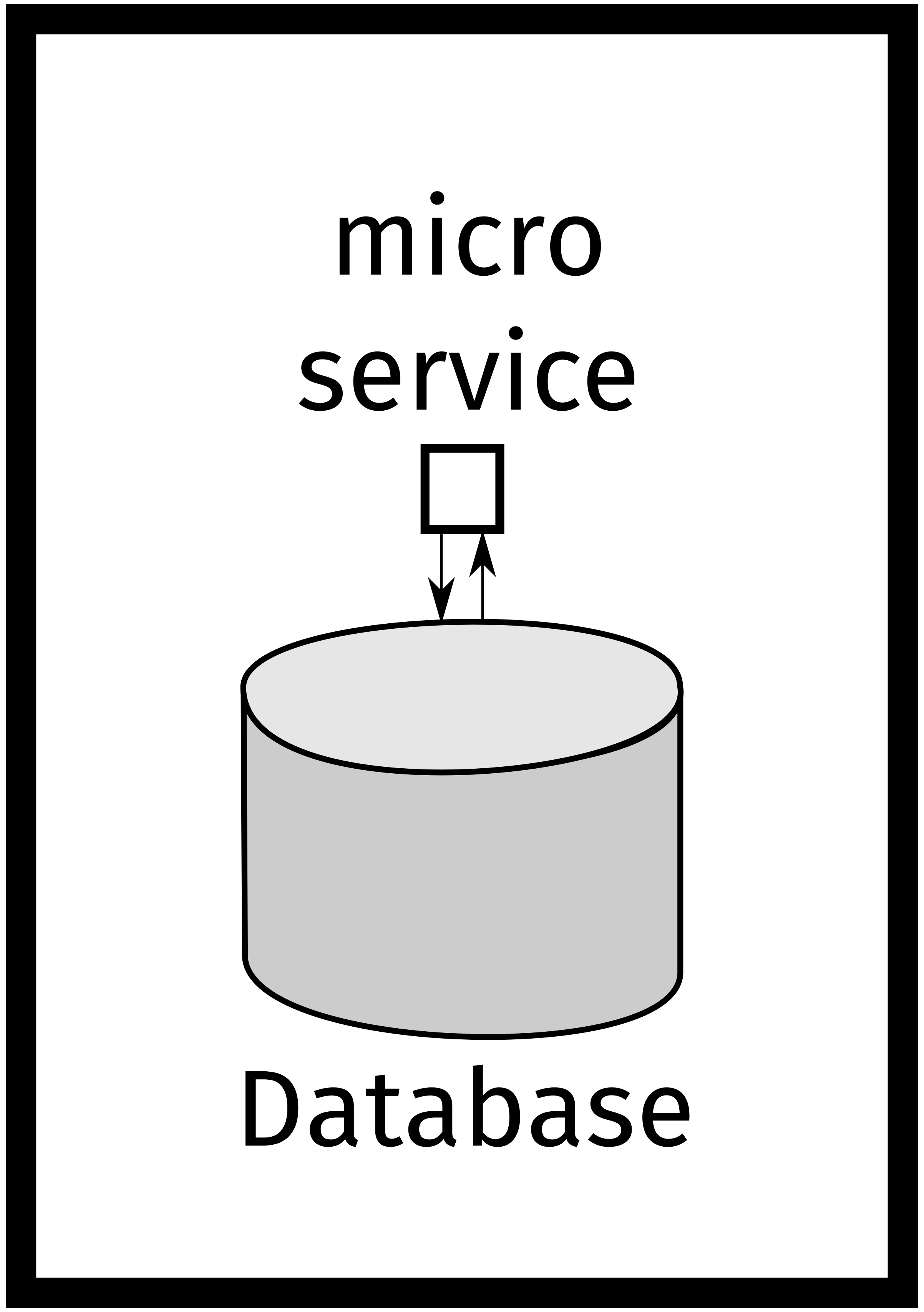
How do you back up a monolith?



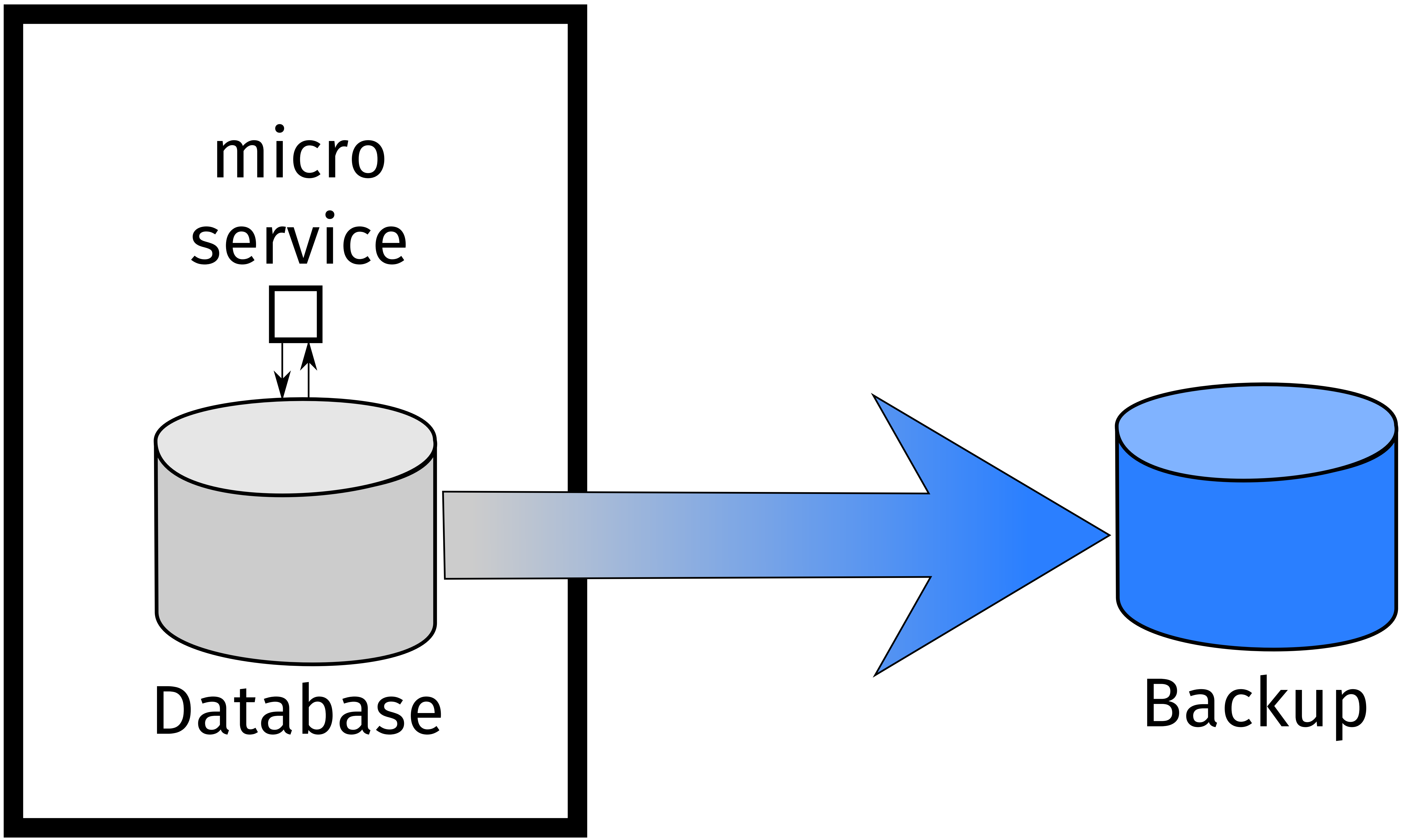
How do you back up a monolith?



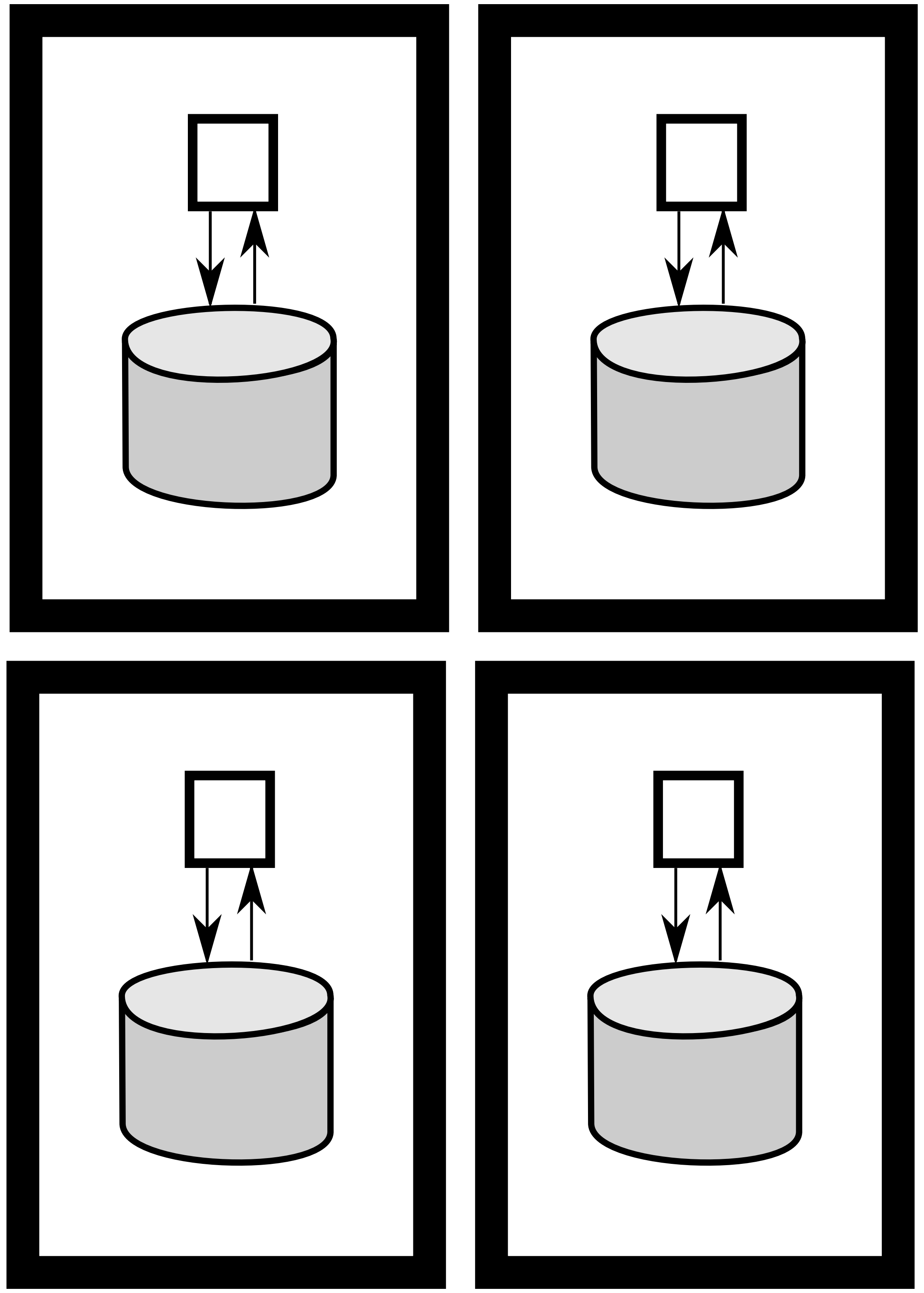
How do you back up one microservice?



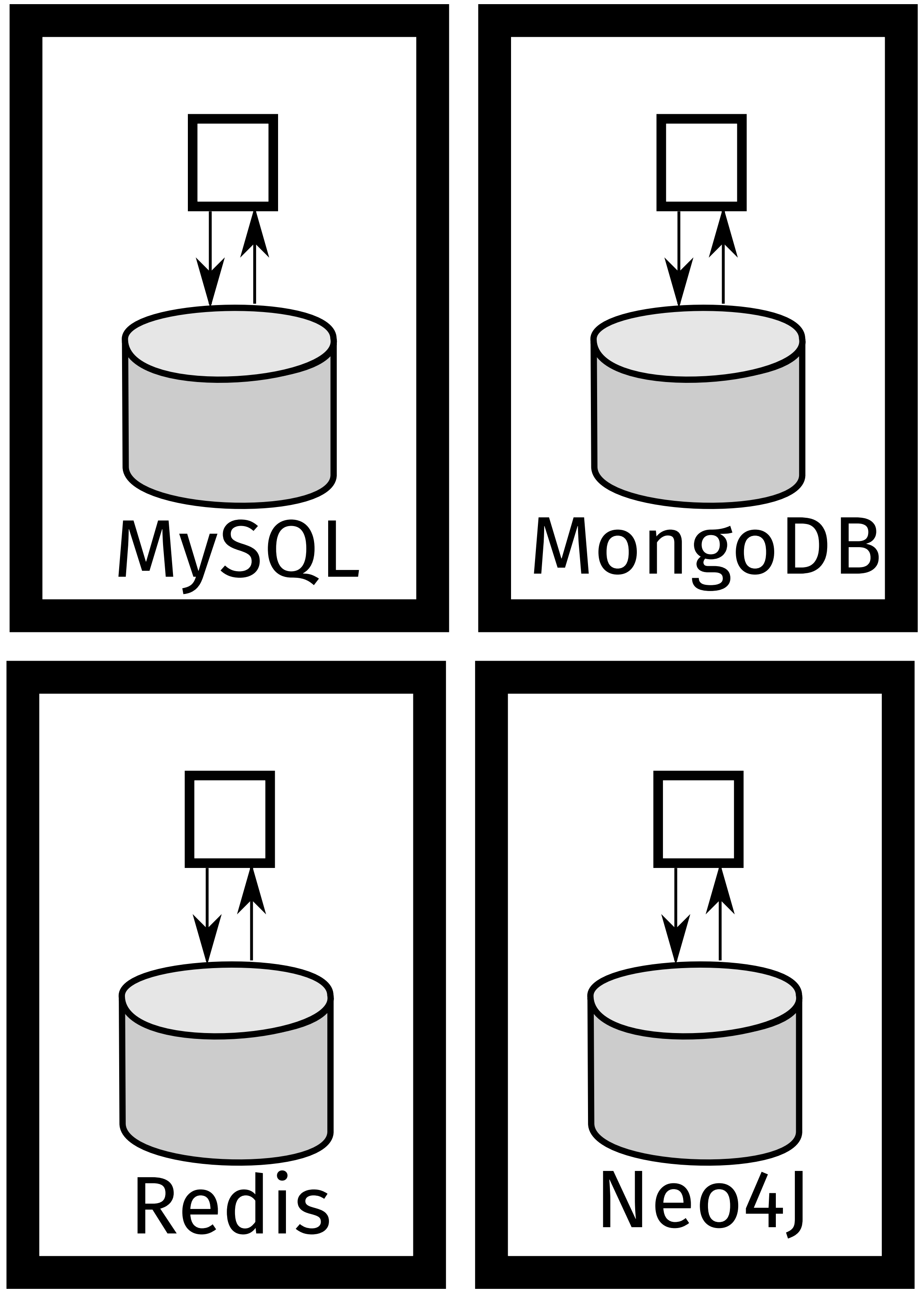
How do you back up one microservice?



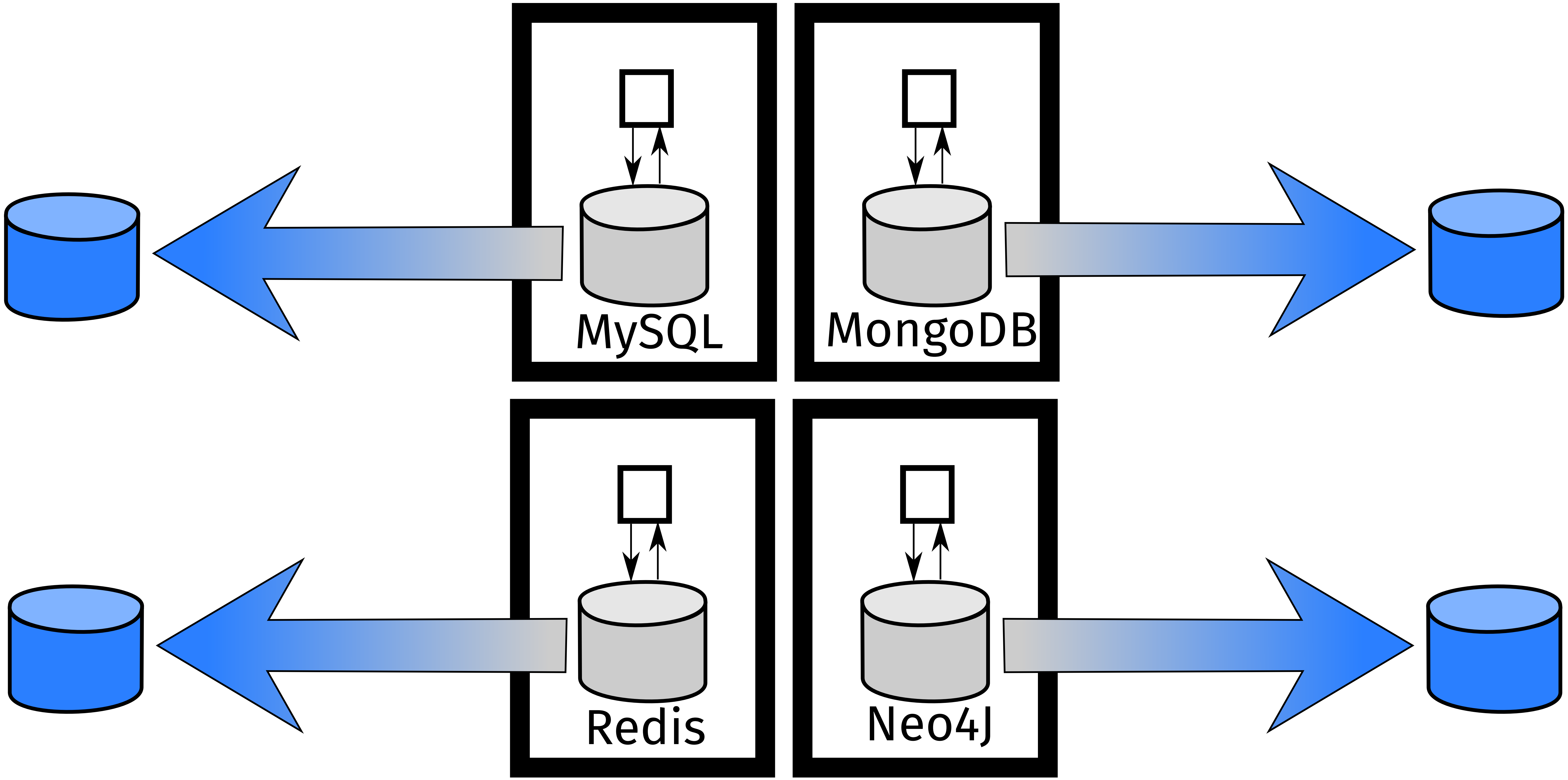
How do you back up an entire microservice architecture?



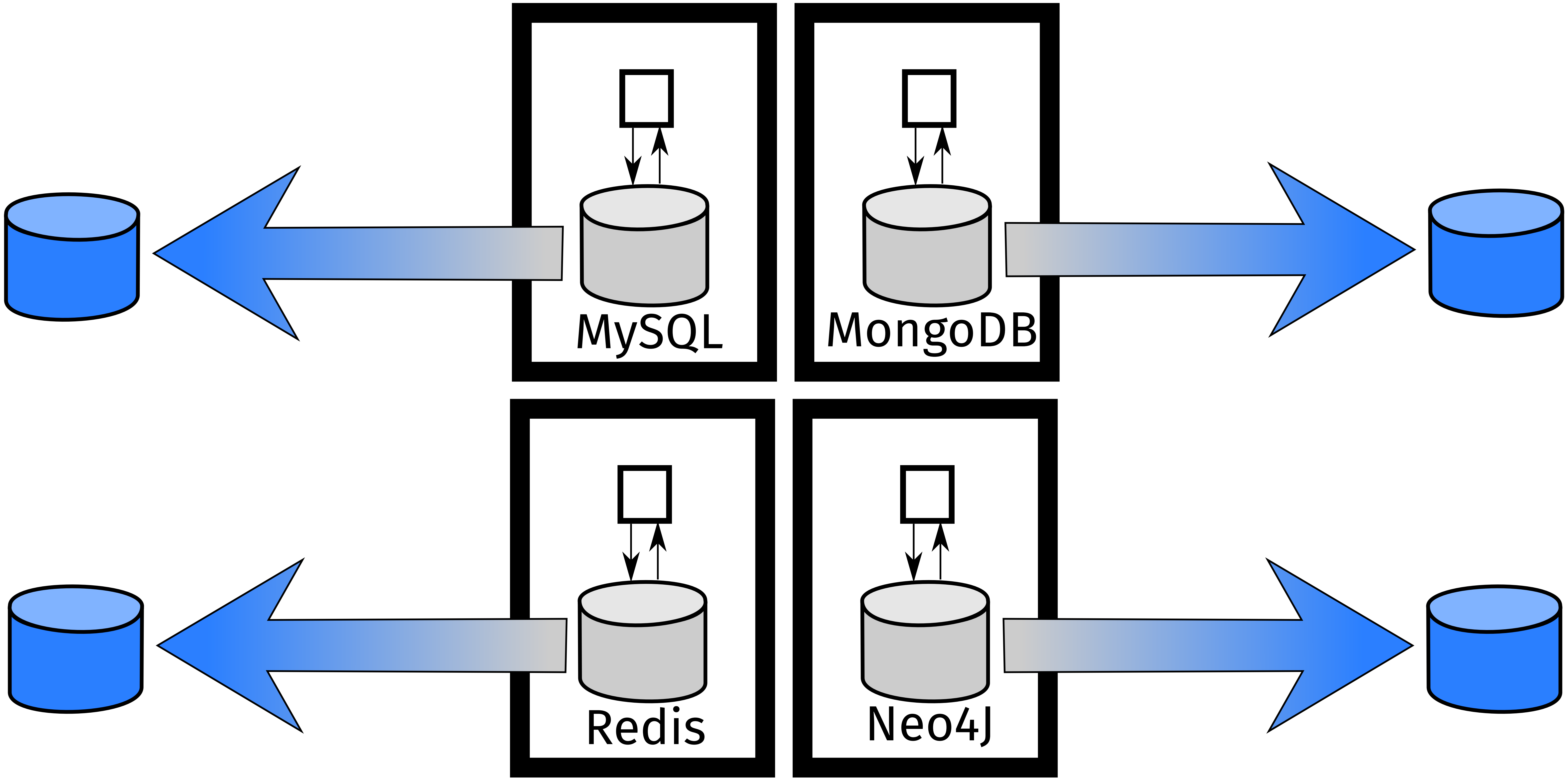
How do you back up an entire microservice architecture?



How do you back up an entire microservice architecture?

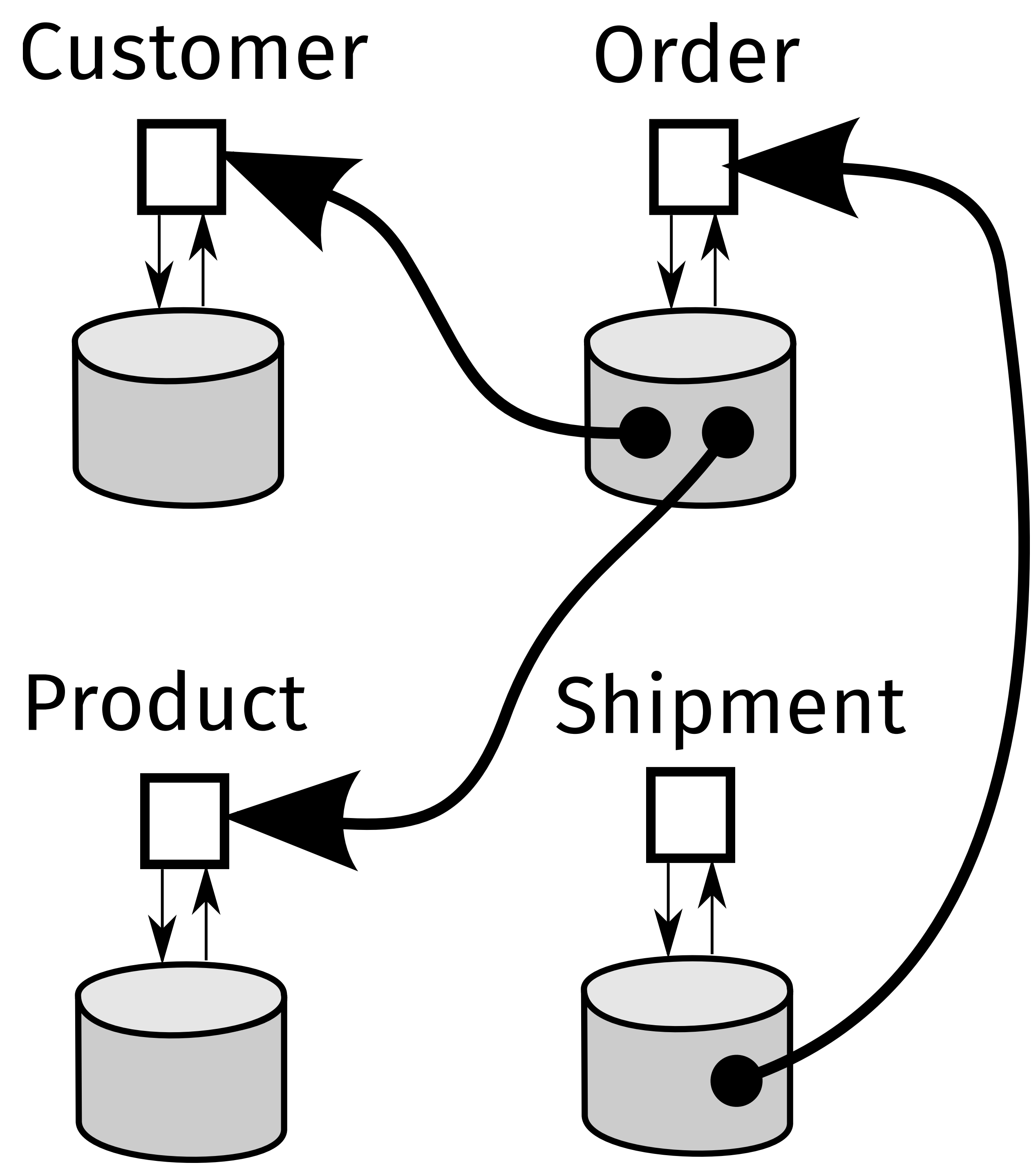


How do you back up an entire microservice architecture?

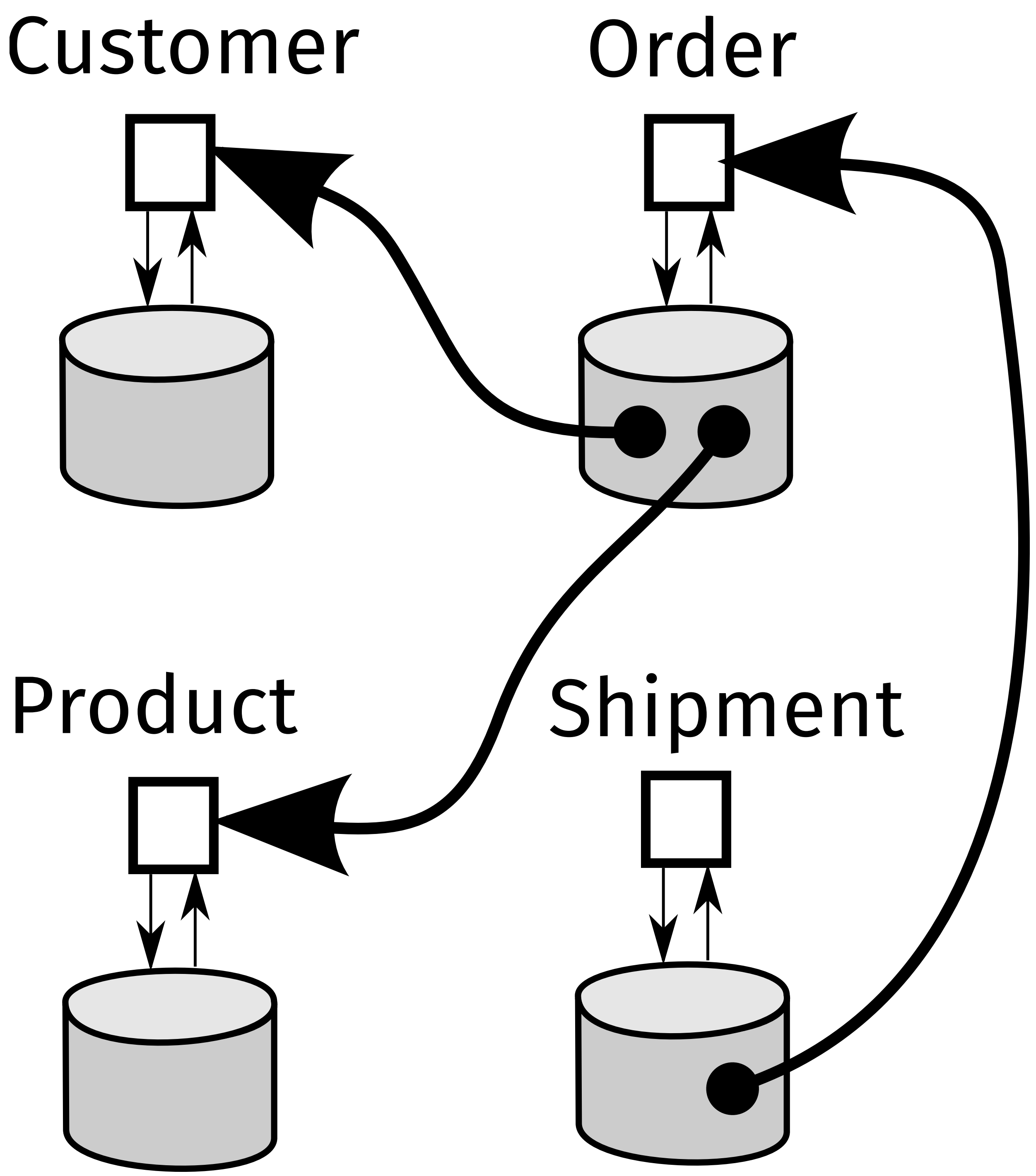


Are you sure?

Example

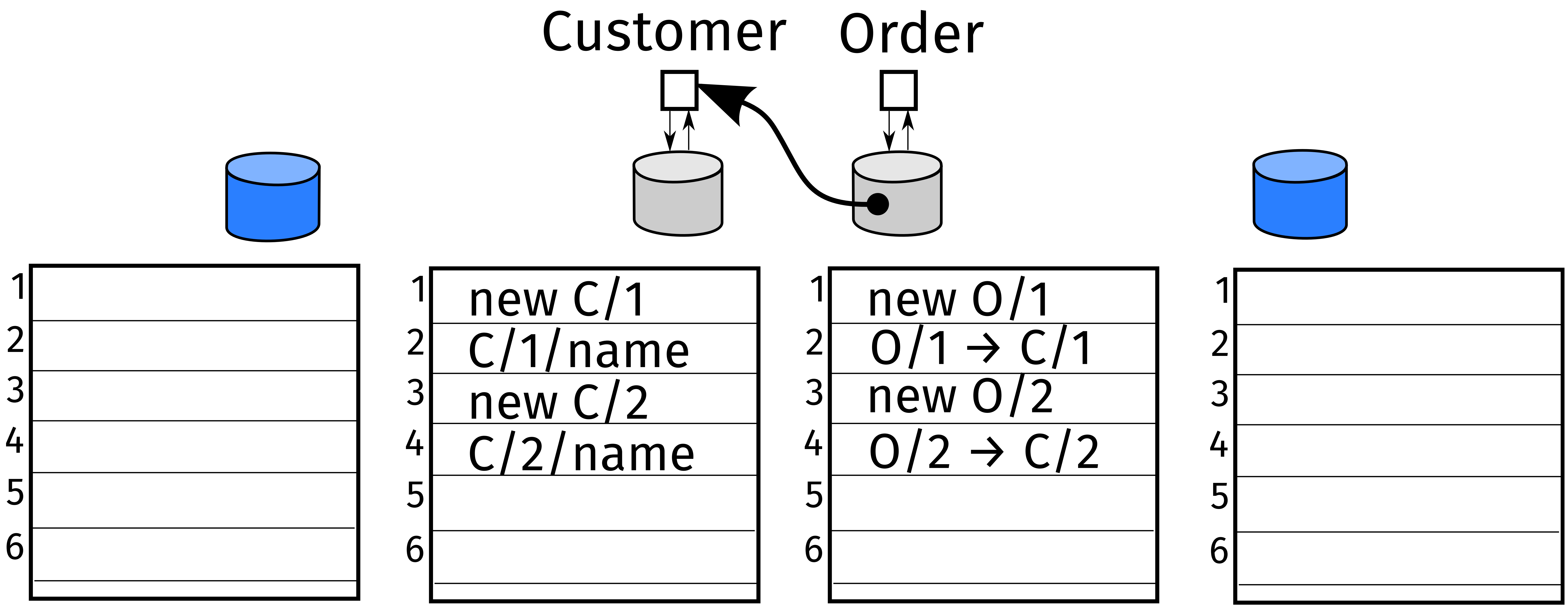


Example

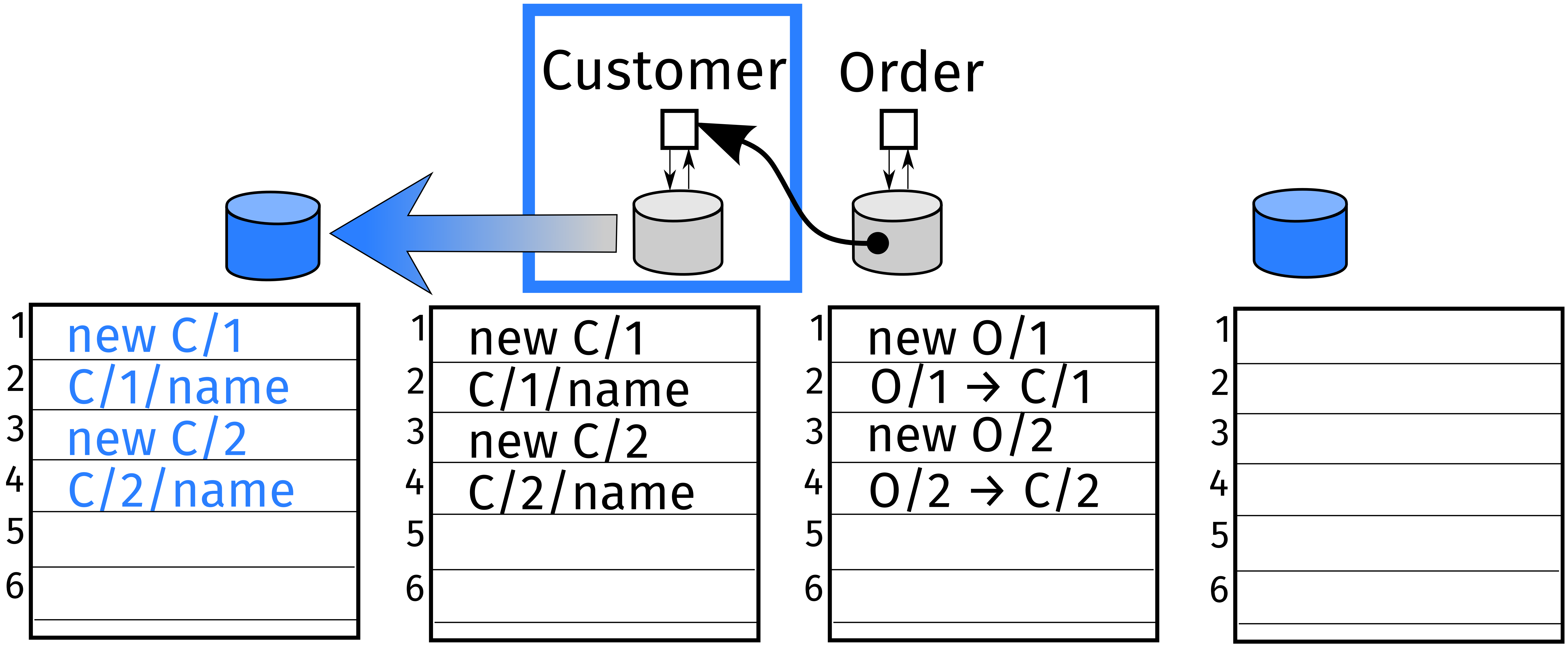


Data relationships across microservices = Hypermedia

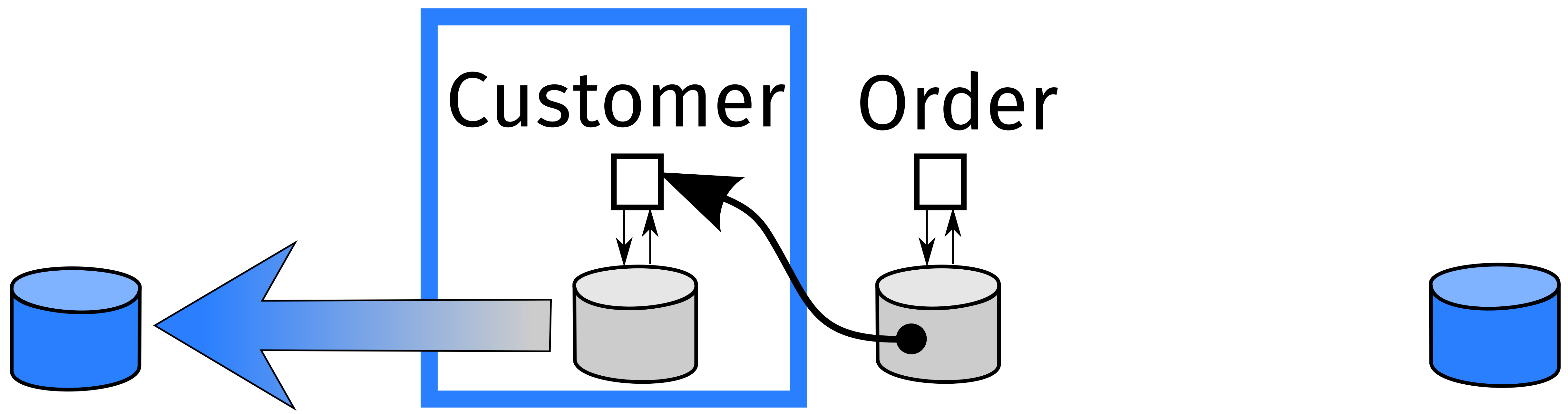
Independent Backup



Independent Backup



Independent Backup



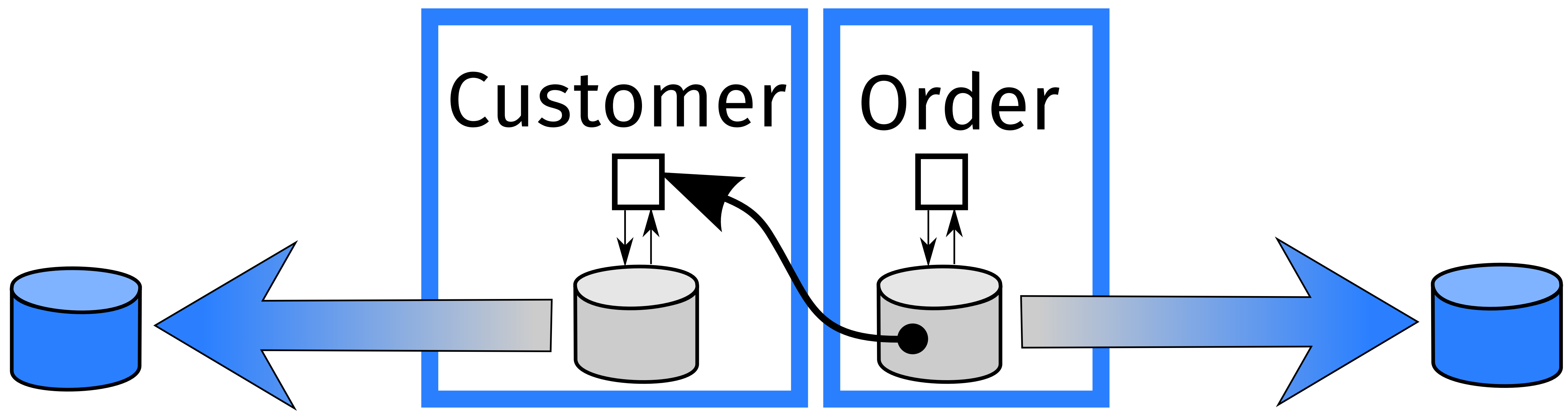
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	
6	

1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	new C/3
6	C/3/name

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	
2	
3	
4	
5	
6	

Independent Backup



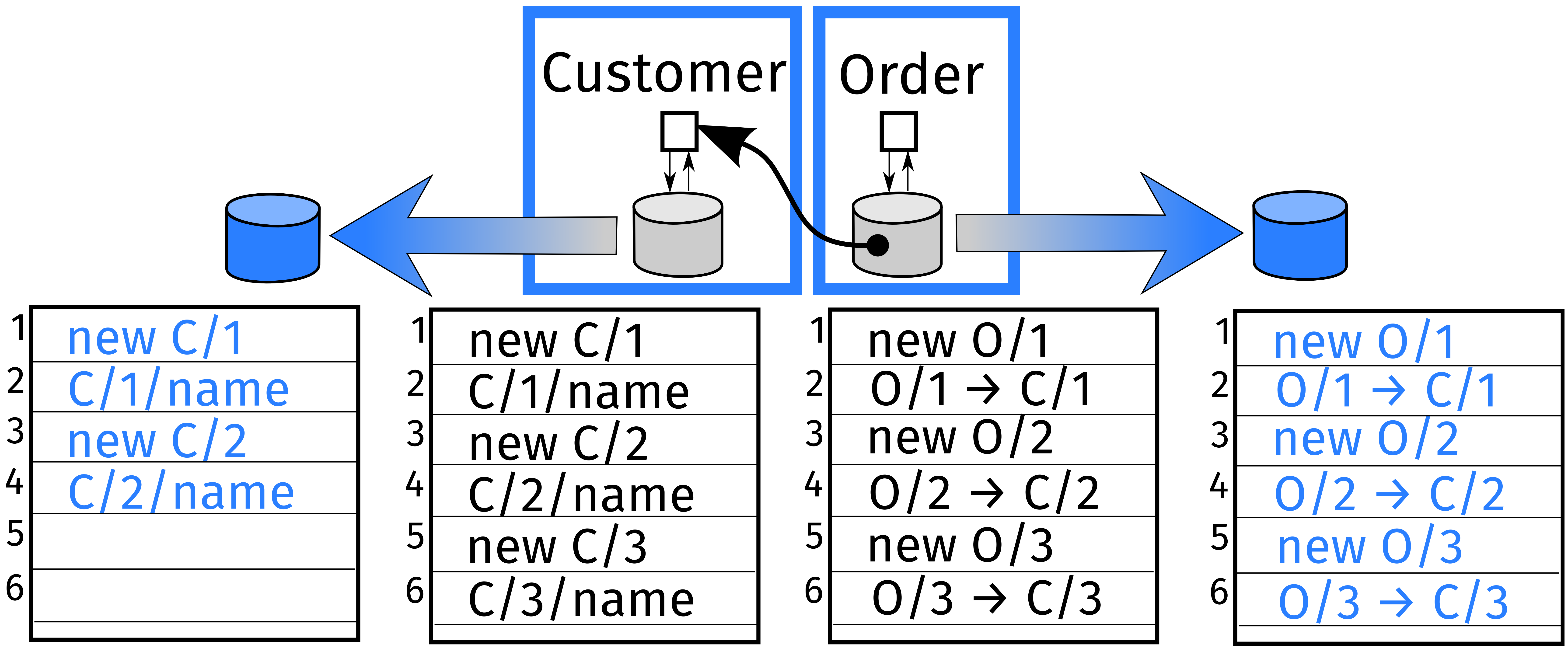
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	
6	

1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	new C/3
6	C/3/name

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

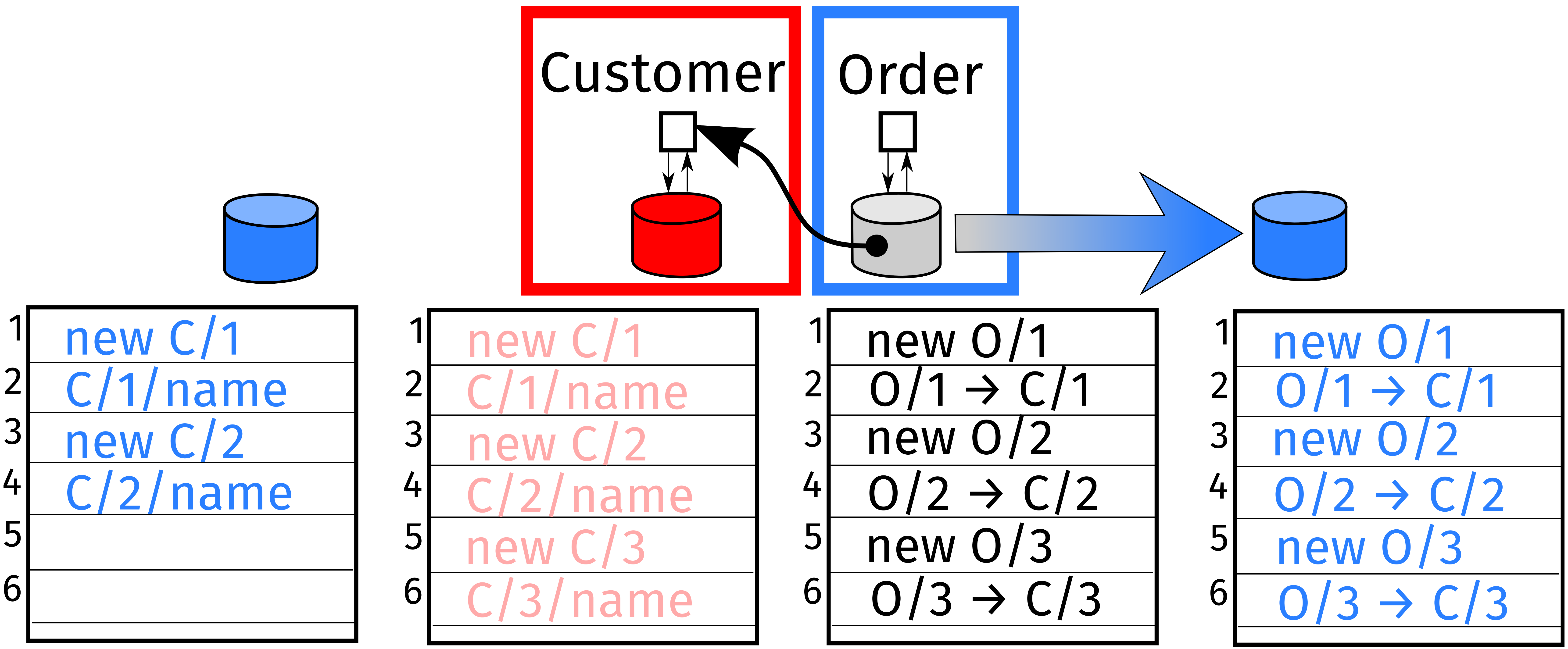
Independent Backup



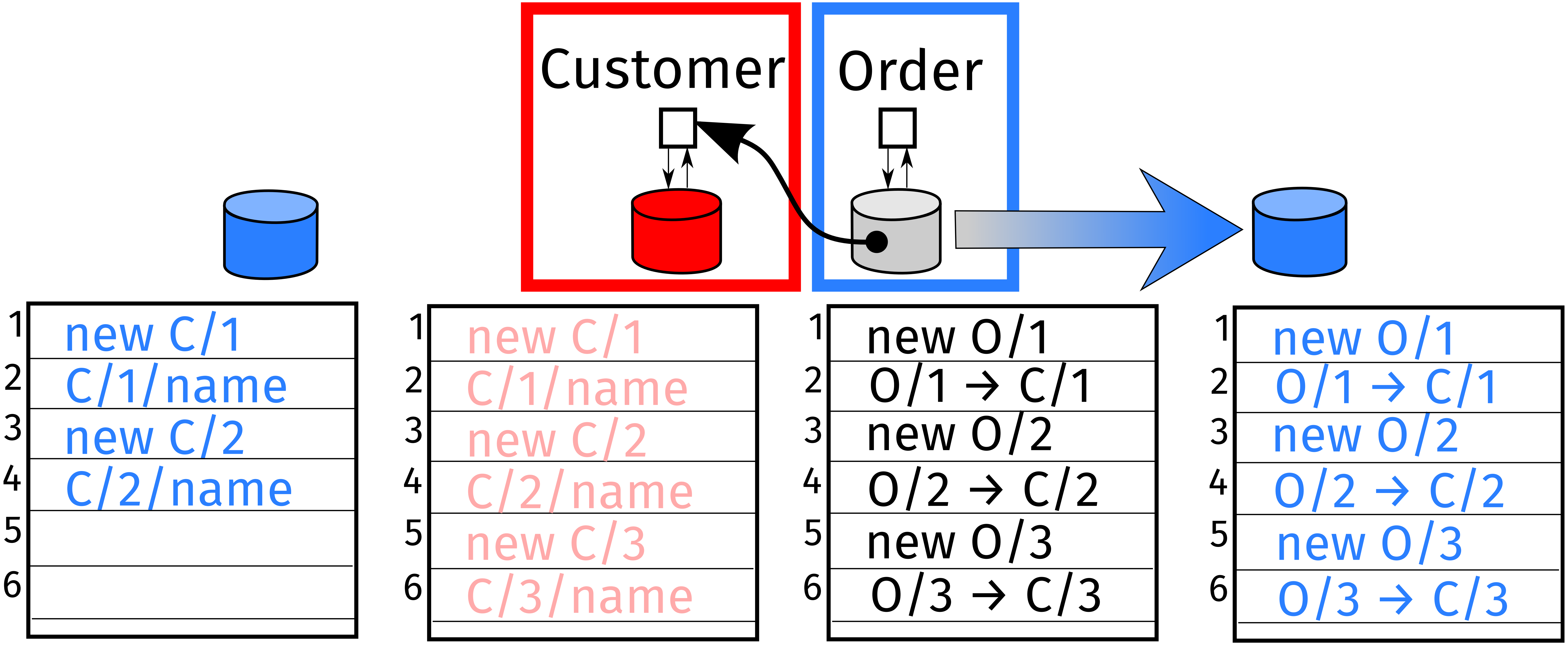
Backups taken independently at different times


Disaster Strikes
CP306
GUARDIA COSTIERA

Disaster Strikes

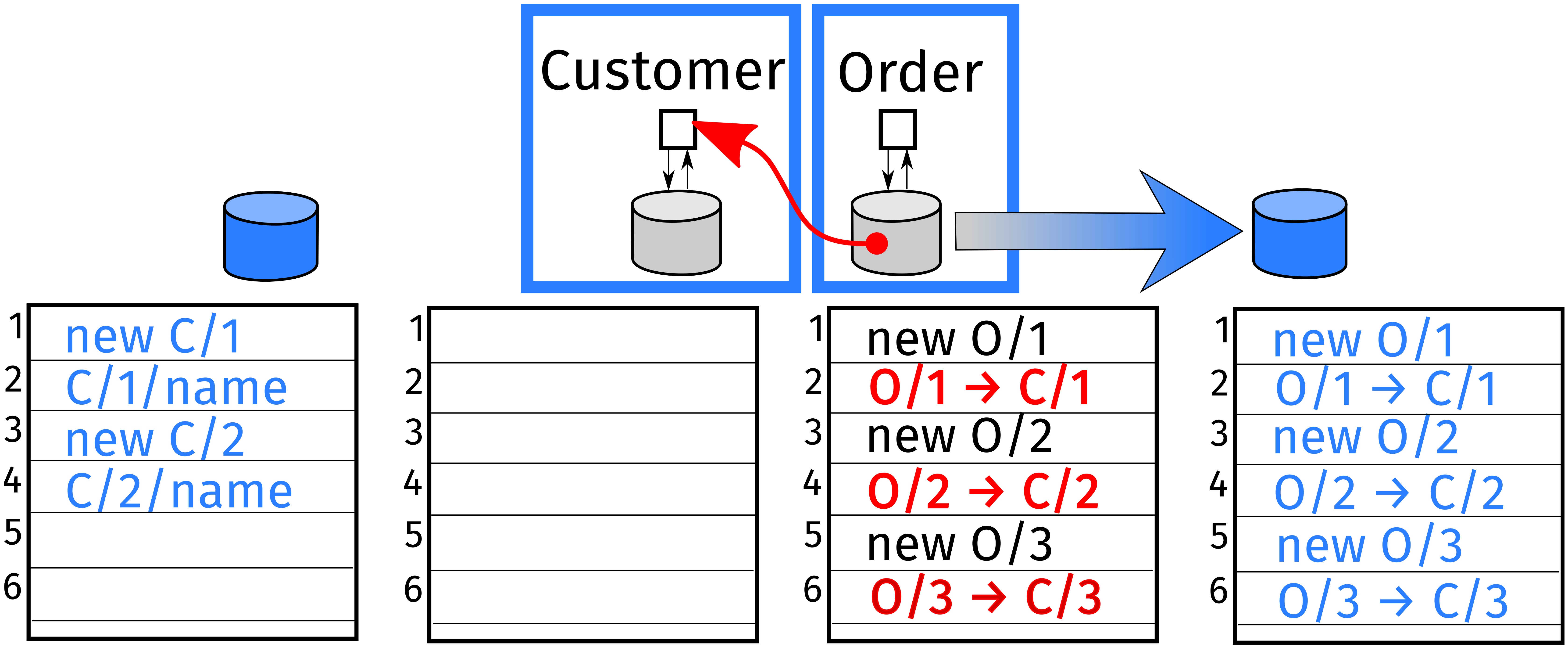


Disaster Strikes

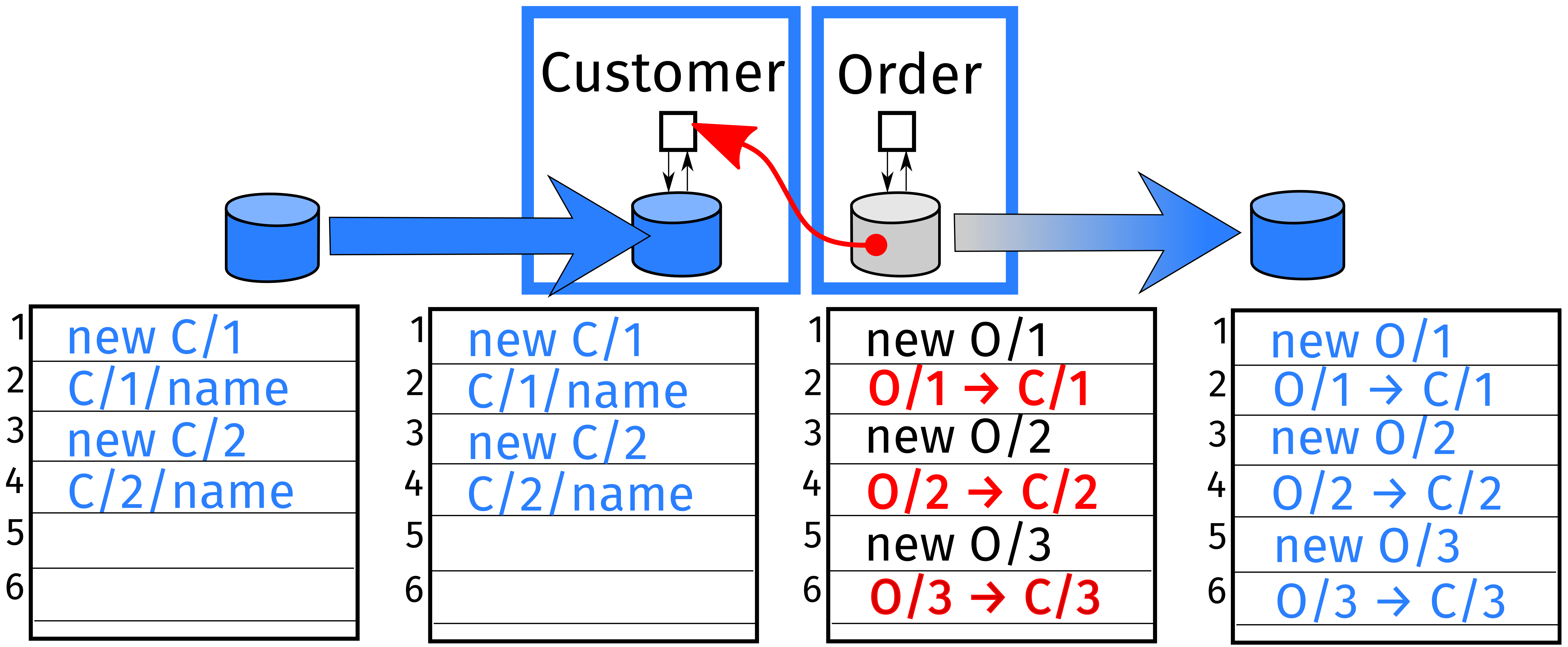


One microservice is lost

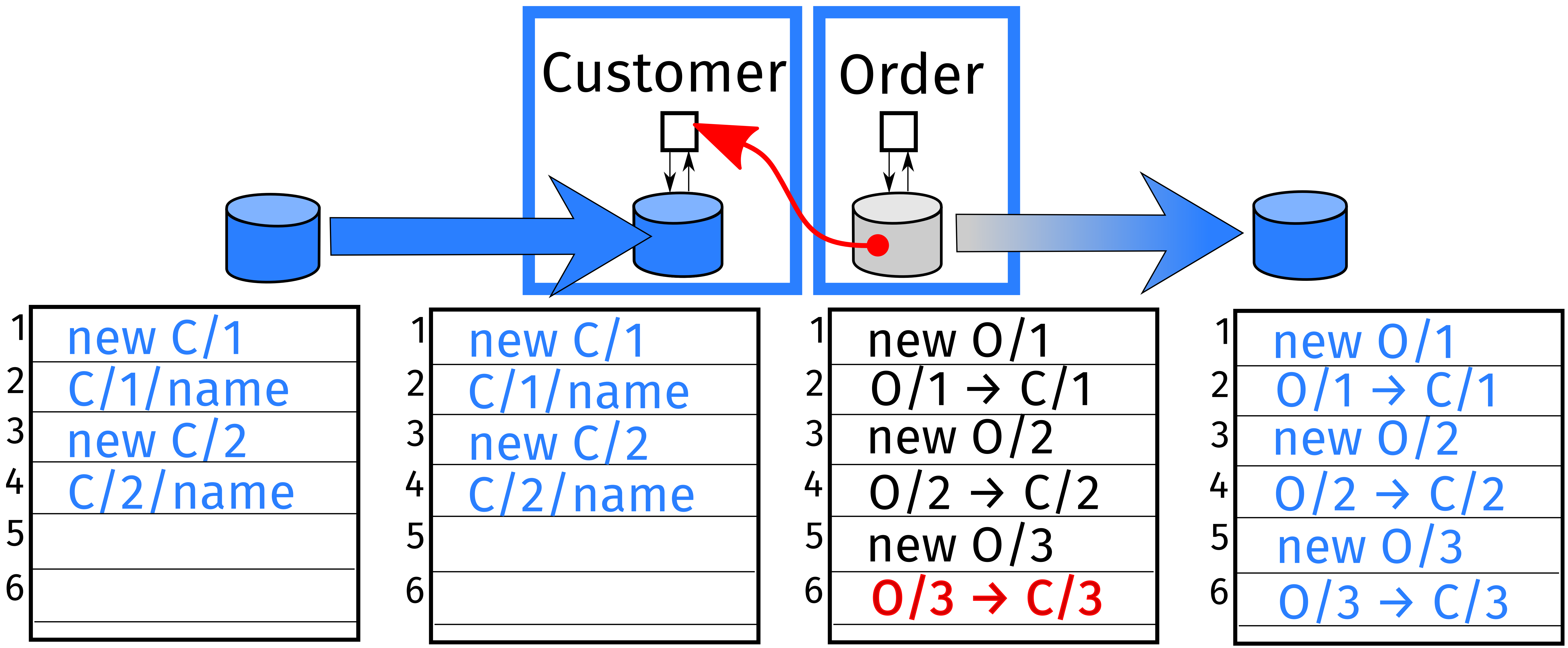
Recovery from Backup



Recovery from Backup

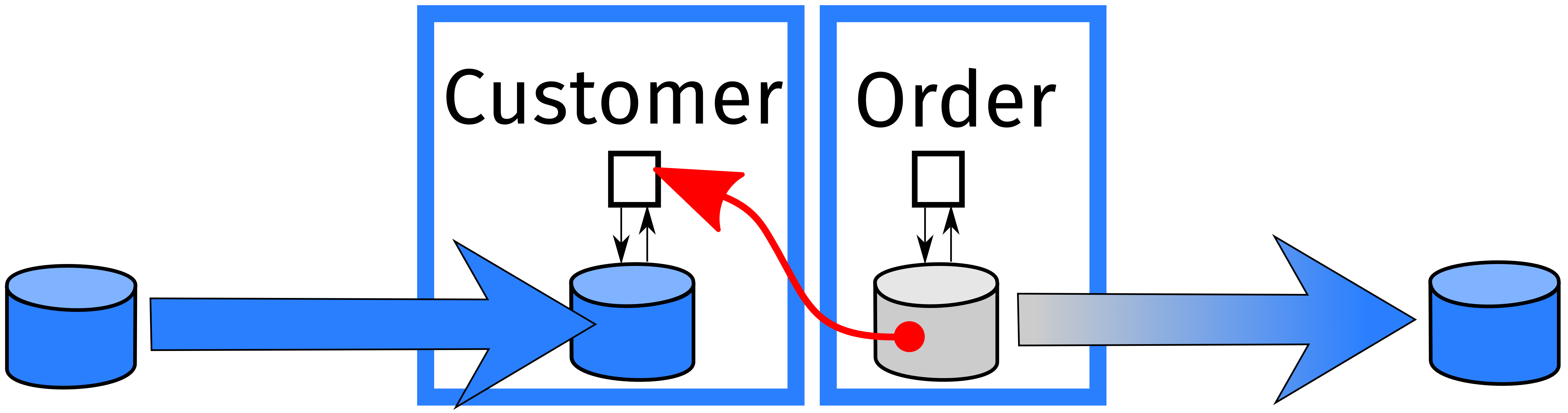


Recovery from Backup



Broken link after recovery

Recovery from Backup

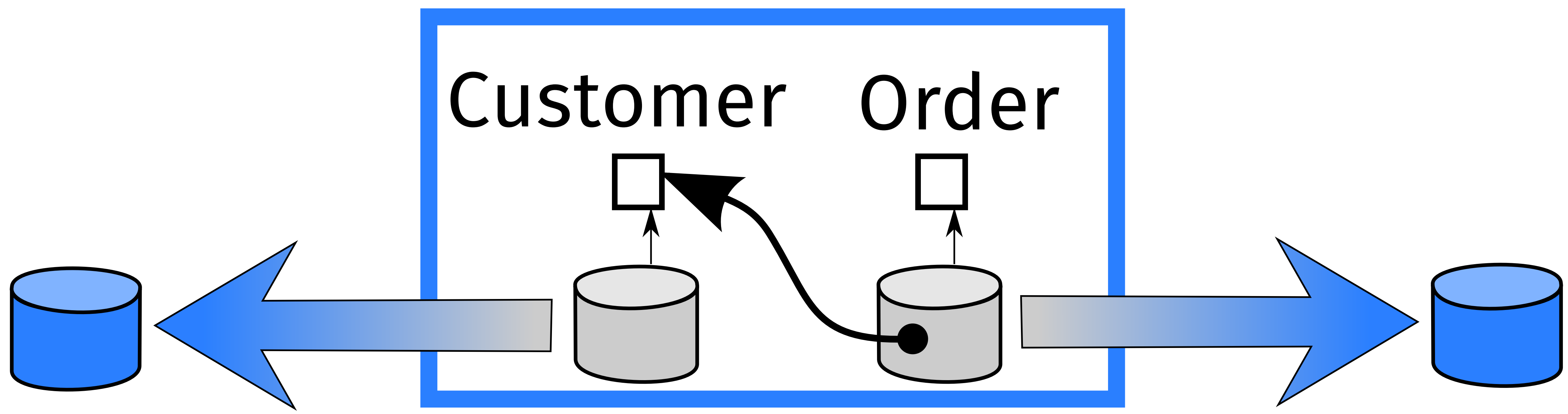


1	new C/1	1	new C/1	1	new O/1	1	new O/1
2	C/1/name	2	C/1/name	2	O/1 → C/1	2	O/1 → C/1
3	new C/2	3	new C/2	3	new O/2	3	new O/2
4	C/2/name	4	C/2/name	4	O/2 → C/2	4	O/2 → C/2
5		5		5	new O/3	5	new O/3
6		6		6	O/3 → C/3	6	O/3 → C/3

Broken link after recovery

Eventual Inconsistency

Synchronized Backups



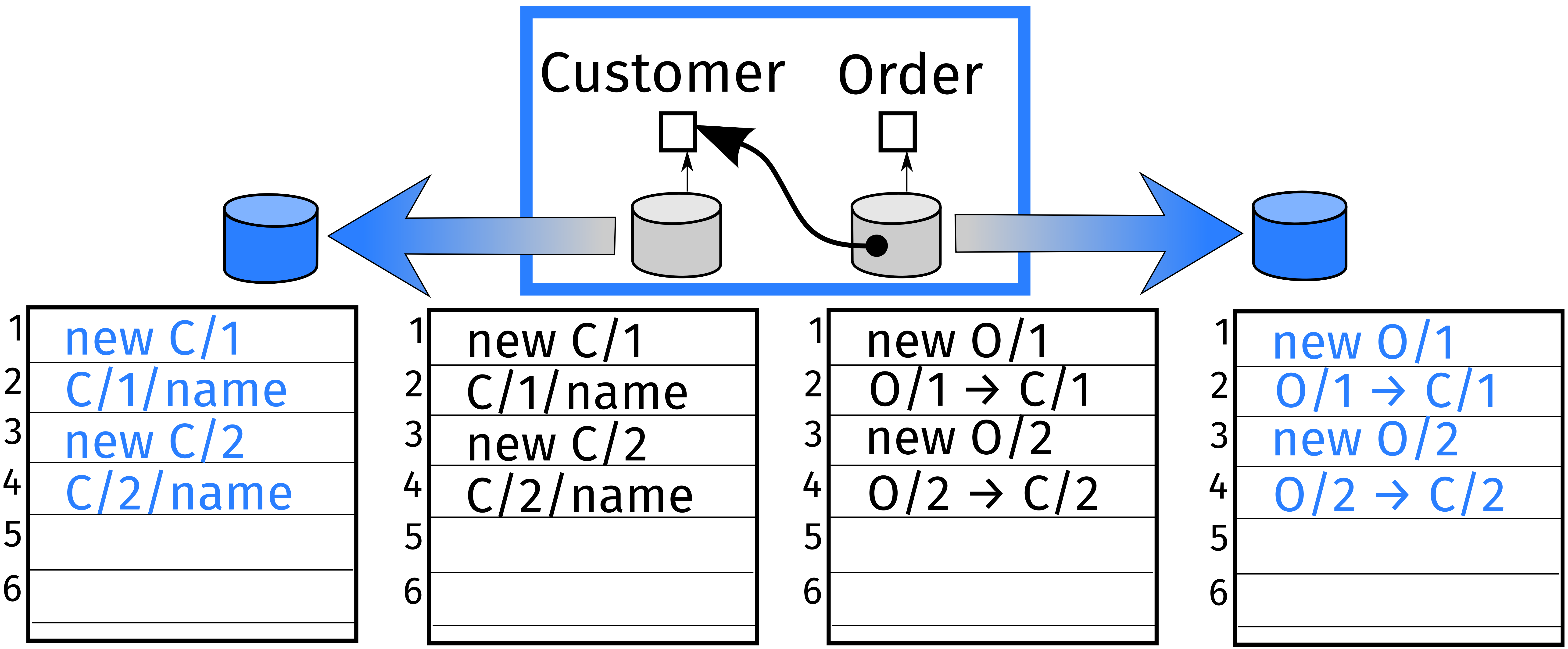
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	
6	

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	
6	

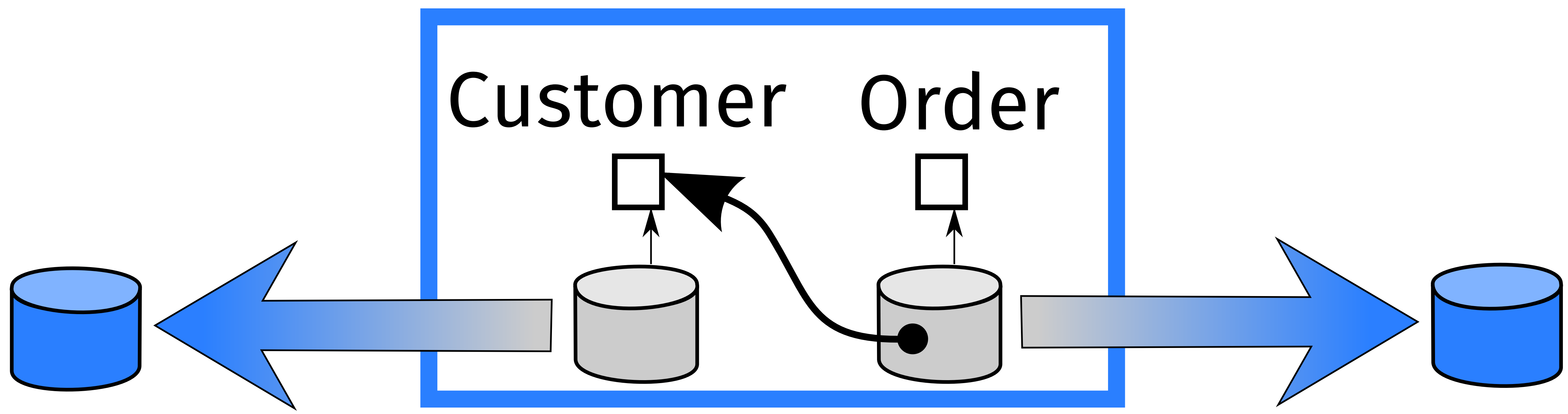
1	new O/1
2	O/1 → C/1
3	new O/2
4	O/2 → C/2
5	
6	

Synchronized Backups



Backups of all microservices taken at the same time.

Synchronized Backups



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

Backups of all microservices taken at the same time.

Limited Autonomy

powered by ASQ

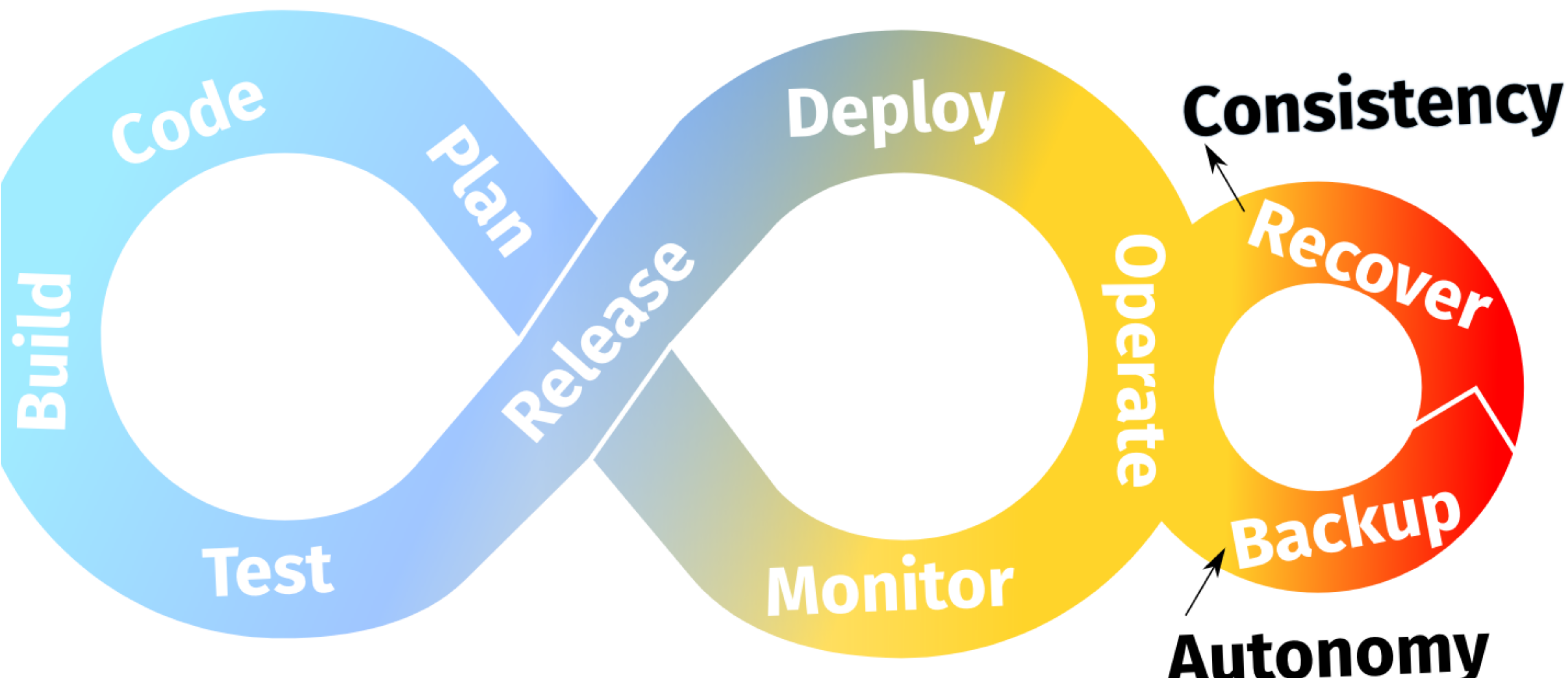
The BAC theorem

The BAC theorem

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

The BAC theorem

When **B**acking up a microservice architecture, it is not possible to have both **C**onsistency and **A**utonomy



Consistency

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

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

Autonomy

Each microservices has an **independent** DevOps lifecycle

Backup autonomy: snapshots taken at different times without any coordination across multiple microservices

Backup

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

Backup

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

No.

Backup + Autonomy

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:

- agree among all microservices on when to perform the backup (limited autonomy)

Backup + Consistency

Taking a consistent backup requires to:

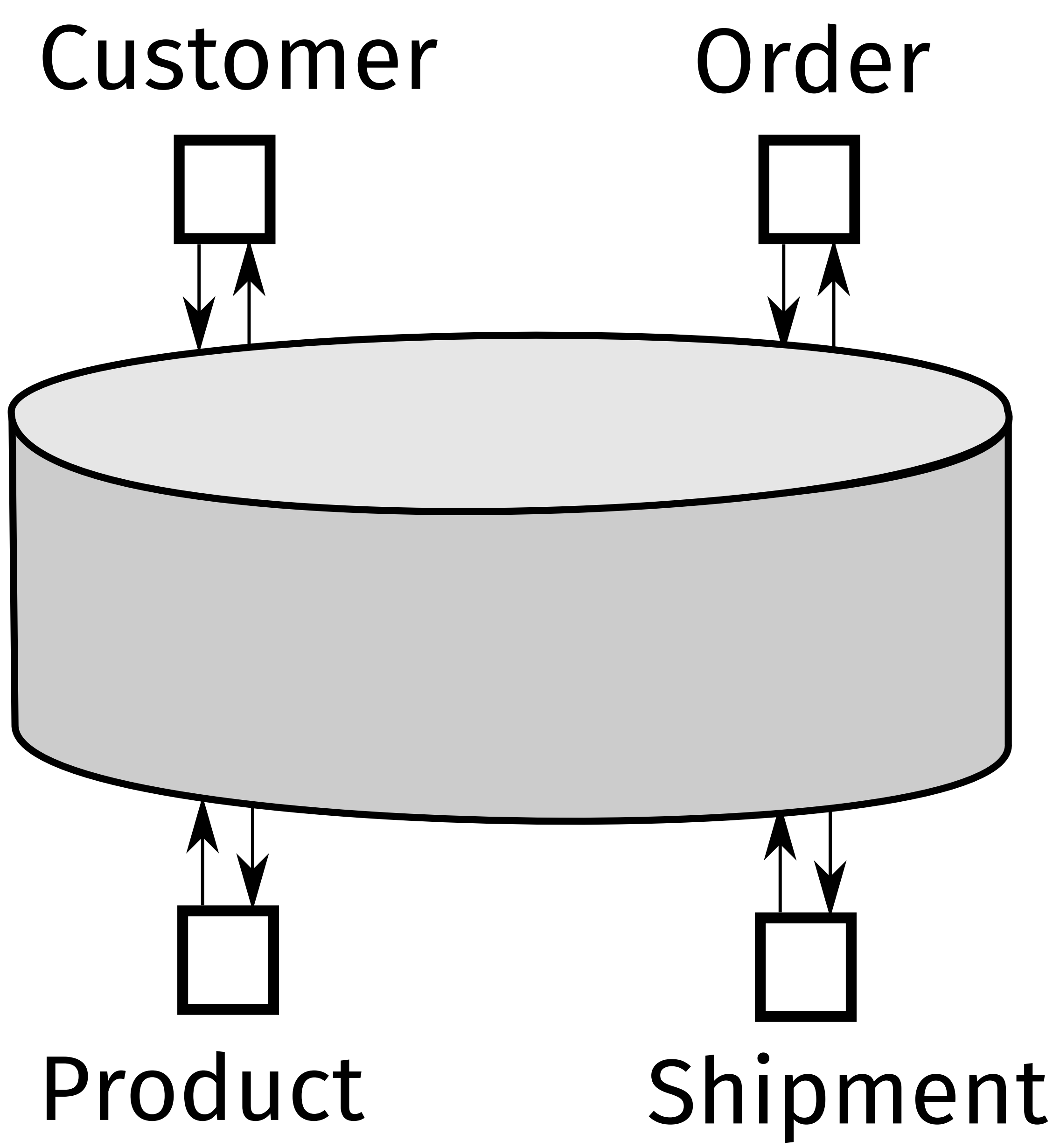
- agree among all microservices on when to perform the backup (limited autonomy)
- disallow updates anywhere during the backup (limited availability)

Backup + Consistency

Taking a consistent backup requires to:

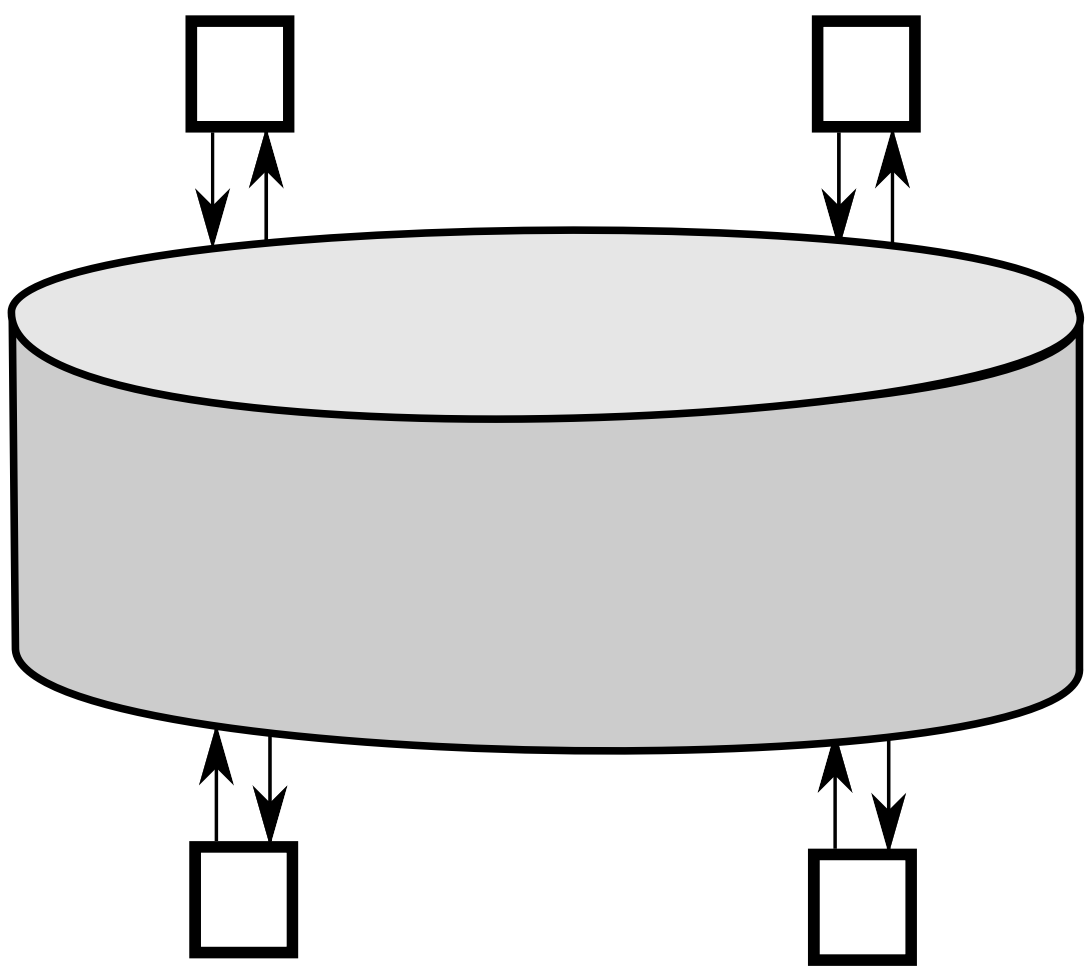
- agree among all microservices on when to perform the backup (limited autonomy)
- disallow updates anywhere during the backup (limited availability)
- wait for the slowest microservice to complete the backup (limited performance)

Shared Database



A centralized, shared database would require only one backup

Shared Database



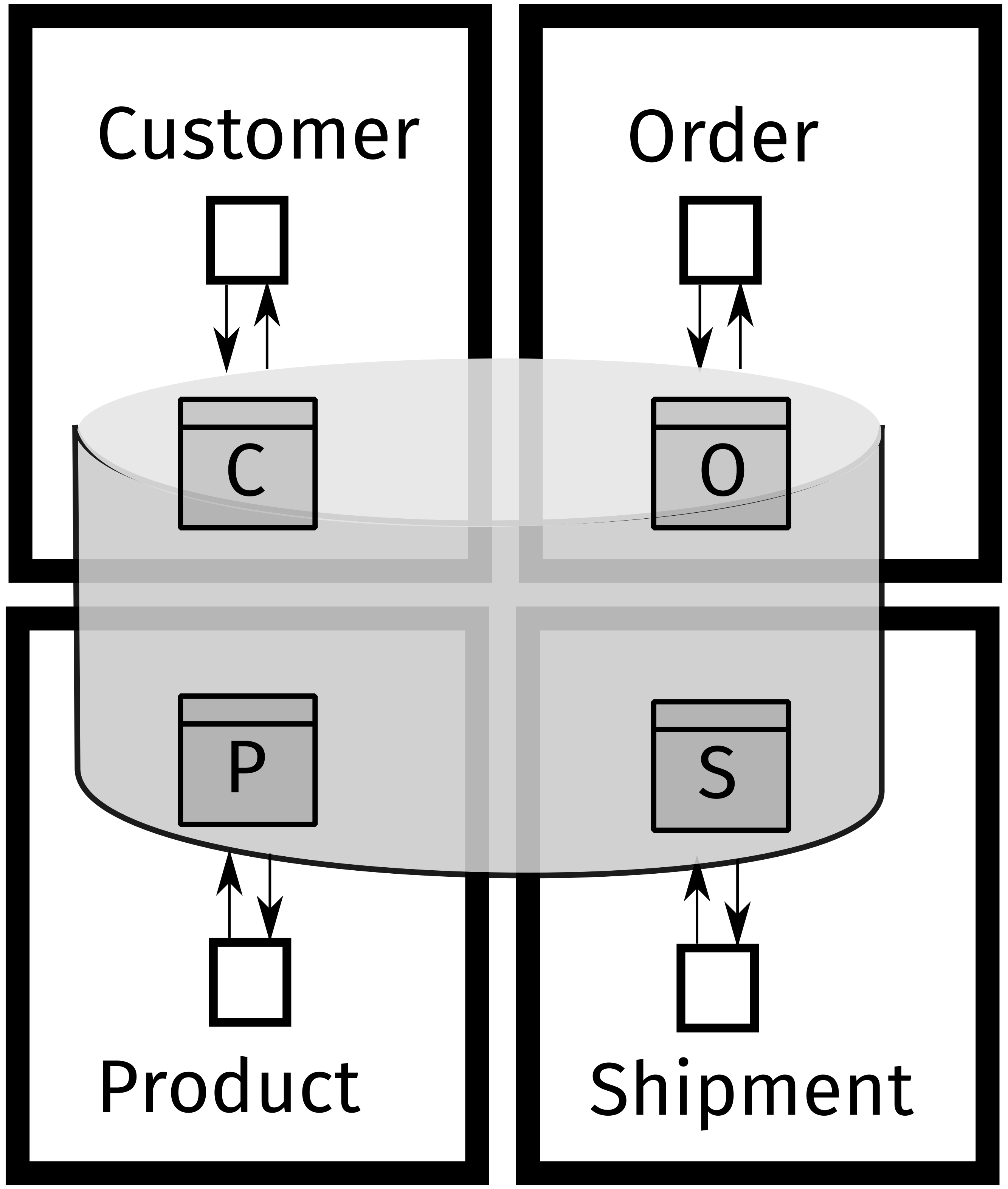
Is this still a microservice architecture?

- Yes
- No

SUBMIT

A centralized, shared database would require only one backup

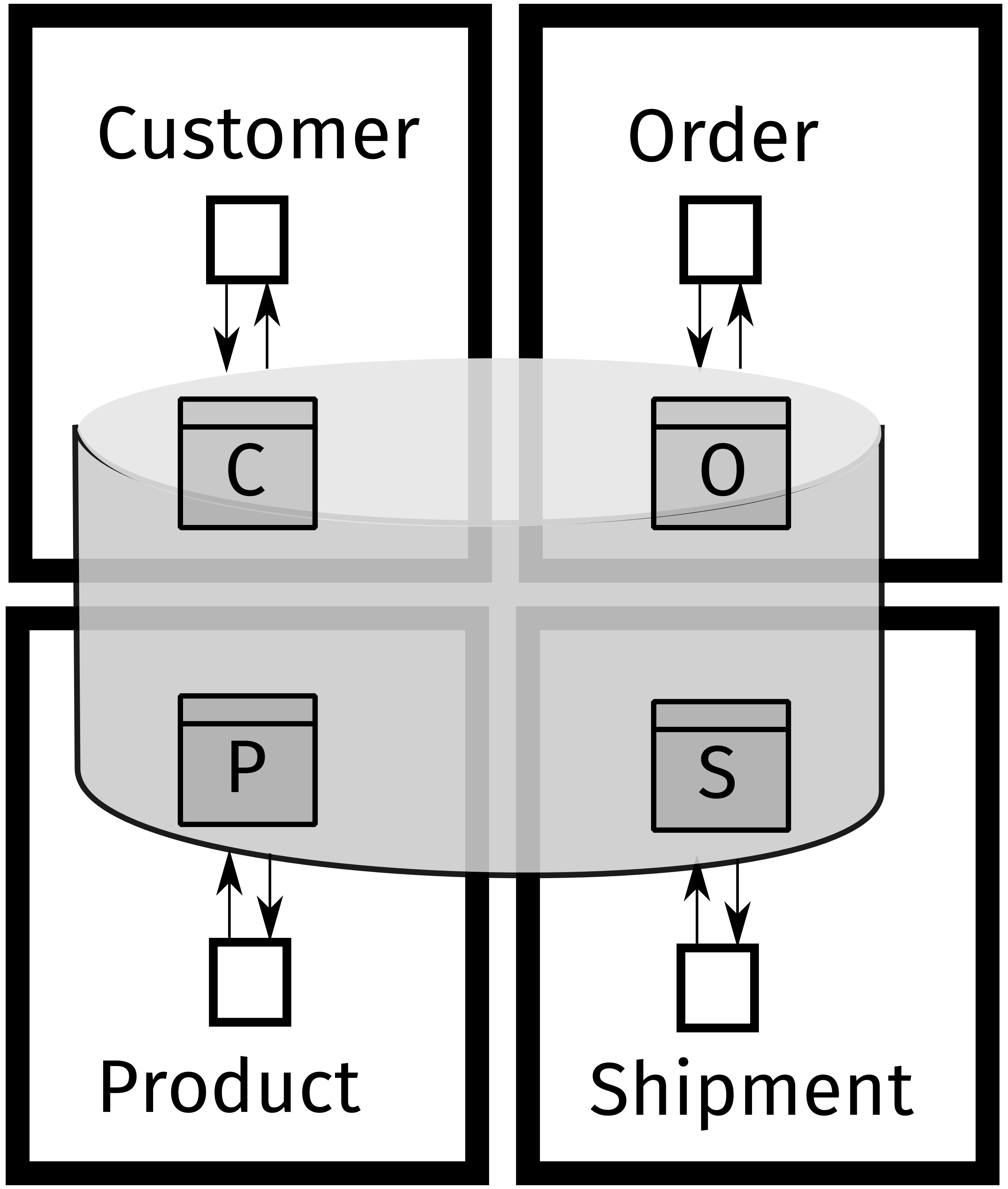
Shared Database, Split Schema



A centralized, shared database would require only one backup

Each microservice must use a logically separate schema

Shared Database, Split Schema

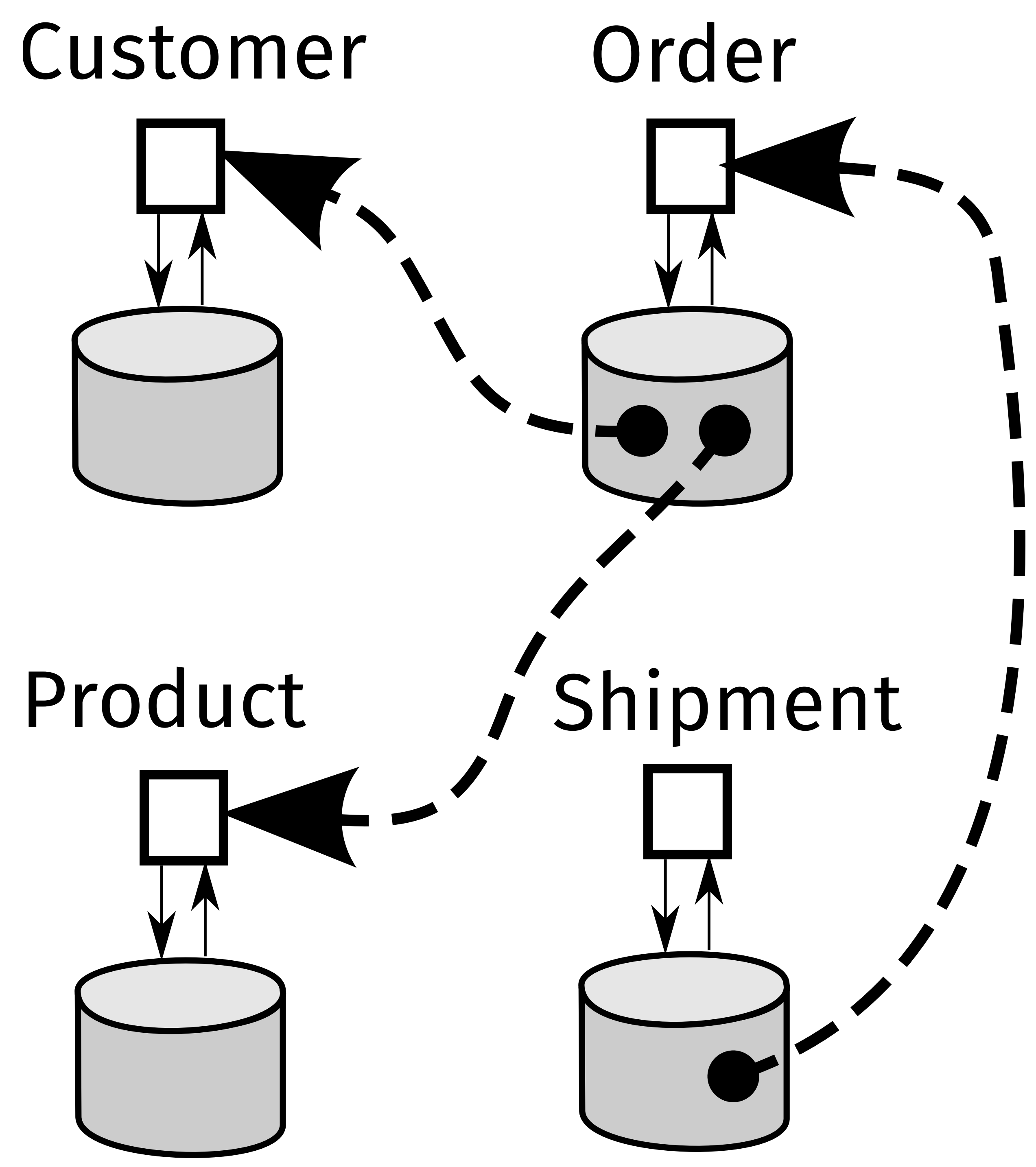


A centralized, shared database would require only one backup

Each microservice must use a logically separate schema

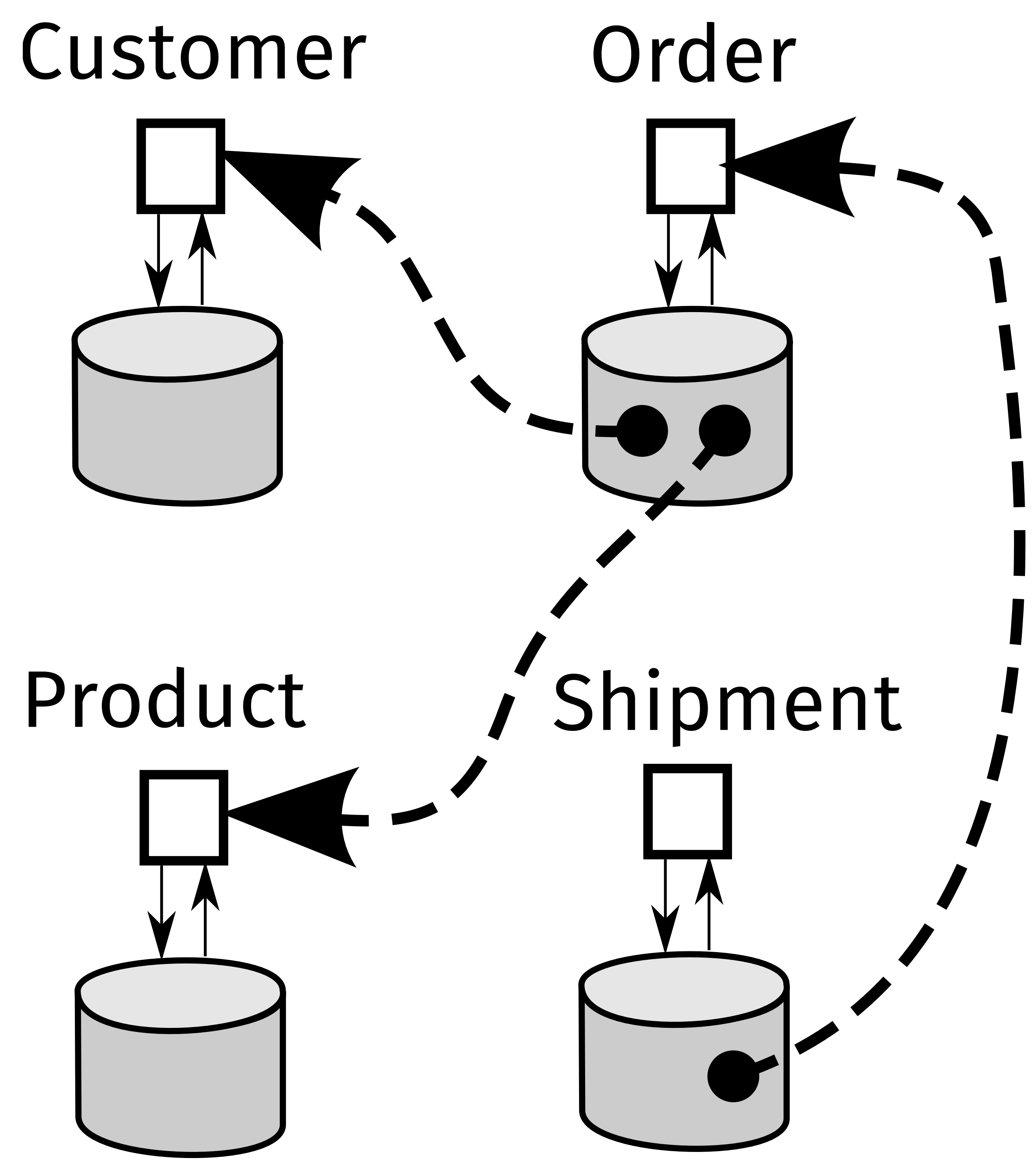
What happened to polyglot persistence?

Links can break



No guarantees for references crossing microservice boundaries

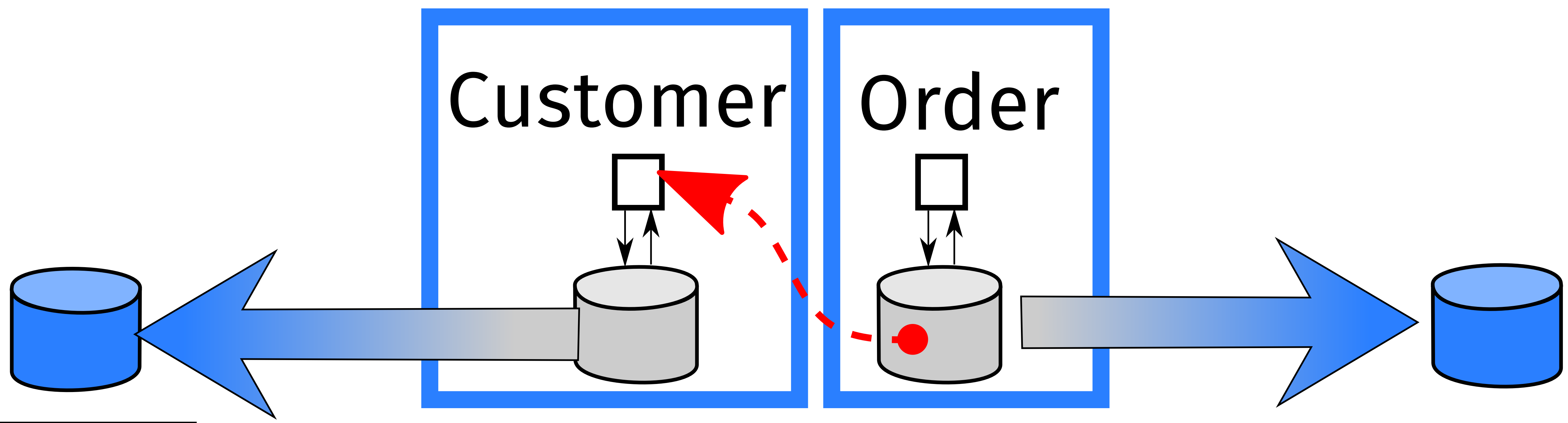
Links can break



No guarantees for references crossing microservice boundaries

Microservices inherit a fundamental property of the Web

Orphan State



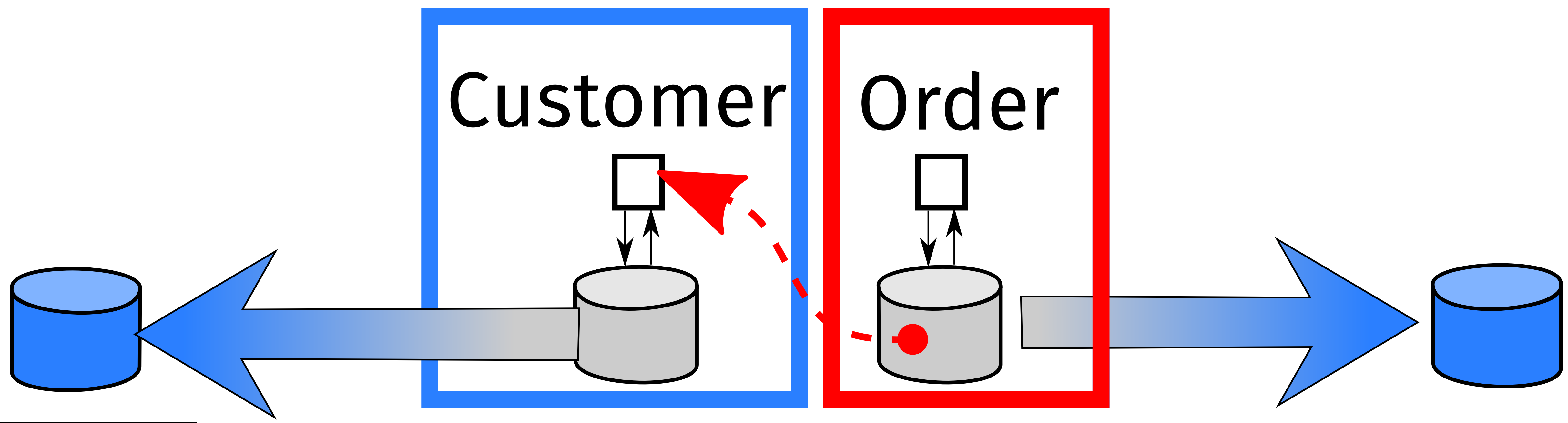
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	new C/3
6	C/3/name

1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	new C/3
6	C/3/name

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	
6	

Orphan State



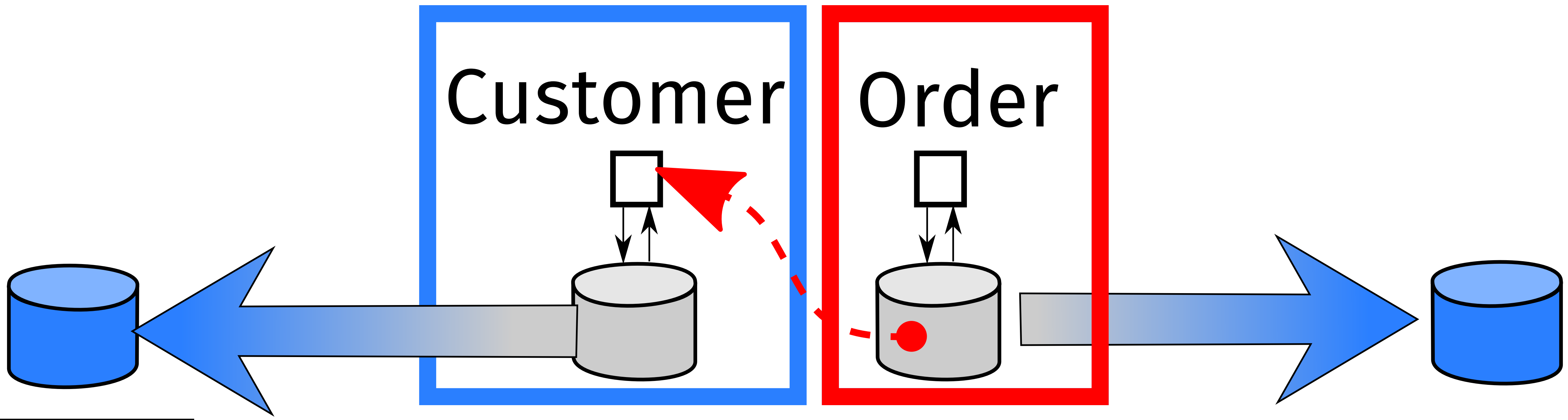
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	new C/3
6	C/3/name

1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	new C/3
6	C/3/name

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

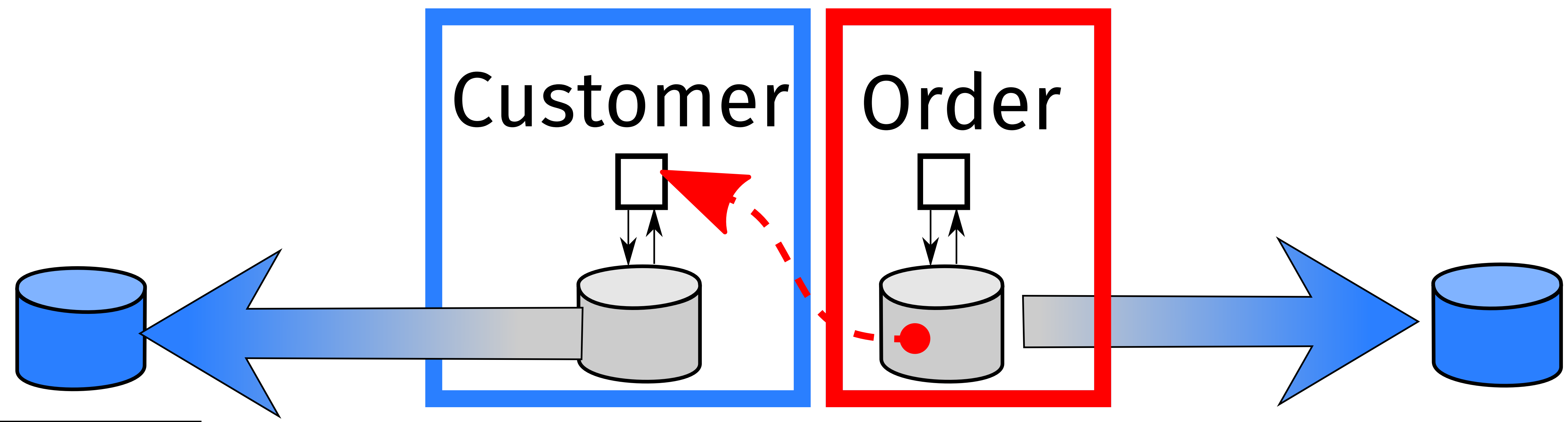
1	new O/1
2	O/1 → C/1
3	new O/2
4	O/2 → C/2
5	
6	

Orphan State



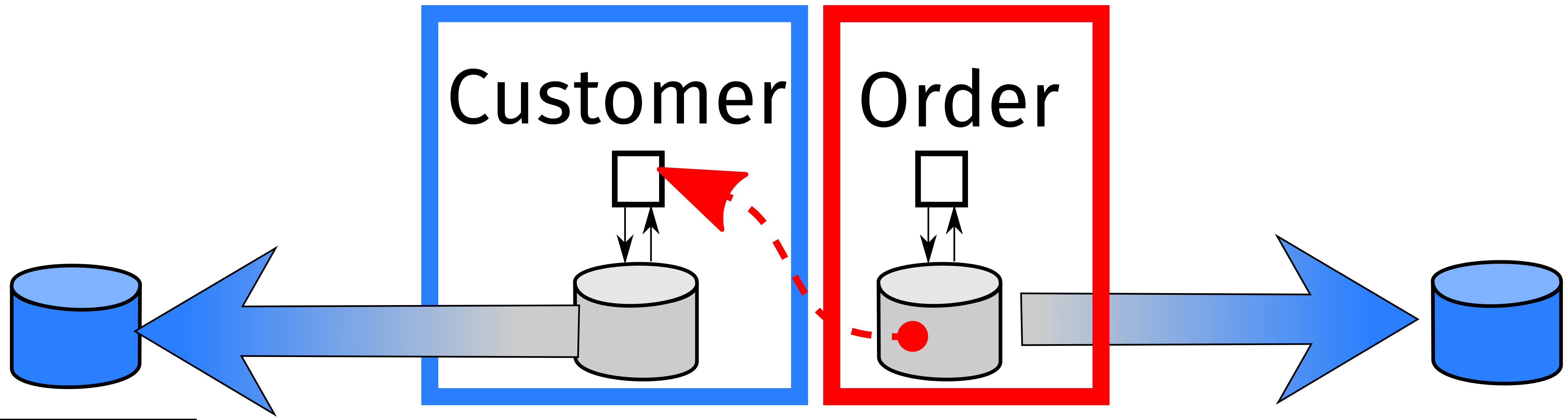
1	new C/1	new C/1	new O/1	new O/1
2	C/1/name	C/1/name	O/1 → C/1	O/1 → C/1
3	new C/2	new C/2	new O/2	new O/2
4	C/2/name	C/2/name	O/2 → C/2	O/2 → C/2
5	new C/3	new C/3		
6	C/3/name	C/3/name		

Orphan State



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

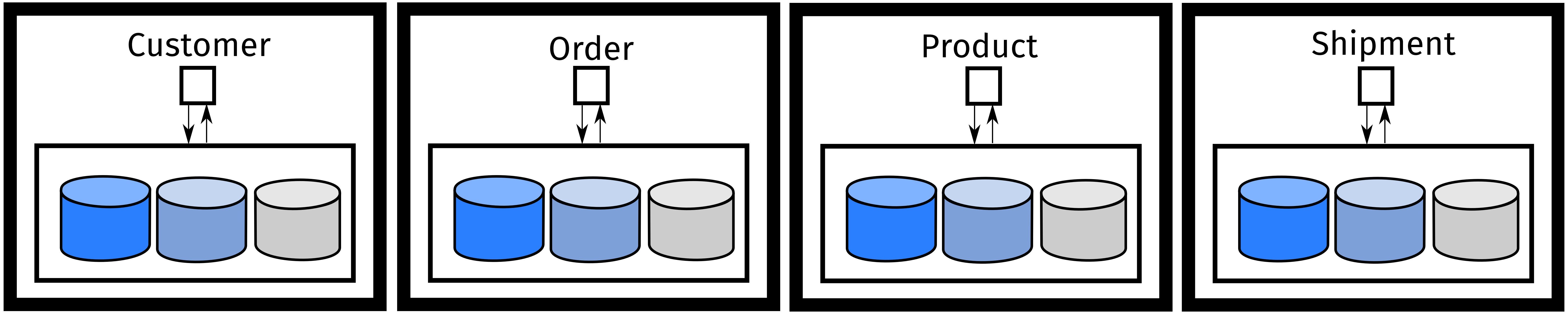
Orphan State



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

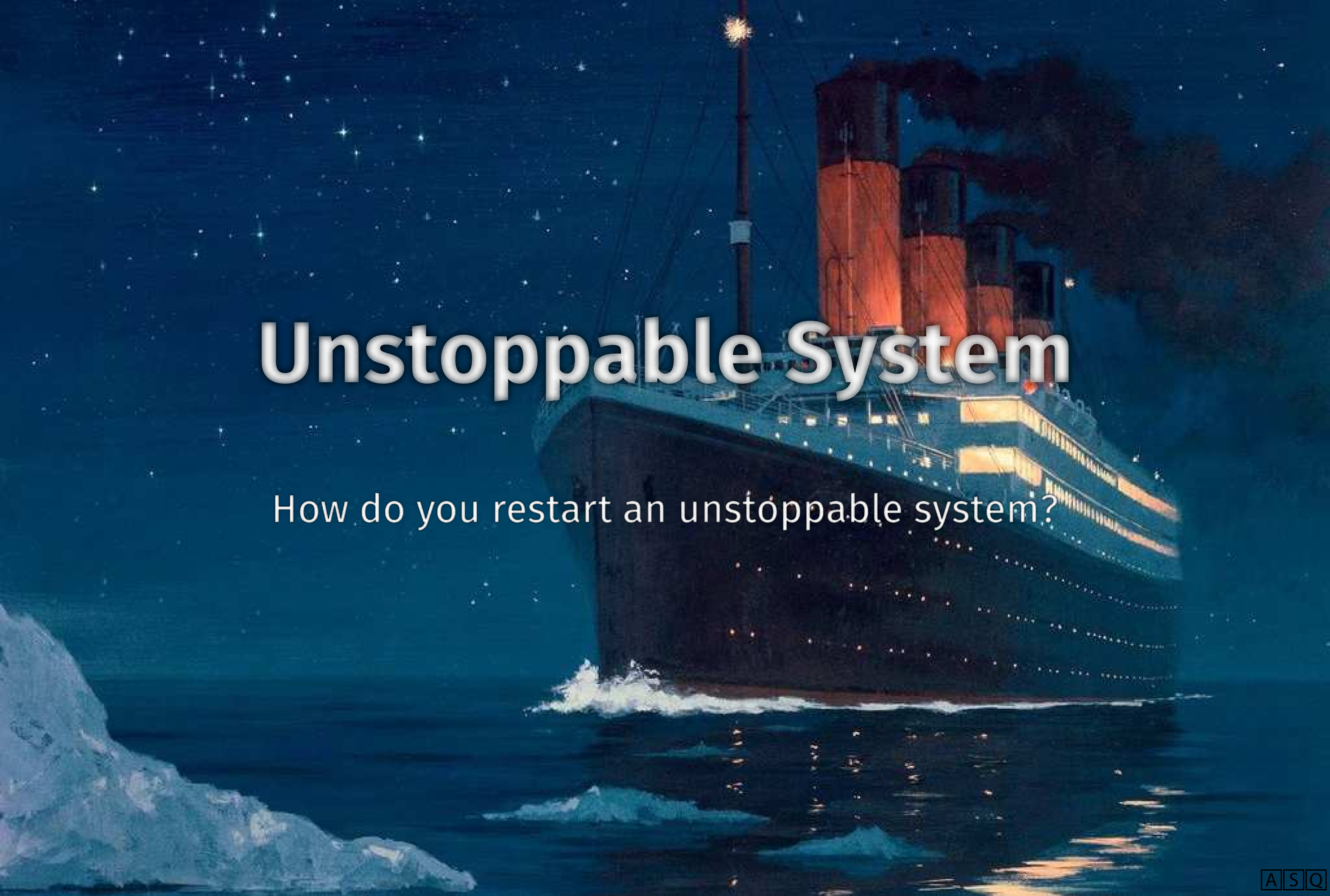
Orphan state is no longer referenced after recovery

Synchronous Replication



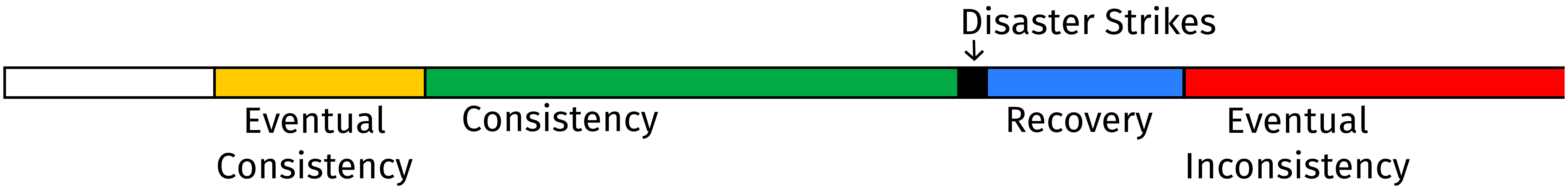
An expensive, replicated database with high-availability for every microservice

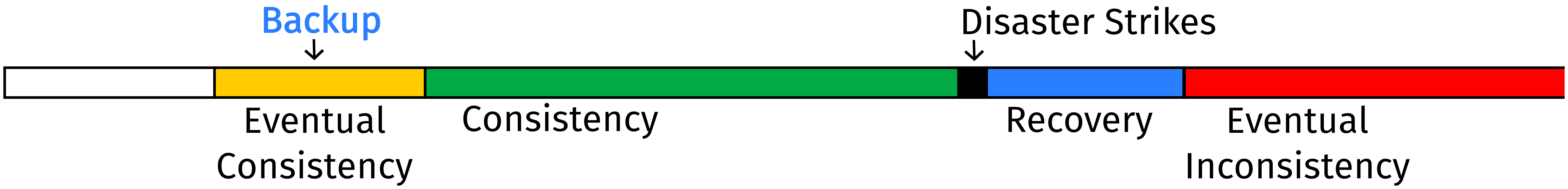

Unstoppable System
ASQ



Unstoppable System

How do you restart an unstoppable system?

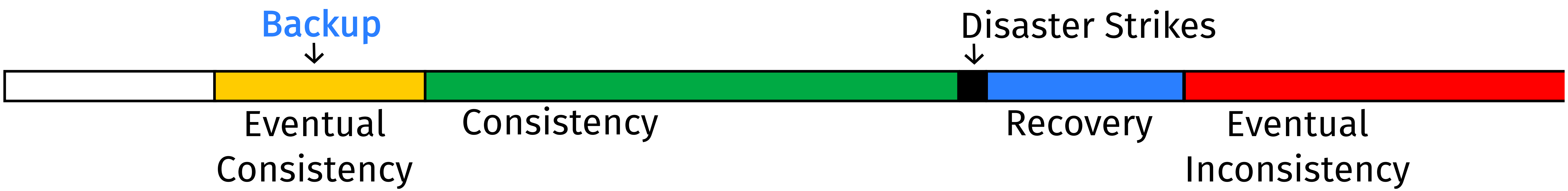

Disaster Strikes
Eventual Consistency
Consistency
Recovery
Eventual Inconsistency



Backup
Disaster Strikes
Eventual
Consistency
Consistency
Recovery
Eventual
Inconsistency

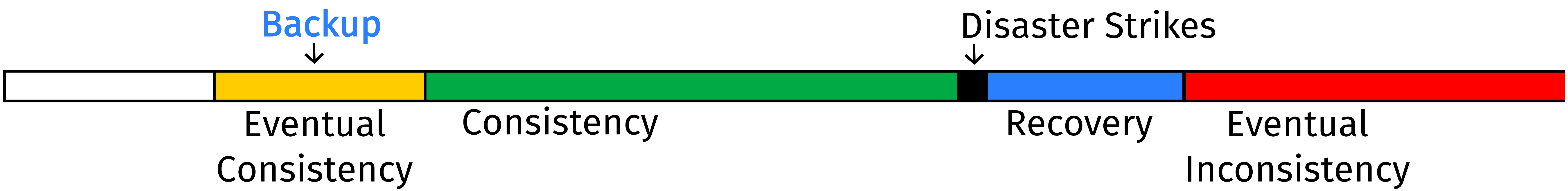
Eventual Consistency

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



Eventual Consistency

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

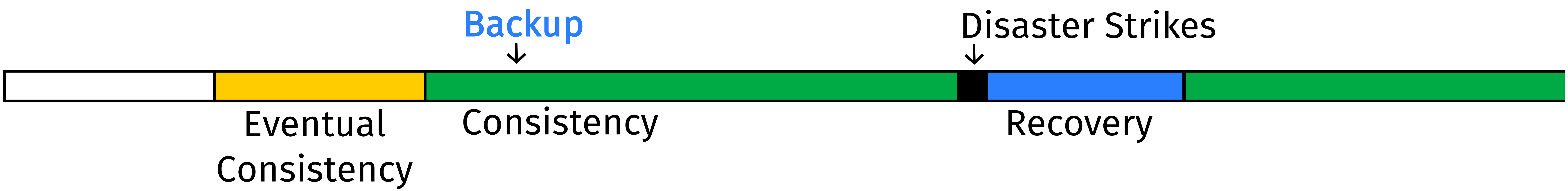


Eventual Inconsistency

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

Eventual Consistency

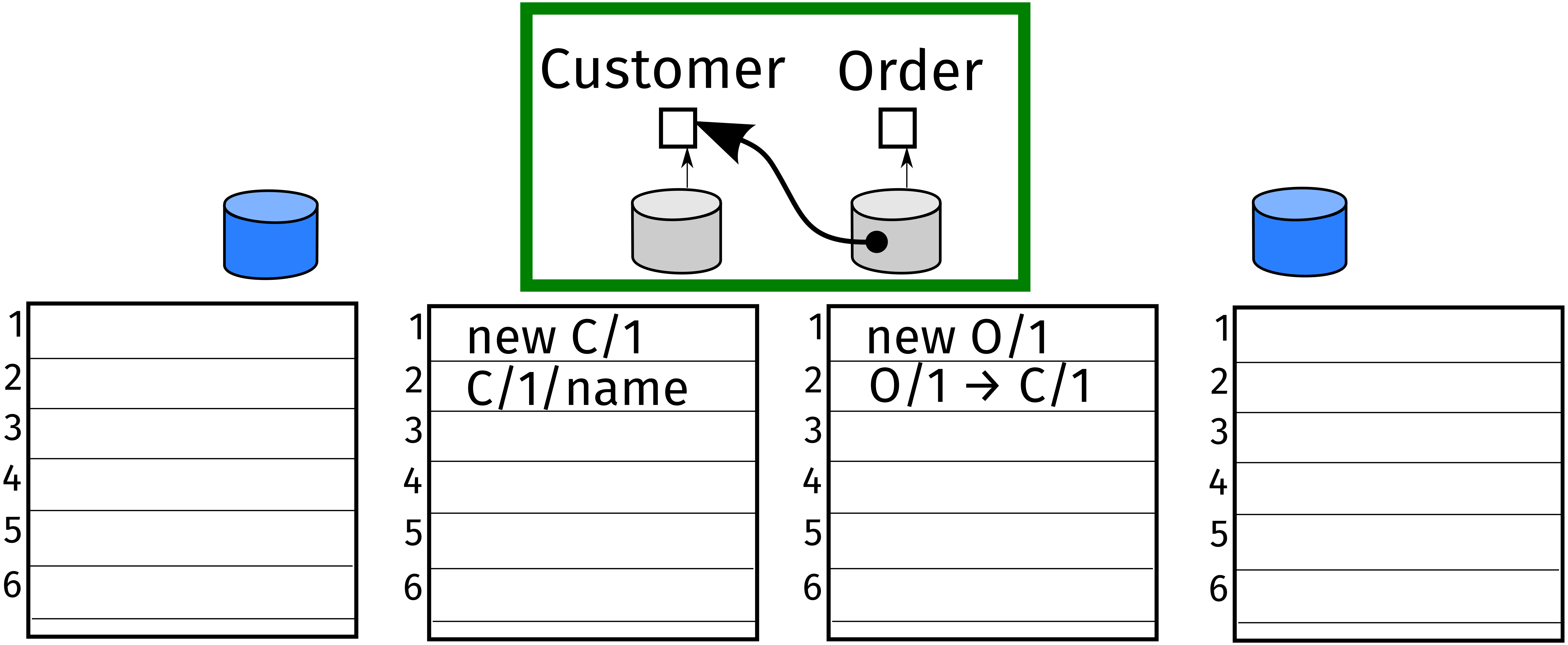
Retries are **enough** to deal with **temporary** failures of read operations, eventually the missing data will be found



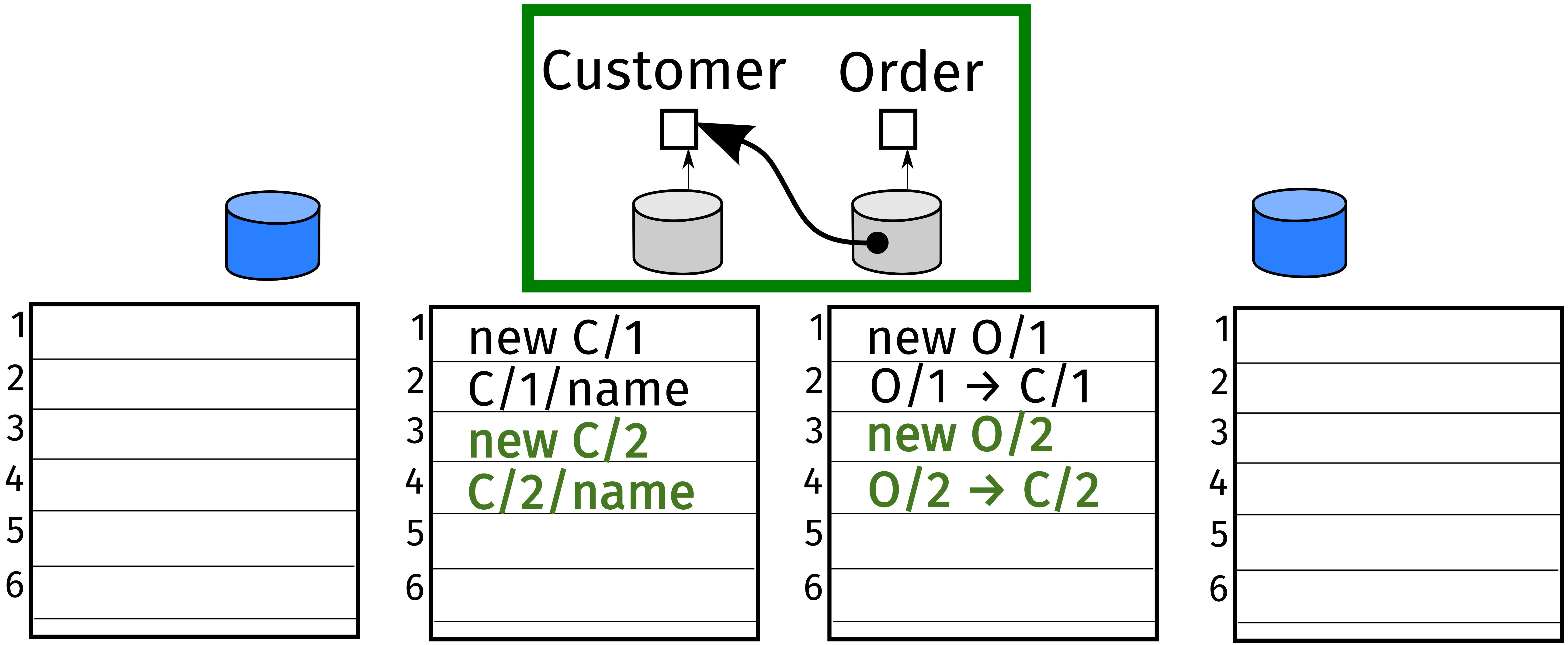
Eventual Inconsistency

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

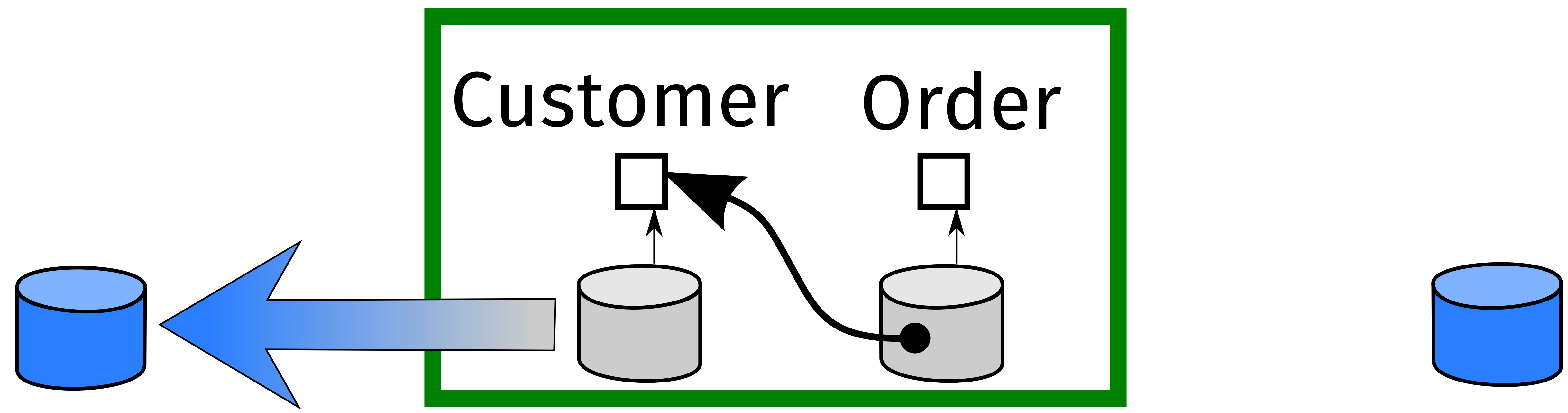
Distributed Transactions



Distributed Transactions



Distributed Transactions



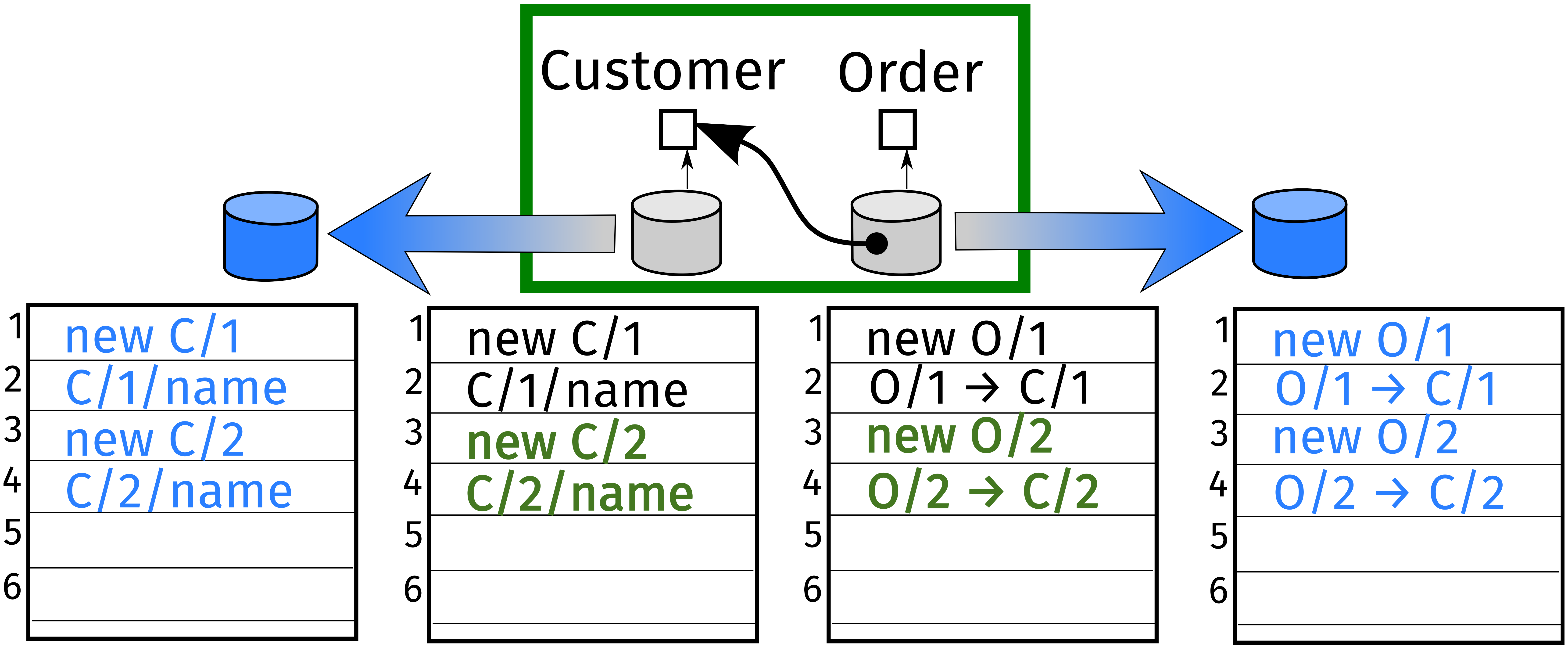
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	
6	

1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	
6	

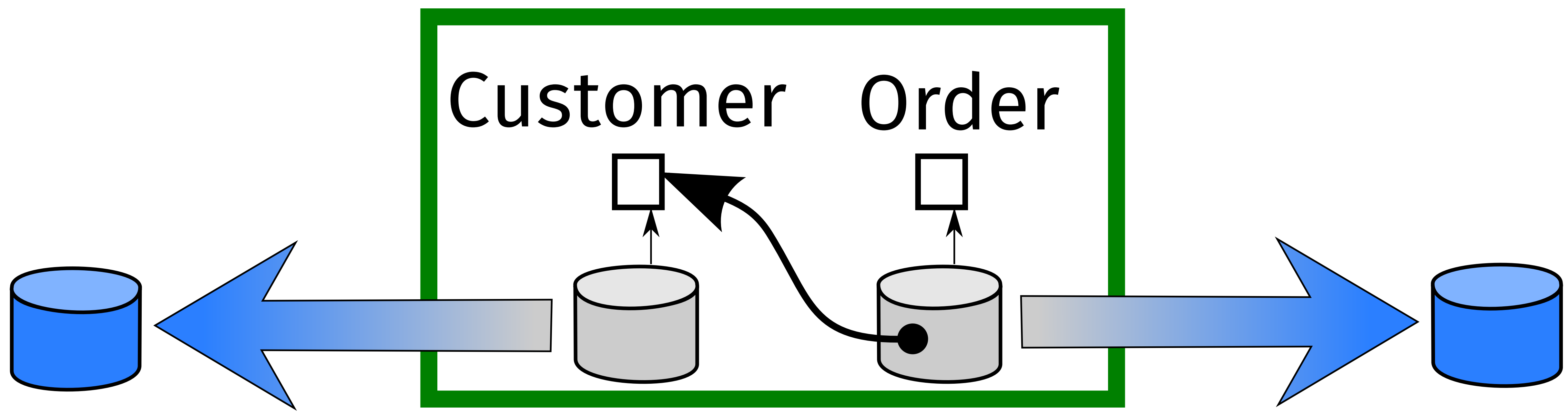
1	new O/1
2	O/1 → C/1
3	new O/2
4	O/2 → C/2
5	
6	

1	
2	
3	
4	
5	
6	

Distributed Transactions



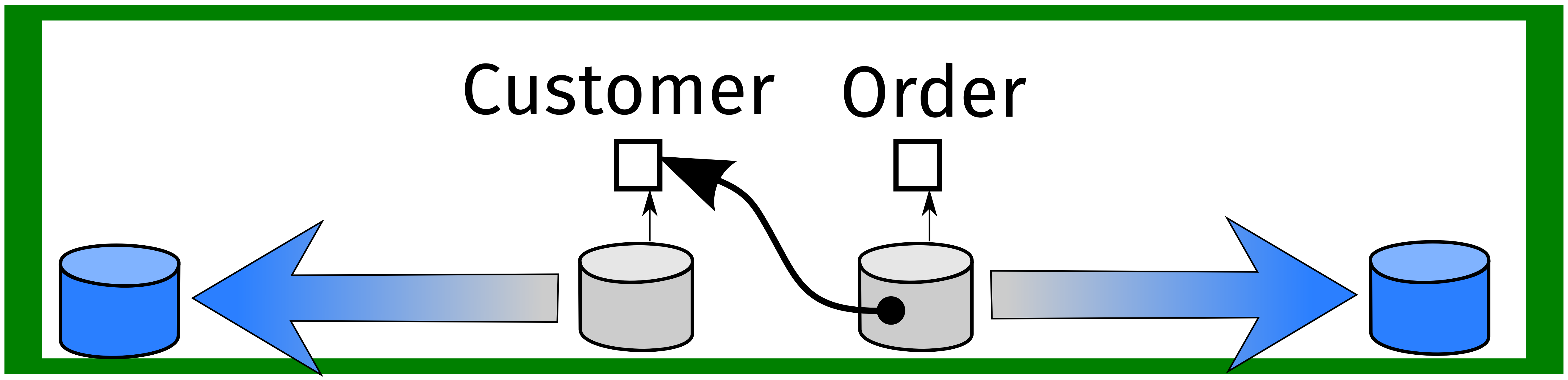
Distributed Transactions



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

Take snapshots only when all microservices are consistent

Distributed Transactions

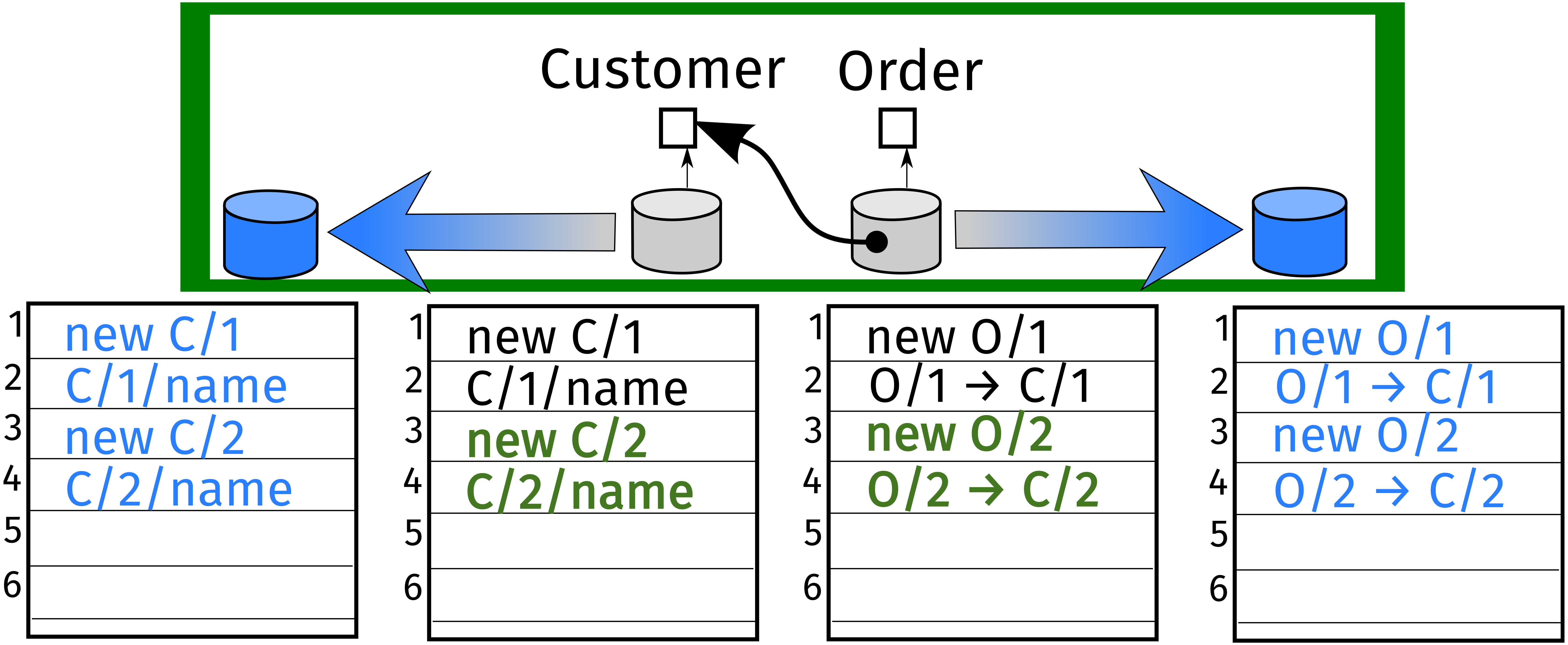


1	new C/1	new C/1	new O/1	new O/1
2	C/1/name	C/1/name	O/1 → C/1	O/1 → C/1
3	new C/2	new C/2	new O/2	new O/2
4	C/2/name	C/2/name	O/2 → C/2	O/2 → C/2
5				
6				

Take snapshots only when all microservices are consistent

Backups taken as part of the distributed transaction

Distributed Transactions



Take snapshots only when all microservices are consistent

Backups taken as part of the distributed transaction

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

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.

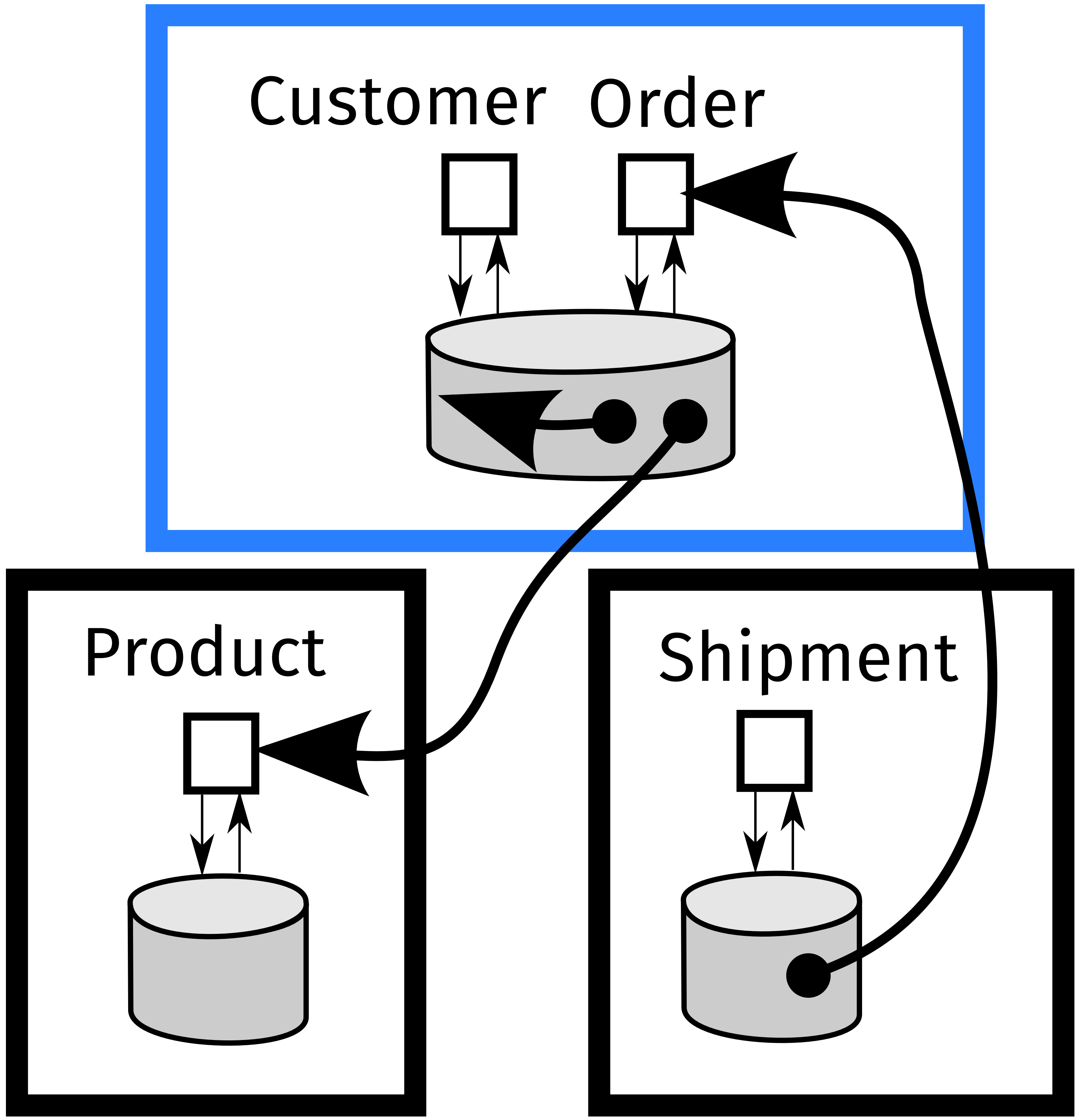
M. Fowler, J. Lewis https://www.martinfowler.com/articles/microservices.html

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

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

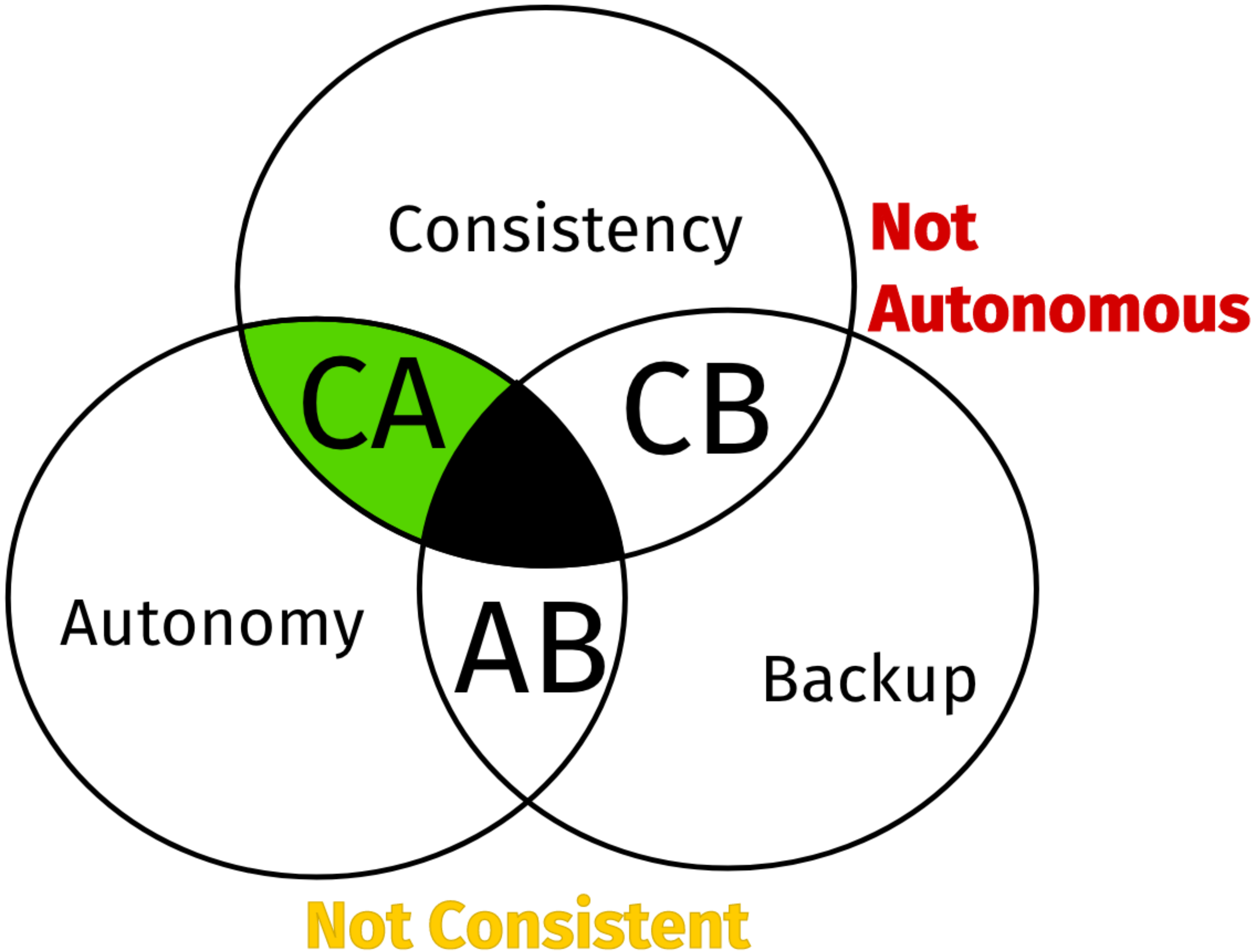
⇒ **Eventual inconsistency (after disaster recovery)**

Does it apply to you?

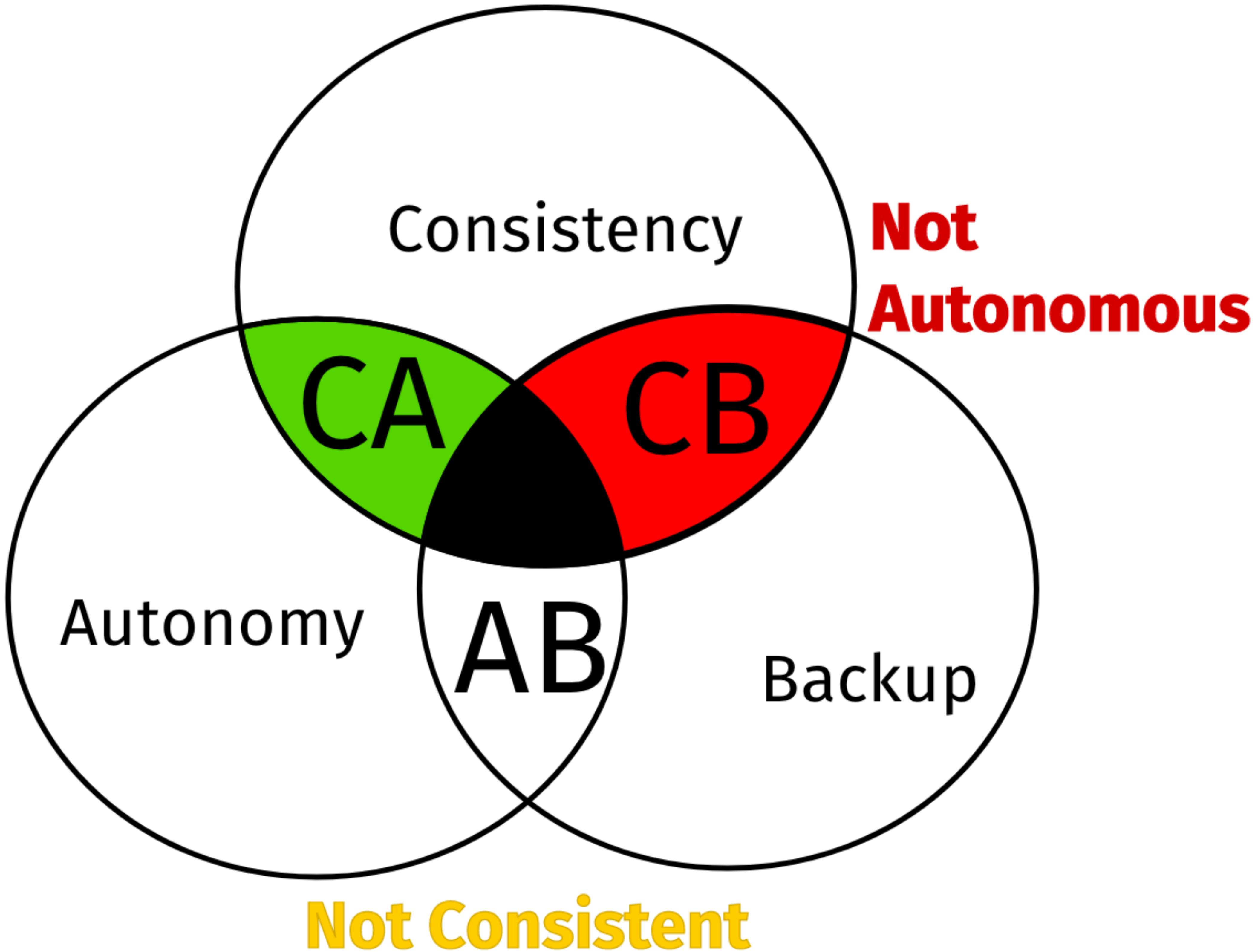
- ☐ More than one stateful microservice
- ☐ Polyglot persistence
- ☐ Eventual Consistency
- ☐ (Cross-microservice references)
- ☐ Disaster recovery based on backup/restore
- ☐ **Synchronized** backups (limited autonomy)

⇒ **Consistent Disaster Recovery**

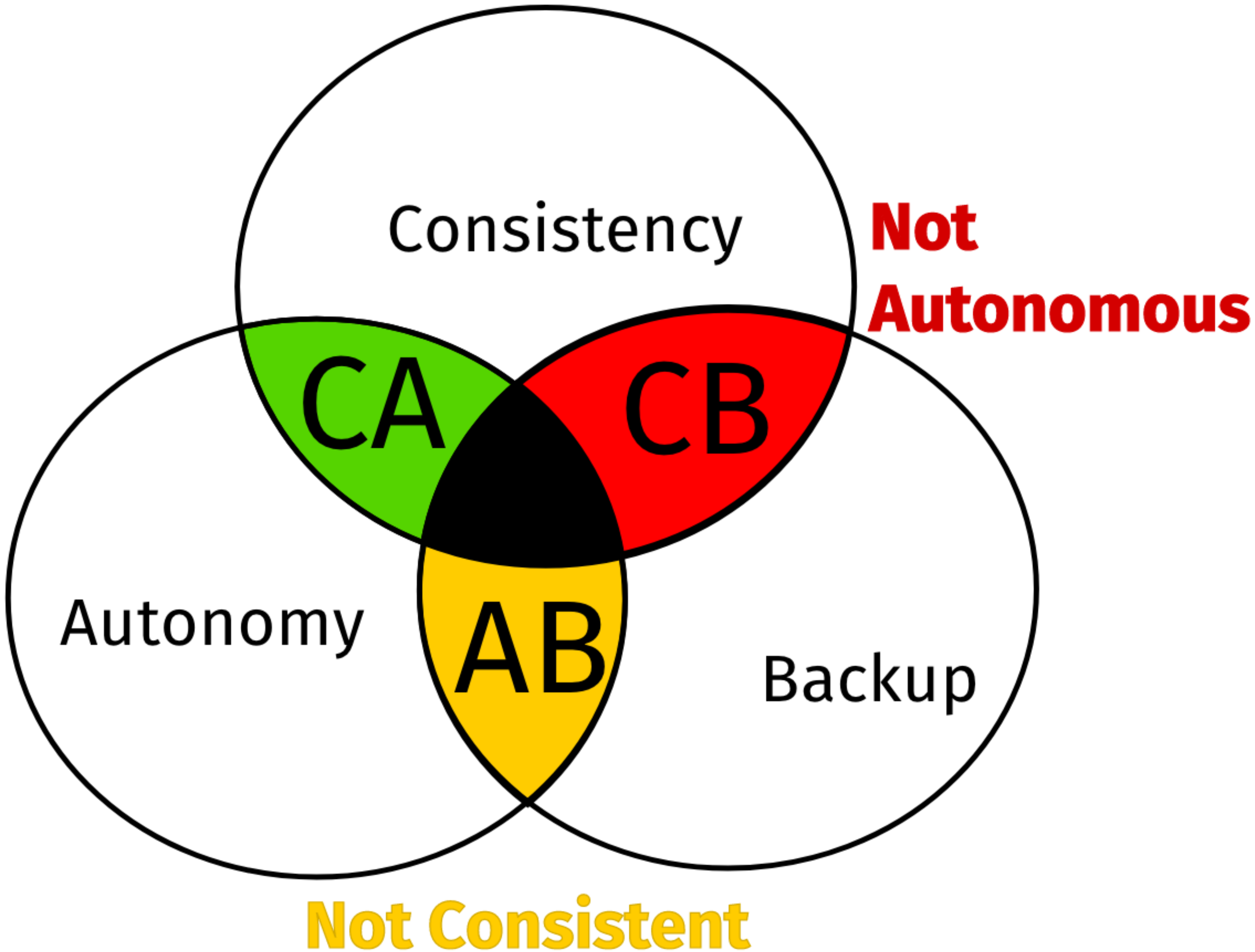
The BAC Theorem



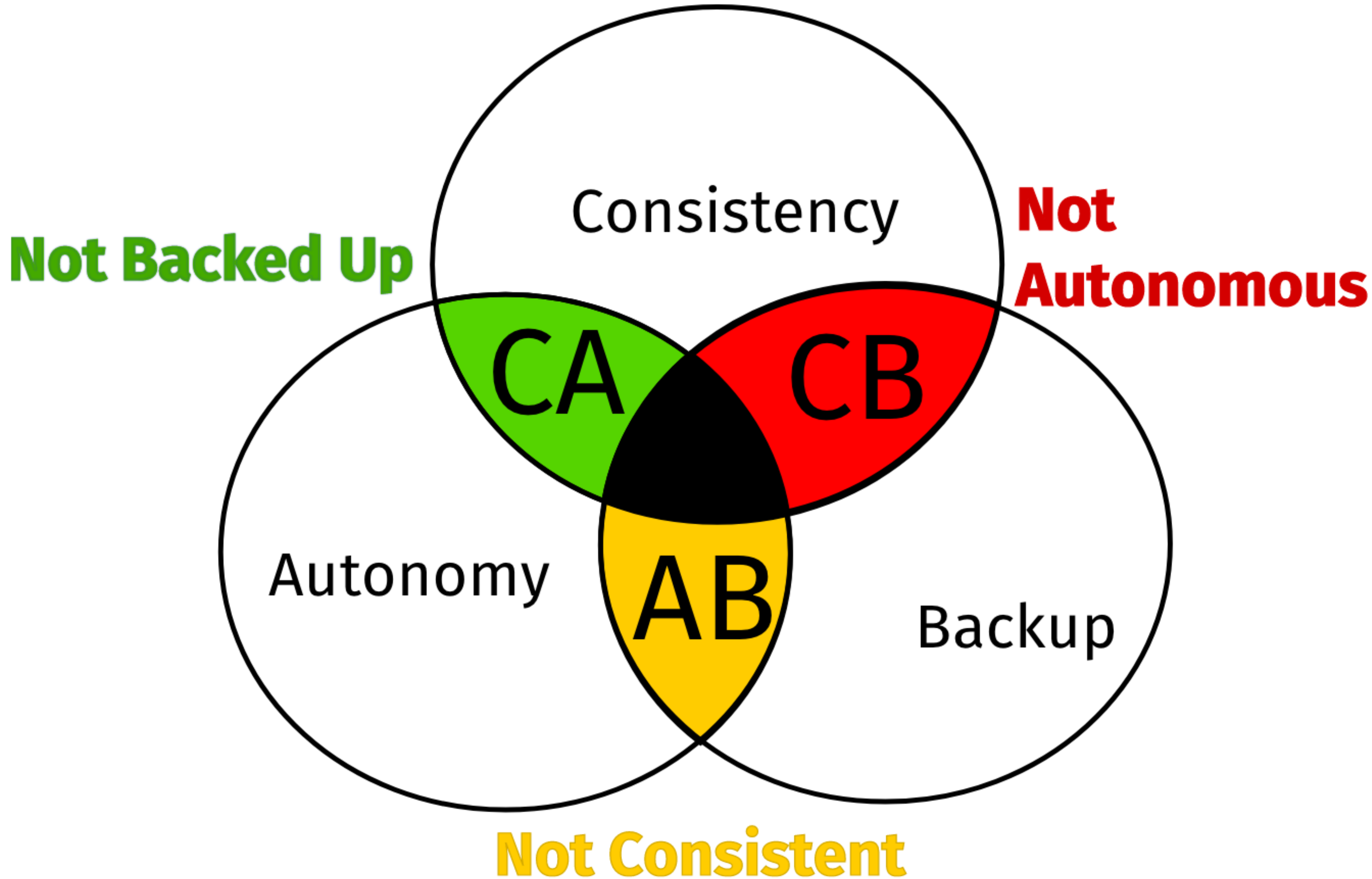
The BAC Theorem



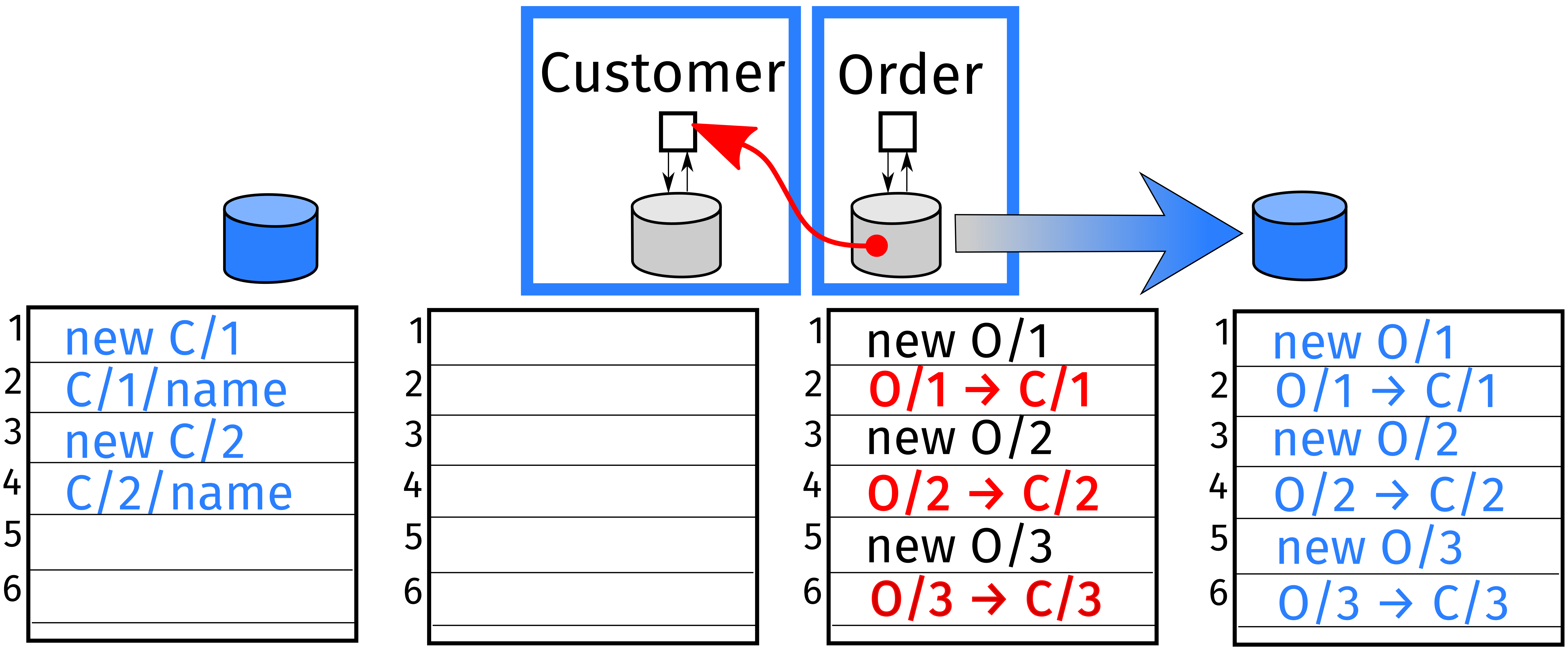
The BAC Theorem



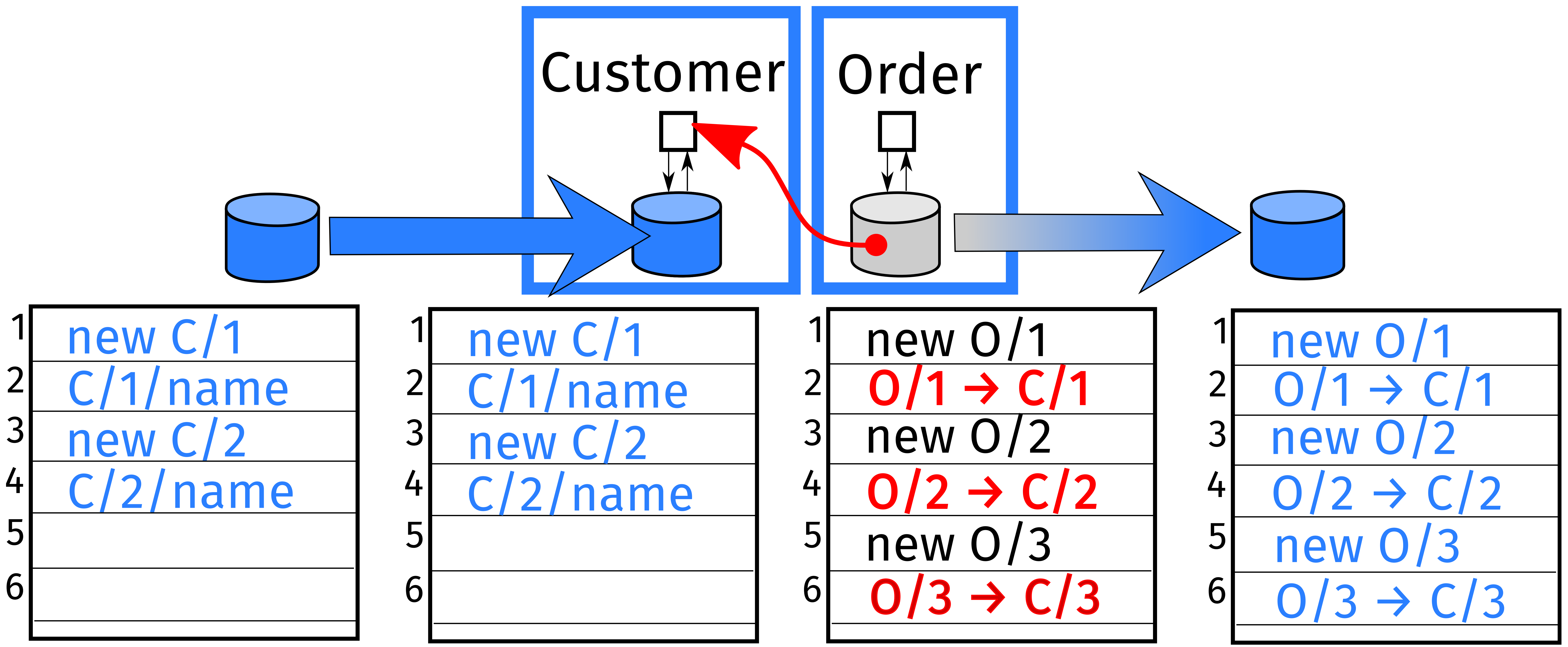
The BAC Theorem



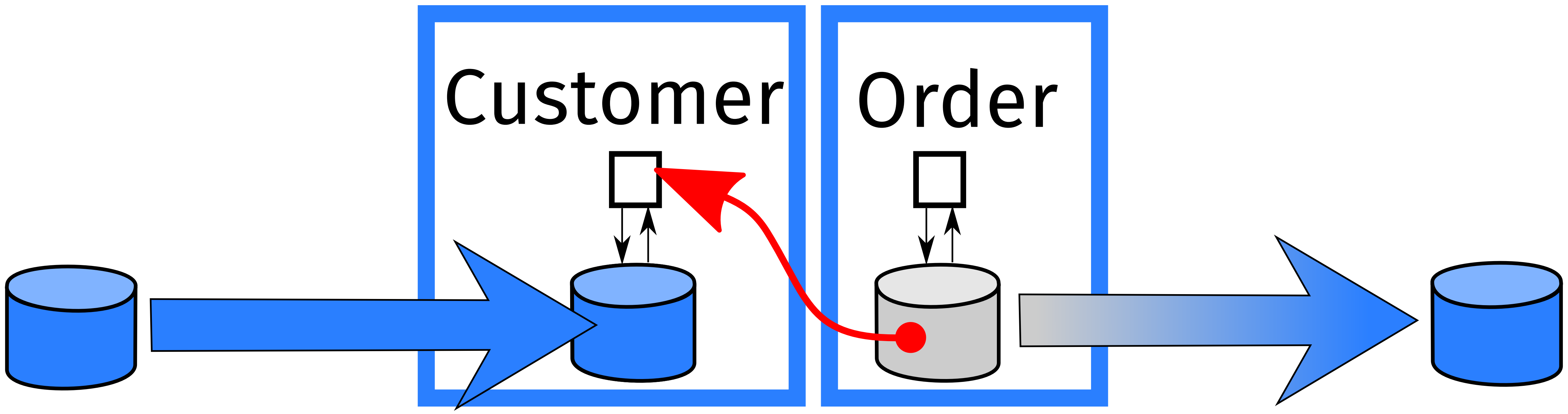
No Backup


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


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



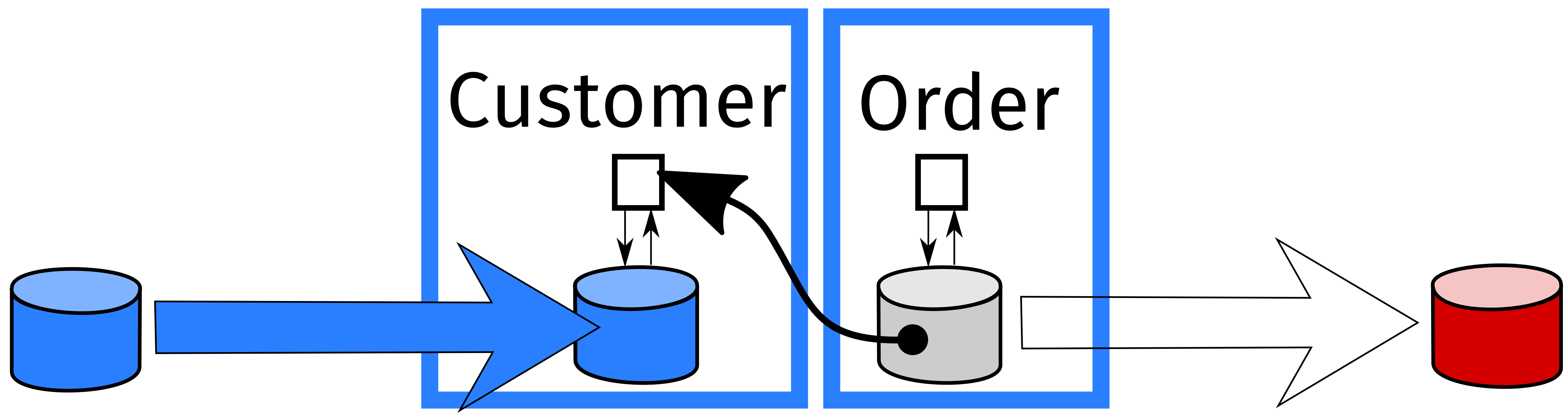
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	
6	

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



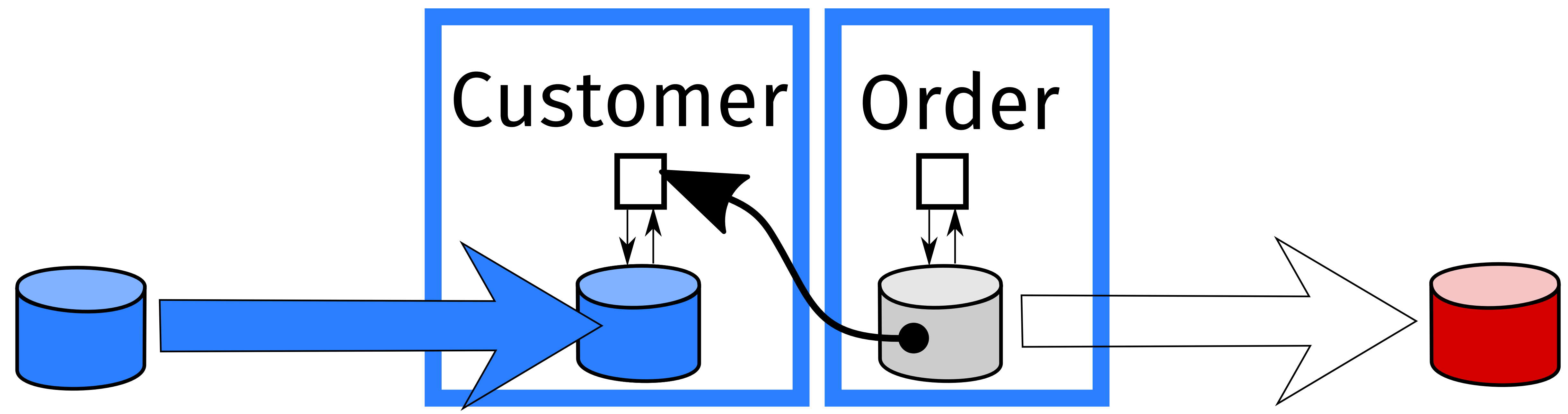
1	new C/1
2	C/1/name
3	new C/2
4	C/2/name
5	
6	

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	
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

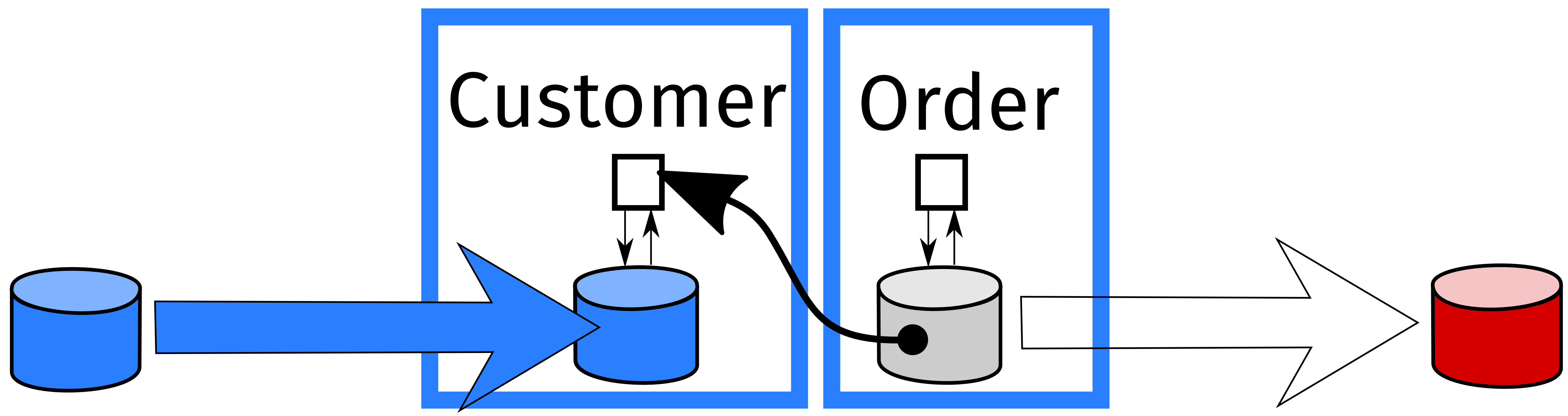
No Backup



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

Trim to the oldest backup

No Backup



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

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

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

The BAC Theorem

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

Corollaries

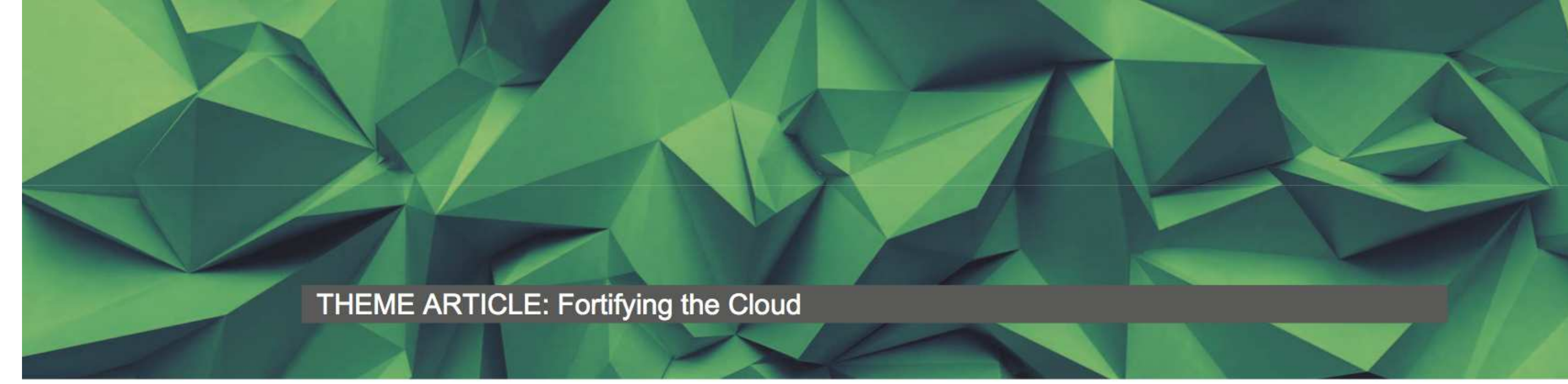
1. Microservice architectures eventually become inconsistent after disaster strikes when recovering from independent backups
2. Achieving consistent backups can be attempted by limiting the full availability/autonomy of the microservices and synchronizing their backups

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

Guy Pardon, Cesare Pautasso, Olaf Zimmermann, Consistent Disaster Recovery for Microservices: the BAC Theorem, IEEE Cloud Computing, 5(1):49-59, January/February 2018

http://design.inf.usi.ch/bac



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.

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
- 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.
- Thomas Erl, Benjamin Carlyle, Cesare Pautasso, Raj Balasubramanian, SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST, Prentice Hall, 2012