UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURES 17-18

Modern Software Architectures

Prof. Luigi Libero Lucio Starace

luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

Software Development Life Cycle

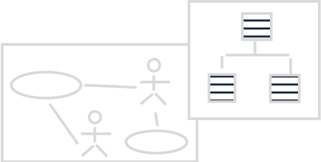
Requirements Engineering → **System Design** → Software and UI Design → Implemen-tation → Testing → Operation and Maintenance

Requirements collected via:

- Interviews with Stakeholders
- Personas
- Stories and Scenarios

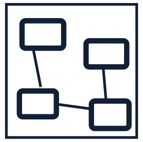
Specified using:

- Use Cases
- Natural Language
- Domain Models
- Mock-ups



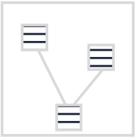
Define System Architecture

- Requirements are allocated to software sub-systems
- Sub-systems are allocated to hardware resources
- Architectural Patterns



Define Subsystems

- Objects required to realize each subsystem are defined.
- Software Design Patterns
- Usability Engineering
- High-fidelity Wireframing



Each Subsystem is implemented

- Source code and other artifacts
- Clean Code
- Frameworks and ORMs
- Focus on Software Quality

Ensure the Software satisfies customers

- Code inspections
- Functional Testing (unit, integration, system testing)
- Usability Testing

System is put into practical use

Maintance will be required at some point

- To fix errors that were not discovered in previous phases
- To adapt the software to changes in requirements on in its environment

Software System Architecture

- The **architecture** of a software system is the set of **principal design decisions** about the system
  - A **blueprint** for a system’s construction and evolution
  - These design decisions have a significant impact on the system’s **structure**, **behaviour** and **interactions**, and **non-functional properties**
- **«Principal»** implies an higher-degree of importance that grants a design decision an «architectural status»
  - Is choosing between MariaDB and MySQL is an architectural decision?
  - What about choosing between a relational or NoSQL database?
  - Architecture is about stuff that is **hard to change** later!

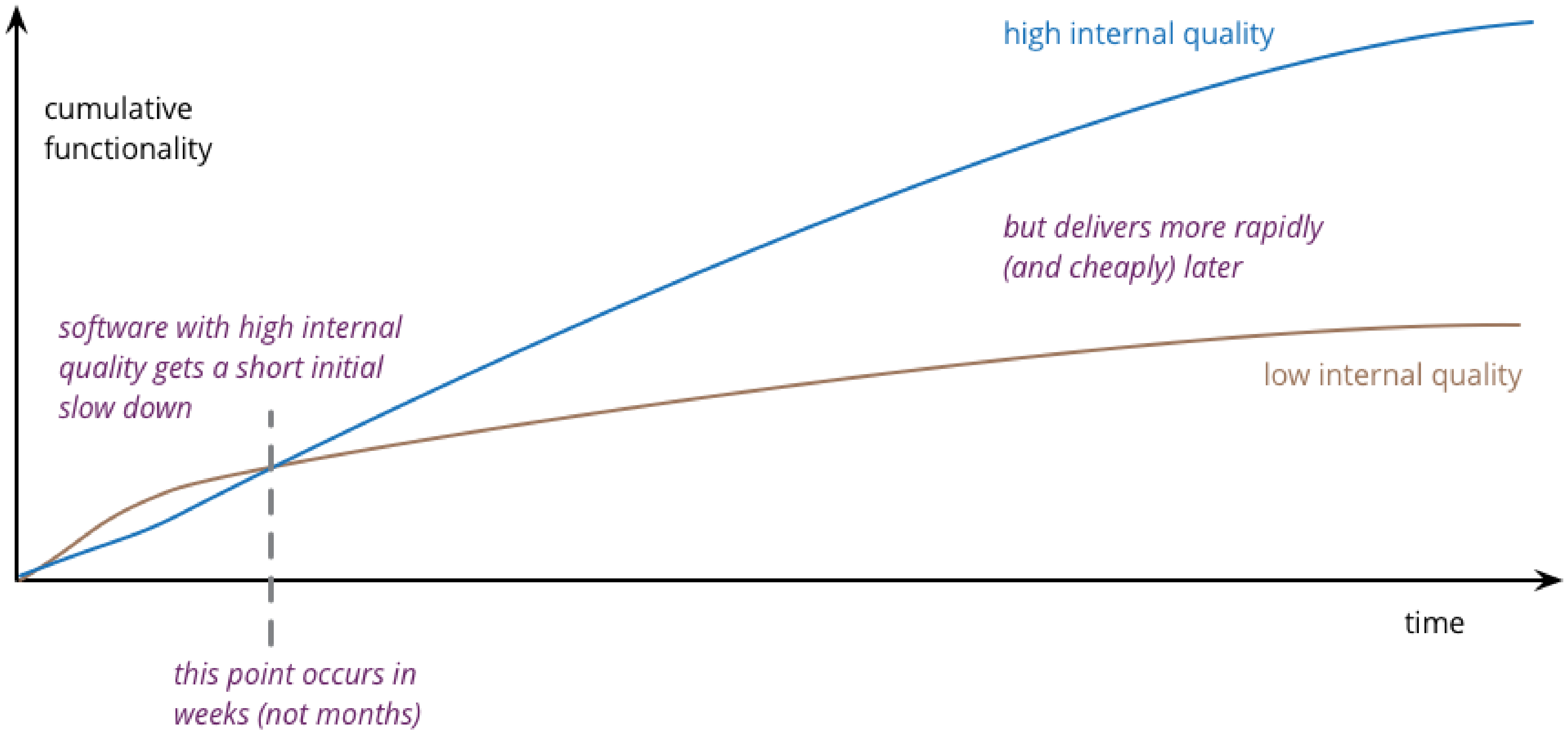
Software System Architecture

- There is no objective way to define what decisions are «**principal**»
- **“Architecture is about the important stuff. Whatever that is”**.
  - The heart of thinking architecturally about software is to **decide what is important** (i.e., what is **architectural**), and then expend energy on keeping those architectural elements in good condition.
  - For a developer to become an architect, they need to be able to recognize what elements are important, recognizing what elements are likely to result in serious problems should they not be controlled.

Why does architecture matter?

- Architecture is a tricky subject for customers and users
    - It’s something they don’t directly perceive
- Nonetheless, architecture has a significant impact on the system’s **structure**, **behaviour** and **interactions**, and **non-functional properties**
- Moreover, poor architecture can lead to systems that are much harder to engineering and evolve, with impacts on costs and delivery of new features
    - High internal quality (e.g.: good architecture) costs more at the beginning, but pays off in the long run!

Why does architecture matter?



Martin Fowler. Is High Quality Software Worth the Cost?
https://martinfowler.com/articles/is-quality-worth-cost.html

Properties affected by architecture

Scalability

Availability

Consistency

Performance

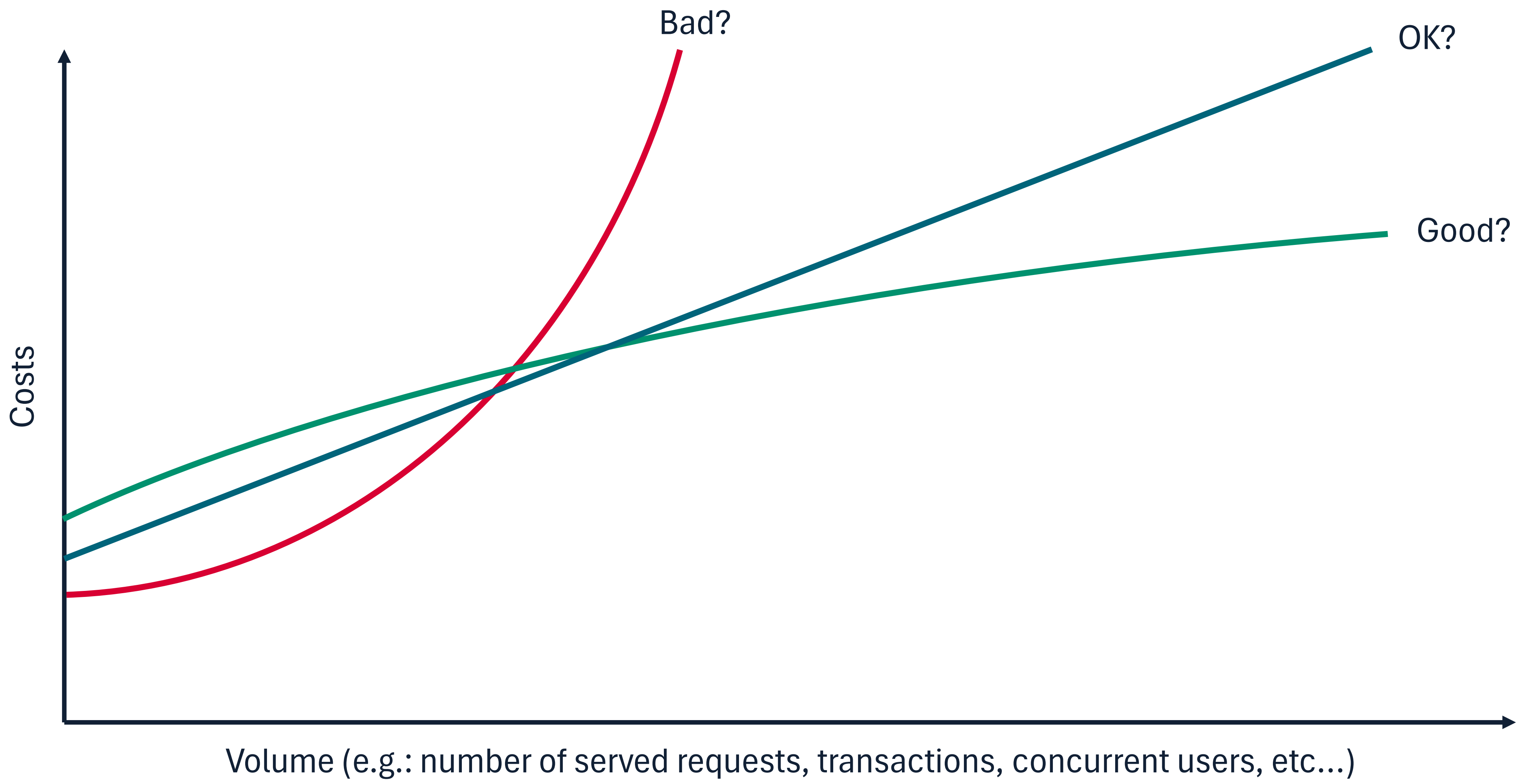
Latency

Throughput

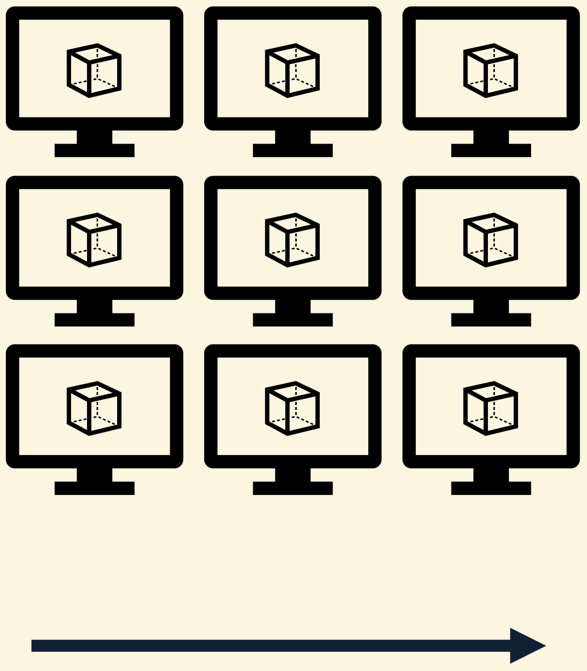
Scalability

- A service is **scalable** if it results in increased **performance** in a manner proportional to resources added.
- Generally, increasing performance means serving **more units of work**, but it can also be to handle **larger units of work**, such as when datasets grow.
- Another way to look at performance vs scalability:
  - If you have a **performance** problem, your system is slow for a single user.
  - If you have a **scalability** problem, your system is fast for a single user but slow under heavy load.

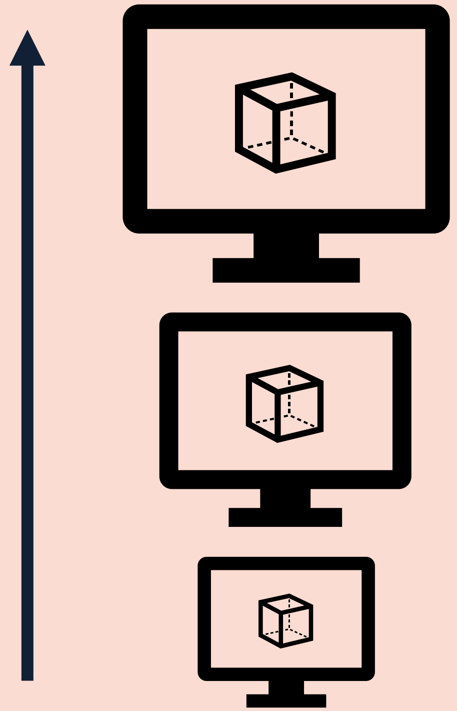
Scalability: it's about money



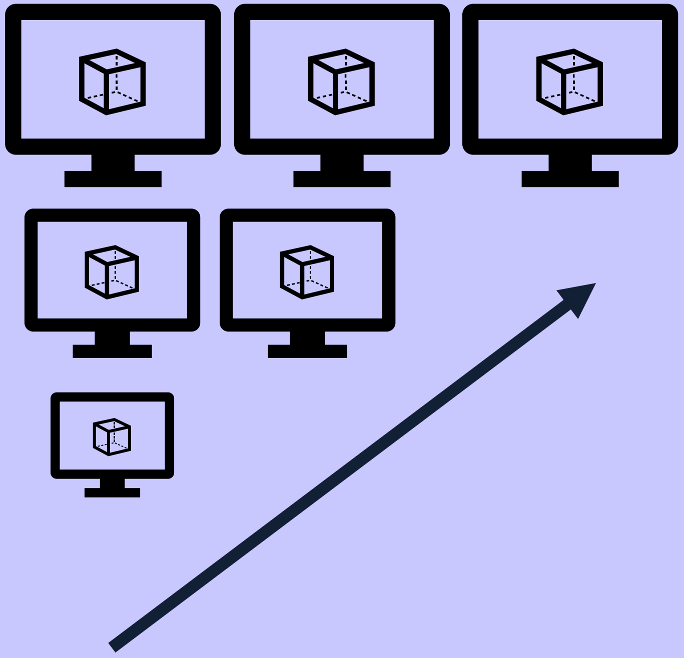
How can we serve more units of work?



Horizontal Scalability
Increase number of computing nodes



Vertical Scalability
Increase computing power of the computing node

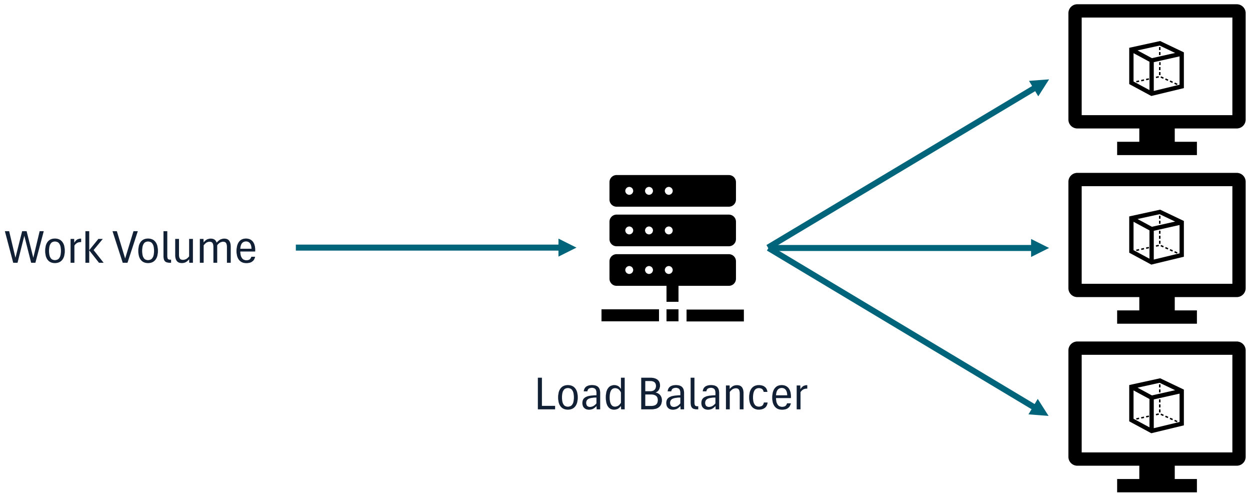


Diagonal Scalability
Increase both number and power of computing nodes

Load Balancing

- When scaling horizontally (or diagonally), a dedicated component distributing incoming requests to the computing nodes is necessary
- Such component is referred to as **Load Balancer**
- Load balancers can route traffic based on various strategies, including:
  - Random (incoming traffic randomly directed to one of the nodes)
  - Least loaded (traffic directed to the least loded node)
  - Round robin
  - Layer 4 (look at transport layer details (e.g.: source IP address, ...)
  - Layer 7 (look at application layer details (e.g.: cookies, sessions, ...)

Load Balancing



Things to keep in mind:

- Load Balancers can be a **performance bottleneck**!
- A single load balancer is a **single point of failure**!

Availability

Availability: definition

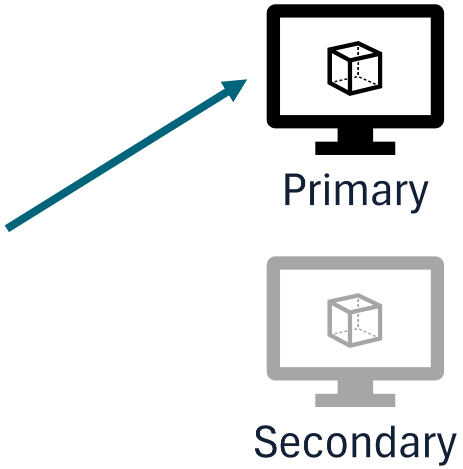
- **Availability** is the proportion of time a system or service is operational and accessible for use
- Availability is crucial for systems whose continuous operation is vital
  - Business-critical systems (financial losses, reputational damage, ...)
  - Safety-critical systems
- What can cause a system to become unavailable?
  - Software issues (the app stops responding)
  - Excessive workloads (system does not have enough computational resources)
  - Loss of network connection
  - Hardware faults

Availability: quantification

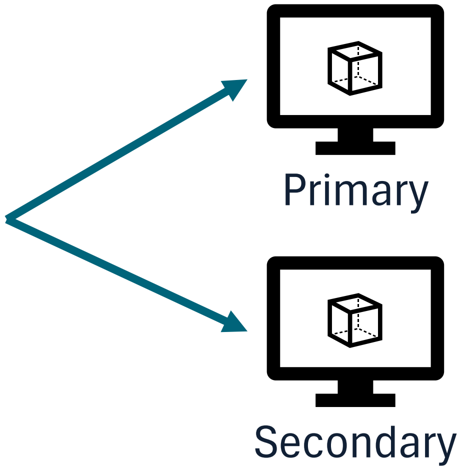
- Availability is often quantified by uptime/downtime as a percentage of time the service is available.
  - Availability is generally measured in number of 9s --a service with 99.99% availability is described as having four 9s.
    - **One nine**: ~36 days / year of acceptable downtime (https://uptime.is/one-nine)
    - **Two nines**: ~3 days 15h / year of acceptable downtime (https://uptime.is/two-nines)
    - **Three nines**: ~8h 41m / year of acceptable downtime (https://uptime.is/three-nines)
    - **Four nines**: ~52m / year of acceptable downtime (https://uptime.is/four-nines)
    - **Five nines**: ~5m / year of acceptable downtime (https://uptime.is/five-nines)
    - **Six nines**: ~31s / year of acceptable downtime (https://uptime.is/six-nines)

How do we achieve high availability?

- **Active-Passive Fail-over**
  - In case of failure of a (primary) component, a different (secondary) instance of that component takes on its work
  - The secondary instance is generally **passive**. It does no work at all unless the primary instance fails.



- **Active-Active Fail-over**
  - Two or more instances of the component exist in the first place, and they all do a share of the work (as in horizontal scaling)
  - If one component fails, the other takes on also its share of the work



Fail-over: Observability and Monitoring

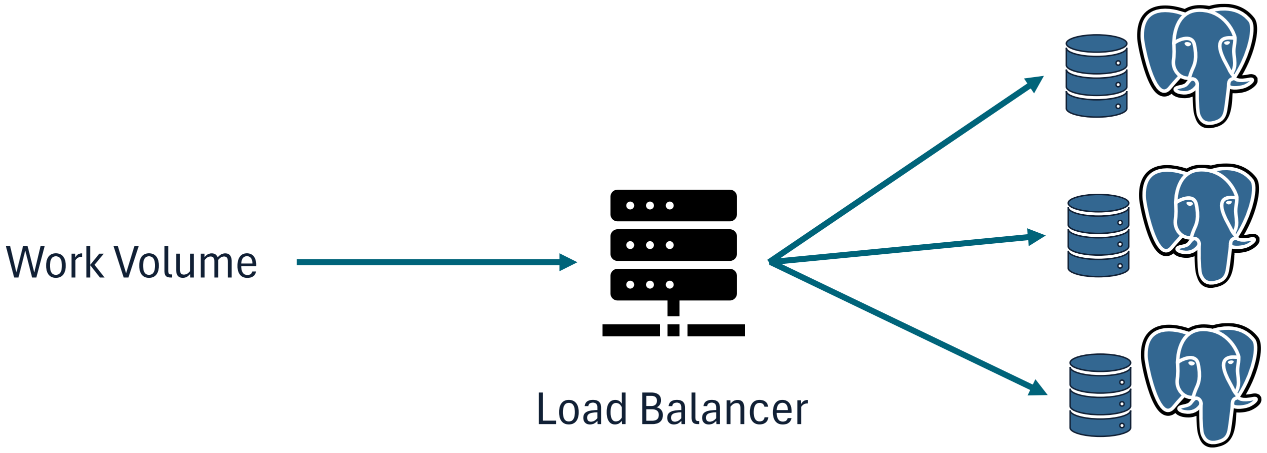
With fail-over approaches, downtime depends on several factors:

- How fast do we notice that a component has failed?
  - How do we even notice that a component has failed?
    - This is typically done using «**heartbeats**»: Periodic «signals» or «checks» to verify normal operation of the system
  - **Observability** (the degree to which we can understand what is going on in a software system) is an important characteristic of a software architecture!
    - Solutions like **OpenTelemetry** are often used
- How much time does the secondary instance take to start?
  - We can have the secondary on a «hot» standby or in a «cold» standby!
  - In active-active failover, secondary instances are already up and running

Data and Consistency

What about scaling data?

- Horizontally scaling application instances was relatively easy
  - Just get a new server with the application on it, and get a load balancer
- Most applications, however, also need data persistence
- What if we need to scale data in a **Relational Database** as well?
  - Not as easy!



RDBMS: ACID properties

ACID is a set of properties of relational database transactions.

- **Atomicity** - Each transaction is all or nothing
- **Consistency** - Any transaction will bring the database from one valid state to another
- **Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially
- **Durability** - Once a transaction has been committed, it will remain so

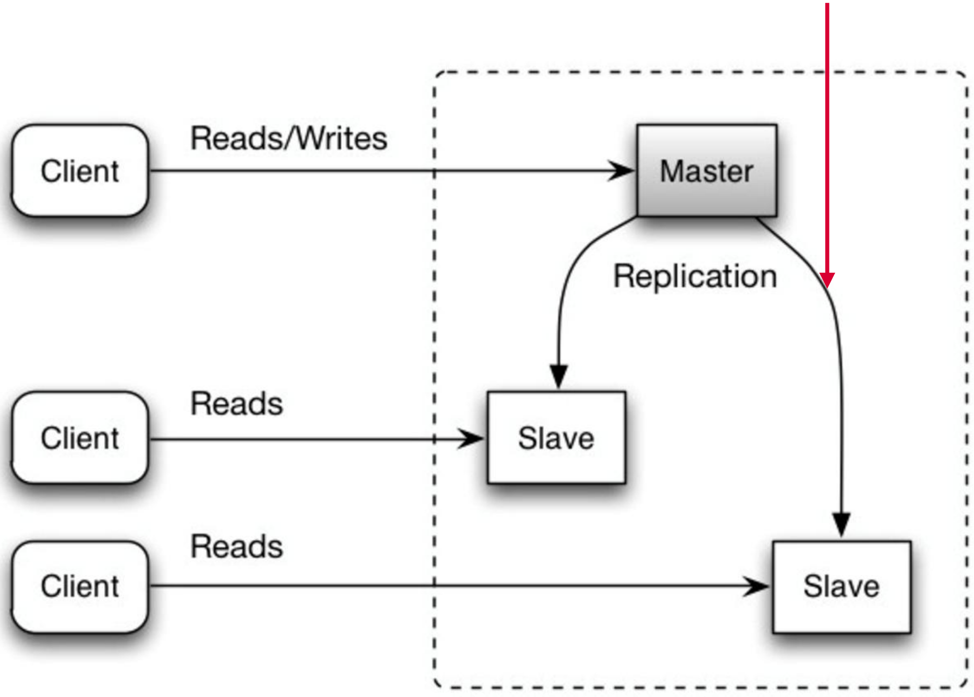
RDBMS Scaling

- The ACID properties make it impossible to simply scale a RDBMS horizontally
- How do we scale a RDBMS then?
  - We can scale it **vertically** (throw more computing resources at it)
  - We can use specific **replication** techniques
    - Master-slave and Master-master
    - Federation and sharding
    - Denormalization

RDBMS Replication: Master-Slave

- Multiple RDBMS instances
  - One master and one or more slaves
- Master serves reads and writes, replicating all writes on each slave
- Slaves act as **read-only replicas**
- Availability: If master fails, the system can still operate in read-only mode (a Slave may be eventually promoted)
- This approach works well when there are many reads and fewer writes

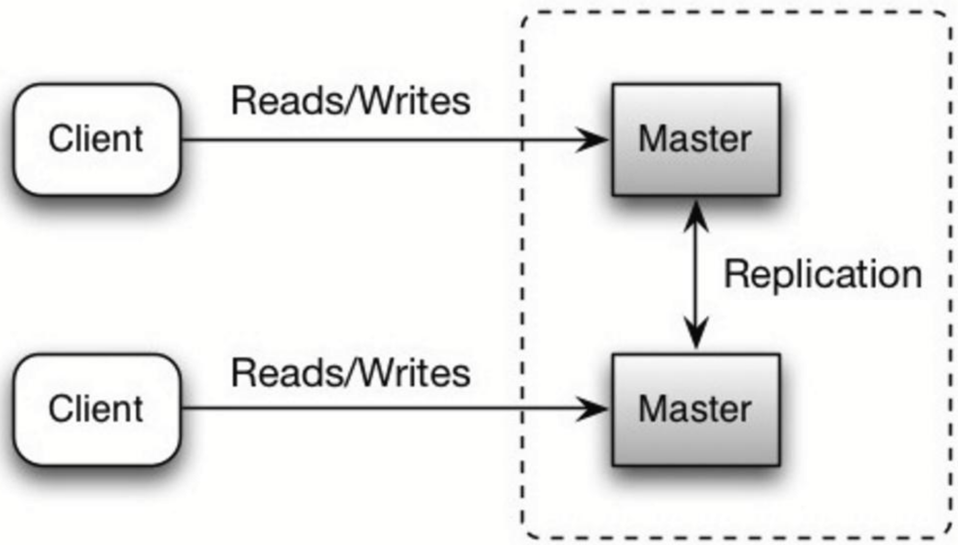
Introduce overhead!



Source: Scalability, availability, stability, patterns

RDBMS Replication: Master-Master

- All RDBMS instances are masters and serve both reads and writes
- For each write, the masters coordinate and synchronize
- Reads are handled independently
- Availability: If one master fails, the system can still operate with the other
- Synchronization introduces an overhead!



Source: Scalability, availability, stability, patterns

RDBMS Replication: Federation

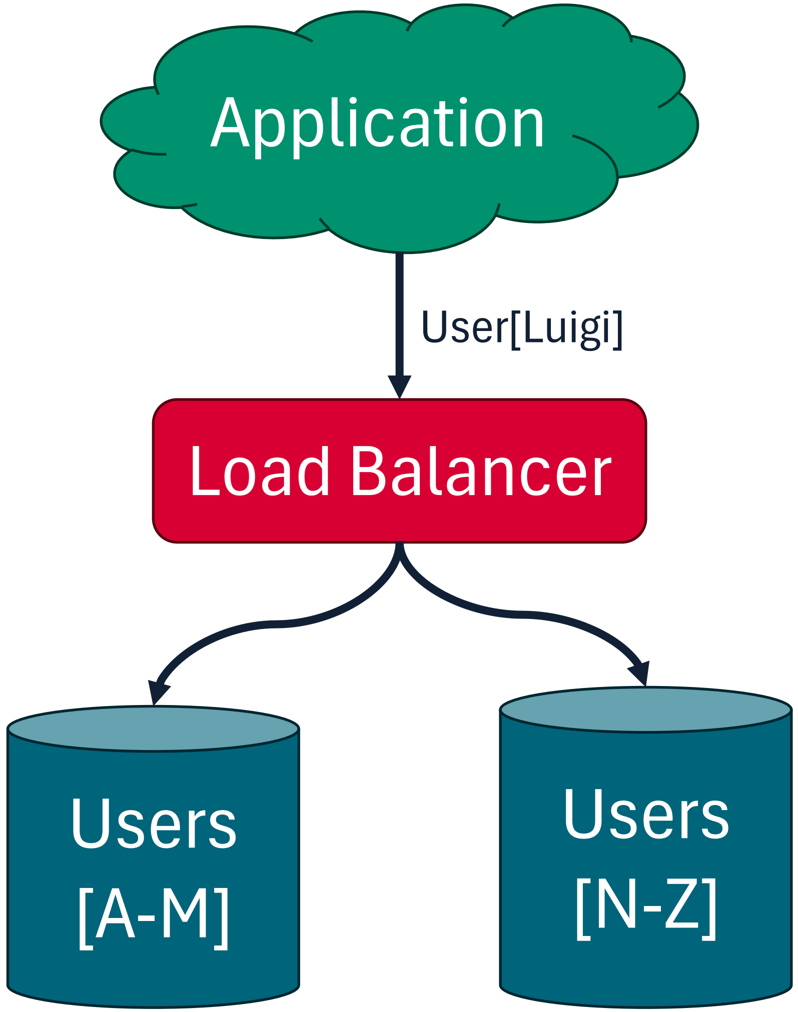
A single large database may be split into multiple databases by function

- For example, instead of a single, monolithic database, you may have:
  - Products database
  - Users database
  - Orders
- This has a few drawbacks:
  - Not effective if your schema includes a very large table
  - More difficult to join data
  - More difficult to enforce cross-table constraints (e.g.: foreign keys)
  - Might require de-normalization

RDBMS Replication: Sharding

Sharding distributes data across multiple databases in such a way that each database only manages a subset of the data

- Application logic becomes more complex to deal with shards
- Shards may become unbalanced
- Joining data is difficult
- Enforcing constraints is more difficult



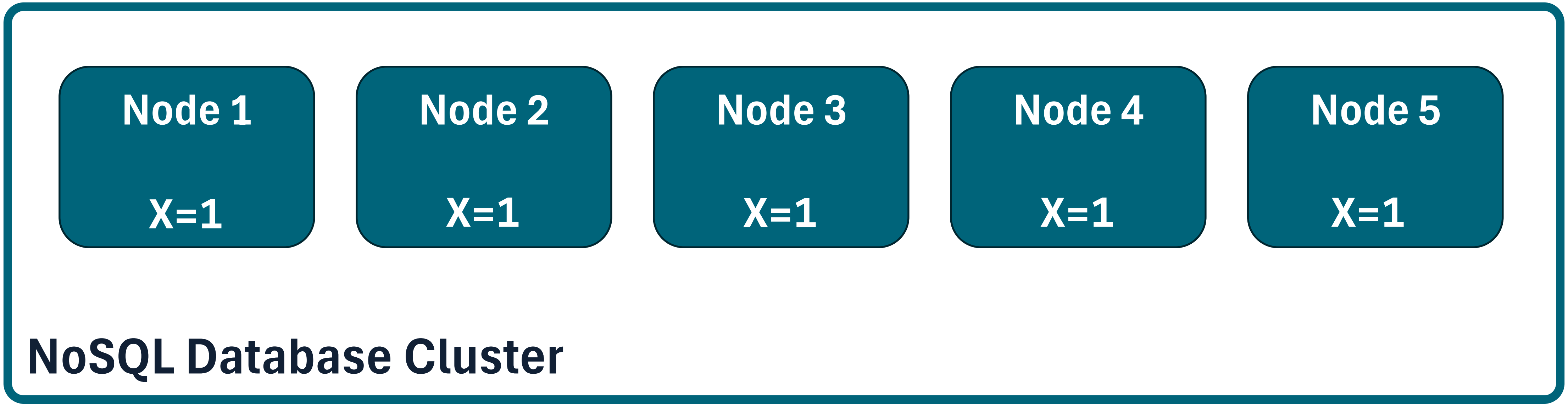
RDBMS Replication: Denormalization

- Denormalization attempts to improve read performance at the expense of some write performance (and loss of normalization, of course!).
- Redundant copies of the data are written in multiple tables to avoid expensive joins.
- Keeping data consistent might be an issue!
- Some RDBMS such as PostgreSQL support **materialized views** which handle the work of storing redundant information and keeping redundant copies consistent.

NoSQL databases

- NoSQL database models are more scalable, at the cost of renouncing true ACID transactions (and some querying performance)
- **BASE** is often used to describe the properties of NoSQL databases.
  - **Basically Available** - the system guarantees availability.
  - **Soft state** - the state of the system may change over time, even without input.
  - **Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

NoSQL Database: Eventual Consistency



Node 1

X=1

Node 2

X=1

Node 3

X=1

Node 4

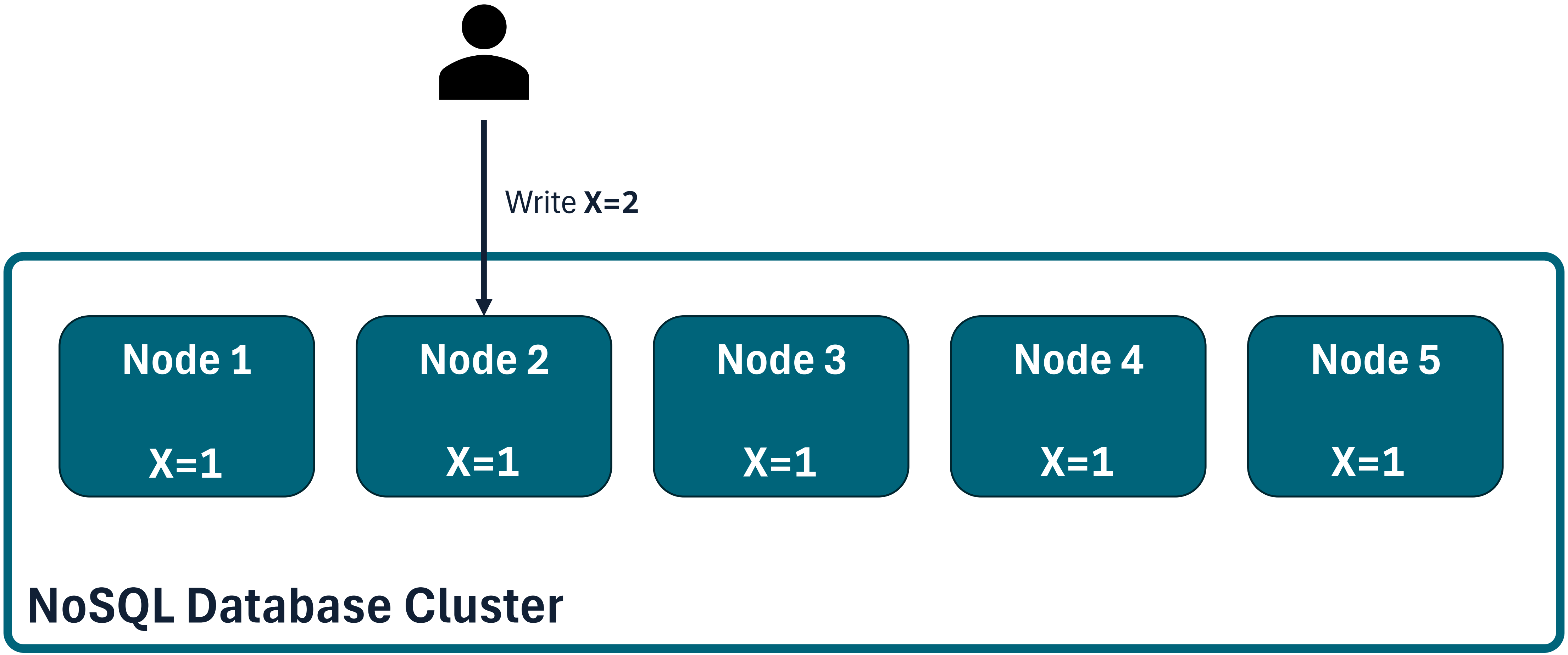
X=1

Node 5

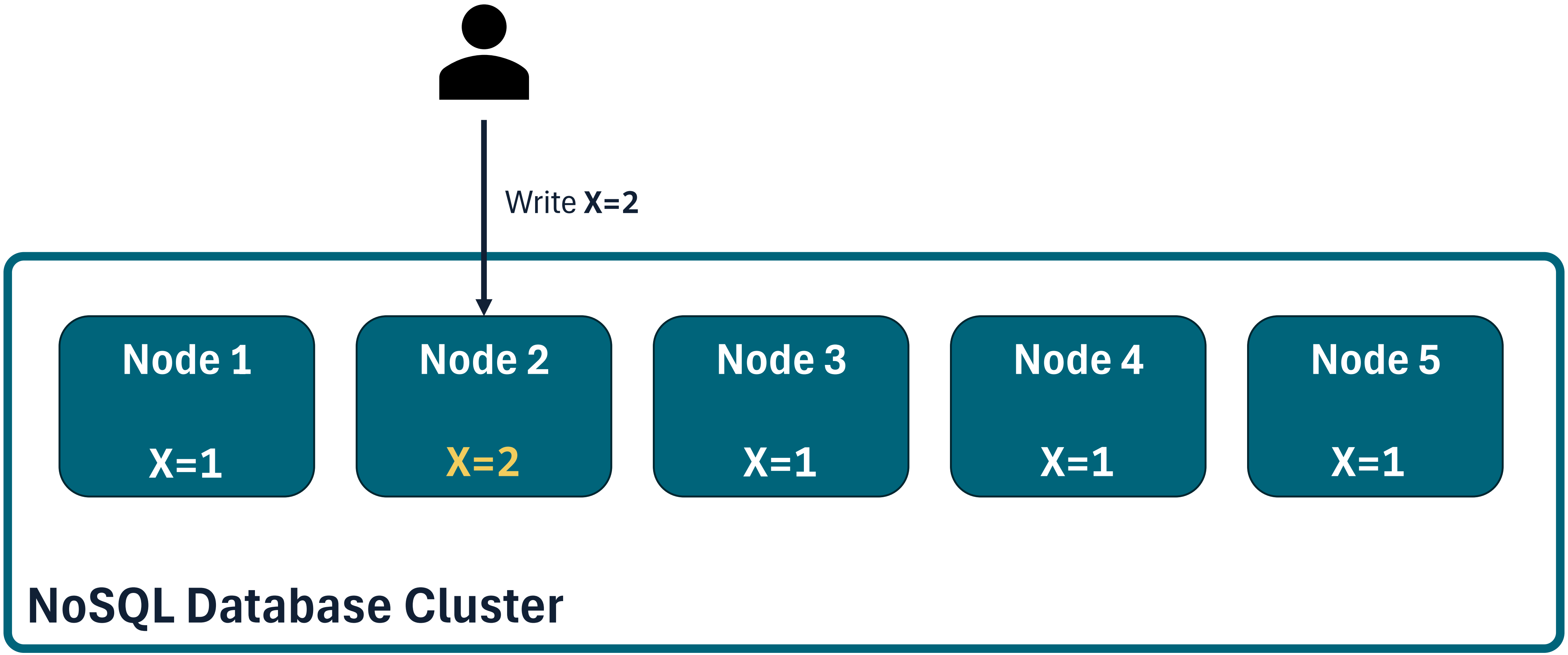
X=1

NoSQL Database Cluster

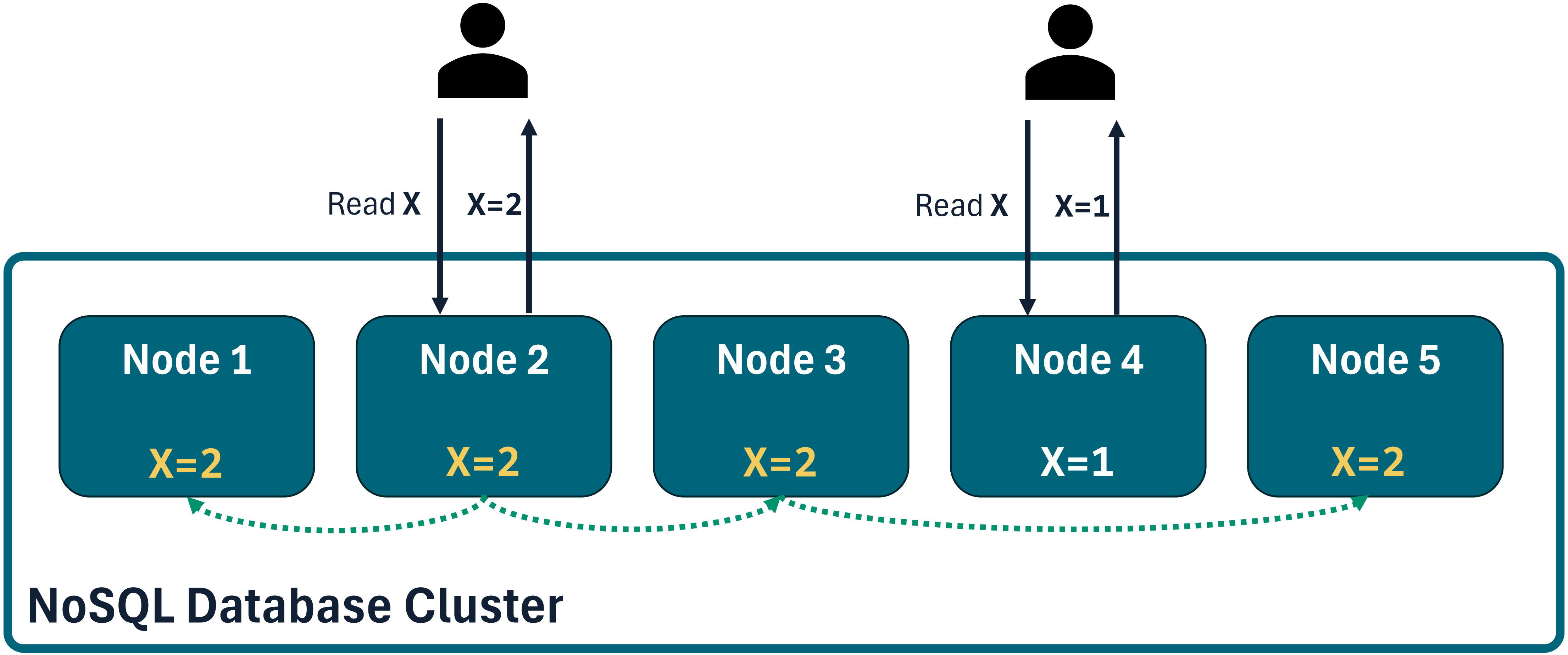
NoSQL Database: Eventual Consistency



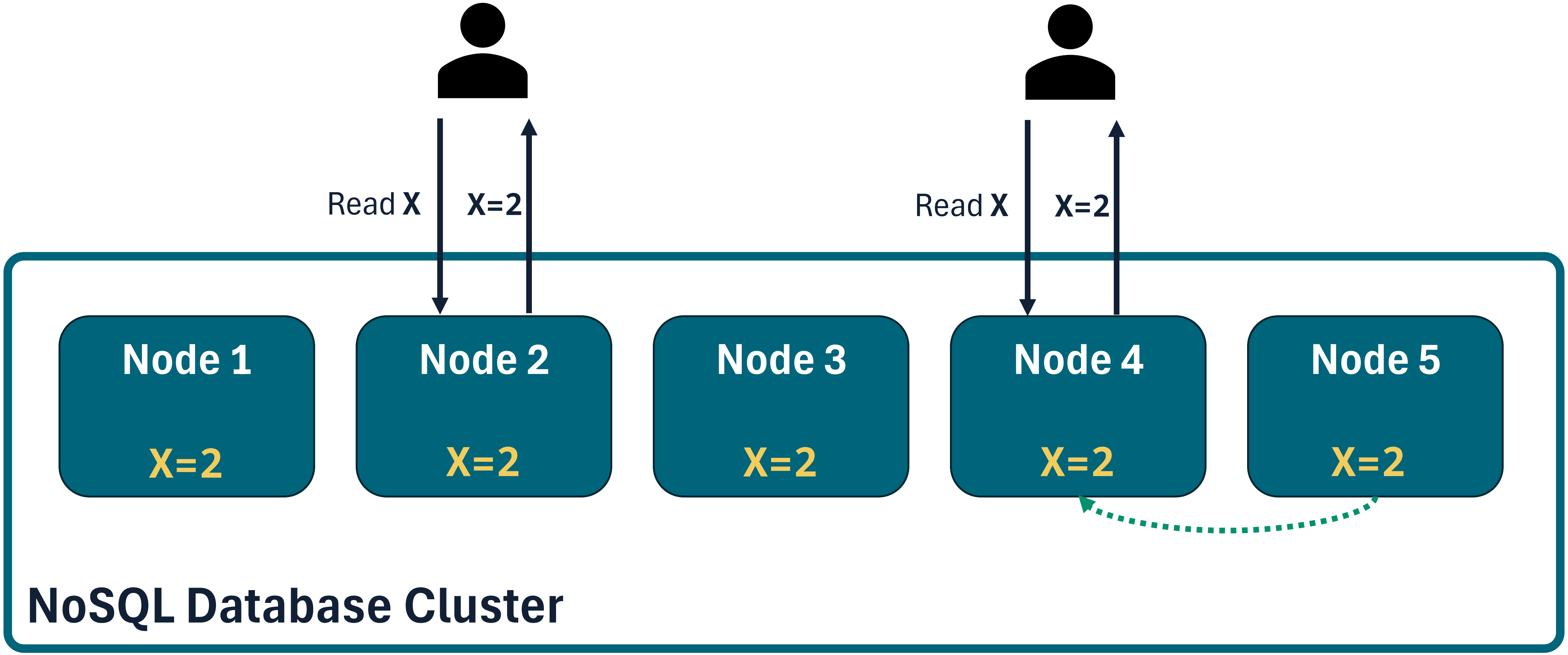
NoSQL Database: Eventual Consistency



NoSQL Database: Eventual Consistency

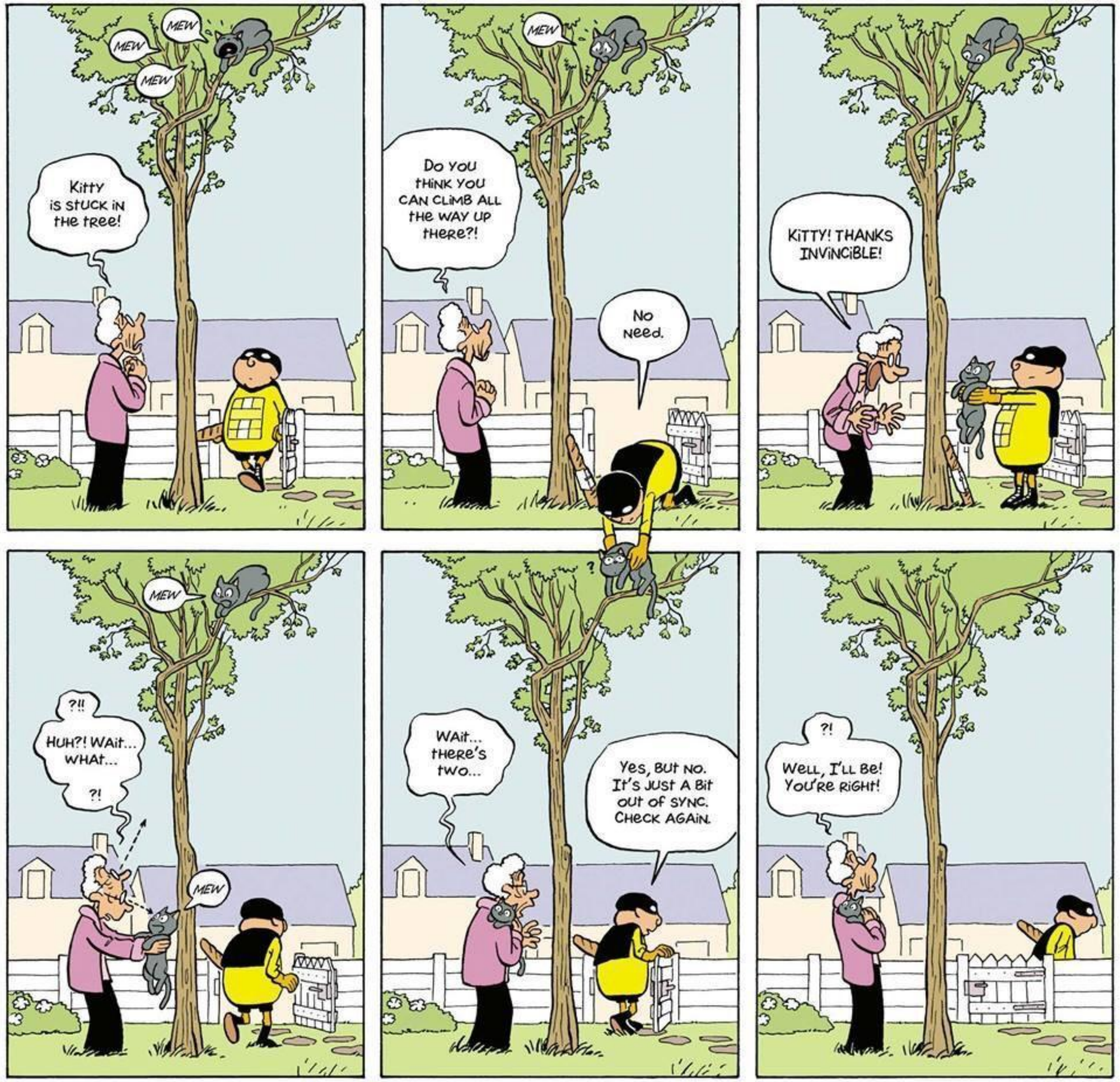


NoSQL Database: Eventual Consistency



Eventual Consistency

- Informal guarantee that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value


MEW
MEW
MEW
Kitty is stuck in the tree!
MEW
Do you think you can climb all the way up there?!
No need.
KITTY! THANKS INVINCIBLE!
MEW
?!!
HUH?! WAIT... WHAT...
?!
MEW
?
WAIT... THERE'S TWO...
YES, BUT NO. IT'S JUST A BIT OUT OF SYNC. CHECK AGAIN.
?!
WELL, I'LL BE! YOU'RE RIGHT!

Imbattable, Pascal Jousselin

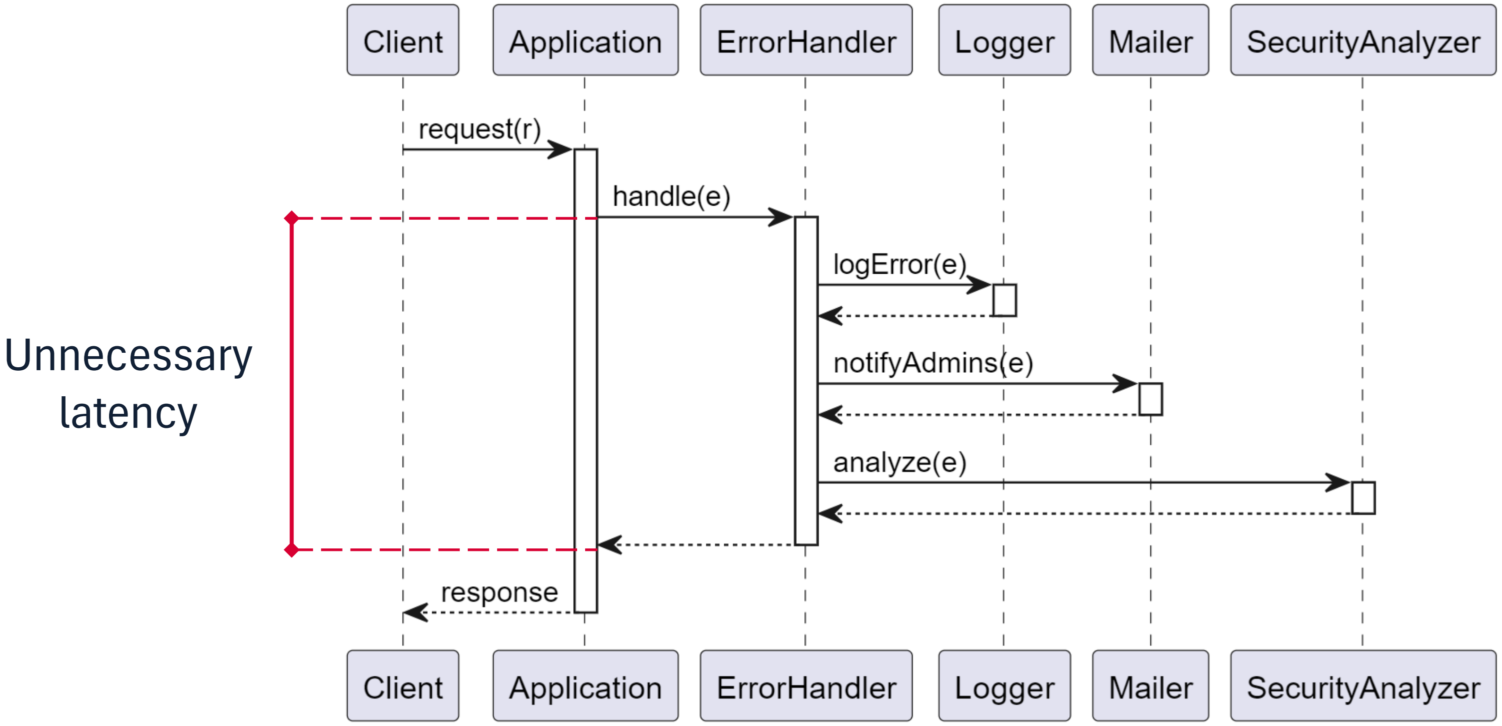
NoSQL databases

- Different kinds of NoSQL database models exist:
- **Key-value stores** (Redis or Memcached): typically operate in-memory
- **Document stores** (MongoDB): centered around documents (e.g.: XML, JSON, …) that store all information for a given object
- **Graph databases** (Neo4J): each node is a record and relations between records are represented as edges
- **Vector databases** (Milvus): specialized in storage and retrieval of large vectors, often coming from AI applications

Asynchronism

Need for Asynchronous Processing

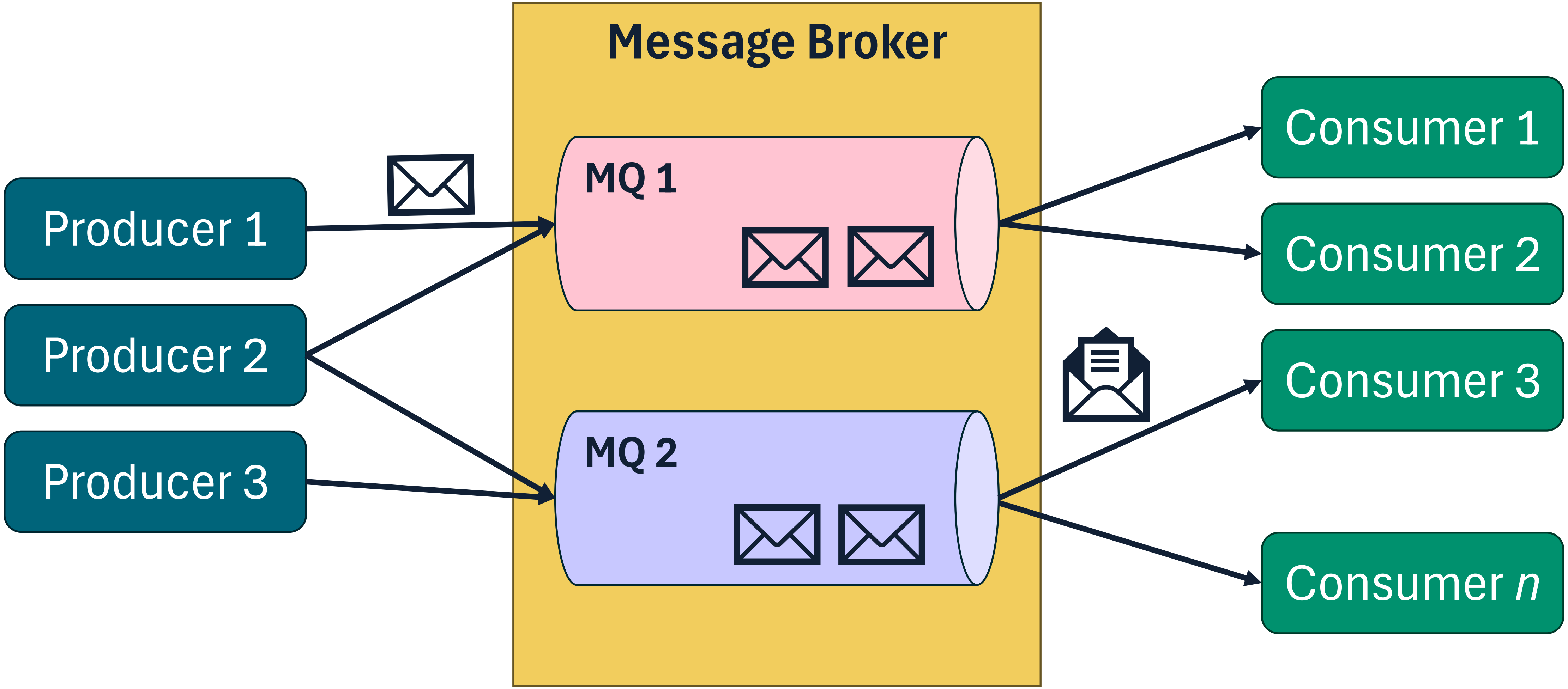
- As systems grow larger and larger, at some point it may be necessary to perform some processing that cannot be performed in-line when serving a request, because it would create unacceptable latency.



Asynchronous Workflows

- Asynchronism can help reduce request times
- **Message Queues** (MQs) are a way to implement asynchronous workflows
  - MQ are components designed to receive, hold, and deliver messages
- When a task is too slow to perform inline (synchronously):
  - An **application publishes** a message to a MQ, with details of the task to perform
  - A **worker** (asynchronously, possibly at a later time) act as a consumer and picks up the message from the MQ, and actually performs the task
  - The user is not blocked waiting for the long task to completed
  - The task is effectively completed in background, when one of possibly many workers becomes available

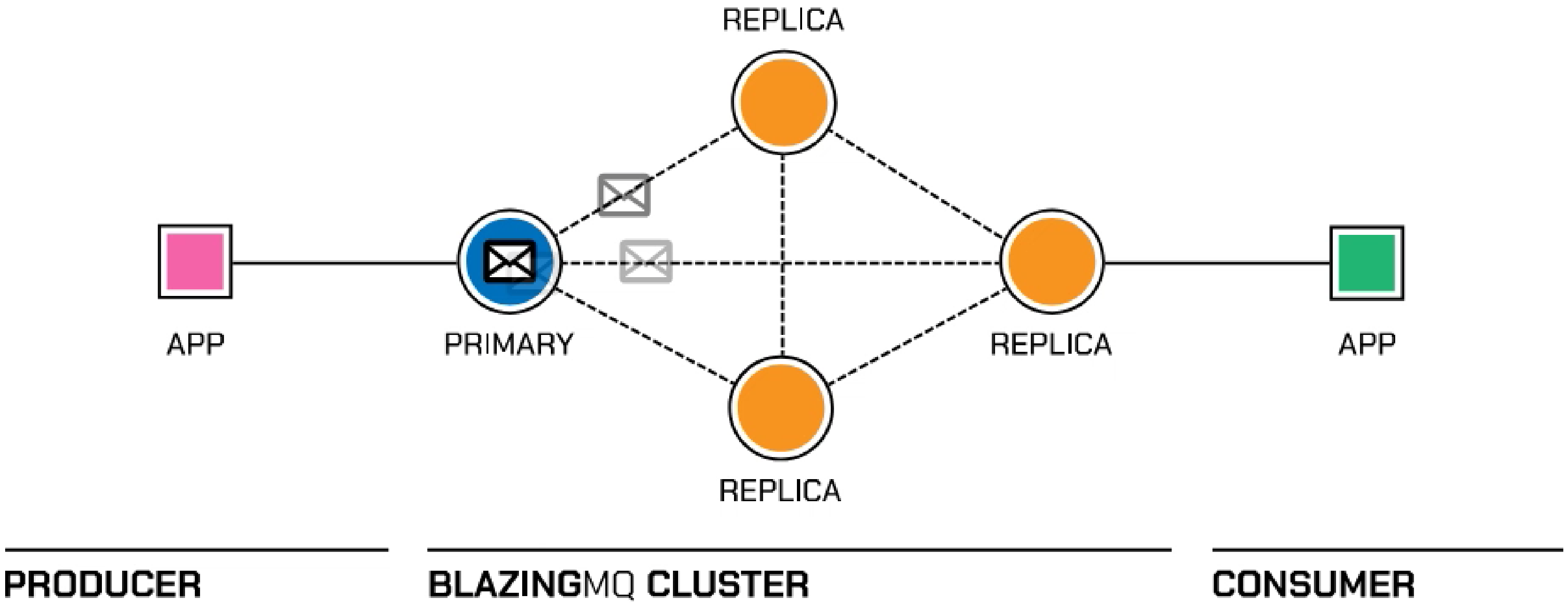
Message Queues and Message Brokers



Message Brokers

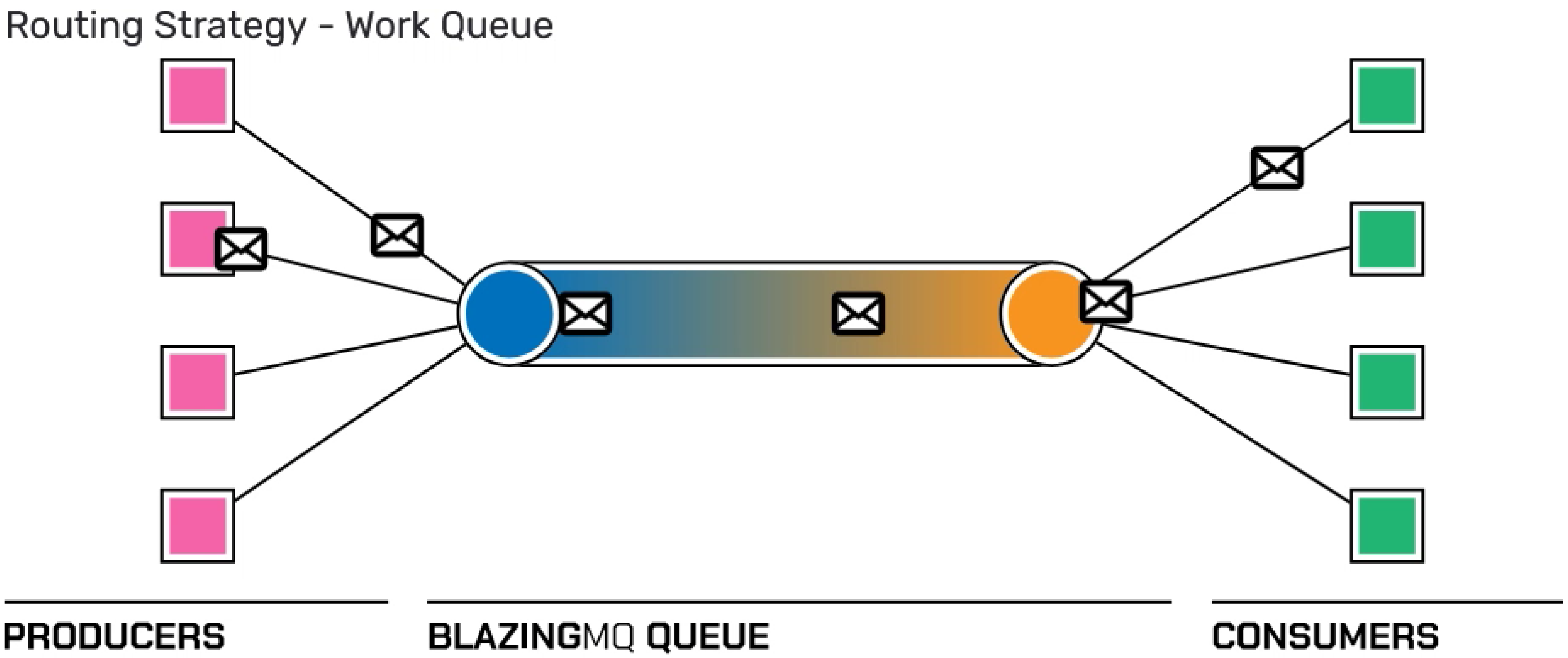
- Popular open-source message brokers include:
    - RabbitMQ, Apache ActiveMQ, BlazingMQ
- Note that Message Brokers are not trivial objects!
    - They are typically internally organized in complex, distributed architectures
    - Message routing strategies may be more complex than simple polling by consumers
    - Different Message Brokers and different configurations may guarantee different properties (e.g.: message delivery may or may not be guaranteed, or a message may be delivered to more than one consumer!)
        - **You need to carefully consider these aspects w.r.t. the problem at hand**

Message Brokers: Complexity



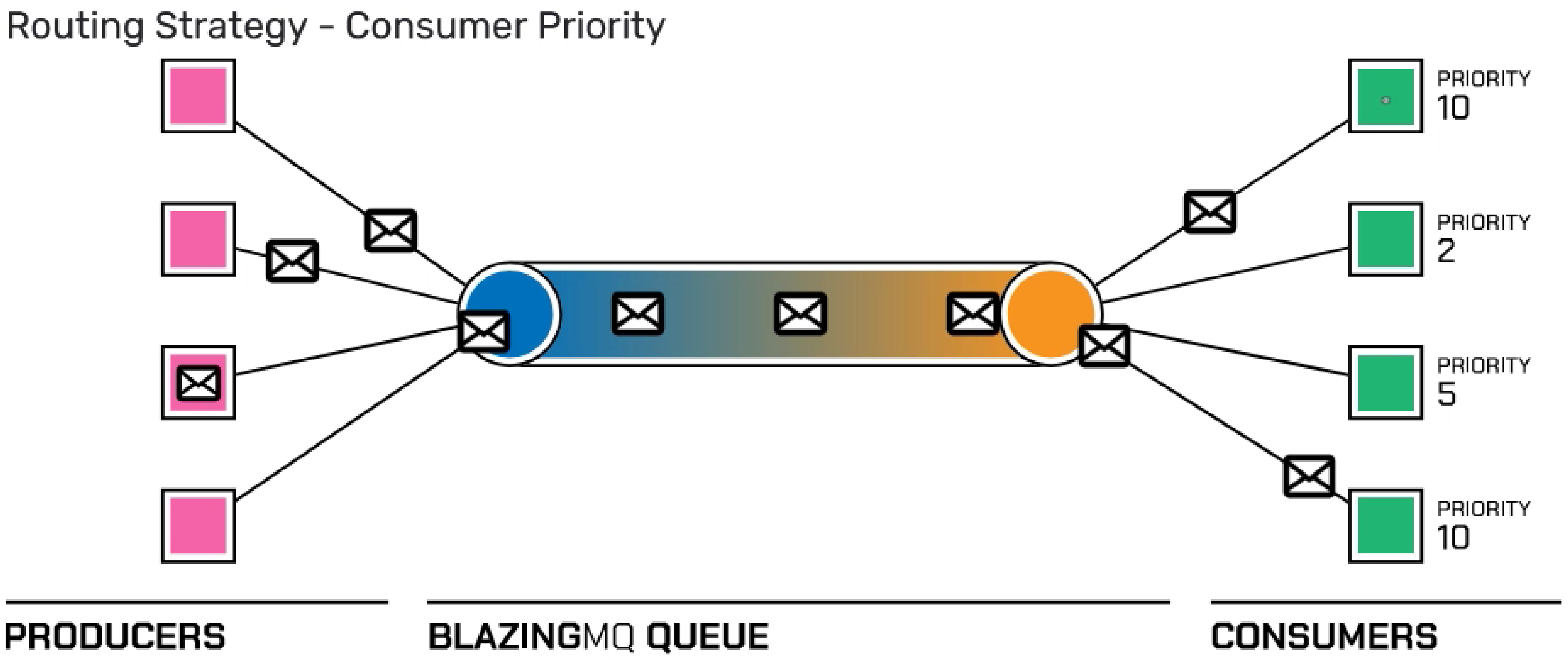
Animation from https://bloomberg.github.io/blazingmq/

Message Brokers: Complexity



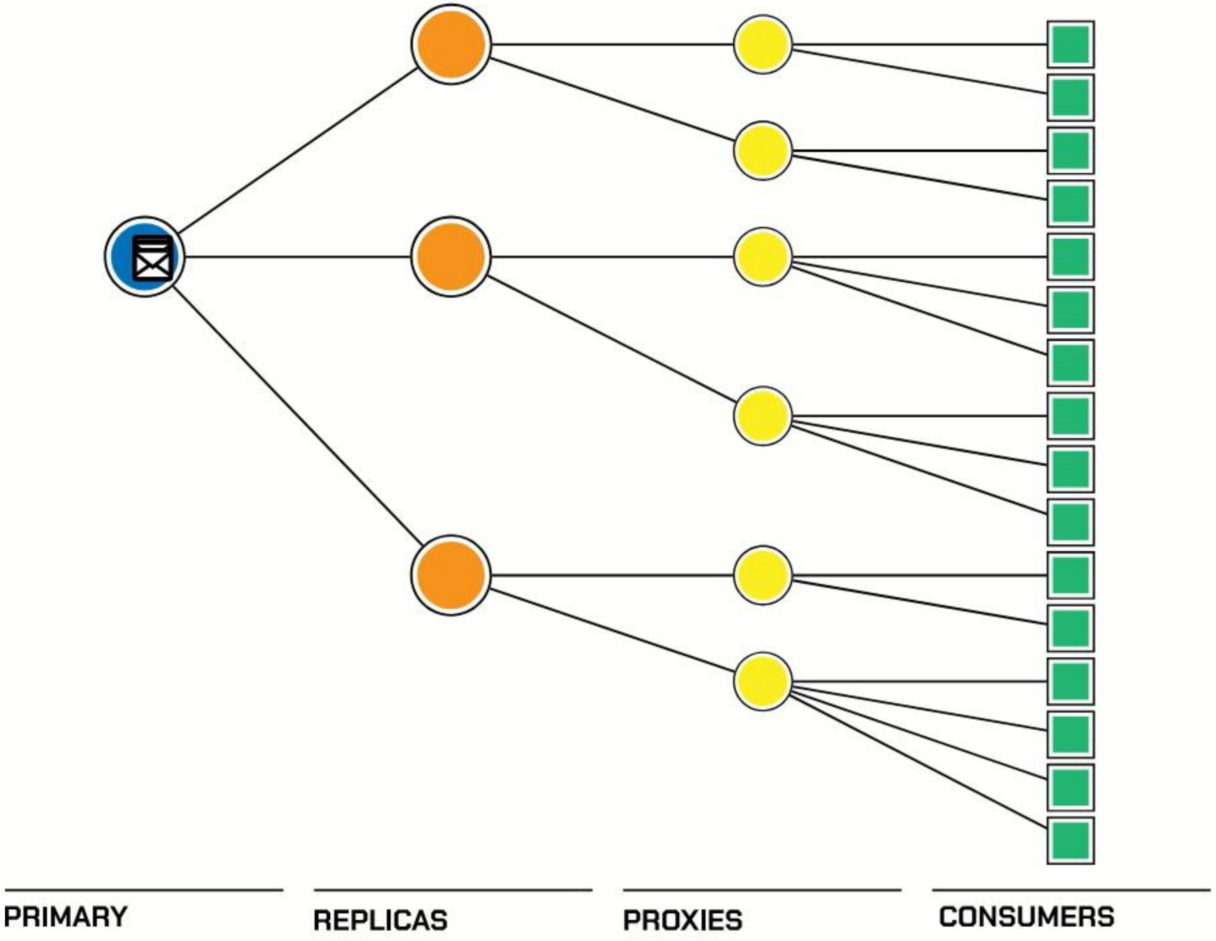
Animation from https://bloomberg.github.io/blazingmq/

Message Brokers: Complexity



Animation from https://bloomberg.github.io/blazingmq/

Message Brokers: Complexity



Animation from https://bloomberg.github.io/blazingmq/

Message Queues and Streams

- Some Message Brokers (e.g.: RabbitMQ) also support **streams**
- Streams can complete the same tasks as queues:
  - Buffering messages from producers that are later read by consumers
- However, streams differ from MQs in two important ways:
  - How messages are **stored**
    - Streams model an append-only log of messages that can be repeatedly read until they expire. Streams are always persistent and replicated. **Non-destructive consumer semantics**.
  - How messages are **consumed**
    - The same message can be read by multiple consumers as many times as they want
    - Consumers can attach themselves at any point of the stream

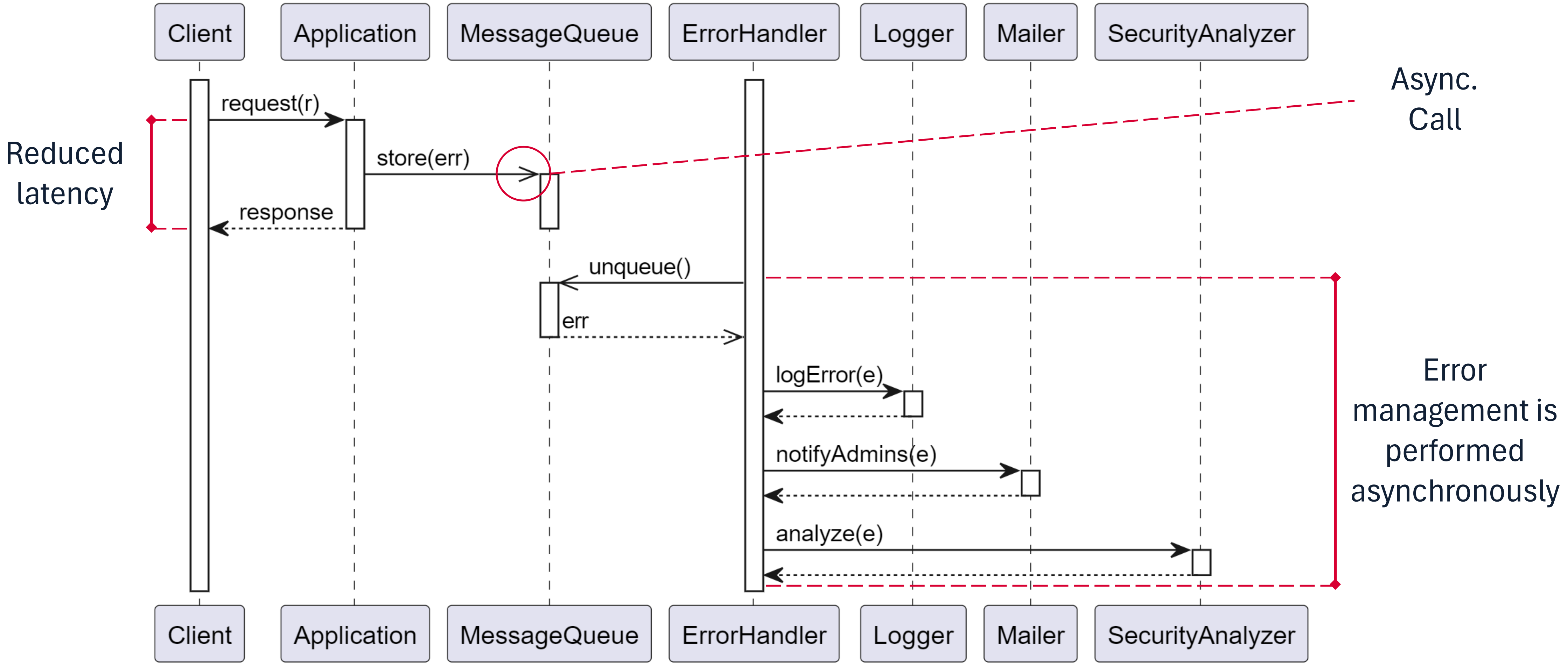
Some use cases for Streams

- **Large fan-outs**
    - To deliver the same message to multiple subscribers users currently have to bind a dedicated MQ for each consumer (which is potentially inefficient)
    - Streams allow any number of consumers to consume the same messages from the same MQ in a non-destructive manner, removing the need for multiple MQs.
- **Large backlogs**
    - MQ are designed and optimized to converge towards the empty state
    - Performance can degrate when there are millions of messages on a MQ.
    - Streams are designed to store larger amounts of data.
- **Replay**
    - Streams allow consumers to re-read messages

Possible scenario: Social Media App

- Upon posting a new picture on a Social Media App:
  - Different versions, optimized for different devices, of the picture need to be generated
  - Advanced AI-pipeline to detect faces and products in the picture is executed
  - A notification about the new picture is sent to all the followers of the user
- No need to make the user wait until all these tasks are completed!
  - Tasks could be published on dedicated MQ, and workers may pick up the tasks when they have time
  - During this time, the client might optionally do a small amount of processing to make it seem like the task has completed
    - E.g.: the picture might appear on the timeline of the user, before the picture is actually delivered to all the followers or converted to all necessary formats!

Asynchronism with a Message Queue



Reduced latency

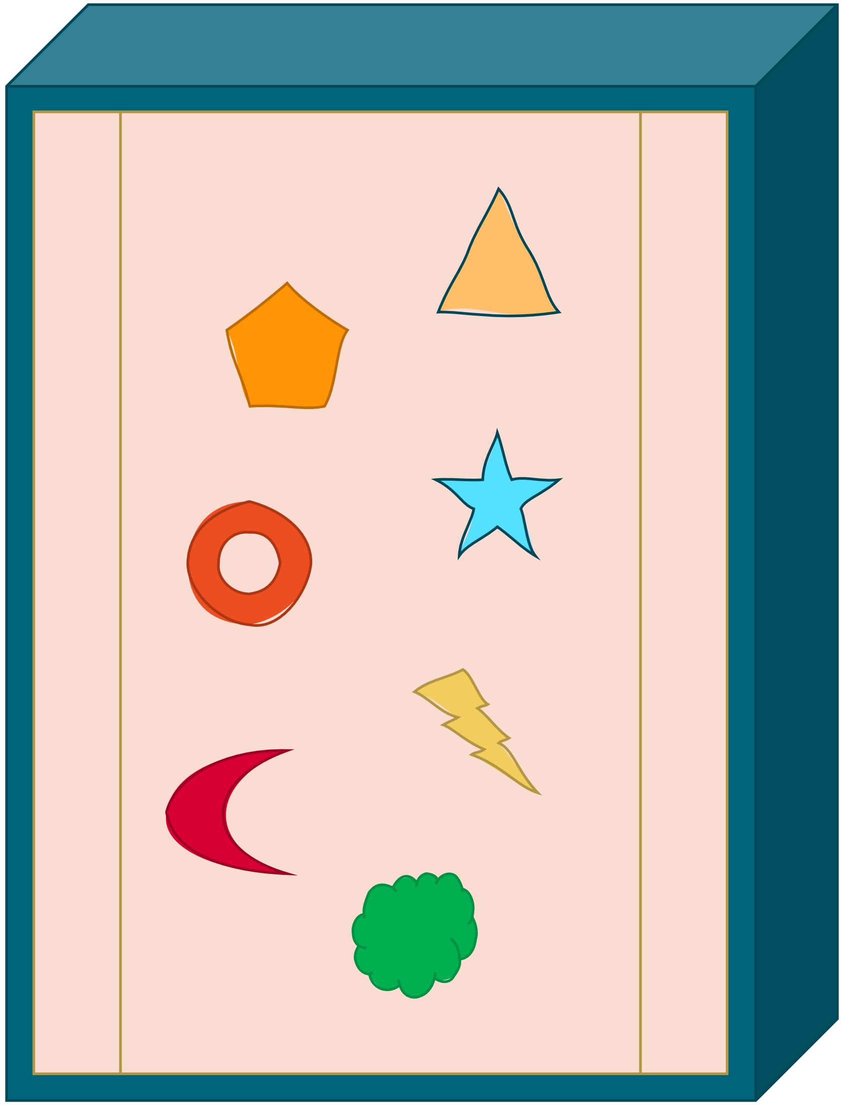
Async. Call

Error management is performed asynchronously

Microservices

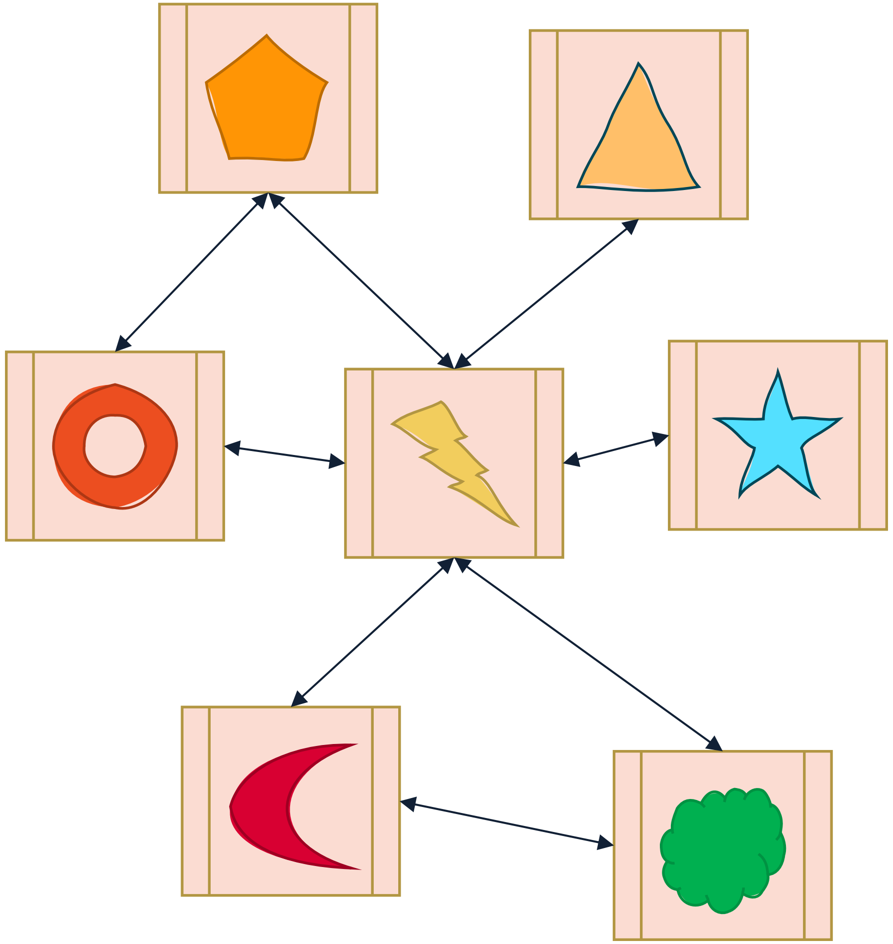
Monolithic Architecture

- Different components/modules tightly packaged in a **single unit**
- Single codebase
- Single, self-contained, deployment unit
- Business concerns are coupled

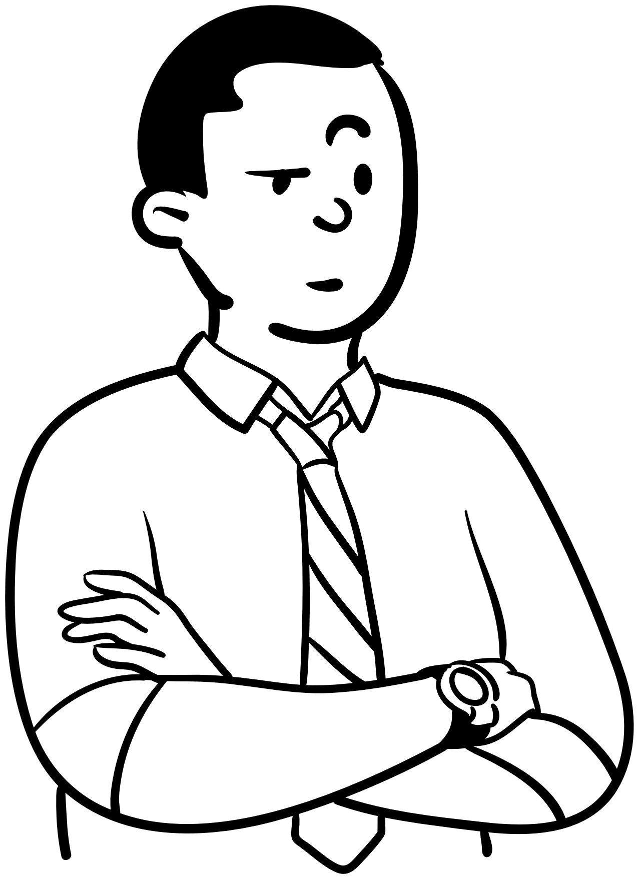


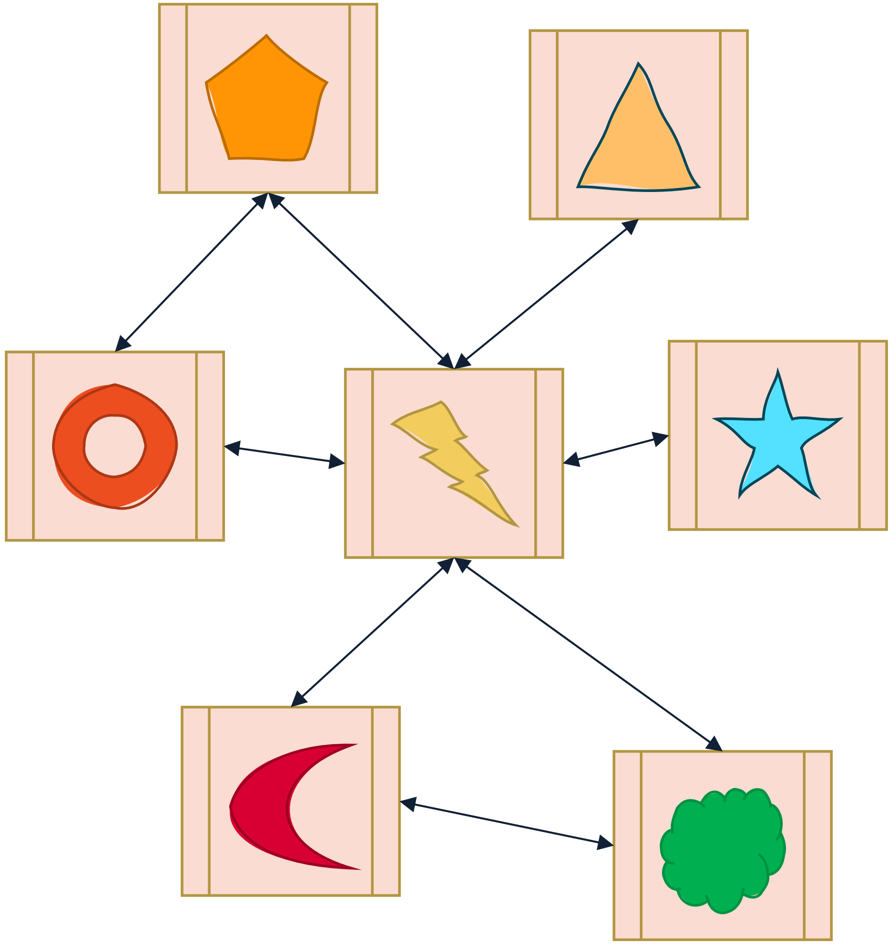
Microservice Architecture

- Software structured as a set of services
- Each service has its own codebase
- Possibly **polyglot**
- Each service is **independently deployable**
- Services organized around business capabilies
- Services are **loosely coupled**

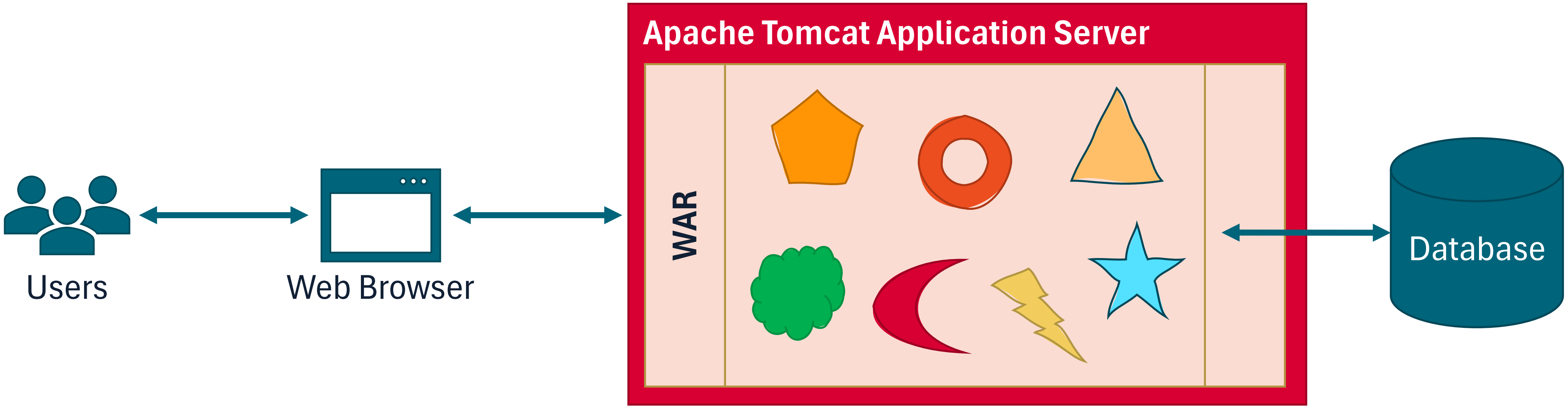


Microservices: Why on hell would I do that?!





Monolithic Architecture

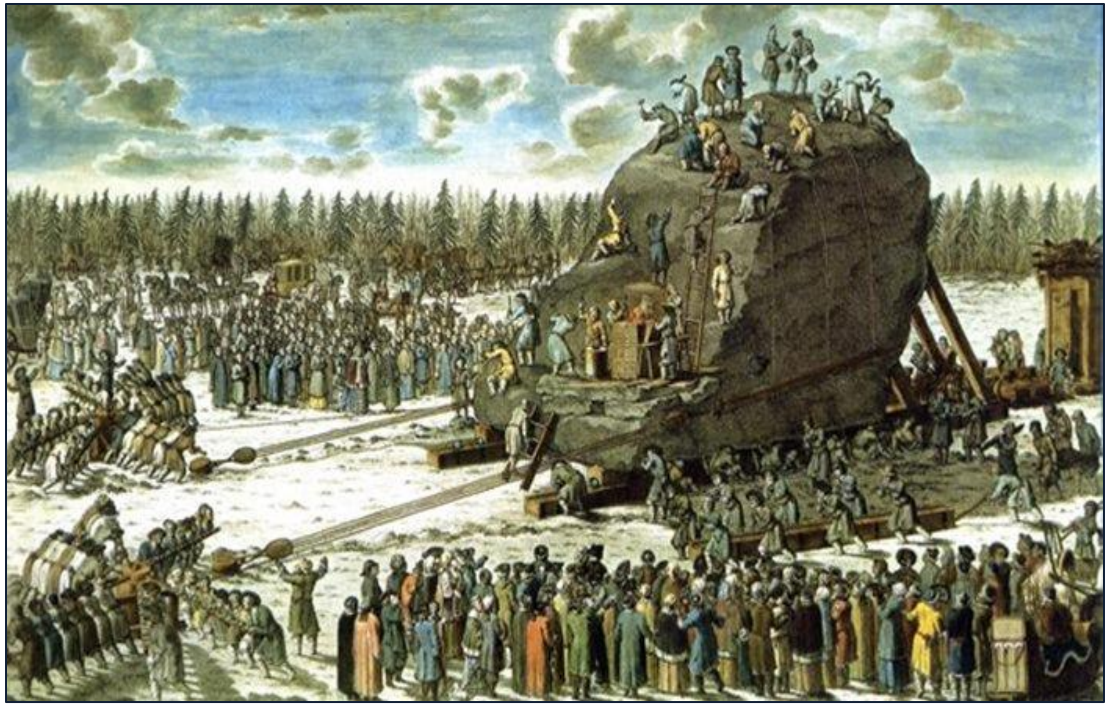


At the beginning (or for software with low complexity):

- **Easy Deployments:** need to build just one artifact
- **Easy Development:** single codebase, easy data management

There’s Monolith and Monolith

AI-generated picture from stockcake.com

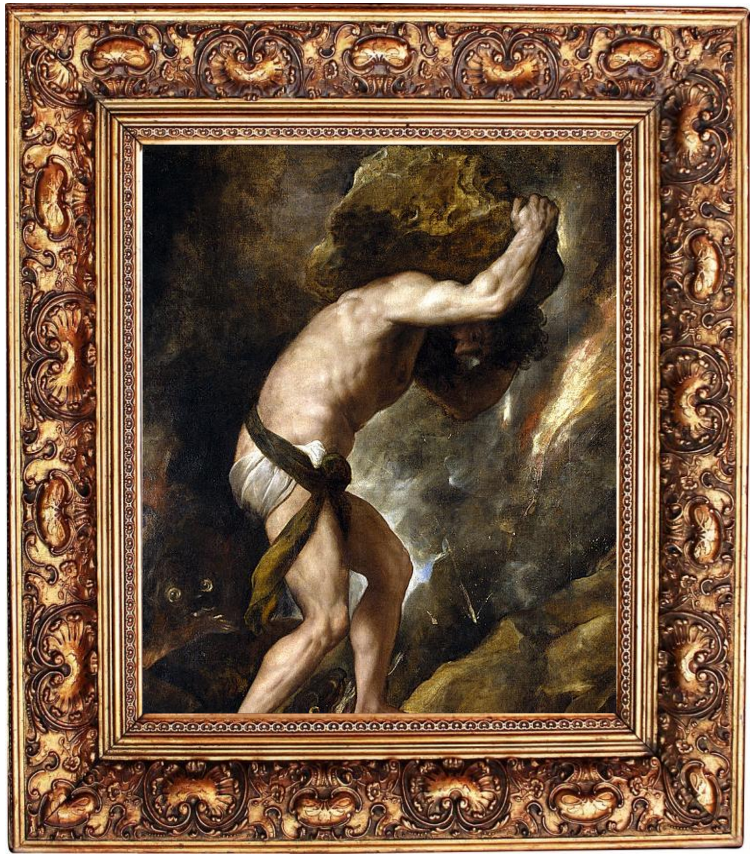


The Transportation of the Thunder-stone in the Presence of Catherine II; Engraving by I. F. Schley of the drawing by Yury Felten, 1770

Challenges with Monolithic Software (1/2)

When a Monolith increases in complexity:

- Inner architecture becomes harder to maintain and evolve
- New releases take longer and longer (months)
- Long time to add new features



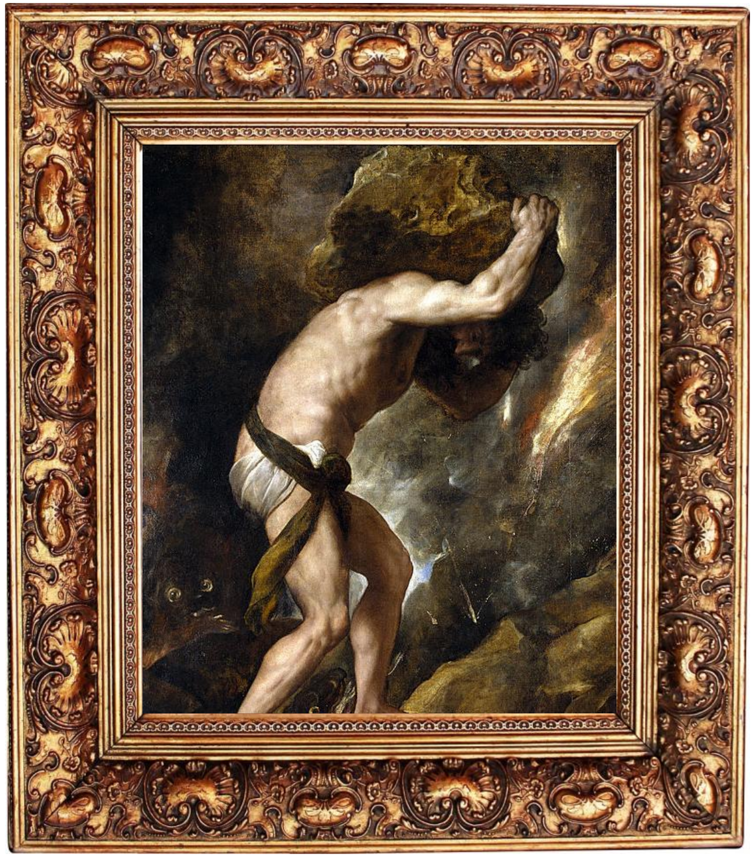
Sisyphus, Titian

Oil painting, 1548 ca.

Museo del Prado, Madrid

Challenges with Monolithic Software (2/2)

- Long and increasingly complex build/test/release lifecycle
  - Who broke the build?
- Deployments become increasingly difficult
  - Who's the owner of the failing module?
- Lack of innovation



Sisyphus, Titian

Oil painting, 1548 ca.

Museo del Prado, Madrid

Microservices: Motivations

- Today’s world is **volatile** and **changes rapidly**
- Businesses must be **agile** and **innovate faster**
- Software must be delivered **rapidly**, **frequently**, and **reliably**

DevOps Research and Assessment (DORA) metrics (link):	
Deployment Frequency	How often releases to production are made
Lead Time for Changes	How much time it takes a commit to get into production
Changes Failure Rate	Percentage of deployments failing in production
Time to Restore Service	How long it takes to recover from a failure in production

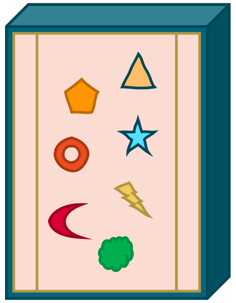
Microservices: Handling Complexity

Splitting the Monolith in a number of Microservices can help make larger systems manageable.

- Each Microservice is significantly smaller than the entire system
  - Easier to understand, to maintain, to evolve, to test…
- Each Microservice can be built and deployed independently
- New features should likely impact a single Microservice
- More frequent deployments (redeploy a single Microservice to add a new feature)

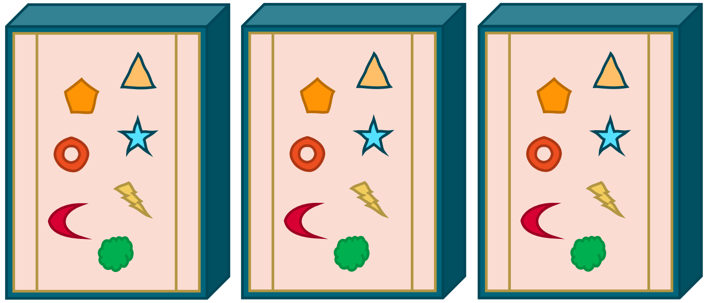
Monolith vs Microservices: Scaling

- Monoliths put all functionality in a single process and can scale by **replicating** the monolith on multiple servers



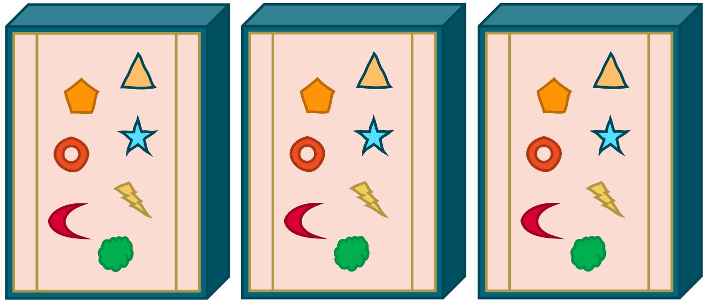
Monolith vs Microservices: Scaling

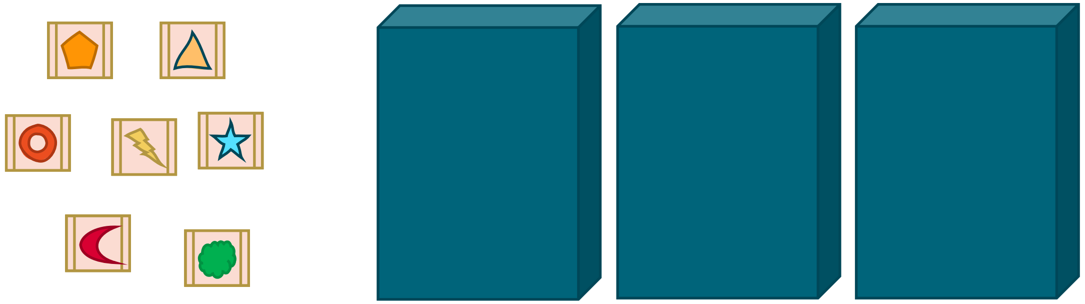
- Monoliths put all functionality in a single process and can scale by **replicating** the monolith on multiple servers



Monolith vs Microservices: Scaling

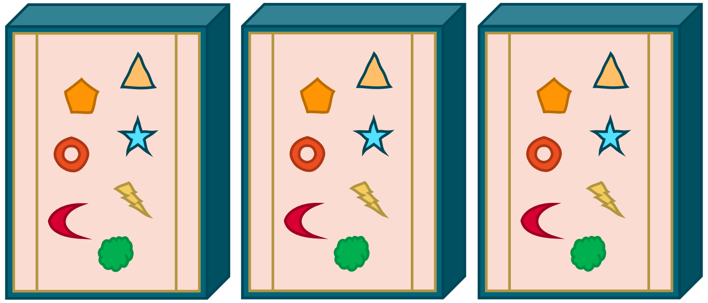
- Monoliths put all functionality in a single process and can scale by **replicating** the monolith on multiple servers
- Microservices put each functionality in a distinct process, and scale by **distributing** services on different servers, replicating as needed

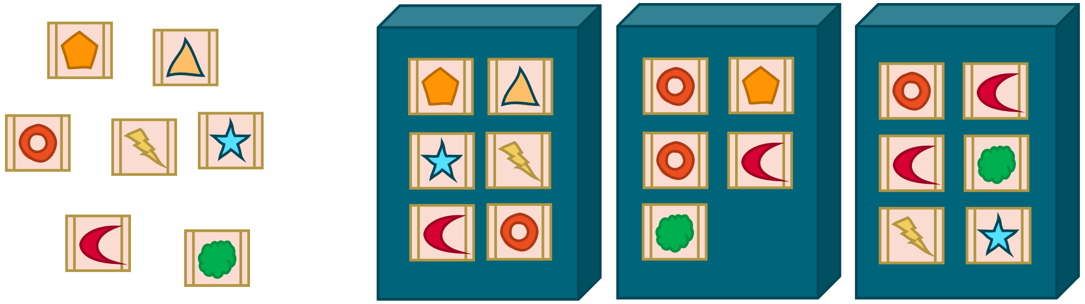




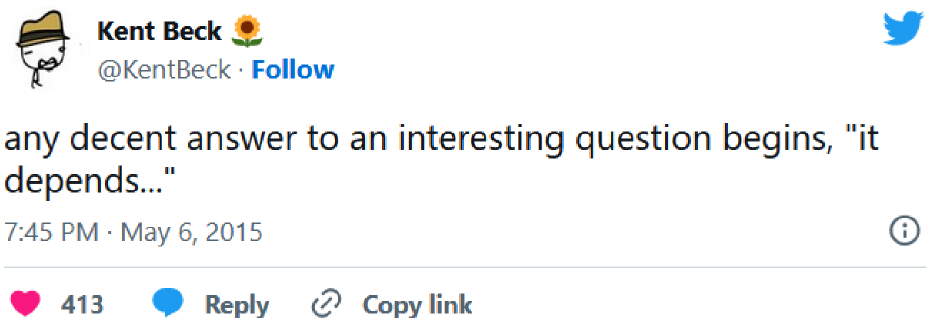
Monolith vs Microservices: Scaling

- Monoliths put all functionality in a single process and can scale by **replicating** the monolith on multiple servers
- Microservices put each functionality in a distinct process, and scale by **distributing** services on different servers, replicating as needed





Should I use a Microservice Architecture?



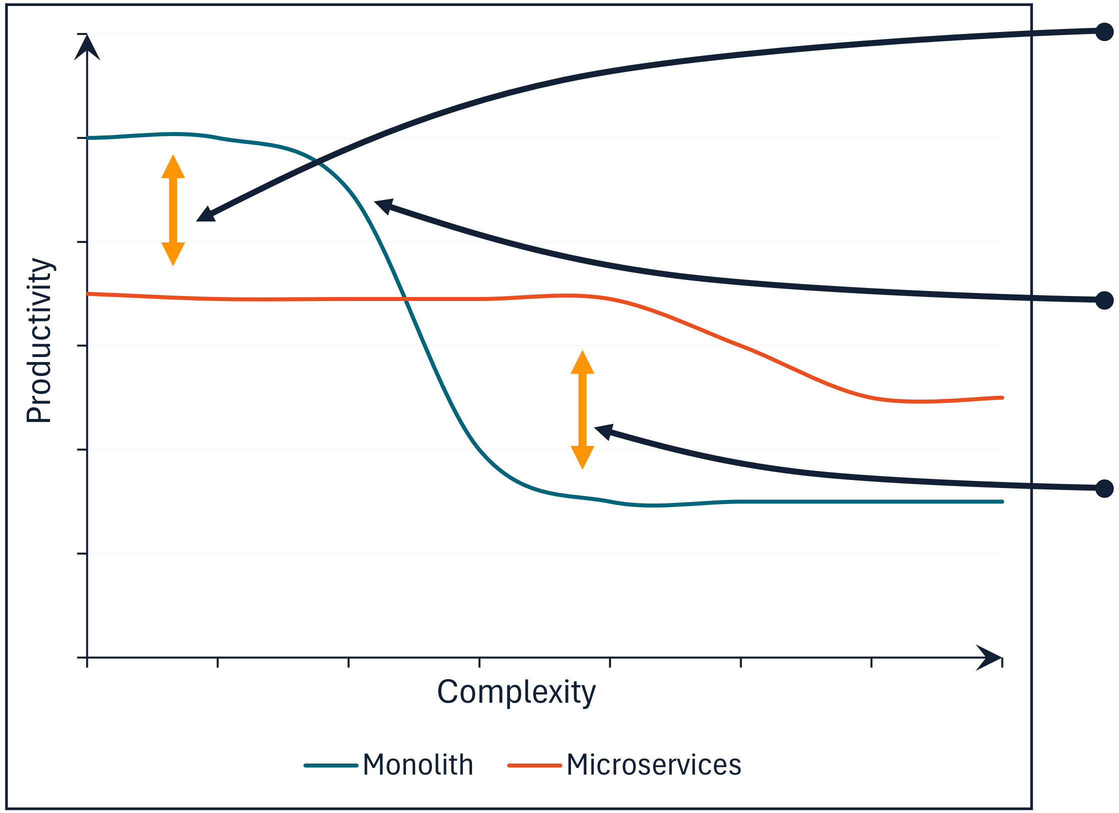
Kent Beck 🌻
@KentBeck · Follow

any decent answer to an interesting question begins, "it depends..."

7:45 PM · May 6, 2015

- Don’t even consider it unless the system is **complex**
- The complexity that drives us to microservices can come from:
  - Sheer size, dealing with large development teams
  - Multi-tenancy
  - Supporting different user interaction models
  - Need for scaling

Monoliths vs Microservices: Productivity



- For less-complex systems, the overhead of microservices reduces productivity
- Complexity kicks in, Monolith productivity drops
- Decreased coupling of microservices reduces the attenuation of productivity

Source: https://www.martinfowler.com/bliki/MicroservicePremium.html

Adopting Microservices

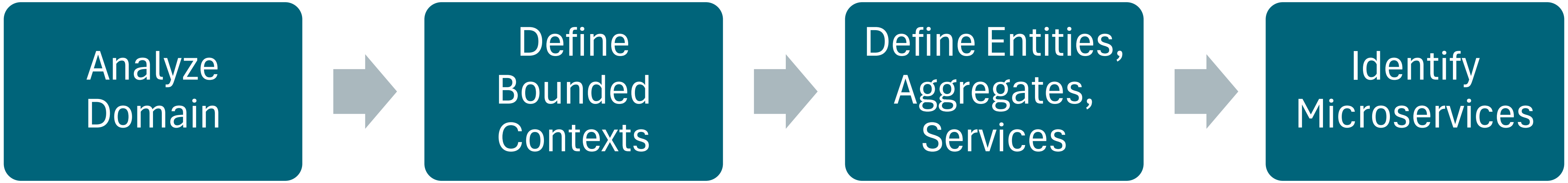
Key points to address

- How do I **organize** the development teams?
- How do I **design** the microservices?
- How do I manage inter-service **communication**?
- How do I manage **data**?

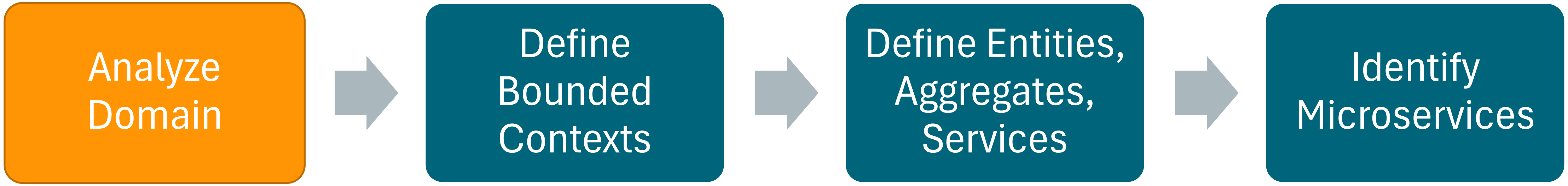
Microservices: Design

Microservices: Design

- Defining boundaries for each service is one of the biggest challenges
- General rule: a service should just do «**one thing**»
- No mechanical process can guarantee the ***right*** result
- **Domain-Driven Development** (DDD) comes in handy
- Nice example here: Using domain analysis to model microservices



Microservices: Design



Analyze Domain

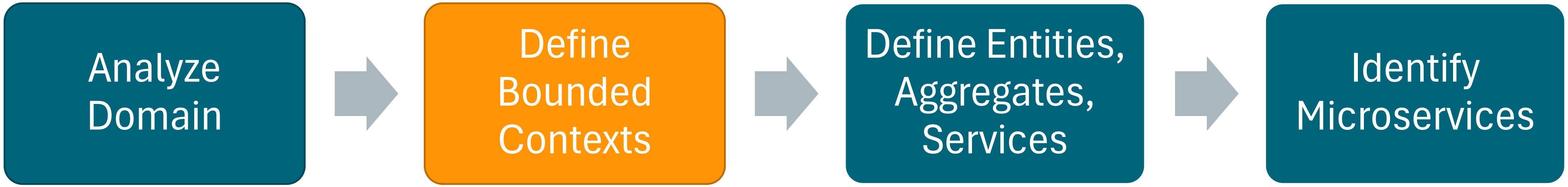
Define Bounded Contexts

Define Entities, Aggregates, Services

Identify Microservices

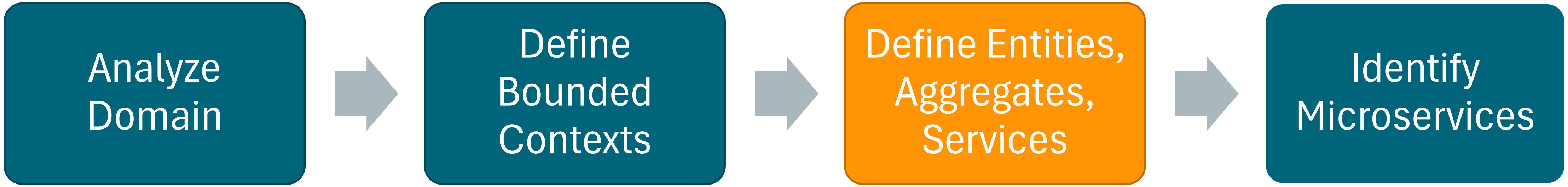
- Understand functional requirements
- Output is an informal definition of the system domain
- Informal definition can be refined into a set of domain models

Microservices: Design



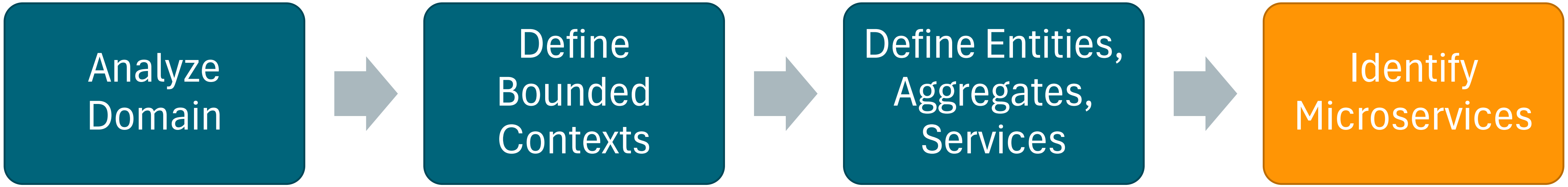
- Each Bounded Context contains a domain model representing a particular subdomain
- Total domain unification might not be feasible nor convenient

Microservices: Design



- For each Bounded Context, apply DDD patterns to define:
  - **Entities**: Objects with an «identity» that persists over time.
  - **Aggregates**: Consistency boundaries around one or more entities. Model transactional invariants.
  - **Domain Services**: Objects that implement logic without holding any state

Microservices: Design



- Start with a Bounded Context. Functionality in a microservice should not span over multiple bounded contexts.
- Aggregates and Domain Services are good candidates for becoming microservices

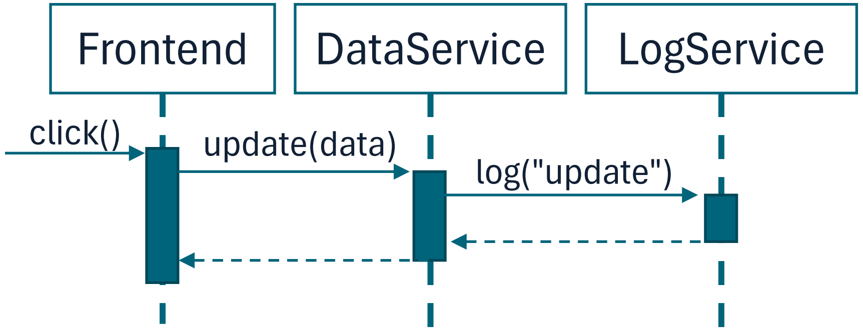
Microservices: Communication

Microservices: Communication

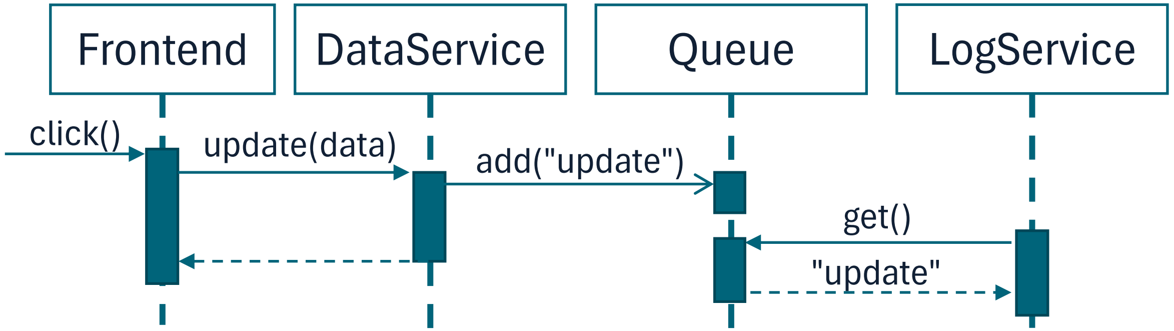
- In Monoliths, communication between different components is achieved with method invocations
- In a Microservice architecture, communication should be **technology agnostic**
- Often, teams leverage protocols and technology on which the World Wide Web is built

Microservices: Communication

- Can be done **synchronously**
  - **REST** (http resource API)
  - **Apache Thrift** (link). Supports multiple protocols (binary, JSON-based, ...)
- Can be done **asynchronously**
  - **Message queues**



Synchronous Communication



Asynchronous Communication

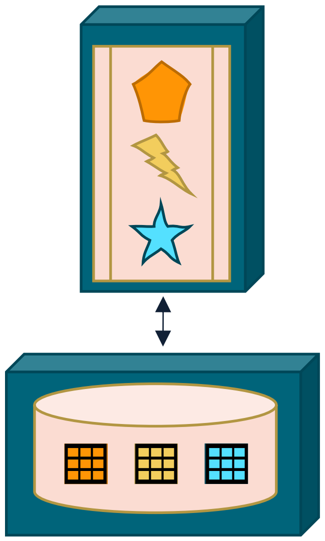
Microservices: Sync vs Async Communication

- Synchronous calls can hinder performance
- In presence of many synchronous calls, the **multiplicative effect of downtime** might manifest.
    - Downtime of the systems becomes the product of the downtimes of individual services
- Either you make your communications asynchronous, or you manage the downtimes

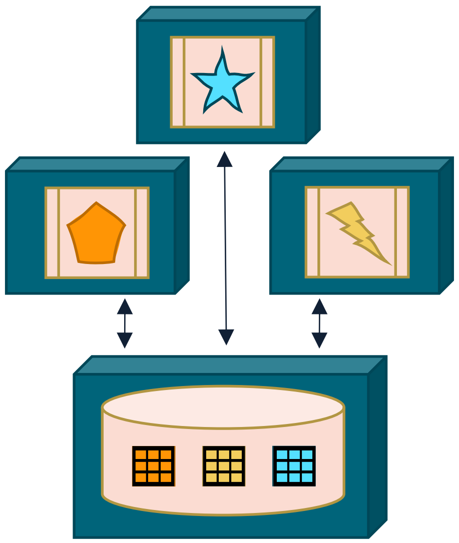
Data management

Microservices: Data management

- Monoliths are typically associated with a **single, centralized database**
  - Often driven by DB licensing models and costs
- Microservices can work with a centralized database…
  - Need to coordinate schema changes across teams



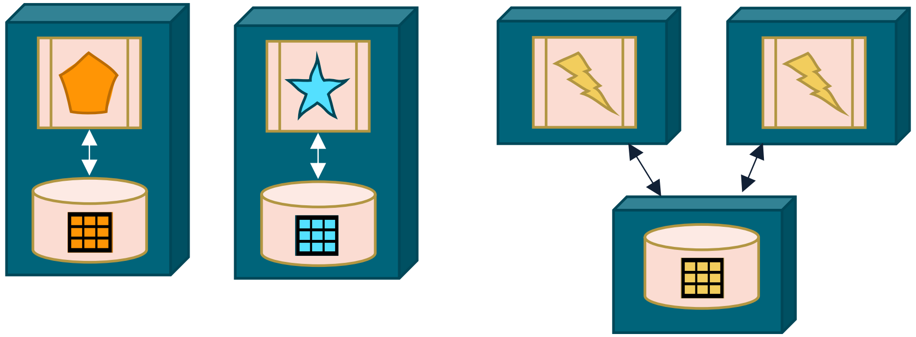
Monolith



Microservices, with shared database

Microservices: Data management

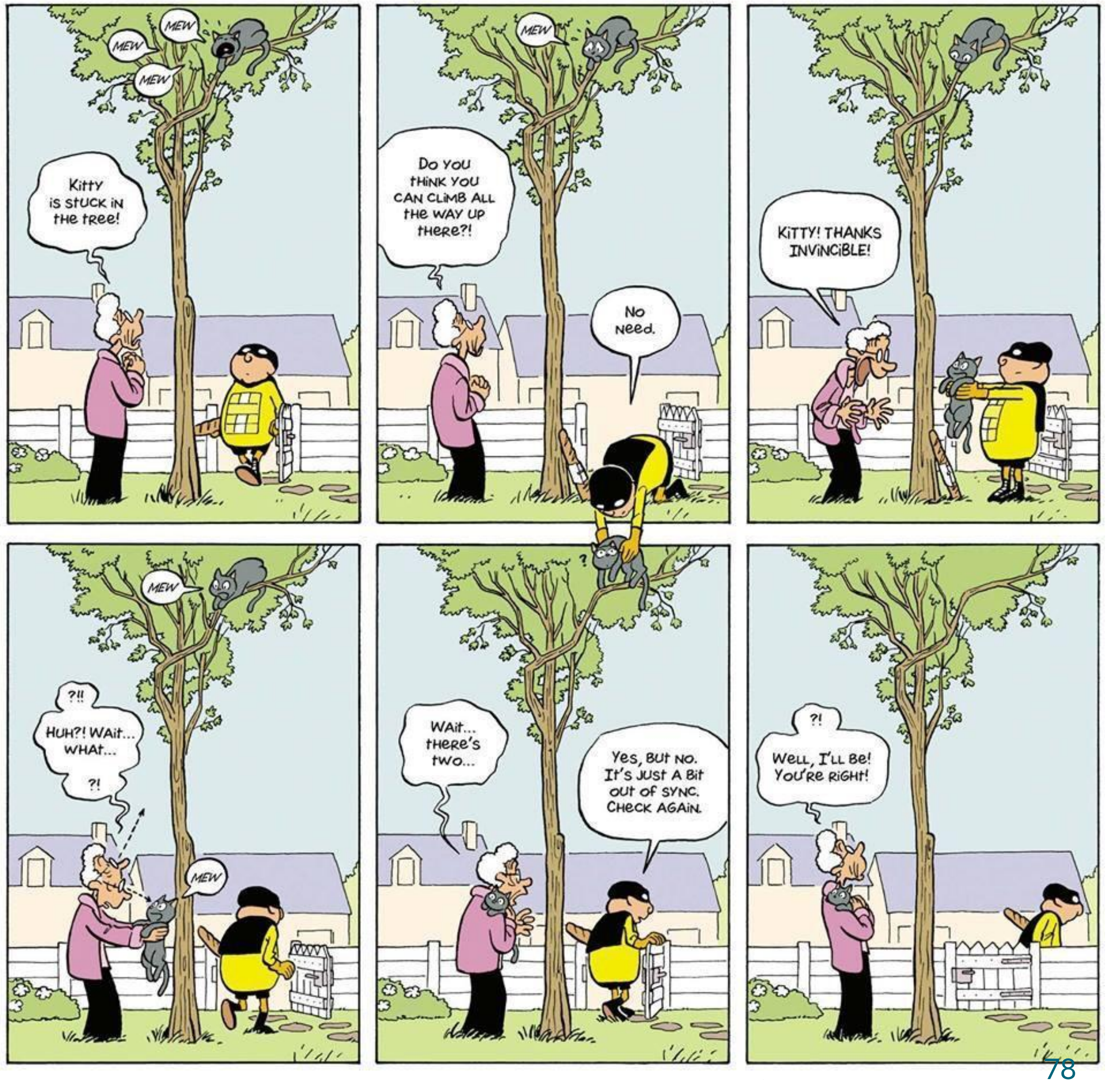
- **Single database per service**
  - Each microservice (and Team) is actually independent
  - Enables polyglot persistence



Microservices, with per-service databases

Microservices: Data management

- When dealing with distributed systems, ensuring **strong consistency** can become harder and harder.
- Might want to settle with **eventual consistency.**
  - At some point in the future, all reads on an entity will return the updated value


MEW
MEW
MEW
Kitty is stuck in the tree!
MEW
Do you think you can climb all the way up there?!
No need.
KITTY! THANKS INVINCIBLE!
MEW
?!!
HUH?! WAIT... WHAT...
?!
MEW
?
WAIT... THERE'S TWO...
YES, BUT NO. IT'S JUST A BIT OUT OF SYNC. CHECK AGAIN.
?!
WELL, I'LL BE! YOU'RE RIGHT!

Imbattable, Pascal Jousselin

Microservices: Conclusions

Microservices: It’s about trade offs

Microservices provide benefits...

✓ Strong module boundaries

✓ Independent deployments

✓ Technology diversity

... but not for free

× Increased complexity

× Eventual consistency

× Operational complexity

So, make sure you have a good reason to adopt microservices! [1]

[1] Mendonça, N. C., Box, C., Manolache, C., & Ryan, L. (2021). The monolith strikes back: Why Istio migrated from microservices to a monolithic architecture. *IEEE Software*, *38*(05), 17-22. https://ieeexplore.ieee.org/document/9520758

Architecture and Development Team Organization

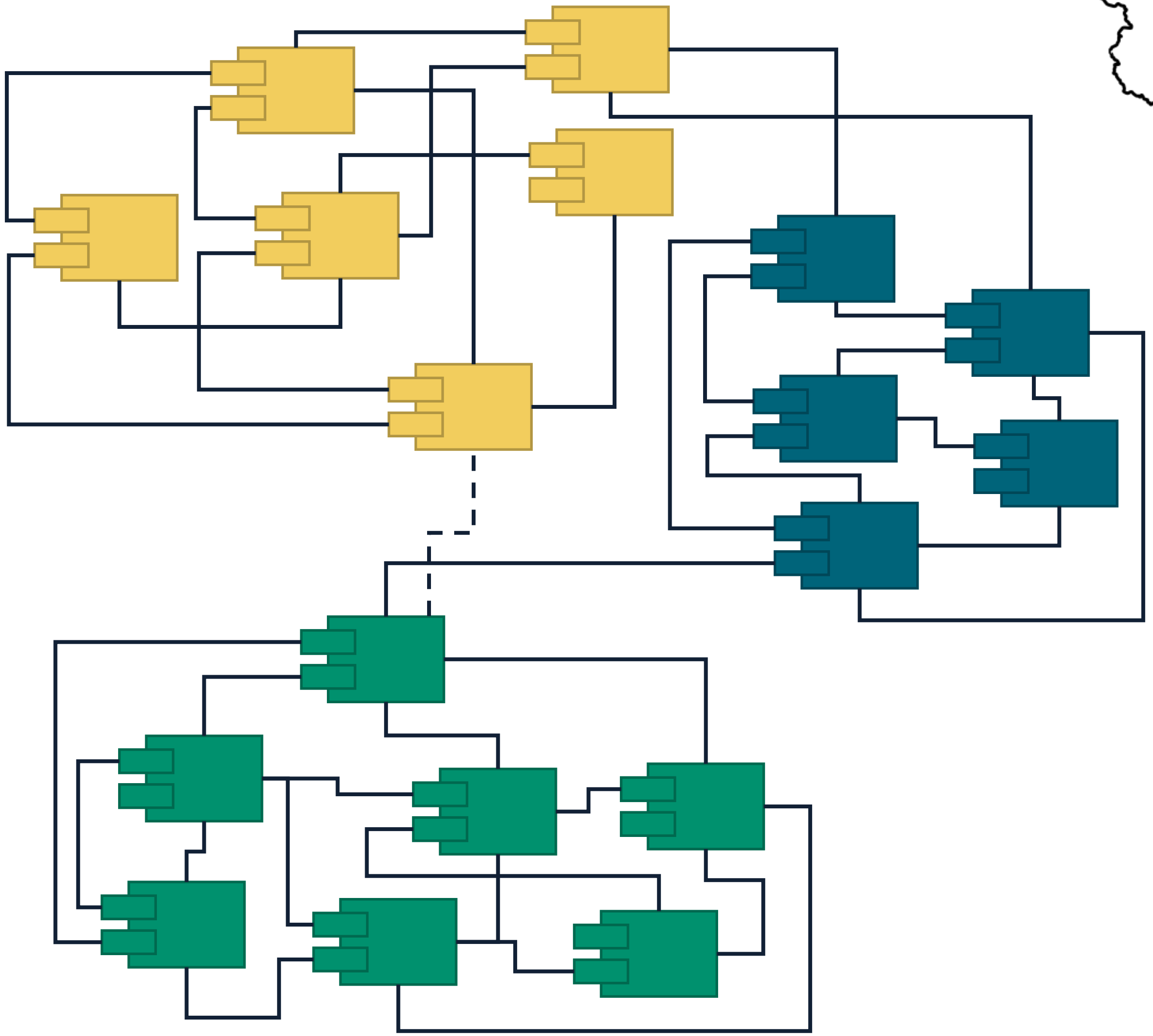
Development Team Organization

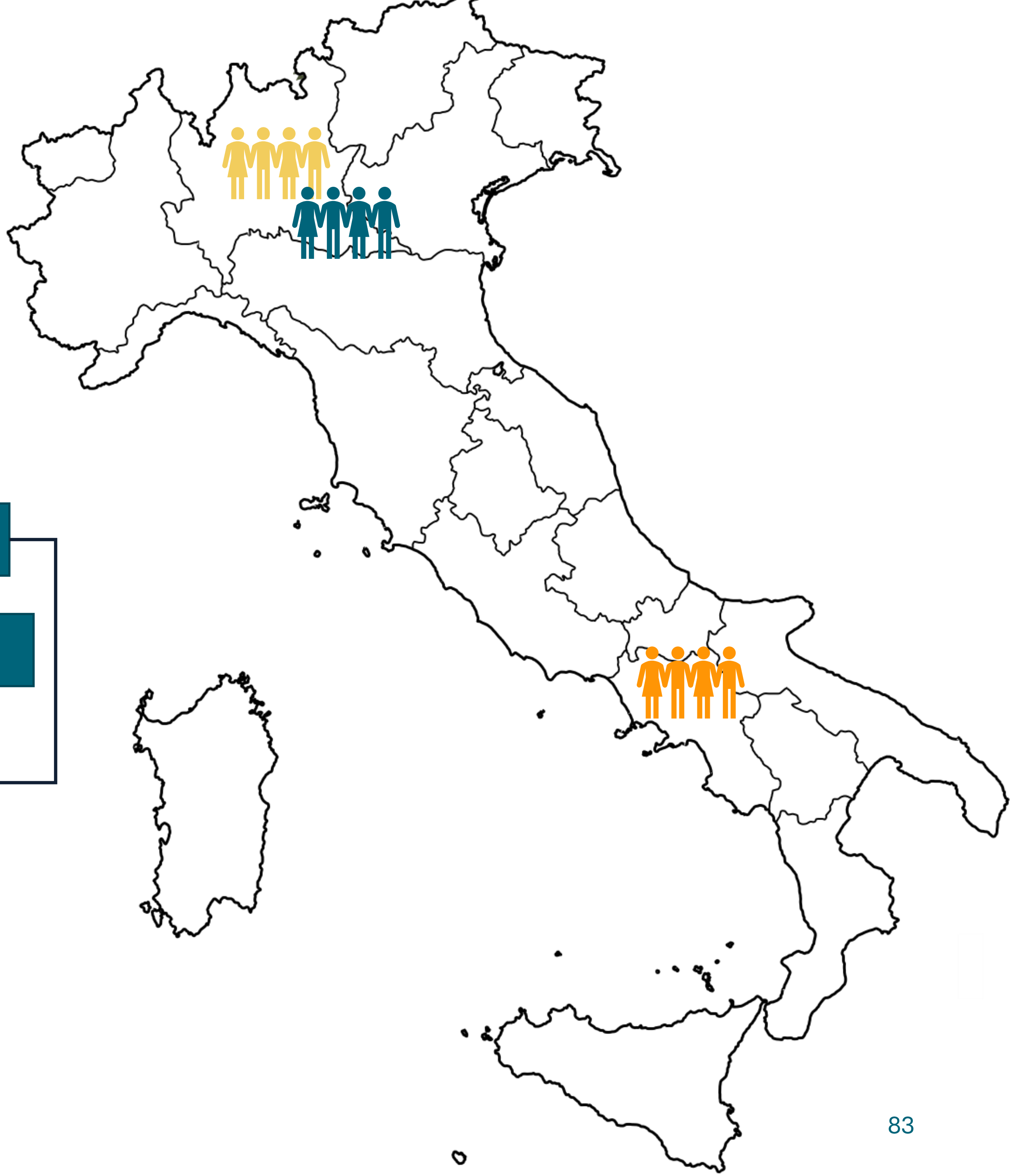
- What does Organization have to do with Architecture?
- Conway's Law:

> *Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.*
>
> -- Melvin Conway [1]

[1] Conway, M. E. (1968). How do committees invent. *Datamation*, *14*(4), 28-31.

Conway’s Law



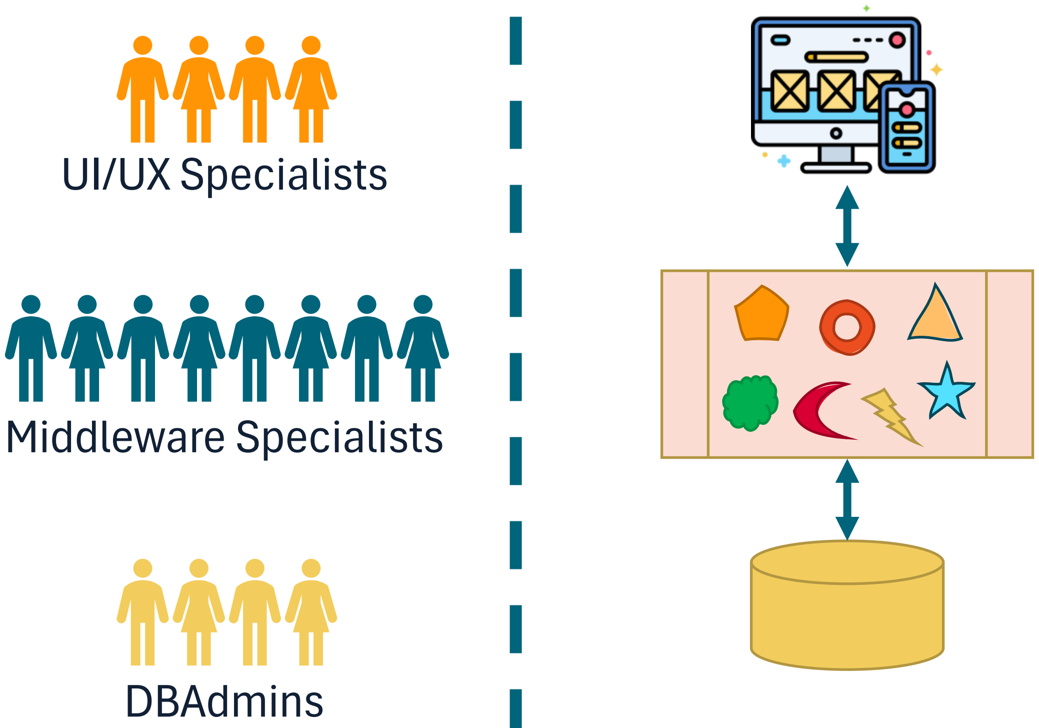


Conway’s Law

- Software coupling is **enabled** and **encouraged** by **human communications**
- If you can talk to the author of some code, it’s easier to truly understand that code
- Which means it is easier to for your code to interact with (and thus be coupled to) that code

How do we organize a large team?

- Often management focuses on the technology layer
- Layered functional teams lead to layered architectures



UI/UX Specialists

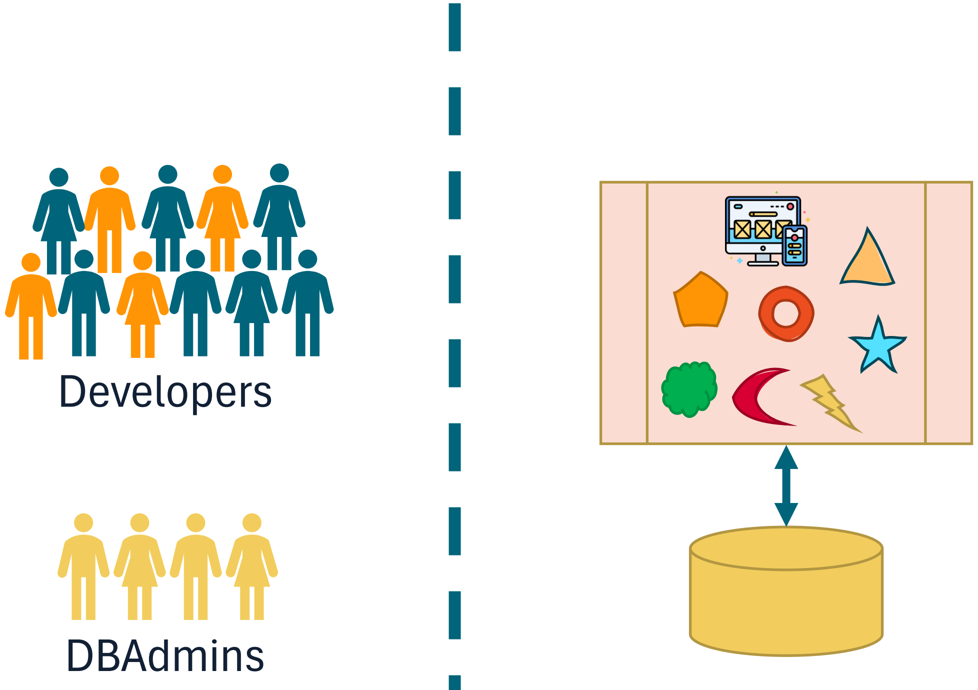
Middleware Specialists

DBAdmins

- What if we need to add a new feature?
- Logic everywhere!

How do we organize a large team?

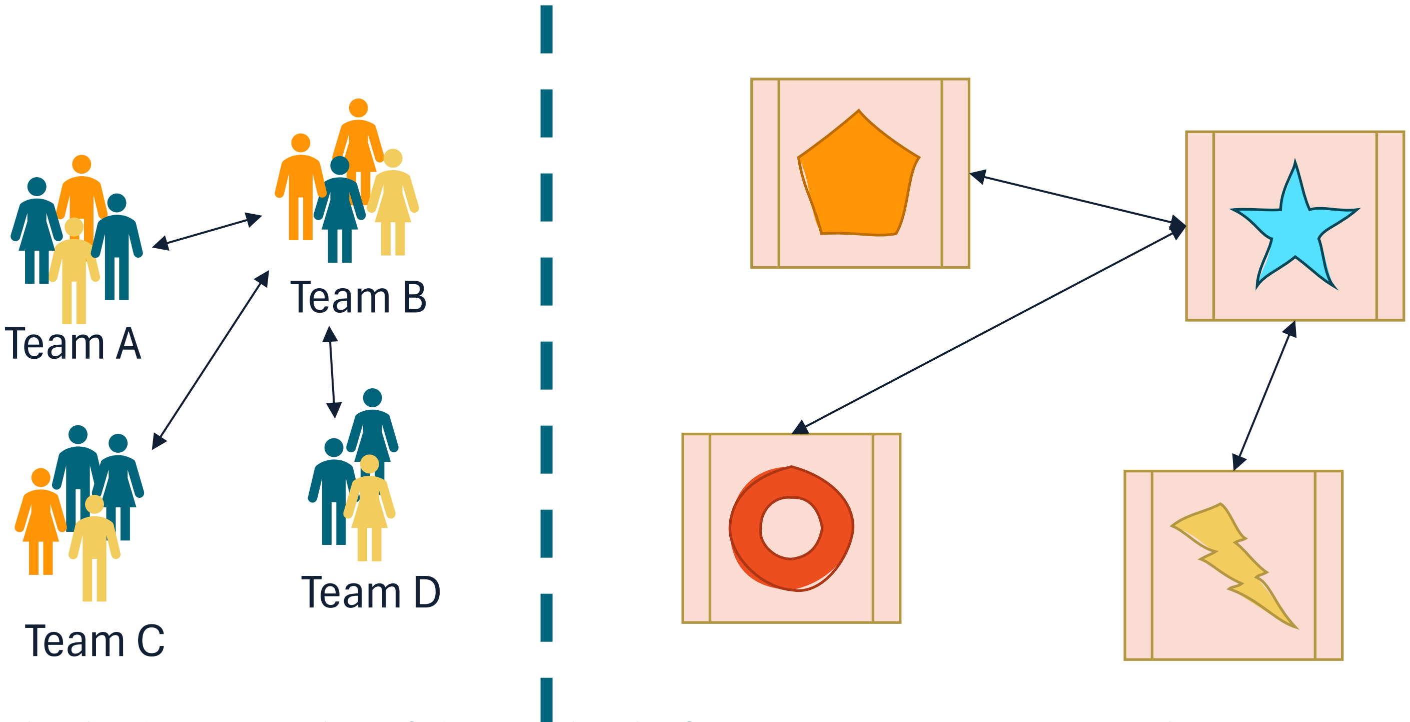
- Often management focuses on the technology layer
- Layered functional teams lead to layered architectures



- What if we need to add a new feature?
- Logic everywhere!

How do we organize a large team?

- Microservices favor small, independent teams
- Teams are **cross-functional**: include full range of skills required



Dealing with Conway’s Law

- **Ignore it:** Don’t take it into account, because you’ve never heard of it, or you don’t think it applies
  - Spoiler alert: it does!
- **Accept it:** Recognize its impact, make sure that you architecture doesn’t clash with the team’s communicational patterns.
- **Inverse Conway Maneuver** [1]**:** Change the communication patterns of the development team to encourage the desired architecture.

[1] https://martinfowler.com/bliki/ConwaysLaw.html

Conway’s Law: Take Home Message

- The decomposition of a system and the decomposition of the development organization cannot be done independently.
- Not only at the beginning of a project, but throughout the entire lifecycle of the project.
- Evolving the architecture and re-organizing the human organization must go hand-in-hand.

READINGS AND REFERENCES

- System Design Primer, Donne Martin
  https://github.com/donnemartin/system-design-primer
- Software Architecture Guide, Martin Fowler
  https://martinfowler.com/architecture/
- Who needs an architect? Martin Fowler, IEEE Software
  https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf