Cloud Computing -- Microservices

Thomas Ropars

Email: thomas.ropars@univ-grenoble-alpes.fr

Website: tropars.github.io

References

The following references were used to prepare these slides:

- Building Microservices by Sam Newman
    - Many figures presented in these slides come from this book
- Set of blog posts by James Lewis and Martin Fowler
    - See this post for a good introduction to microservices
- Other research papers are cited on corresponding slides

Introduction

Definition

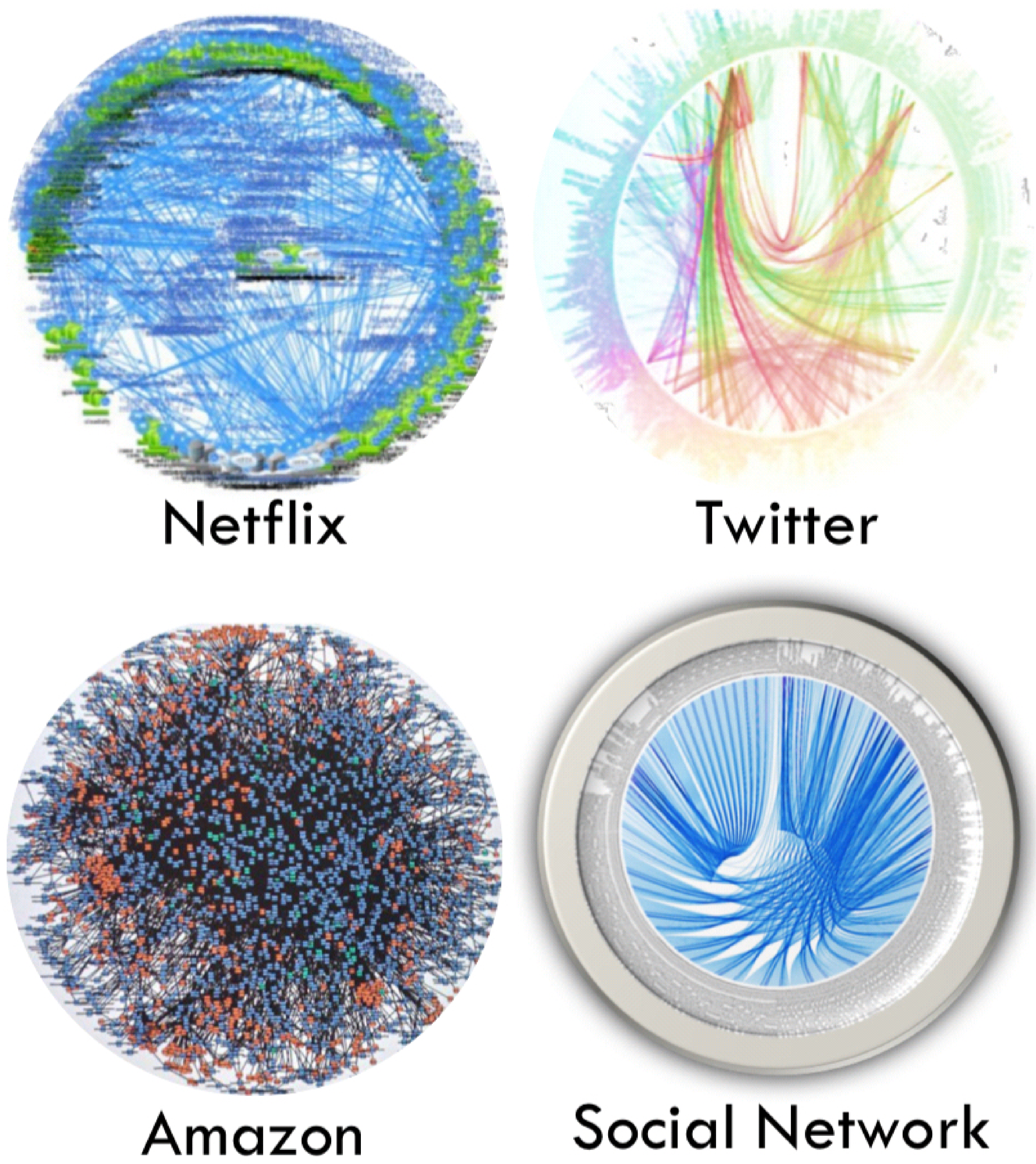
By J. Lewis and M. Fowler

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process 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.

The key concepts

- Architectural style
- Suite of small services
- Built around business capabilities
- Independently deployable
- Communicating with lightweight mechanisms
- Written in different programming languages
- Use different data storage technologies

Examples of large-scale microservices apps



See: Gan, Yu, et al. "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems.", ASPLOS 2019.

Example of a media application

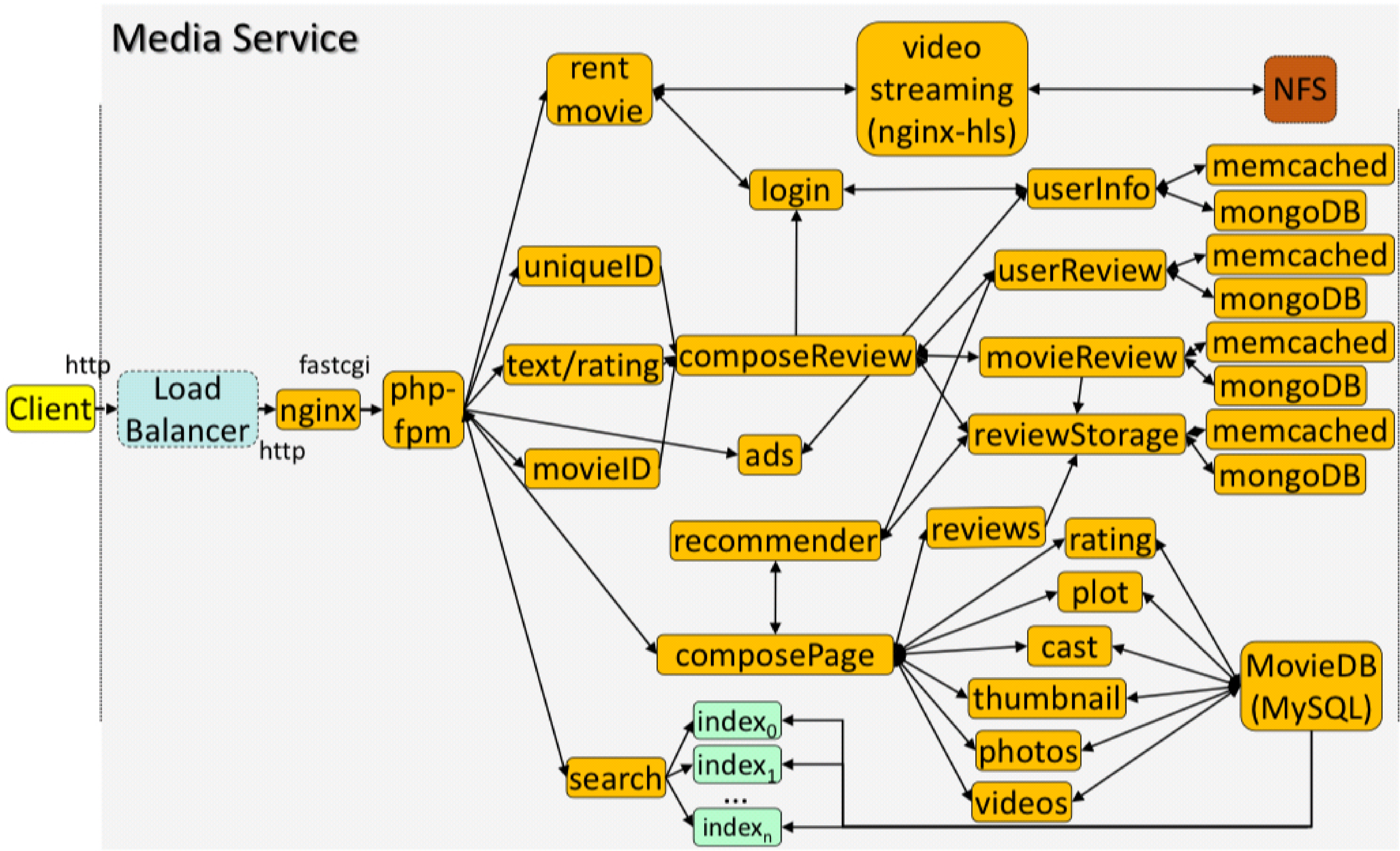


Figure 5. The architecture of the *Media Service* for reviewing, renting, and streaming movies.

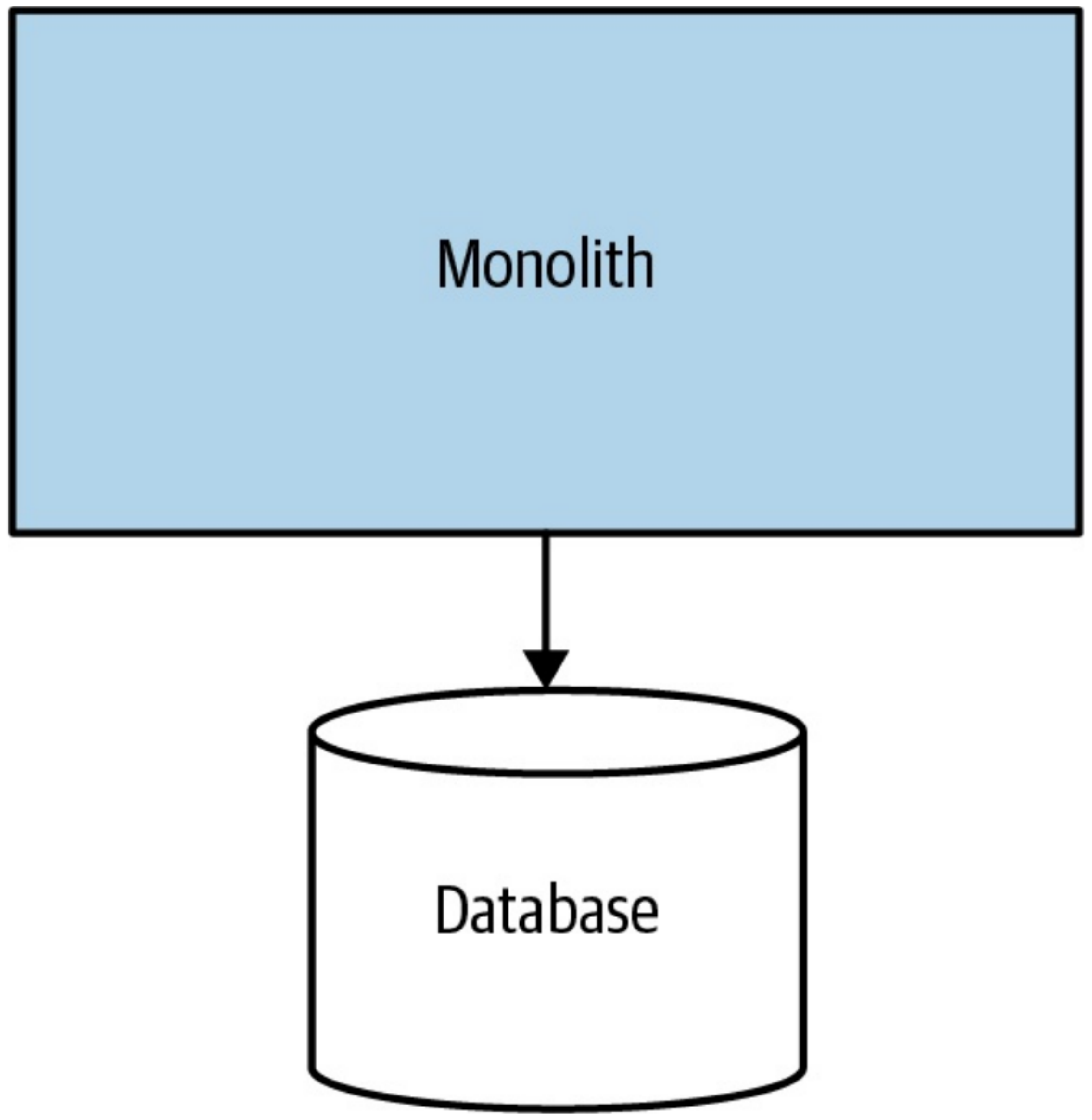
See: Gan, Yu, et al. "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems.", ASPLOS 2019.

The Key Concepts of microservices

Alternative to the monolithic architecture

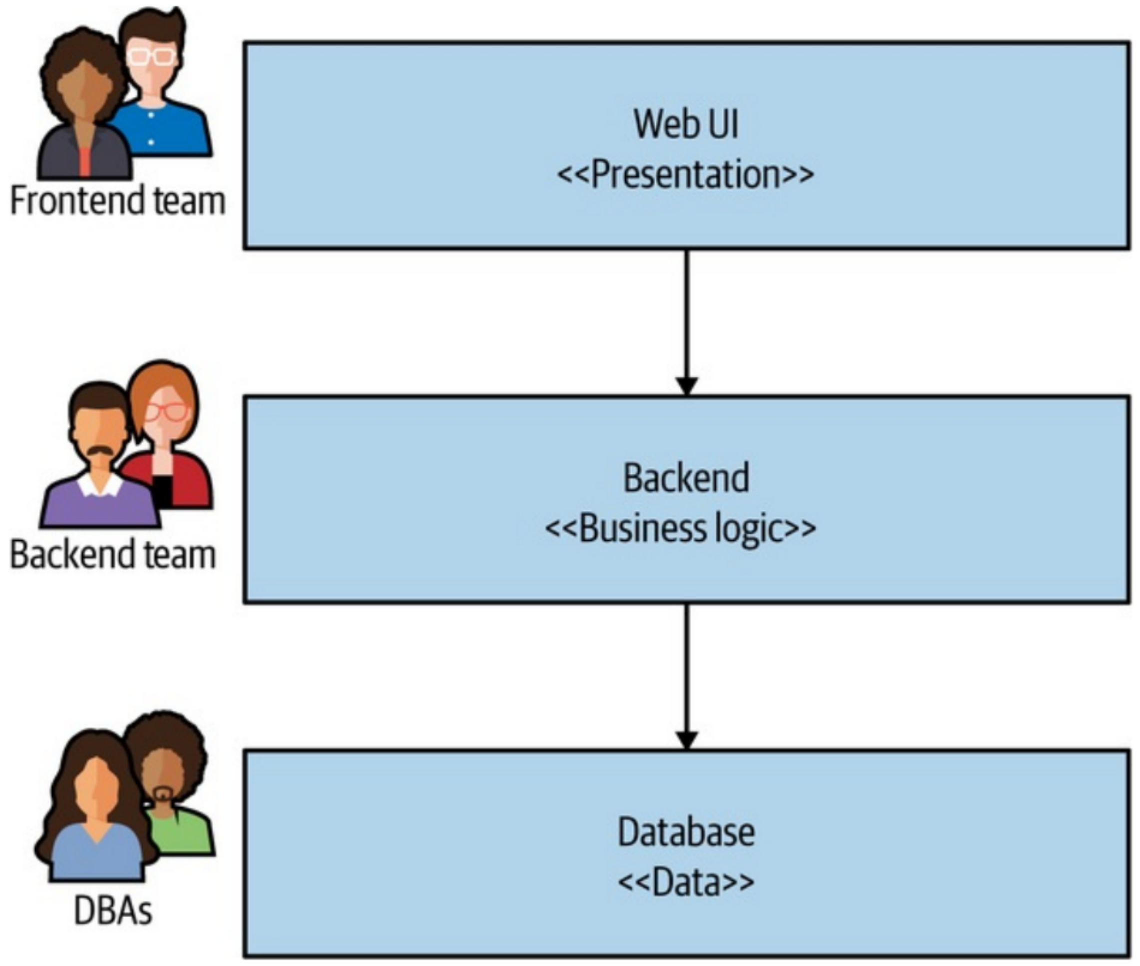
Monolith -- Definition

- A monolith system is one in which all functionality must be deployed together
- Example: The single process monolith

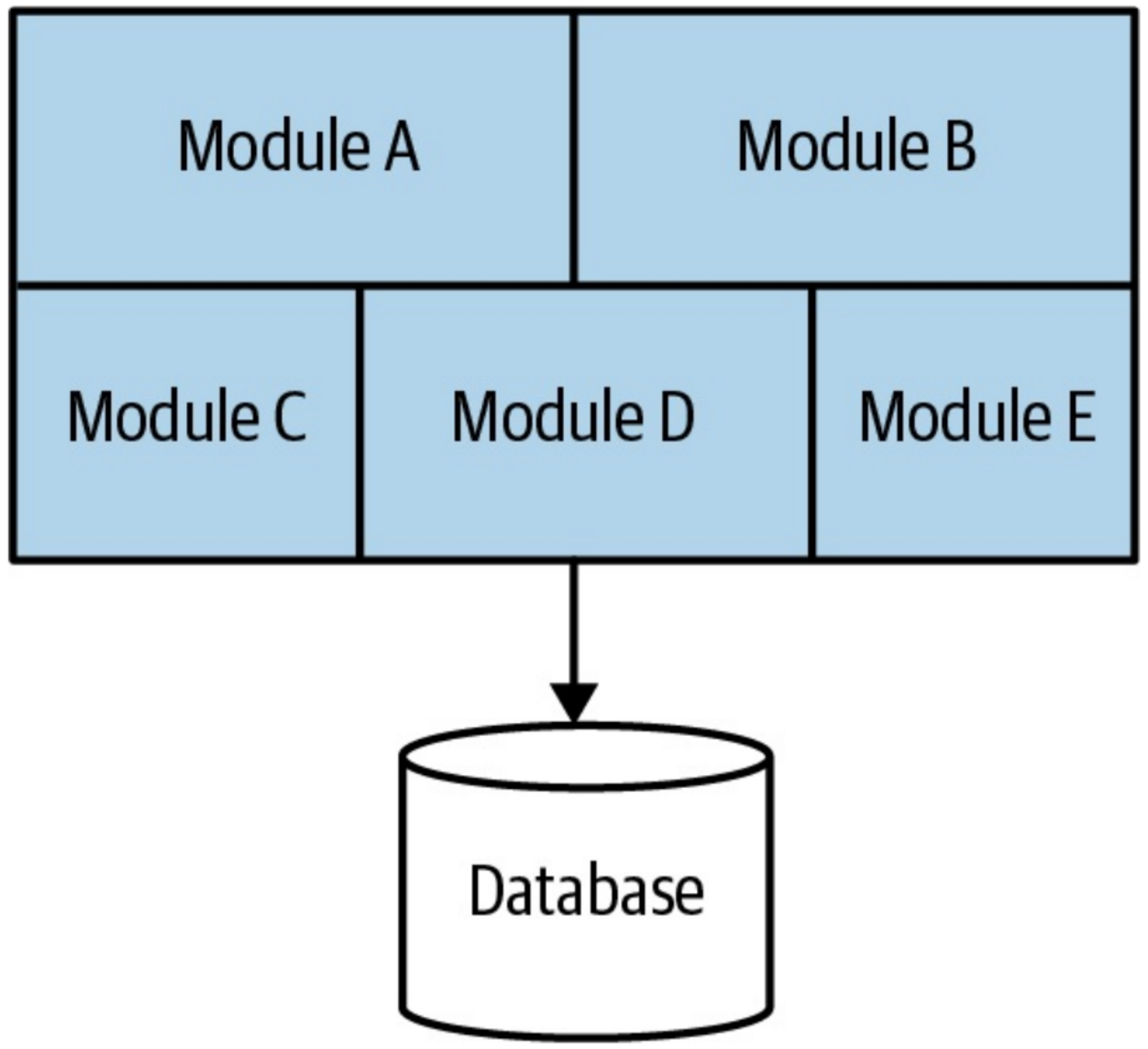


About the 3-tier architecture

- A traditional approach for web applications: The 3-tier architecture
  - Web UI
  - Backend (a monolith)
  - Database



Modular Monolith



- Still a monolith since all components are deployed together
  - In general deployed as a single process
- Problems that might appear:
  - Difficulty in determining the boundaries of the modules
    - Who owns what and who decides what?
  - Delivery contention
    - Delivery delayed because another team also wants to update a module

The key concept of microservices

- Independent deployability
- Owning their own state
- Alignment with organization + structured based on business domains
- Size

Independent deployability

The goal

- Being able to re-deploy a microservice without re-deploying the others
  - Being able to modify a service without modifying the others
  - Reduce the time to release new features

How this is achieved

- The microservices must be loosely coupled
- Clear definition of the interfaces of a service
  - Concept of `information hiding`: hiding as much information as possible inside a component and exposing as little as possible via external interfaces

A note on information hiding

- It can be shown that if a programmer has access to some information, they will take advantage of it

Consequence:

- No matter how careful people are initially, the result will eventually be tightly coupled services
- The only solution is to make as little information as possible accessible from outside a service

See David Lorge Parnas. "Information distribution aspects of design methodology." (1971).

Owning their own state

- Avoid shared databases
    - If a service needs an information from another service, it always needs to go through the pre-defined interface

Why it is important:

- Contributes to `information hiding`
- Required to ensure independent deployability

Alignment with the organization of the company

- *Old-style* organization of IT companies: Group people based on their core expertise
    - A team of database admins
    - A team of backend devs
    - A team of frontend devs

- Be aware of the Conway's law!

A note on Conway's law

```
Any organization that designs a system (defined broadly) will produce a design
whose structure is a copy of the organization's communication structure.
```

Consequence for designing web application

See https://martinfowler.com/bliki/ConwaysLaw.html

A note on Conway's law

```
Any organization that designs a system (defined broadly) will produce a design
whose structure is a copy of the organization's communication structure.
```

Consequence for designing web application

- A small group of developers that works in the same office
    - Will produce a monolith
- Organization based on core expertise
    - Will produce a 3-tier architecture (still monolithic)

Note that designing an architecture that does not align with the organization structure would be highly counter-productive:

- A lot of communication overhead
- Tension

See https://martinfowler.com/bliki/ConwaysLaw.html

Problem with the 3-tier architecture

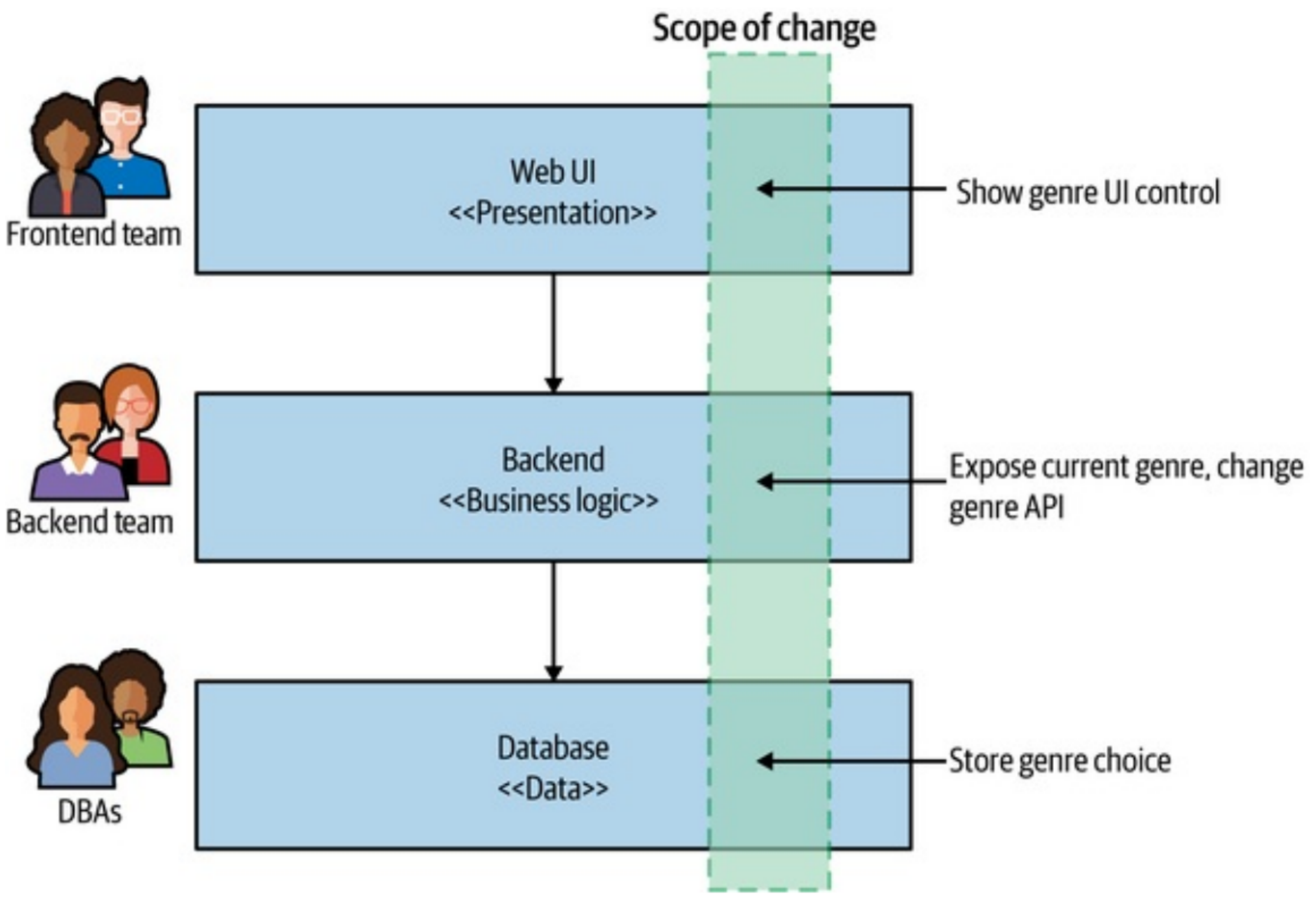
A scenario

- We are developing a music streaming service
- We want to add a feature: each user can specify its favorite music style in its profile

Problem with the 3-tier architecture

A scenario

- We are developing a music streaming service
- We want to add a feature: each user can specify its favorite music style in its profile

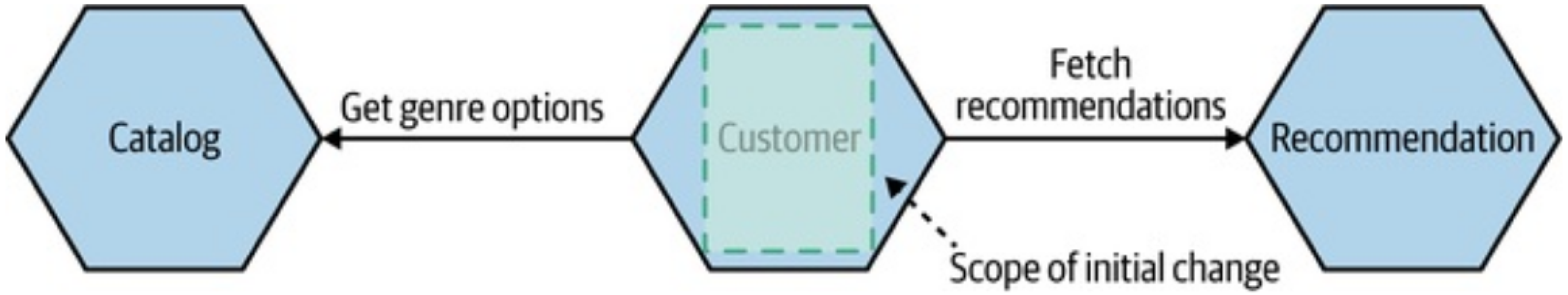


- Multiple teams need to be involved
- Modifications need to be deployed in the right order

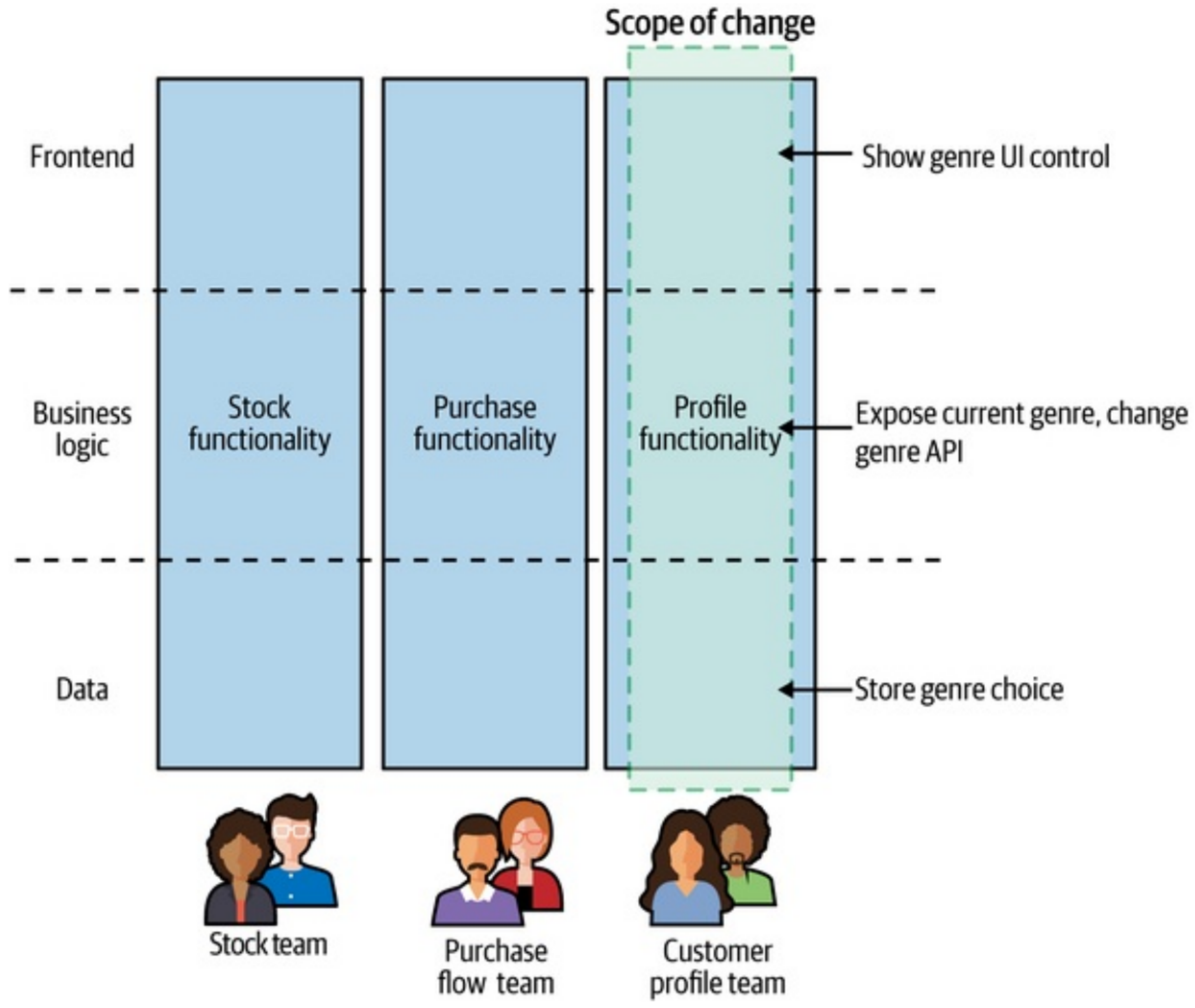
Structure around business domains

- Define microservices boundaries based on business domains
  - Approach inspired from *Domain-Driven Design*

For our previous scenario



About our previous scenario



- A single team is in charge the *Profile* functionality
  - This team includes frontend, backend, database experts
  - Nowadays companies' organization:
    - Small poly-skilled teams (5 to 10 persons)
    - Goal: Avoid difficult interactions between siloed teams
    - This vision is even more effective with distributed and remote workers

The size of microservices

A difficult question

- Each microservice should remain small enough that its code can be understood
    - And fully managed by a *small* team
        - Including deployment if possible ("Run what you wrote")
- Having too many microservices has several drawbacks:
    - Performance issues
    - Complex interactions between services
        - Still loosely coupled ?
        - Deployment issues
- A solution: focus on the business domains more than the size

A note on "Devs run what they wrote"

Idea of decentralized management

- No central entity is in charge of the deployment
    - Idea made popular by Amazon and Netflix
    - Each team is fully responsible of its microservice(s)

Objective: Improve code quality

- You build it, you run it
    - Force programmers to focus on the quality of their code
    - Situation: You are woken up in the middle of the night because your code does not work

Advantages of microservice architectures

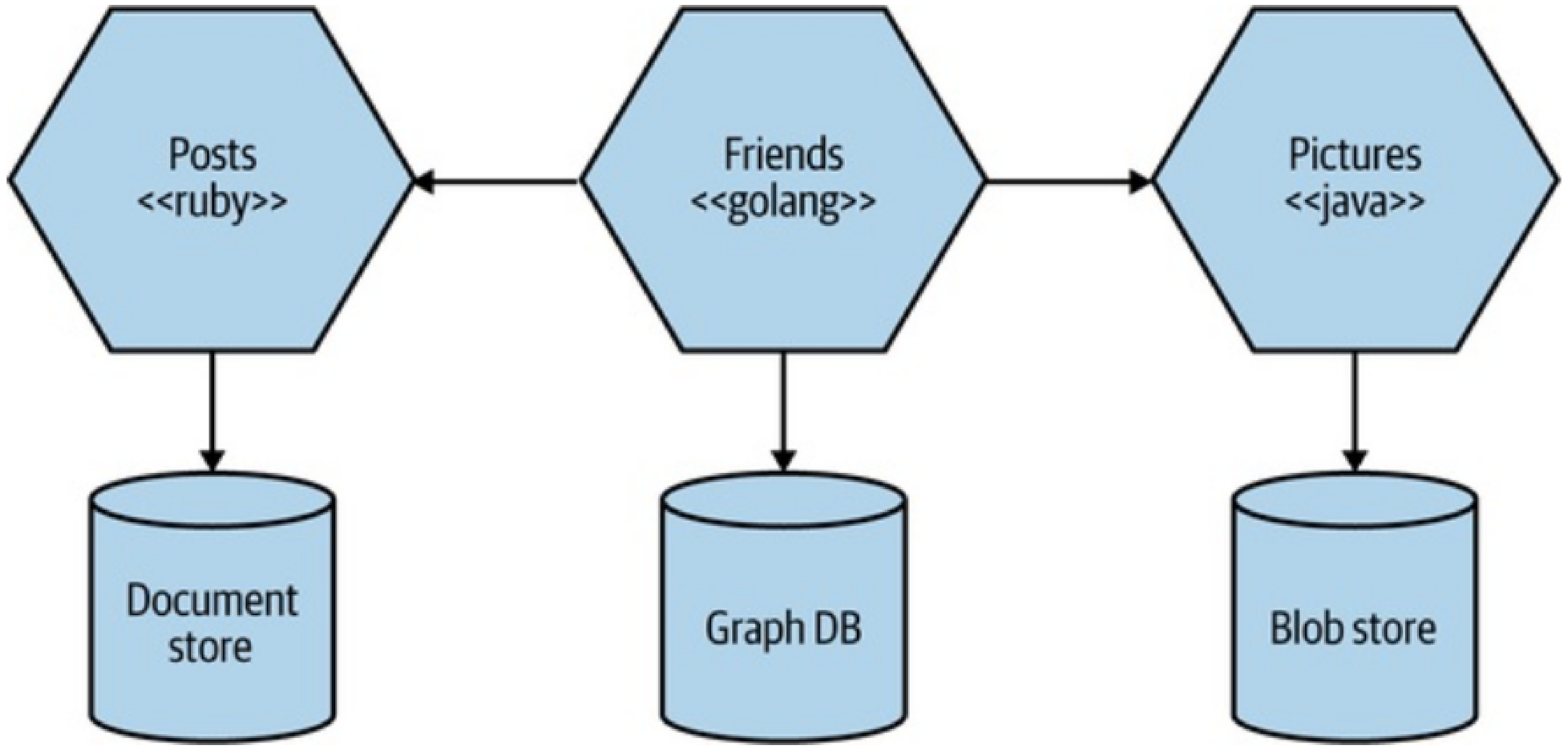
Advantages of microservice architectures

- Technology heterogeneity
- Robustness
- Scaling
- Ease of Deployment

Advantages of microservice architectures

Technology heterogeneity

- Each component can be implemented in a different language
  - Since internal details are hidden, we can change the techno at any point
- Database technologies can be different between components
- It aligns with:
  - One team per microservice
  - Decentralized management



Advantages of microservice architectures

Robustness

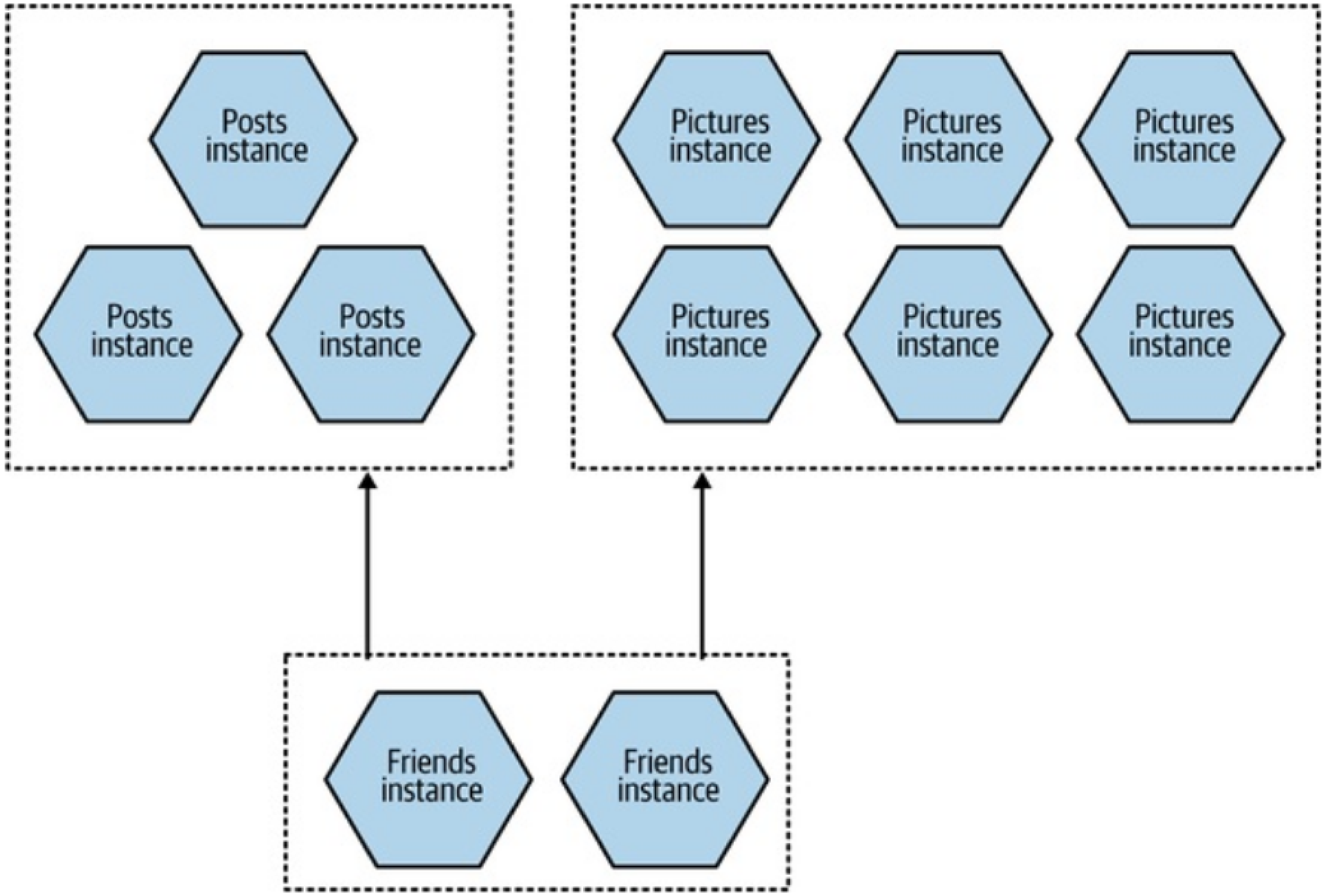
- Failure of microservice != failure of application
    - Parts of the application can continue working even when some components fail
    - Require a good isolation between components

- Warning: new types of failure might appear
    - Network failures

Advantages of microservice architectures

Scaling

- Each microservice can be scaled independently



Advantages of microservice architectures

Ease of deployment

- Each microservice can be deployed independently
- It is important for:

Advantages of microservice architectures

Ease of deployment

- Each microservice can be deployed independently

- It is important for:
    - Decentralized management
    - **Allow new features to be deployed as soon as possible**
    - **Being able to apply DevOps approaches**
        - Continuous integration
        - Continuous delivery

Other advantages

Alignment with the organization of the company

- Sweet spot team size/productivity

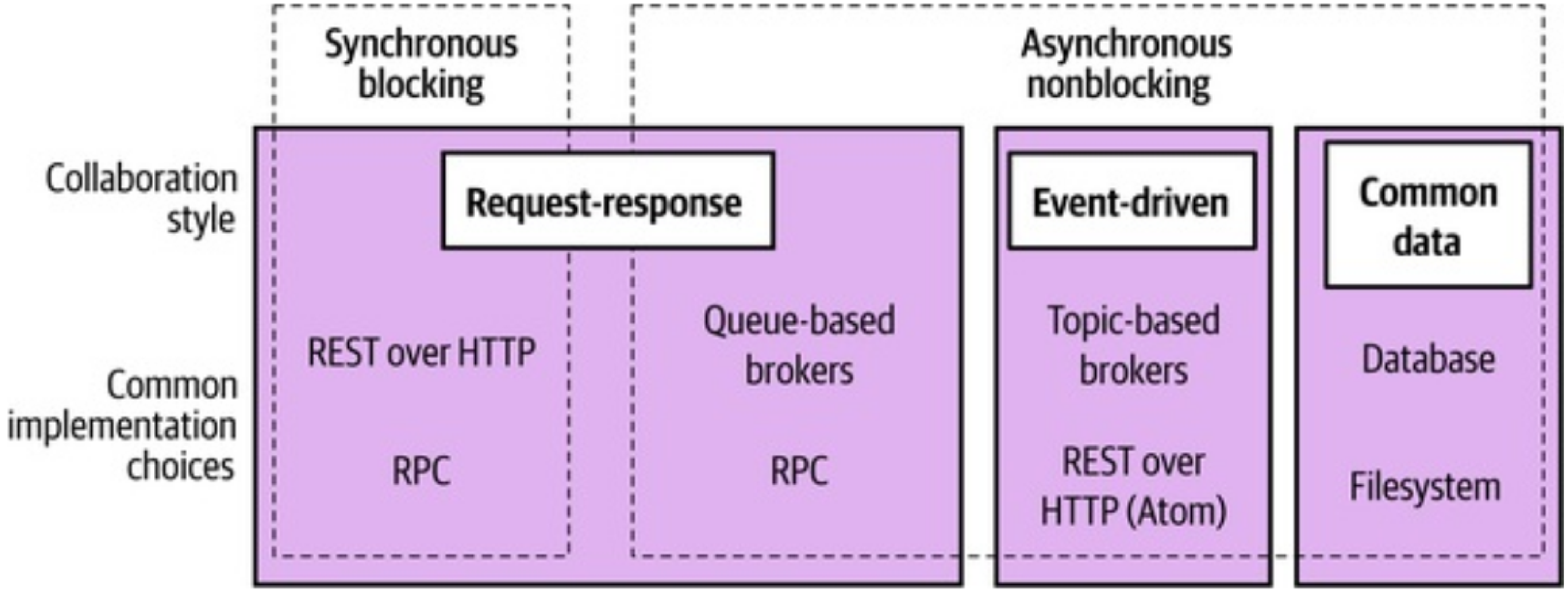
Reusability

- Microservices can be reused in other applications

Some important technologies

Communication between microservices

Different communication paradigms



Communication between microservices

Several technologies

- Remote Procedure Calls
  - gRPC (Protocol Buffer for data serialization)
  - High performance but use of an explicit schema that can lead to coupling between components

- REST + HTTP
  - Lower performance
  - Default choice for small-scale applications

- Message queues/brokers, pub-sub
  - Techno: Kafka, Rabbit-MQ, Zero-MQ, etc
  - Provide extra fault tolerance (avoids message lost)

Enabling technologies related to deployment

- Containers + Orchestration

- Services provided by Cloud providers
  - Managed databases
  - Message brokers
  - Serverless

Discussion about the challenges

Complex development and maintenance

- Multiple technologies
- Complex relations between services
    - Debugging?
- Complex deployment with a large number of services
    - Monitoring?

Require experienced developers

- A monolithic approach is the best choice in many cases

Complex data management

- Data are distributed over multiple databases
    - How to manage the application state?
    - May have to deal with low level of consistency (eventual consistency)

Solutions to modify the application state:

- Distributed transactions
    - Difficult topic
- Sagas
    - A global transaction as a sequence of local transactions
    - *Compensating transactions* need to be defined in case a transaction fail (instead of classical rollback)

About the complexity of the applications

Studies in large companies

- Alibaba
  - 5% of microservices are used by more than 90% of online service
- Meta
  - 18K services; 12M service instances [Meta]
  - The number of service instances doubled in 2 years
  - Analysis of number of services calling (fade-in) and called by each service (fade-out)

Statistic	Fan-in	Fan-out	Fan-in (Reg)	Fan-out (Reg)
Min	1	1	1	1
Median	4	4	3	6
Average	14	12	19	15
P99	86	101	324	158
Max	14,084	5,865	2,968	1,069

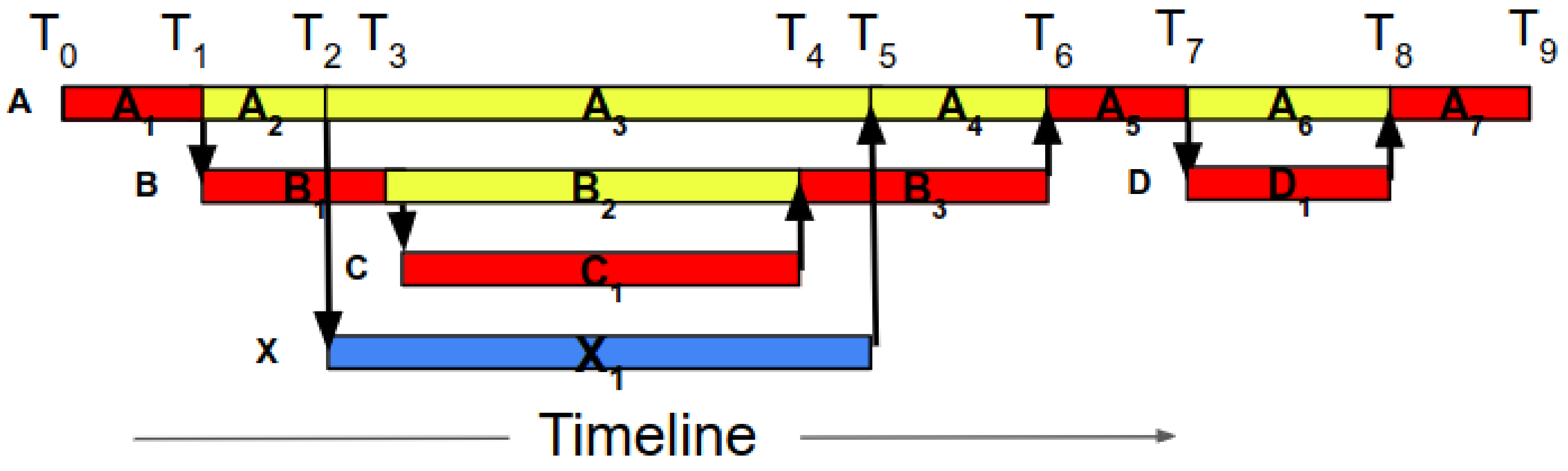
See: Huye, Darby, Yuri Shkuro, and Raja R. Sambasivan. "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows." 2023 USENIX Annual Technical Conference.

See: Luo, Shutian, et al. "Characterizing microservice dependency and performance: Alibaba trace analysis." SoCC 2021.

About the complexity of monitoring

Tracing

- Trace the set of calls generated by a request to a service
  - Useful for debugging
  - Useful to try analyzing performance issues
- Set of tools to trace RPC calls
  - OpenTracing, Jeager, etc.



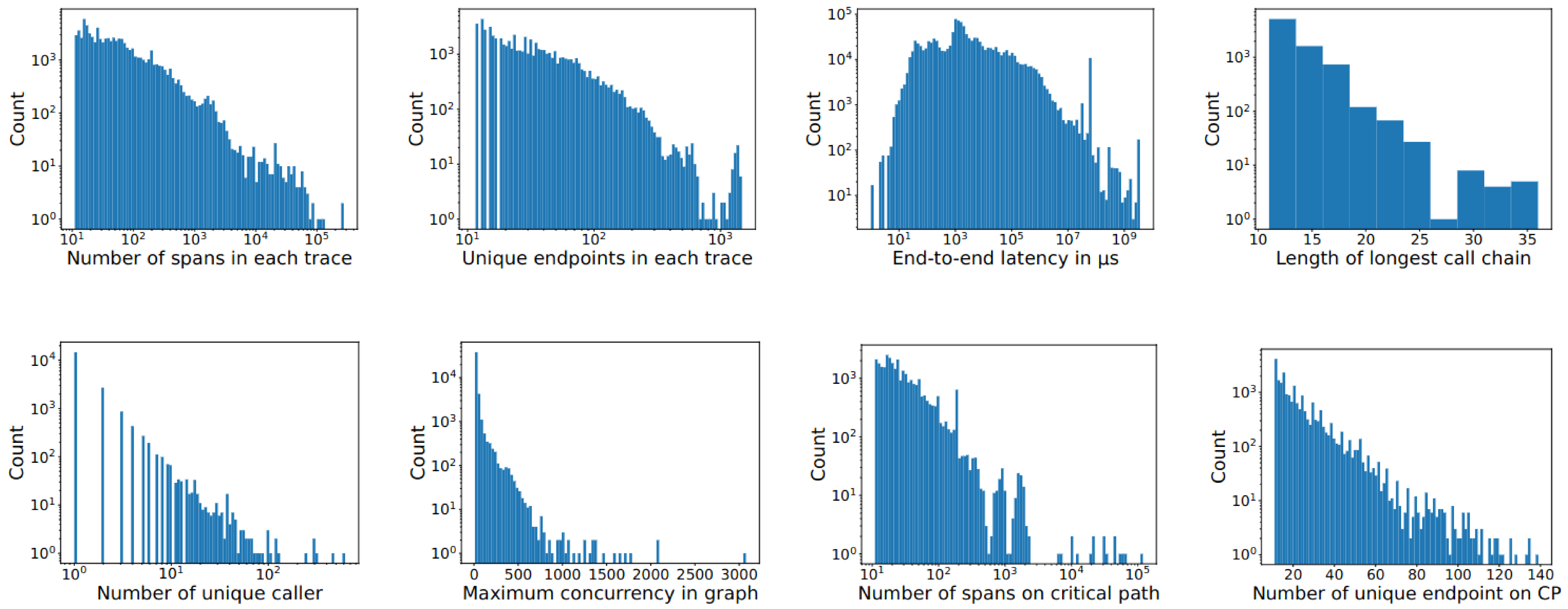
About the complexity of monitoring/debugging

Study of the Uber application

See: Zhang, Zhizhou, et al. "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures." 2022 USENIX Annual Technical Conference.

About the complexity of monitoring/debugging

Study of the Uber application



- Manual debugging becomes impossible
- Goal of the paper: Automatic tool for Critical-Path Analysis

See: Zhang, Zhizhou, et al. "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures." 2022 USENIX Annual Technical Conference.

About performance

Tail at scale

- Tail latency is a major issue in distributed systems (cf "end-to-end latency" in previous slide)
    - Because of the accumulation of many small delays, the response time for some requests might be very bad

Monolithic apps might perform better with respect to tail latency

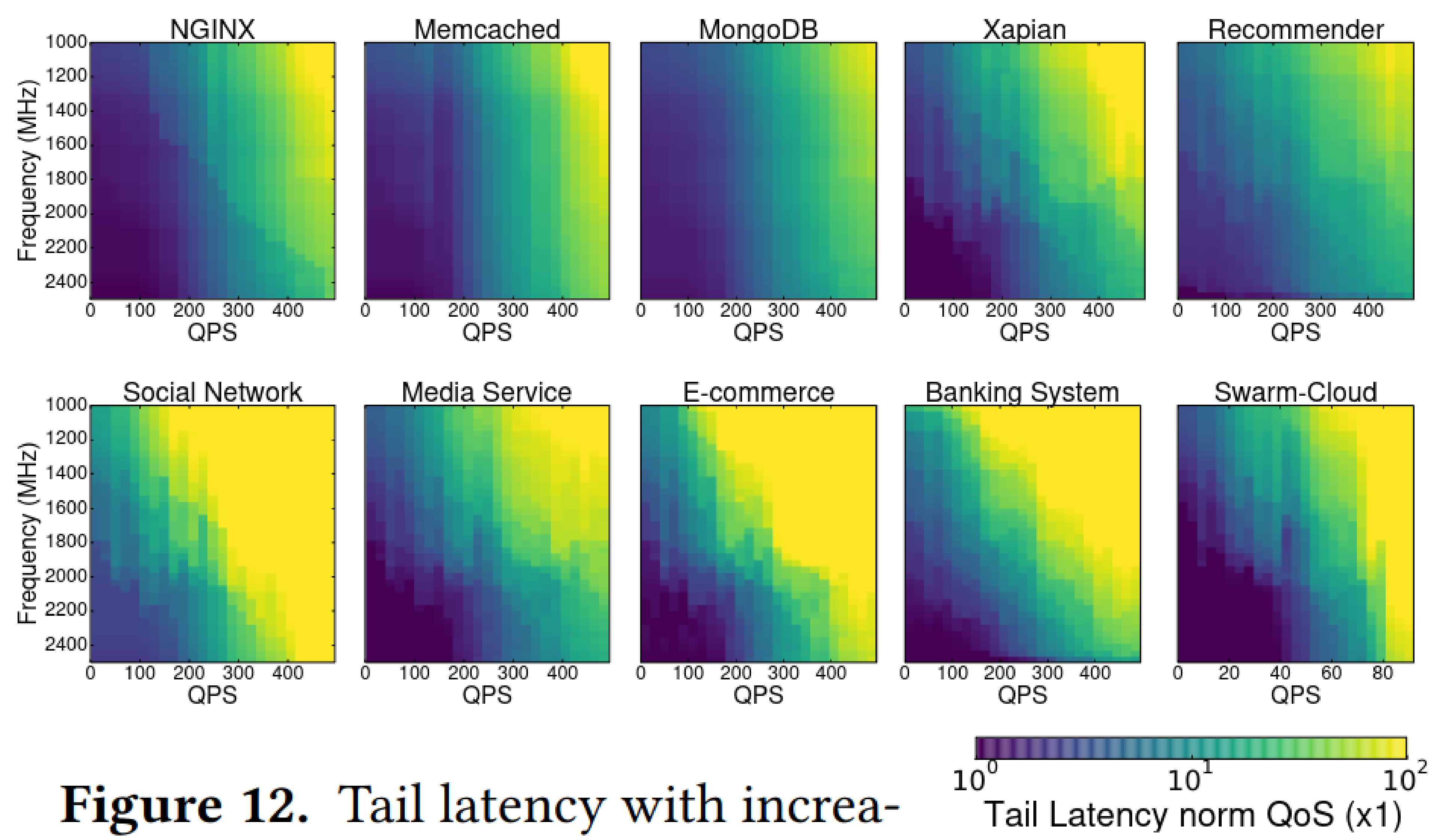
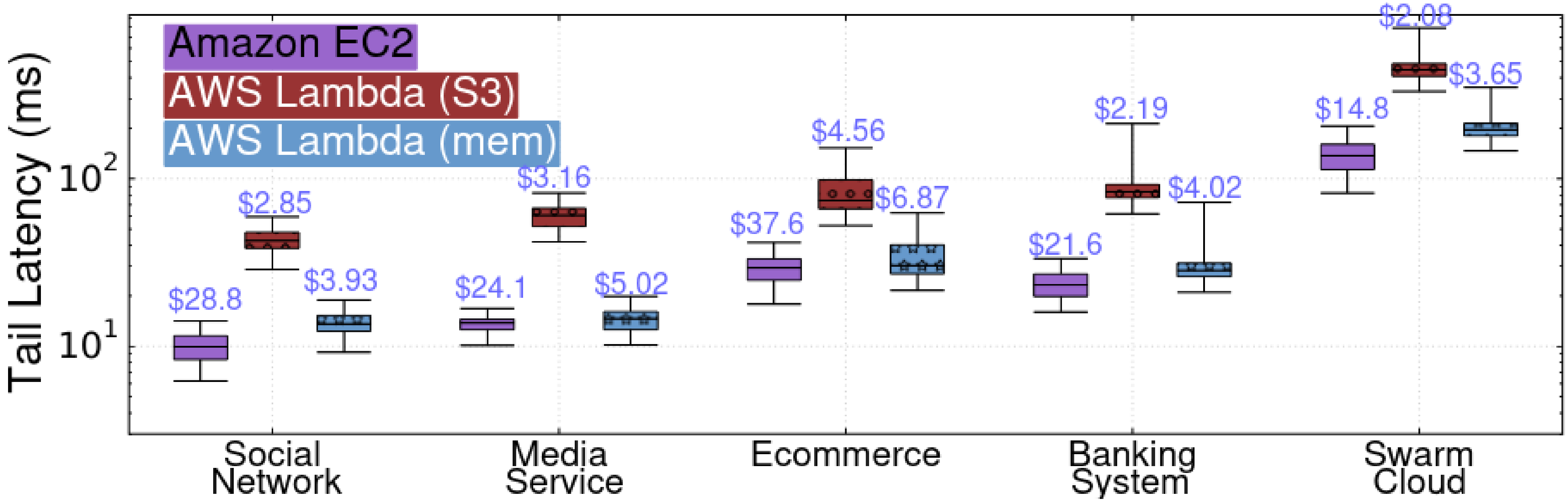


Figure 12. Tail latency with increasing load and decreasing frequency

See: Gan, Yu, et al. "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems.", ASPLOS 2019.

Tail at scale

Impact of a serverless approach on cost and tail latency



- Serverless can significantly reduce costs but with a negative impact on tail latency
- The Lambda (mem) approach is an ad-hoc solution to use the memory of additional VMs to transfer data between microservices

See: Gan, Yu, et al. "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems.", ASPLOS 2019.

About performance

Autoscalling

- How to decide which service to scale and when?
- Scale horizontally or vertically?

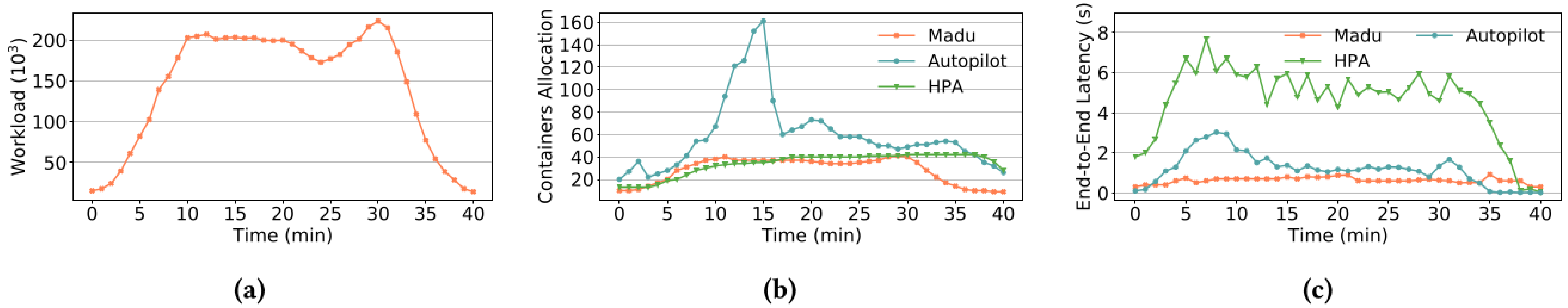


Figure 14: Comparison between reactive and proactive schemes using Media service. (a) Smoothly-fluctuated workload from Alibaba traces. (b) Resource allocation overtime. (c) 95th percentile end-to-end latency of online services.

In this experiment, the bad decisions of HPA have a 6x impact on latency

See: Luo, Shutian, et al. "The power of prediction: Microservice auto scaling via workload learning." SoCC 2022.