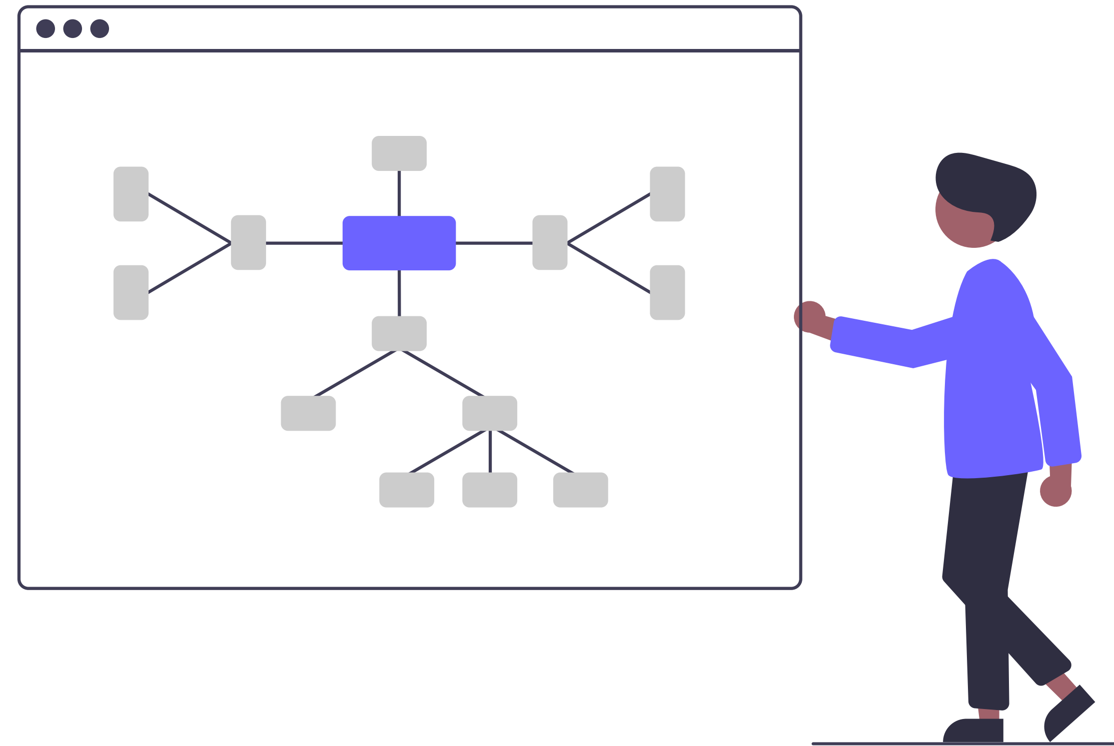
Pitfalls of Microservices

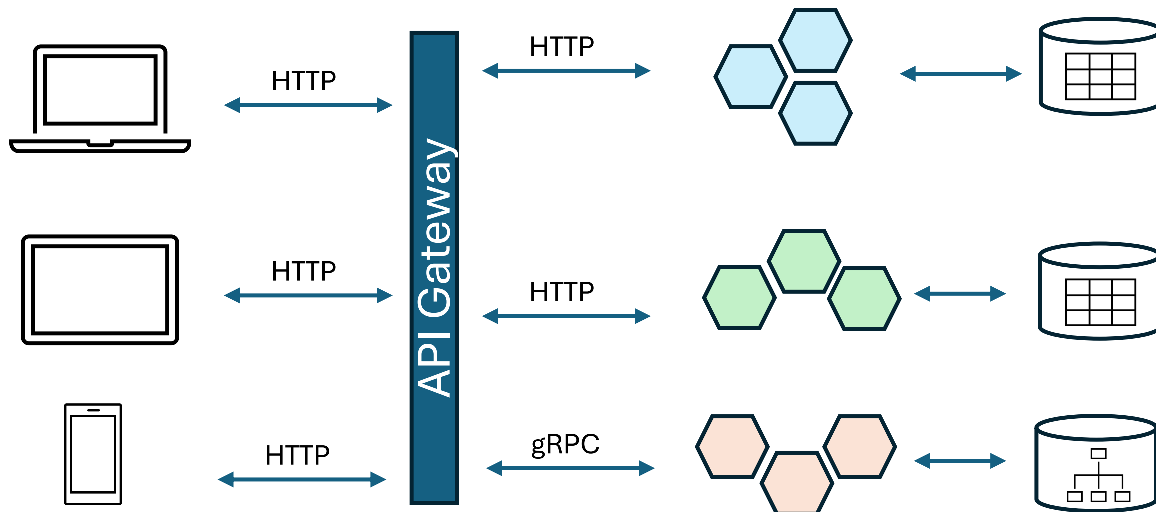
by Marian Veteanu



Introduction to Microservices

What are Microservices?

- A software architecture style where applications are broken down into smaller, independent services.
- Each service handles a specific business function and communicates with others over a network.



Why Microservices?

- When executed right, microservices offer SaaS products attributes such as scalability, flexibility, and independent deployment of services.

Why not Microservices?

- As awesome as may sound, microservices are <u>not</u> for everyone. Keep reading to understand some of the microservices pitfalls before you start your next project.

Microservices are not a silver bullet

While microservices can offer scalability, flexibility, and better fault isolation, they are not always the best solution for every product.

Simple applications

For small, straightforward applications with limited complexity and a single team, a monolithic architecture may be more appropriate.

Limited scalability needs

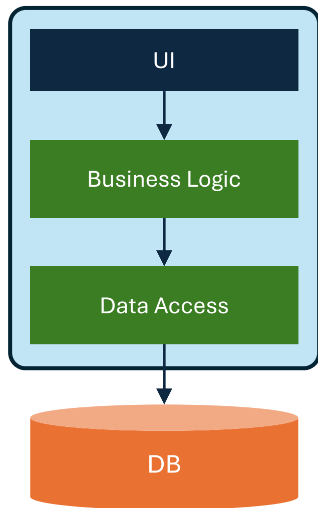
If the product does not require frequent scaling or doesn't anticipate high traffic, the complexity of microservices may add unnecessary overhead.

Resource constraints

Microservices require extensive infrastructure, monitoring, and operational resources. Small teams or startups might lack the resources to handle the added complexity.

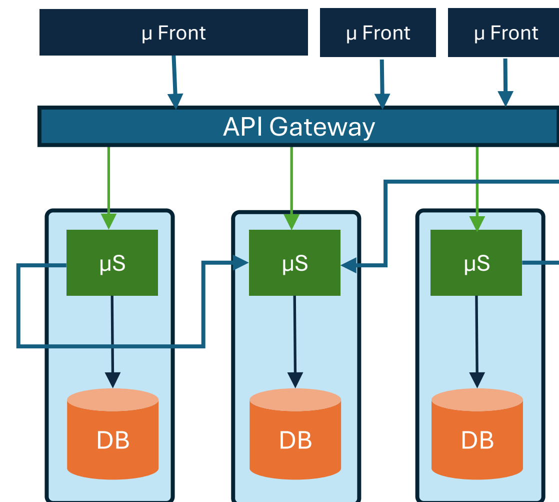
Pitfall 1: Increased Complexity

- Microservices introduce operational complexity compared to monolithic architectures.
- More services to manage and monitor.
- Multiple databases and networking issues.



Monolith architecture

VS



Microservices architecture

Each microservice is a separate entity that requires its own set of configurations for deployment, networking, security, logging, and resource allocation.

Deploying or scaling a service requires coordination across many independent teams.

Due to increased complexity, microservices are generally recommended for big-scale SaaS products. If the product is sold for on-premises installation, the additional overhead can create complications for your customers' IT personnel.

Pitfall 2: Overhead in small teams

Challenge:

Microservices architecture might be overkill for **small teams** or simpler applications.

Before deciding on a microservices architecture think if you have the right resources to develop, manage, monitor, and deploy each service.

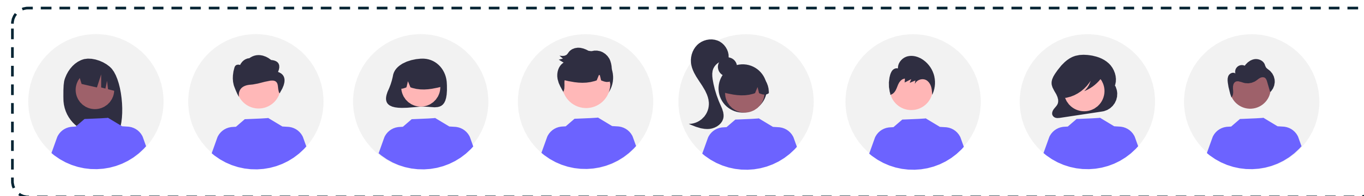
Challenge:

Teams must align on standards and practices.

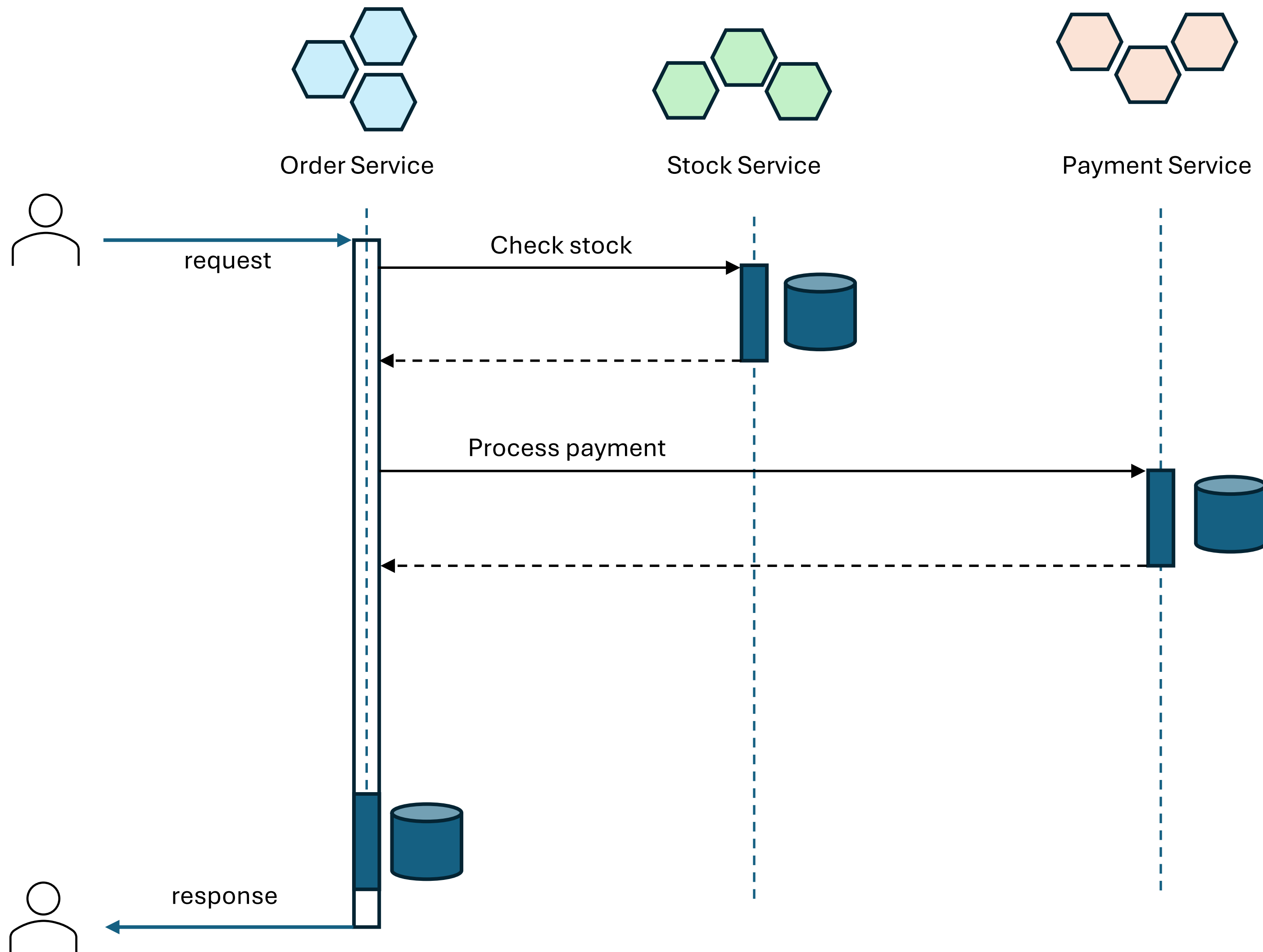
- Microservices enable different teams to use different tools, programming languages, and methodologies.
- Lack of a consistent approach can lead to fragmentation and inefficiency.

Example: A team working on Service A uses Go, while another team uses Node.js for Service B, causing a learning curve during cross-team collaboration.

R&D Department



Pitfall 3 – Data Consistency and Integrity



Example from: https://dilfuruz.medium.com/data-consistency-in-microservices-architecture-5c67e0f65256

Challenge:

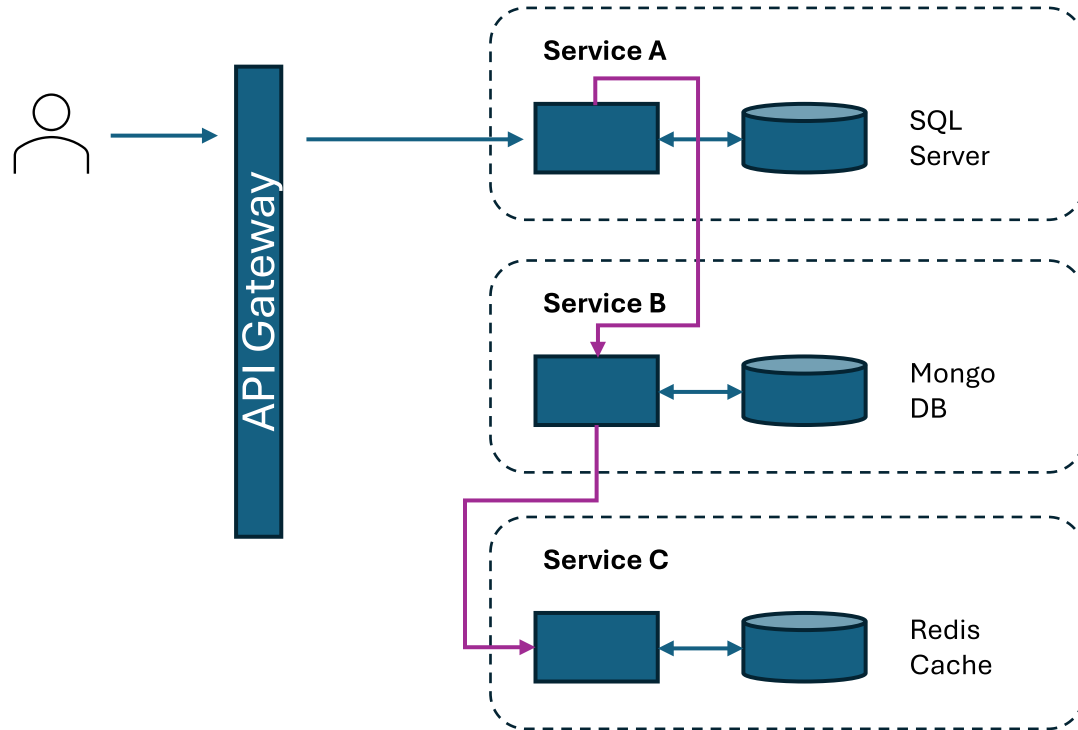
Maintaining consistency between distributed services.

Traditional databases provide consistency guarantees (ACID), but microservices often use separate DBs.

Example:

An order service updates the inventory service, but the payment service fails — how do you roll back?

Pitfall 4 – Latency and Performance



Challenge:

Increased network overhead and potential latency.

Inter-service communication introduces network latency.

Serializing and deserializing data across services.

Example:

Complex service chains where service A calls service B, which calls service C, leading to slow response times.

Pitfall 5 – Need to Implement Advanced Resilience Patterns

Circuit Breakers + Retries + Timeouts provide a robust mechanism for improving the fault tolerance of microservices.

Circuit Breakers

A circuit breaker is a pattern used to detect service failures and prevent a cascade of failures across services.

How it works:

When a service call fails repeatedly, the circuit breaker "opens," preventing further attempts to call the failing service.

If the service is healthy again, the circuit breaker "closes," allowing normal traffic flow.

Benefits:

Prevents overloading failing services.
Protects upstream services from cascading failures.

Timeouts

Setting timeouts for service calls ensures that services don’t hang indefinitely waiting for a response from a slow or failing service.

How it works:

When making a call to another service, a timeout value is set.

If the service does not respond within the timeout period, the call is aborted, and the failure is handled by a retry or fallback mechanism.

Benefits:

Prevents services from waiting indefinitely for a response.
Ensures resources are not consumed by long-running or hung requests.

Retry Mechanisms

Automatic retries allow microservices to handle transient failures, such as network timeouts or temporary service outages.

How it works:

When a service call fails, the retry mechanism attempts to call the service again after a short delay.
The number of retry attempts and the delay between retries can be configured based on the expected failure recovery time.

Benefits:

Improves fault tolerance for services with intermittent issues.
Avoids unnecessary service outages for temporary network or service glitches.

Example: If a payment service is down, retries can be made, and after several failures, the circuit breaker opens to prevent further traffic.
Timeouts ensure calls do not hang indefinitely while waiting for the service to respond.

Pitfall 6 – Service Dependencies and Orchestration

Challenge:

Hidden service dependencies create complex relationships.

Circular dependencies between services can cause cascading failures.

Service orchestration requires robust management tools.

Example:

A single service failure might trigger failures in downstream services if dependencies are not properly managed.

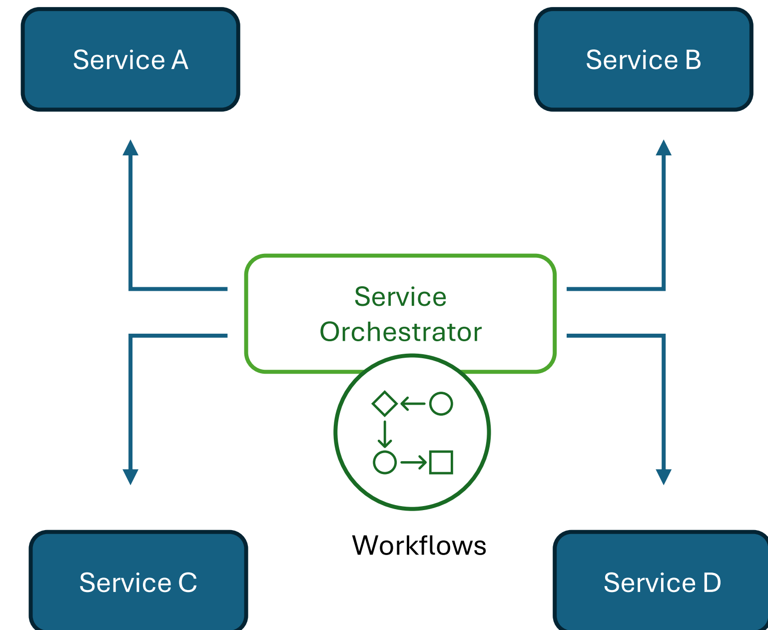


Diagram from: https://cloud.google.com/blog/topics/developers-practitioners/service-orchestration-google-cloud

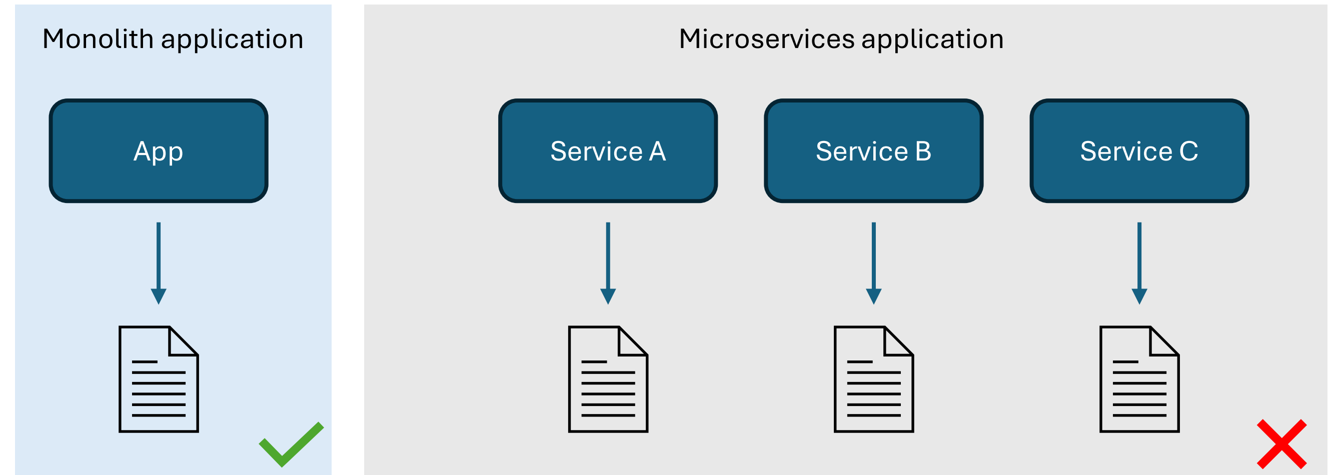
Pitfall 7 – Monitoring and Debugging

Challenge:

Difficult to trace failures across services.

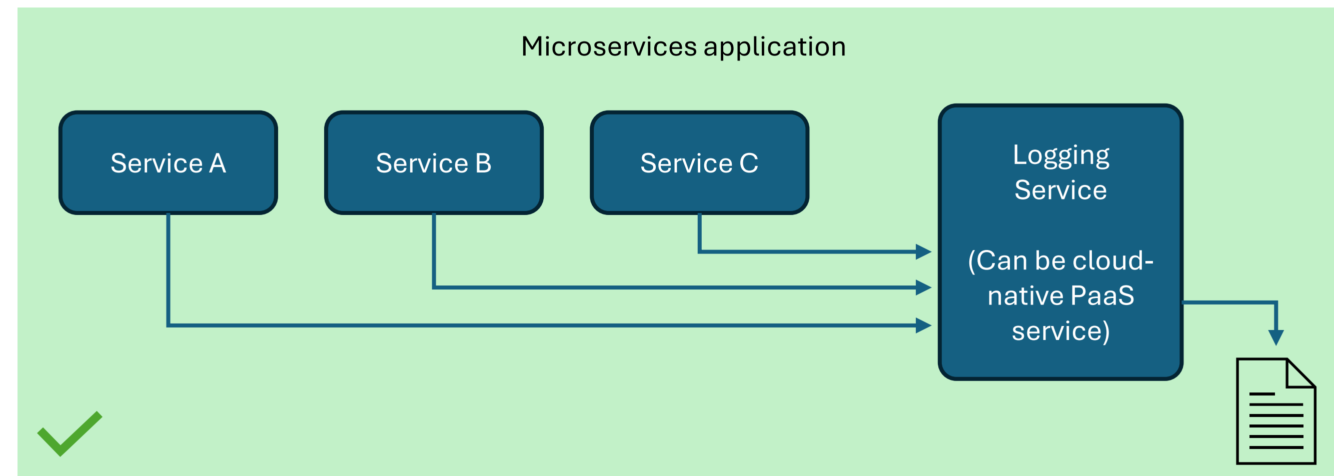
Debugging issues in one service might require correlating logs from multiple services.

Centralized logging becomes a must.



Example:

A user faces an issue in the UI, but the root cause could be hidden deep within the microservices ecosystem.



Pitfall 8 – Version Management

Challenge:

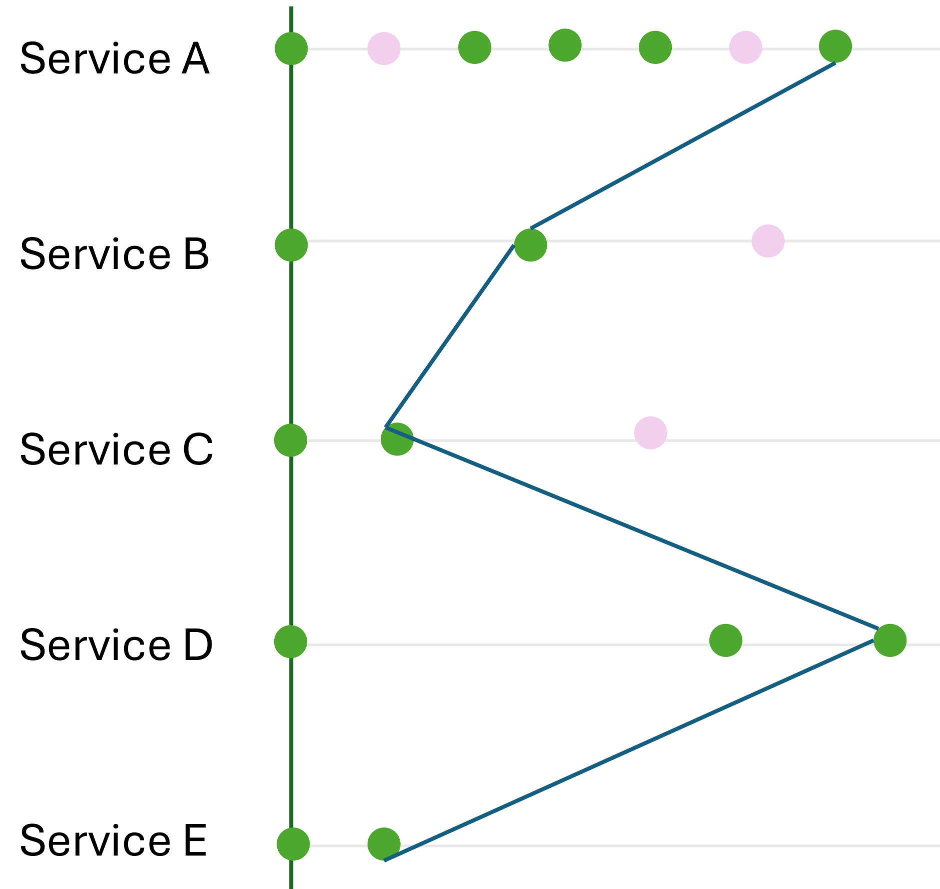
Coordinating versioning, API compatibility, and service dependencies.

CI/CD pipelines must account for many services and their interactions.

Example:

Service A is deployed but depends on an old version of Service B, causing compatibility issues.

Keeping dependent services compatible when updating a single service is tricky.



Pitfall 9 – Security

Challenge:

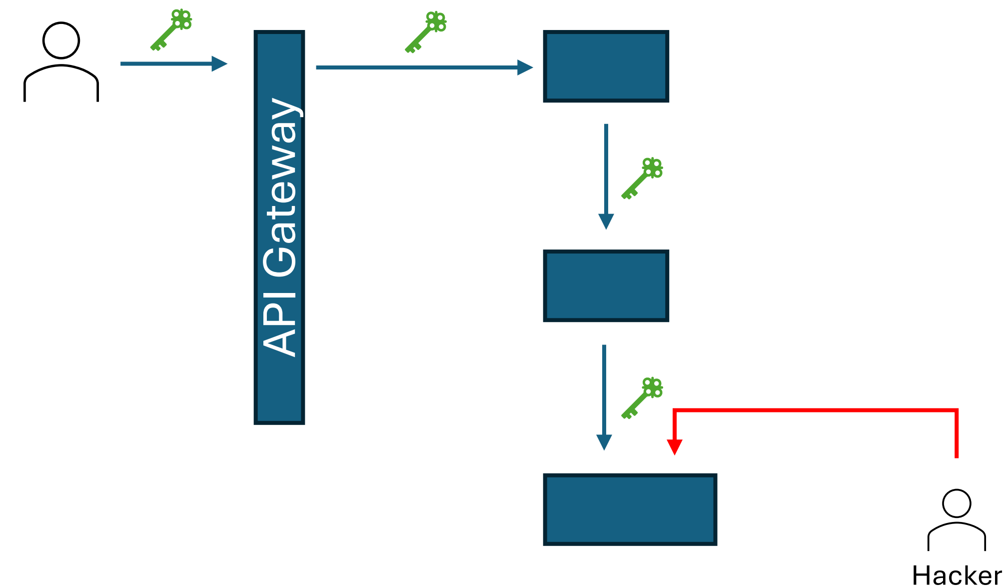
Distributed services increase the attack surface.

It is recommended to implement each service with its own security measures (authentication, authorization).

Securing inter-service communication is vital.

Example:

A compromised service could be used as an entry point to attack other services.



Study results: Drivers and Barriers for Microservice Adoption

Drivers for microservice adoption in different industries

	Overall		Development / Consulting	Energy / Industry	Financial Services	Retail / E-Commerce	Other / Unknown
	Score (mean)	SD	Score (mean)	Score (mean)	Score (mean)	Score (mean)	Score (mean)
High Scalability and Elasticity	2.14	0.72	2.19	2.18	2.15	2.67	1.89
High Maintainability	2.11	0.75	2.31	2.10	2.15	2.17	1.89
Short Time to Market	2.07	0.82	2.12	1.45	2.10	2.67	2.17
Enabler for CD and DevOps	1.61	0.89	1.69	1.27	1.70	1.83	1.56
Suitedness for Cloud and Docker	1.55	0.89	2.00	1.09	1.35	1.50	1.67
Organizational Improvement	1.37	0.81	1.31	1.18	1.50	1.83	1.22
Polyglot Programming	1.28	0.90	1.00	1.82	1.15	2.00	1.11
Polyglot Persistence	1.27	0.83	1.06	1.09	1.40	1.33	1.39
Attractiveness as Employer	0.87	0.86	1.00	0.55	0.85	1.67	0.72

Barriers for microservice adoption in different industries

	Overall		Development / Consulting	Energy / Industry	Financial Services	Retail / E-Commerce	Other / Unknown
	Score (mean)	SD	Score (mean)	Score (mean)	Score (mean)	Score (mean)	Score (mean)
Insufficient Ops Skills	1.71	0.82	2.06	1.27	1.53	1.67	1.89
Resistance by Ops	1.66	0.87	1.88	1.64	1.68	1.33	1.56
Insufficient Dev Skills	1.63	0.95	1.69	1.00	1.79	1.67	1.78
Deployment Complexity	1.41	0.79	1.62	0.73	1.35	1.00	1.83
Compliance and Regulations	1.21	1.11	1.38	0.45	1.50	1.17	1.22
Compatibility Issues	1.13	0.96	1.06	1.09	1.05	0.50	1.50
Consistent Backups	1.13	0.88	1.00	0.73	1.60	1.00	1.00
Maturity of Technology	0.97	0.78	1.06	0.73	1.05	0.83	1.00
Resistance by Devs	0.97	1.06	1.06	1.18	1.10	0.50	0.78
Support Contracts / Licenses	0.89	0.77	0.56	0.73	1.20	0.50	1.06

Data extracted from paper:

Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany

Study results: Microservices issues, advantages and disadvantages

Table 1: The Issues Identified

Issue	Description	Votes
Wrong Cuts	Decomposing the monolith into microservices is a complex task. Understanding which is the most effective decomposition strategy and where to "cut" the existing system is still not clearly defined.	12
Continuous Architecture	The continuous changes required to the architecture require a lot of attention and often create issues, especially if the architecture is not documented.	8
Monitoring	Monitoring microservices require a distributed monitoring strategy while monoliths are easier to monitor	3
Testing Contracts	Testing the whole system is more complex. Testing interfaces must remain stable, a set of testing contracts should be clearly defined between microservices.	3
Versioning	Microservices versioning often create issues for connected services. As example, API should be deprecated and not simply changed, however, the API versioning is still an issue that would require a simpler mechanism.	2
State Management	Microservices are stateless, however in some cases we do need to keep track of the application state.	2
System Effort Estimation	The effort estimation of the whole system is more complex and less accurate compared to monolithic systems.	

Table 2: The Advantages Identified

Advantage	Description	Votes
Simple Scalability	Because of their nature, microservices are easier to scale than monoliths	10
Clear Boundaries	Clear boundaries reflect clear responsibilities, both for the service point of view (a microservice should do only one thing) and from the development team (the team is clearly responsible of exactly one microservice without having shared responsibilities with other teams.	6
Independent Deployment	Microservices can be deployed independently from the rest of the application while teams working on monoliths need to synchronize to deploy together.	4
Quick Feedback Loops	Independent deployment support fast releases and therefore allowing quick customer feedbacks.	4

Table 3: The Disadvantages Identified

Disadvantage	Description	Votes
Need of senior developers	The architectural decisions are complex and moreover, starting the development of a microservices-based system require highly skilled developers.	12
Difficult to learn	Because of the technologies adopted by microservices (orchestration, distributed system, API management and others) microservices require a lot of training.	12
Increase of system complexity	The spread of microservices and the increase of size of the ecosystem of microservices commonly lead to an increase on the system complexity.	2

Data extracted from paper:

Microservices in Agile Software Development: a Workshop-Based Study into Issues, Advantages, and Disadvantages

Marian Veteanu
Technology Architect and Product Leader

Looking to see how I can add value to your organization? Message me!

https://www.linkedin.com/in/mveteanu/
https://x.com/mveteanu