DATE 25 03 22

Microservices

- These are an architectural and organizational approach to s/w development where s/w is composed of small independent services that communicate over well-defined API's. These services are owned by small, self-contained teams.
- These are the result of problems with two architectural paradigms:

(1) **Monolith →**
- Original architecture
- All s/w components are executed in a single process.
- No distribution of any kind.
- Strong coupling b/w all classes.
- Usually implemented as Silo.

Pros - Easier to design
Performance

(2) **Service Oriented Architecture (SOA) →**
- First coined in 1998
- Apps are services exposing functionality to the outside world.
- Services expose metadata to declare their functionality.
- Usually implemented using SOAP and WSDL.
- Usually implemented with ~~ESB~~ ESB.

DATE

Pros → sharing data + functionality
Polyglot b/w services

★ Single technology platform →
- With monolith, all the components must be developed using the same development platform.
- Not always the best for the task.
- Can't use specific platform for specific features.
- Future upgrade is a problem - need to upgrade the whole app.

★ Inflexible deployment →
- With Monolith, new deployment is always for the whole app.
- No way to deploy only part of the app.
- Even when updating only one component - the whole codebase is deployed.
- Forces rigorous testing for every deployment.
- Forces long development cycles.

★ Inefficient Compute Resources →
- With monolith, computer resources (CPU and RAM) are divided across all components.
- If a specific component needs more resources - no way to do that.
- Very inefficient.

DATE

☆ Complicated & expensive ESB

- With SOA, the ESB is one of the main components.
- Can quickly become bloated and expensive.
- Tries to do everything.
- Very difficult to maintain.

☆ Lack of tooling →

- For SOA to be effective, short development cycles were needed.
- Allow for quick testing and deployment
- No tooling existed to supported this.
- No time saving way achieved.

☆ Microservices Architecture →

- Microservices is an architectural design for building a distributed app. using containers. They get their name because each func. of the application operates as an independent service. This architectural allows for each service to scale or update without disrupting other services in the app.

Characteristics →

(i) Componentization via services →

- Modular design is always a good idea.
- Components are the parts that together compose the s/w.
- Modularity can be achieved using:

- Libraries → called directly within the process.
- Services → called by out-of-process mechanism (Web API, RPC).

- In Microservices we prefer using services for the componentization.
- Libraries can be used inside the service.
- Motivation:
  - Independent deployment
  - Well defined interface

(2) Organized around business capabilities →

- Traditional projects have teams with horizontal responsibilities - UI, API, Logic, DB, etc.
- With Microservices, every service is handled by a single team, responsible for all aspects.
- With Microservices, every service handles a well-defined business capability.
- Motivation:
  - Quick development
  - Well-defined boundaries.

(3) Products not Projects →

- With traditional projects, the goal is to deliver a working code.
- No lasting relationship with the customer.
- often no acquaintance with the customer.
- After delivering - the team moves on to the next project.
- With Microservices - the goal is to deliver a working product.

385: 1683
929: 2781

DATE

- A product needs ongoing support and requires close relationship with the customer.
The team is responsible for the microservice after the delivery too.
- Motivation:
  - Increase customer's satisfaction.
  - Change developer's Mindset.

④ Smart endpoints and dumb pipes →
- Traditional SOA projects used 2 complicated mechanisms:
  - ESB
  - WS-* protocol.
- Made inter-service communication complicated and difficult to maintain.
- Microservices systems use "dumb pipes" - simple protocols.
- Strive to use what the web already offers.
- Usually - REST API, the simplest API in existence.
- Motivation:
  - Accelerate development
  - Make the app easier to maintain.

⑤ Decentralized Governance →
- In traditional projects there is a std. for almost anything
  - Which dev platform to use.
  - Which db to use
  - How logs are created
  - And more.

- With Microservices each team makes its own decision.
  - " same as above.
  - "
  - "

- Each team is fully responsible for its service.
  - "you build it, you run it".
- and so will make the optimal decision
- enabled by the loosely coupled nature of Microservices
- Multi dev platform is called polyglot
- Motivation:
  - Enables making the optimal technological decision for the specific service.

(6) Decentralized Data Management →
- Traditional systems have a single db.
- Stores all the system's data from all the components.
- With Microservices each service has its own db.
- Motivation:
  - Right tool for the right task - having the right db is IMP.
- Encourages isolation.

(7) Infrastructure automation →
- The SOA paradigm suffered from lack of tooling.
- Tooling greatly helps in deployment like:
  - Automated testing
- For Microservices automation is essential.

DATE

- Short deployment cycles are a must.
- Can't be done manually.
- There are a lot of automation tools:
  - Azure devops
  - Gitlab
  - Jenkins
- Motivation:
  - Short deployment cycles

(8) Design for failure →

- With microservices there are a lot of processes and a lot of n/w traffic.
- A lot can go wrong.
- The code must assume failure can happen and handle it gracefully.
- Extensive logging and monitoring should be in place.
- Motivation:
  - Increase system's reliability

(9) Evolutionary design →

- The move to microservices should be gradual.
- No need to break everything apart.
- Start small and upgrade each part separately.

DATE

★ Communication patterns →

- Efficient communication b/w services is crucial.
- It's imp. to choose the correct commu. pattern.

- 1-to-1 sync
- 1-to-1 Async
- Pub-Sub / Event driven.

★ Design the architecture →

- Service's architecture is no diff. from regular s/w.
- Based on the layers paradigm.

★ Testing microservices →

- Testing is imp. in all systems and all architecture types.
- With microservice it's even more imp.
- It poses additional challenges.

Types →

- Unit tests
- Integration tests
- End-to-end tests

Challenges →

- Microservices systems have a lot of moving parts.
- Testing and tracing all the services is not trivial
- Testing state across services.
- Non-functional dependent services.

DATE

★ Service Mesh →

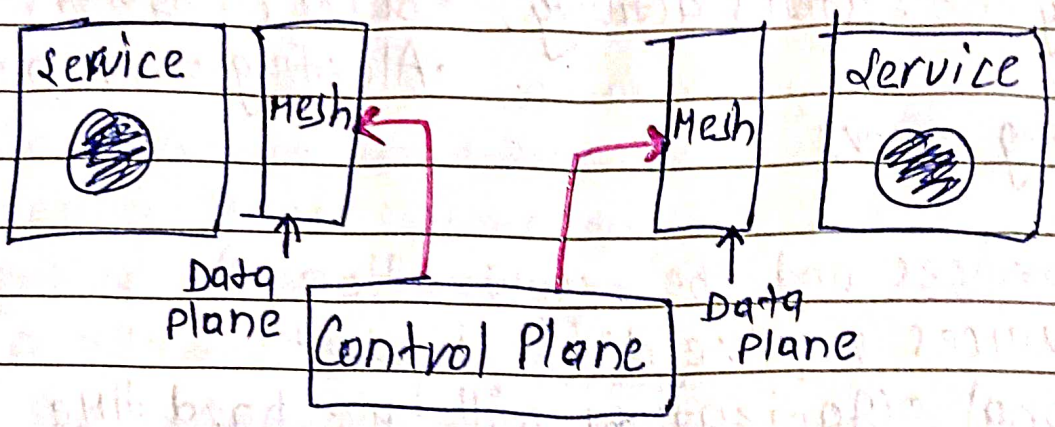
- Manages all service-to-service communication.
- Provides additional services.
- Platform agnostic (usually...).

- **Problems solved →**
  - Microservices communicate b/w them a lot.
  - The communication might cause a lot of problems and challenges.
    - Timeouts
    - Security
    - Retries
    - Monitoring.
- < S/w components that sit near the service and manage all service-to-service communication.
- Provides all comm. services.
- The service interacts with the service mesh only.

- **Services →**
  - Protocol version
  - Comm. security
  - Authentication
  - Reliability (timeouts, retries, health checks, circuit breaking).
  - Monitoring
  - Service discovery.
  - Testing (A/B testing, traffic ~~split~~ splitting)
  - Load balancing.

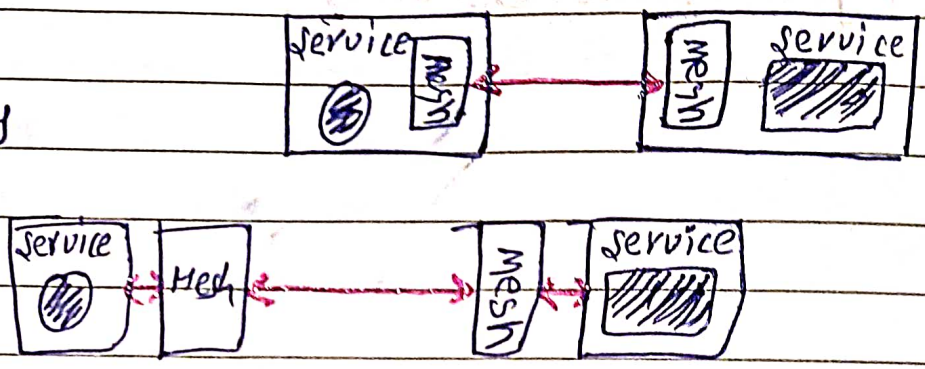
DATE

- Architecture →



- Types →

1. In-Process
2. Sidecar



In Process	Side Car
• Performance	• Platform
	• Code agnostic

↑

More popular

★ Logging and Monitoring →

- Extremely imp. in microservices
- Flow goes through multiple processes.
- Hard to get wholistic view.
- Hard to know what's going on with the services.

DATE

Logging

- Recording the system's activity.
- Audit
- Documenting errors

Monitoring

- Based on system's metrics
- Alerting when needed.

Microservices and the organization →

- Microservices require diff. mindset.
- Traditional organizations will have hard time succeeding with microservices.