ASJ.

MICROSERVICES

Monolith → The Original architecture.
- All s/w component are executed together.
- Strong coupling b/w all layers.
- implemented as Silo
- Easier to design
- Faster than distributed

Service Oriented Architecture →
- Coined in 1998
- Sharing & giving capiblity.
- Implemented using SOAP & WSDL.
- Implemented with ESB (Enterprise Service Bus).

Adv
- Sharing data & functionality.
- Polyglot Between Services. (platfor independent)

Problem with above & Architecture.
- Problem relevent to technology, deployment, cost & more

② Single technology platform - Monolith
- Not always best for the task
- Can't use specific platform for the specific users
- Future upgrade is a problem - ie need to upgrade the whole app.

② Inflexible deployment →
- New deployment → then whole app.
- No way to deploy one part of app, even when updating one component, need to deploy whole code base
- Force regerous testing with every deployment
- Long development cycle.

(iii) Inefficient Compute Resources -
- With monolith Compute resources (CPU & RAM) are divided across all components.
- If a specific component needs more resources - no way to do that
- Only choice is that we have to increase the whole power of computer resources.

(iv) Large & Complex problems
- With monolith, codebase is large & complex
- Little change can affect other components
- Testing not always detects all the bugs
- Very difficult to maintain.

(v) Complicated & Expensive ESB
- With SOA, the ESB is main component
- This ESB may quickly become bloated & expensive
- Difficult to maintain ESB
- .

(vi) Lack of Tooling :-
- For SOA to be effective, short development cycle were needed.
- Allow for quick testing & deployment.
- No tooling existed to support this.
- No time saving was achieved.

Microservice Architecture :-

- Problem with monolith & SOA lead to new paradigm
- Need of simple API.
- Microservices appeared in 2011

* Characteristics :-

① Componentization →

- Modular design is always a good idea.
- Modularity can be achieved using • Libraries • Services (web API)
- In Micro... we prefer using services for the componentization
- Libraries can be used inside the service.
- * Using services make our component independently deployable.
- well defined interface.

② Organized around business capabilities :-

- Traditional projects have team with horizontal responsibility UI, ⇔ API, Logic, DB etc.
  :- thus slow inter-group communication.

with microservices every service is handled by a single team, responsible for all aspects.

- Team has one goal. * Single team responsible for single business ~~com~~ capability.
  * Quick development. • well defined boundaries.

③ Products Not Projects →

- Earlier no communication with Customer. & After ~~Boot~~ delivery - team moves into the next project

* But with Microservices ⇒ The goal is to deliver a working product.

- needs ongoing support & close relationship with the customer

- The team is responsible for the service after the delivery too.

** Increased customer satisfaction · changing developer mindset.

④ Smart end points & Dumb Pipes :-

* User Simple protocol for communication to other service
- Usually → REST API,
- Accelerate the development, Maker app easier to maintain.

⑤ Decentralized Governance :-

- Each team maker its own decision like - which dev platform to use, database to use, etc.
- Each Team is fully responsible for its services

⑥ Decentralized Data Management →

- Traditional system have a single D.B
- With Microservice, each service has own D.B but with distributed db & transactions, we may have cases of duplication & more
- right DB for right task
- Encourage Isolation

⑦ Infrastructure Automation :-

- Automated testing, deployment with help of tools like Azure Devops, Gitlab, Jenkins.

⑧ Design for Failure :-

- With microservice there are lot of process & a lot of network traffic.

* To resolve it, we use Monitoring tools. like - Kubernetes.

① Eureka ② Main
Zuul-server

Problem solved by Microservices :-
① Single technology Platform
• Decentralized governance allows each teams to decide which platform to use, which DB to use.

② Inflexible deployment :-
* each service can be deployed independently

③ Inefficient Compute resources :-
• Each service runs in its own process.

④ Large & Complex :-
• Each services are isolated

① Zuul server → CMD → mvn spring-boot:run

Sept 6 # Designing Microservices Architecture.

#.> Mapping the components :-
• Defining the various components of the systems
• Components means services.

Mapping should be based on:
① Business Requirements :-
• The collection of requirements around a specific business capablity. eg. Order Management.
• Add, remove, update, calculate amount.

② Functional Autonomy →

eg →

(iii) Data entities →:
- Services is designed around well-specified data entities eg. orders, items. . order stores the customer ID.

(iv) Data Atomicity →
eg. Employee service that relies on Address service to retrieve employee data.

(2) Defining Communication Pattern →.
- Efficient communication b/w services is crucial
- important to choose correct communication pattern

- 1-to-1 Sync → A service call another service & waits for the response.

- 1-to-1 Async → . A service call another service & continues working.
- Does not wait for Response - Fire & Forget.
- Usually when the first service wants to pass a message to the other service

- Pub-Sub / Event Driven →: . A service wants to ~~no~~ notify other service about something.
- The service has no idea how many services listen
- Does not wait for response - Fire & Forget.

(3) Select Technology Stack →.
- Focus on Backend & storage platform.
- Make concrete decision based on hard evidence.

Backend development platform.

Language	App Type	Type System	cross platform	Community	Perform	learning curve
.NET	All	Static	No	Large	OK	Long
.NET Core	web Apps web API Console service	Static	Yes	Medium & growing	Great	Long
Java	All	Static	Yes	Huge	OK	Long
node.js	web apps web API	Dynamic	Yes	Large	Great	Medium
PHP	web Apps web API	Dynamic	Yes	Large	OK	Med.
Python	All	Dynamic	Yes	Huge	OK	Short

Storage Platform.

① Relational Database → SQL Server, MySQL, PostgreSQL

② NoSQL Database → MongoDB, amazon DynamoDB, ...

③ Cache → inmemory database → for fast access, data which changes frequently. eg. redis

→ Stores Data in tables.

→ Emphasis on scale & performance, • No schema • Stored in Jason Format

④ Object Store → Stores un-structured, large data • documents, Photos, Files

eg. amazon S3, MINIO, Microsoft Azure

④ Design the Architecture :—

• Based on layers paradigms :—

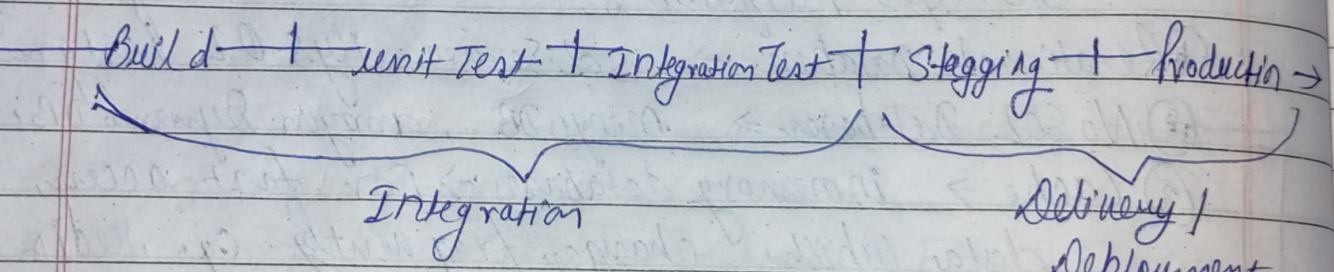
* Layer represent horizontal functionality.

VS / SI	→ Expose API → Json Handling, Auth
Business Logic	→ Validation → Enrichment, Computations
Data Access Layer	→ Connection Handling → Querying / Saving data → Transaction Handling

* Purpose of Layer :-
  - Forces well formed & focused code.
  - Layer makes our code Modular i.e. changes can be made easily.

Deploying Microservices.

* CI/CD → The full automation of the integration & delivery stages.



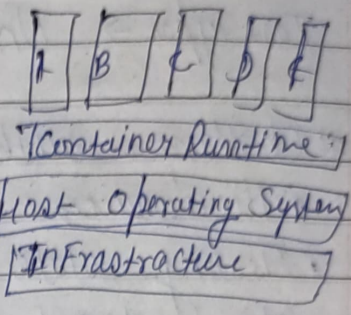
CI/CD

* Faster release cycle
* Reliability
* Reporting

CI/CD Pipelines :-
- Pipelines are heart of CI/CD process.
- Usually defined using YAML, with UI representation.

As the Architect →
- Make sure there is a CI/CD engine
- Help shape the steps in the pipeline.

Containerry :- • This packaging model.
• Packages s/w, its dependencies & configuration files
• Can be copied b/w machines.
• Uses the underlying o/s.
• Uses same o/p as host



light weight ←

why containerry?
* • Predictability
• Performance
• Density → One server can run 1000s of containers

Docker → : Most popular container environment.
Docker Daemon → is the docker server
image → Contains s/w to run
container → instances of image.
Registory → Repository of images. Can be public or private
cli → Manages the images & containers

* Docker cli capablity is to ability to pull images.
* Dockerfile → Contains instructions for building custom image

Containerry Management →
Kubernetes → Most popular container management platform
• Provides all aspect of container management —
-Routing, Scaling, High-availability, Automated-deployment, Configuration-management.

Testing Microservices :-

* An important thing in all systems & all architecture type

Test types → Unit test, Integration Test, End-to-End Test

Challanges with Microservices testing.
- Microservices systems have a lot of moving parts
- Testing state across services.
- Non-functional dependent services

① Unit Tests :- Tests individual code units such as - method, interface etc
- Usually automated.
- Developed by the developers

in Microservices →
- Test only in-process code
- Use same framework & methodologies.

(ii) Integration Test →
- We test the services functionality.
- Cover allmost all code path in the service.
- Some path may include accessing external objects like database, other services
- Developed & conducted by the QA team.
- Should be automated

(iii) End to End Tests →
- Test the whole flow of the system
- Touch all services
- Test for end state.

Service Mesh :-

Service Mesh manages all service-to-service communication

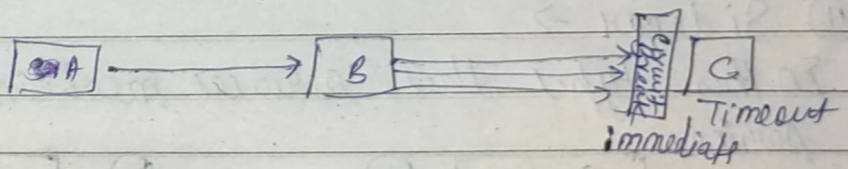
Problems solved by Mesh :-

- As Microservices communicates between them a lot
- The communication might cause a lot of problems to challenges like - timeouts, security, retries, Monitoring

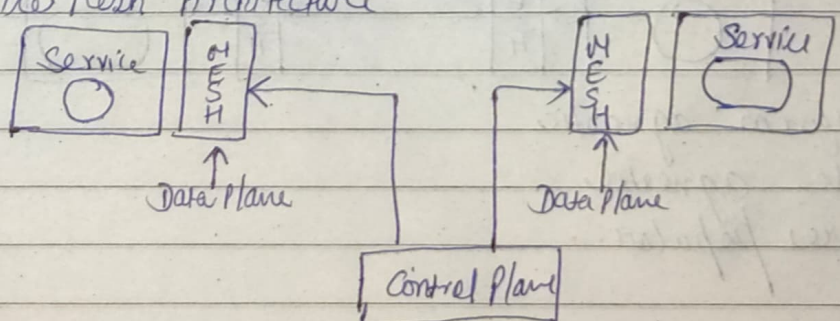
Services provided by Service Mesh -

- Protocol Conversion
- Communication security
- Authentication
- Reliability
- Monitoring (timeouts, retries checks, Circuit breaking)
- Service Discovery
- Testing (A/B testing, traffic splitting)
- Load balancing.

Circuit Breaker → • Prevents cascading failures when a service fails.



Service Mesh Architecture

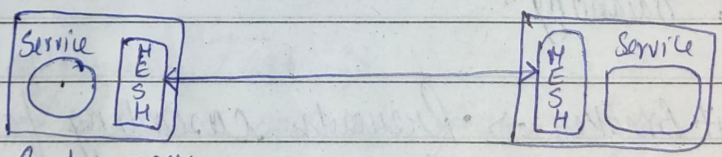


Data plane - Does all heavy lifting like, it converts protocol, handles security, implement circuit break, retries timeout

Control Plane → Controls to manages data plane
- It make sure that data plane works according to actual need
- eg → Defines what kind of protocol needed, what kind of security needed.

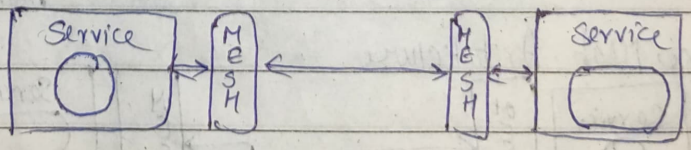
Types of Service Mesh :-

(i) In-Process :- & service
We have service mesh is implemented as a software component which is a part of service process itself. Then if services wants to communicates then it simply calls the method in mesh.



- Performance

(ii) Sidecar →
In this type the service mesh is outside the service



- platform agnostic
- Code agnostic
- More popular.