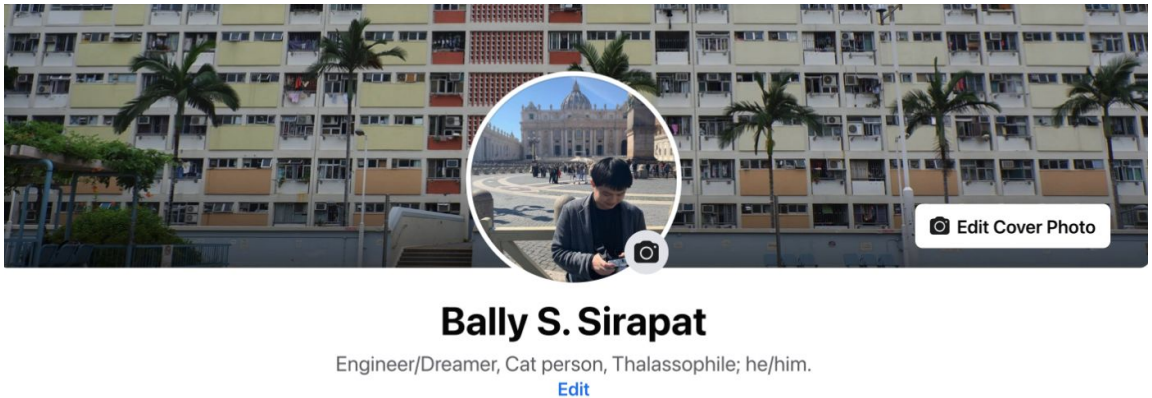
A Journey to Microservices

Or how to create microservices for newbie

Who am I?

- Sirapat Siribandit (Ball)
- Senior software engineer @ agoda
- Question, discussion and feedback are always welcomed



Agenda

- Part 1 - Introduction
- Part 2 - Microservices, How?
- Part 3 - Workshop 1
- Part 4 - From monolith to microservices
- Part 5 - Testing and deployment
- Part 6 - Dive deep into microservices pattern
- Part 7 - Workshop 2
- Part 8 - Epilogue

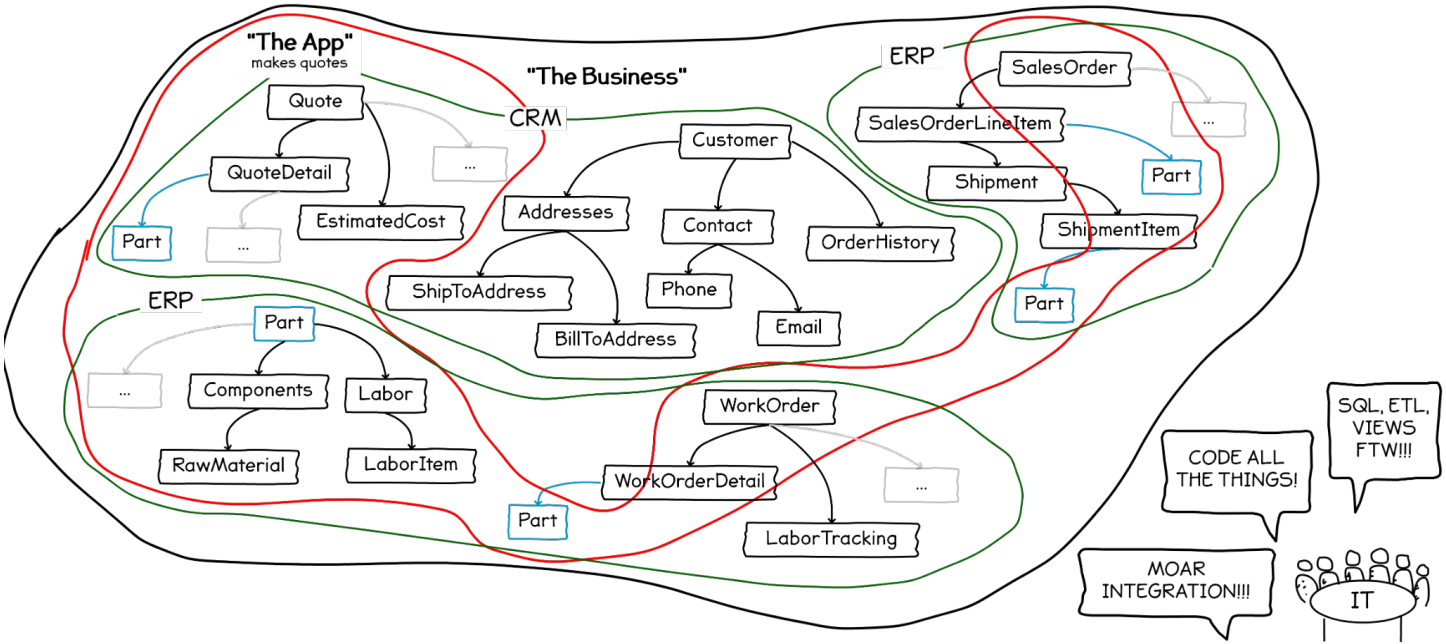
Part 1: Introduction

What is microservices?

- Architecture style or software architecture pattern.
- Structures an application into collection of services
  - Highly maintainable and testable
  - Single purpose
  - Loosely coupled, high cohesion
  - Independently deployable
  - Organized around business capabilities
  - Owned by a small team
- Normally people/organizational culture problem
- Is it fit in your context
- Organizational culture reflect company microservices architecture

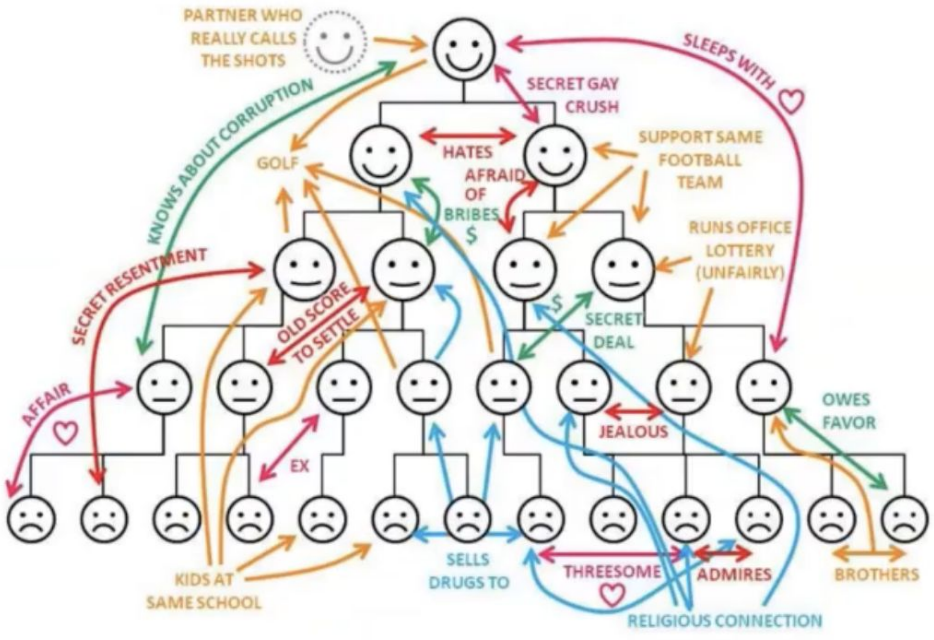
But more important question. Why microservices?

Architecture hell



s-inno.cdgs.co.th/sinno/index.php/technology/19-2017-03-23-06-52-23/app-dev-group/210-microservices-journey

Organizational culture



From book Team of teams

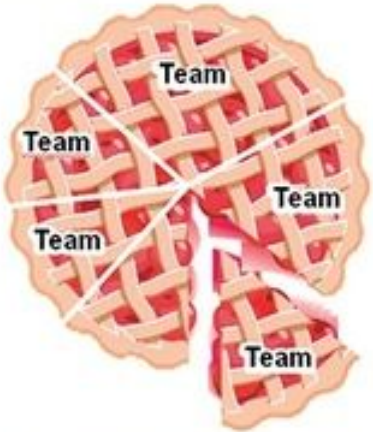
1990s and earlier

Pre-SOA (monolithic)
Tight coupling

For a monolith to change, all must agree on each change. Each change has unanticipated effects requiring careful testing beforehand.

2000s

Traditional SOA
Looser coupling



Elements in SOA are developed more autonomously but must be coordinated with others to fit into the overall design.

2010s

Microservices
Decoupled

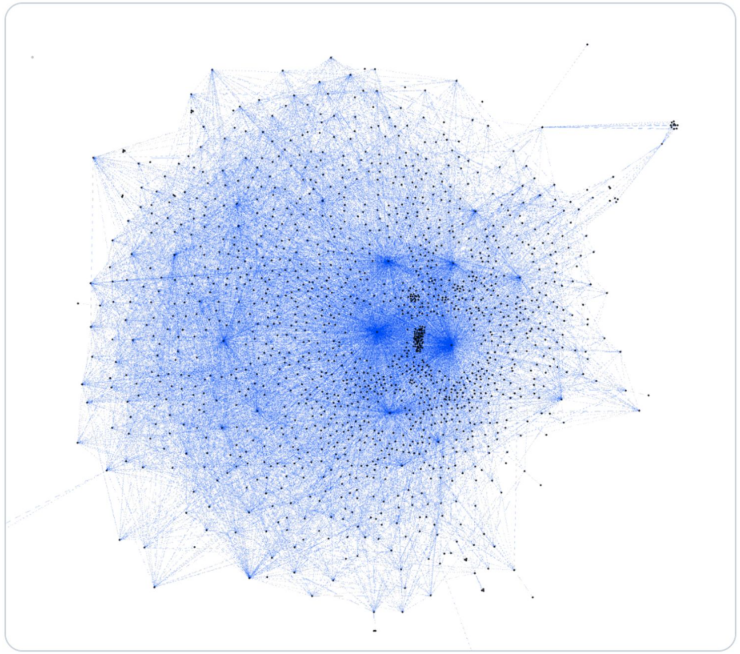
Developers can create and activate new microservices without prior coordination with others. Their adherence to MSA principles makes continuous delivery of new or modified services possible.

www.quora.com/Whats-the-difference-between-service-oriented-architecture-SOA-and-microservices

Architecture size

Jack Kleeman
@JackKleeman

1500 microservices at @monzo; every line is an enforced network rule allowing traffic



3:47 PM · Nov 1, 2019 · Twitter Web App

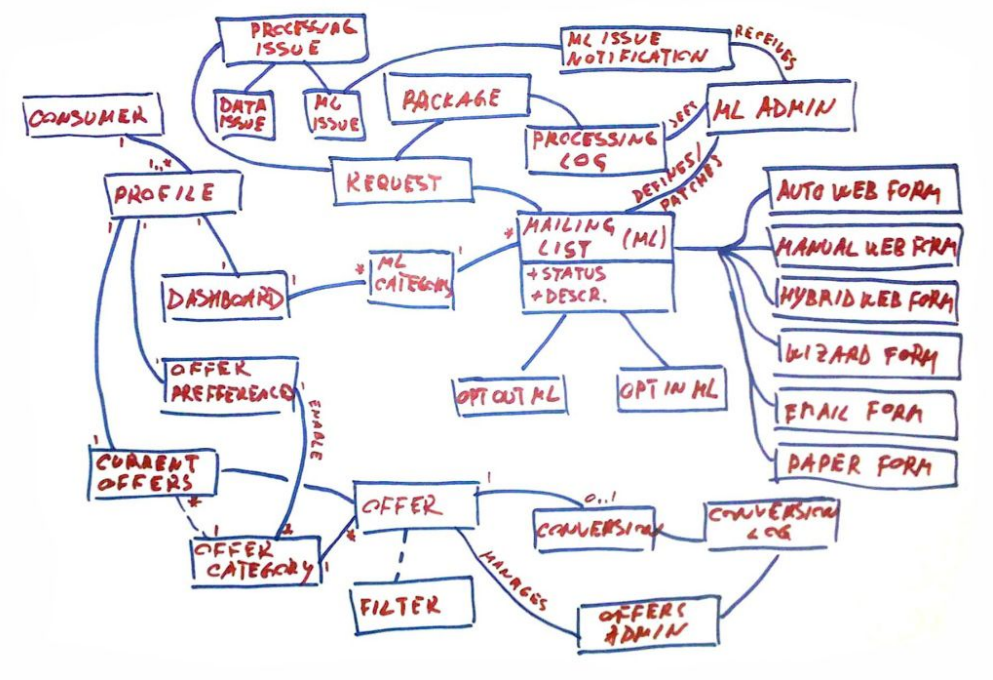
642 Retweets **2.7K** Likes

What is service

- Standalone, Independently deployable software component which implement some useful functionality.

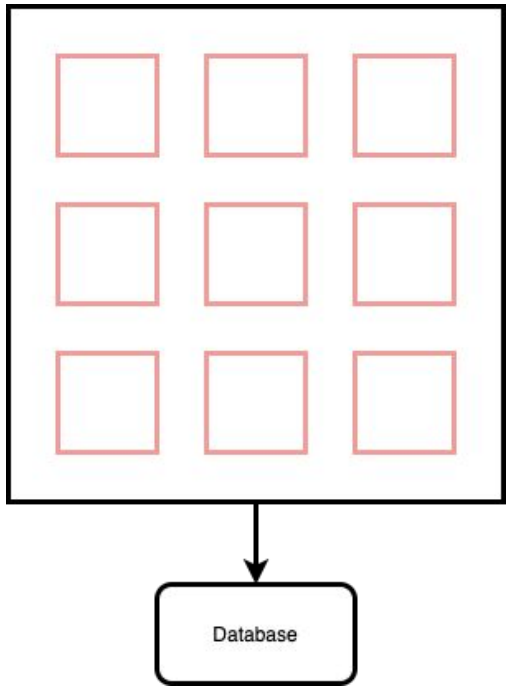
Monolith

Very big monolith

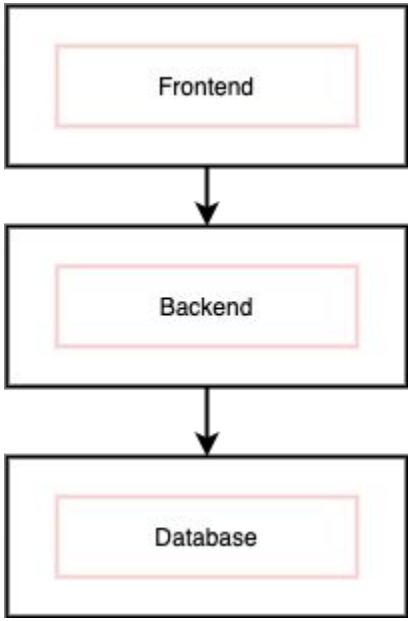


www.scaledagileframework.com/domain-modeling/

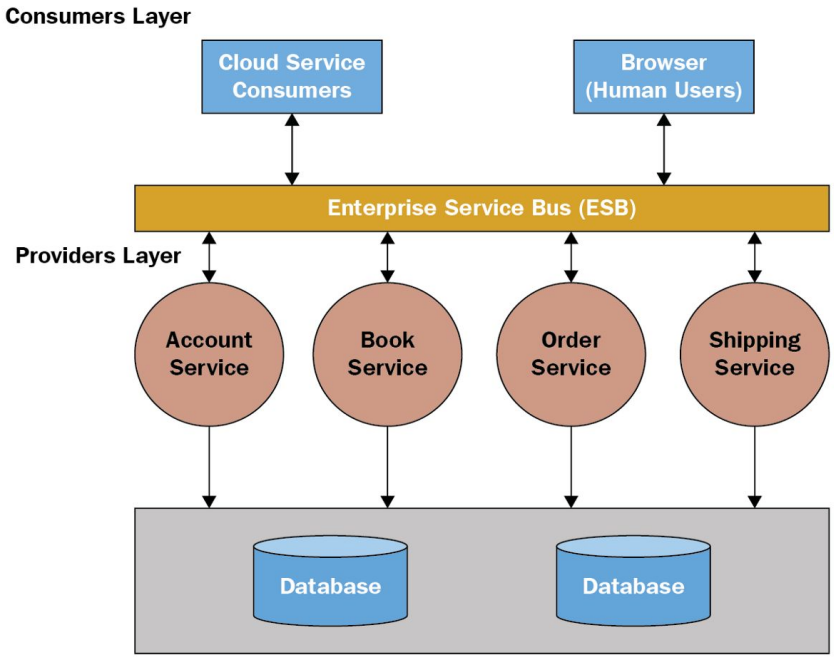
Modular monolith



Layered architecture

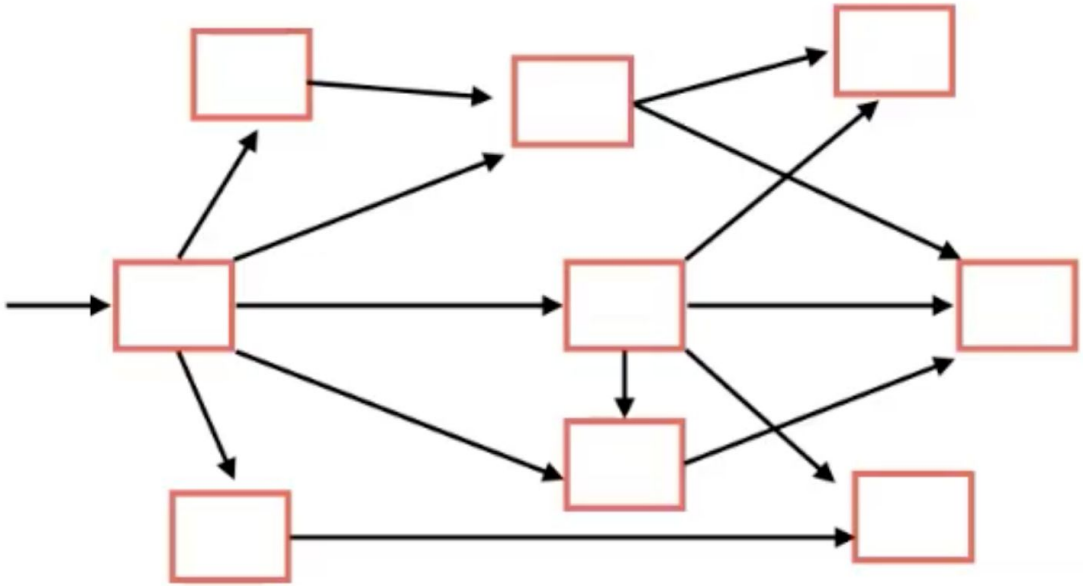


Service oriented architecture

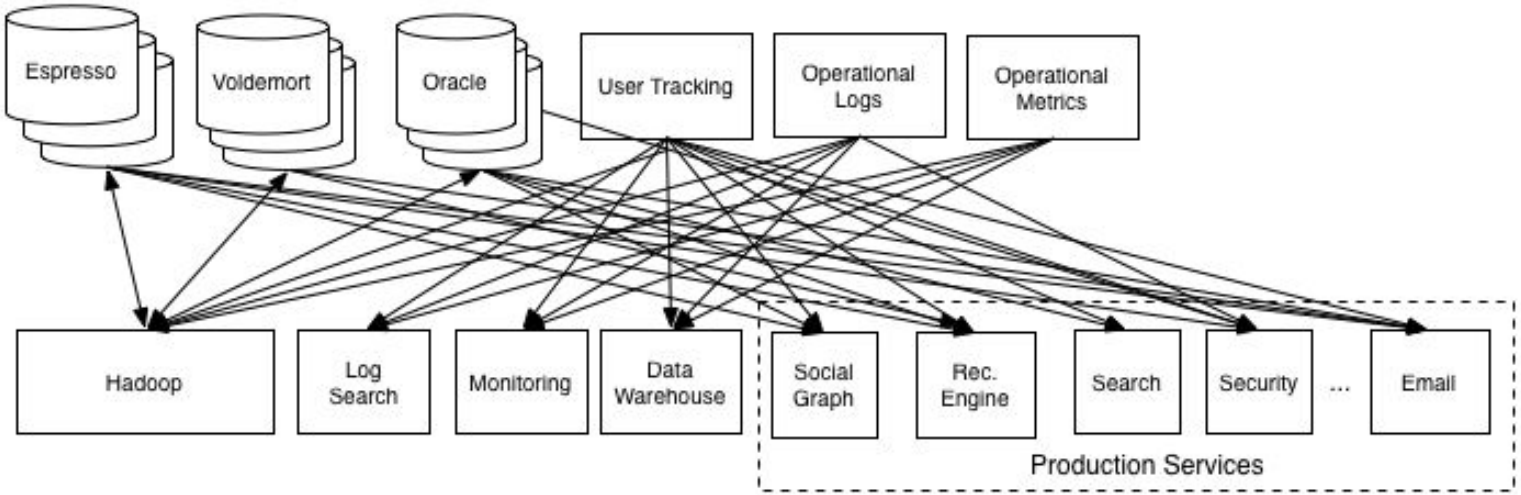


ooneunhye.medium.com/service-oriented-architecture-soa-%E0%B8%84%E0%B8%B7%E0%B8%AD-%E0%B8%AD%E0%B8%B0%E0%B9%84%E0%B8%A3%E0%B9%80%E0%B8%AB%E0%B8%A3%E0%B8%AD-f462e60c2198

Distributed monolith



Distributed monolith



medium.com/@joaovasques/your-distributed-monoliths-are-secretly-plotting-against-you-4c1b20324a31

Moving from monolith

- Monolith -> Modular monolith -> Microservices

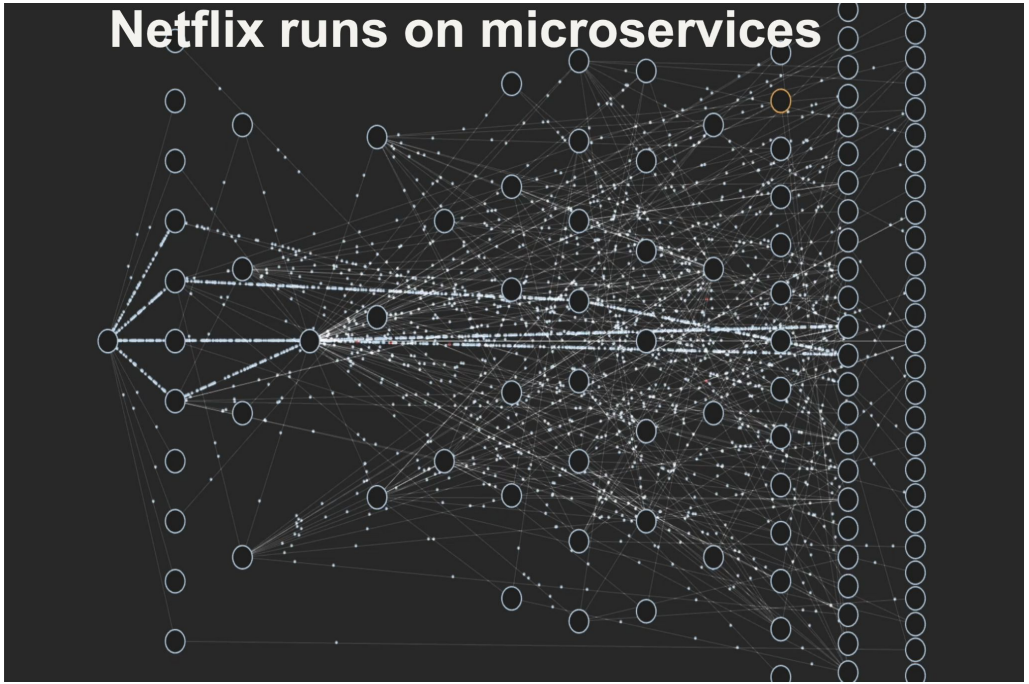
Microservices

- Focus on business first
- Loosely coupled
- Easy to develop by small team
- Deploy and scale independently
- Highly maintainable and testable

Why microservices? aka why not monolith.

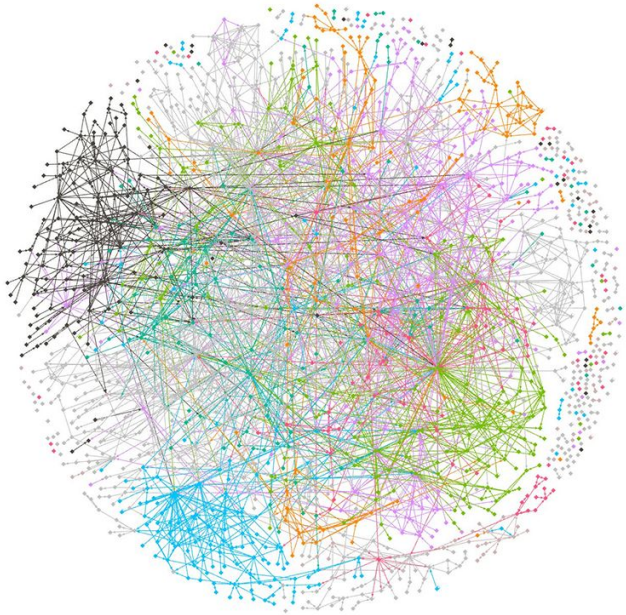
- Improve team autonomy
- Reduce time to market
- Team scaling
- Infrastructure scaling, cost effective for load
- Robustness
- Technology heterogeneity
- From engineering team boundaries to Domain boundaries
- Sociotechnical

Microservices example - Netflix



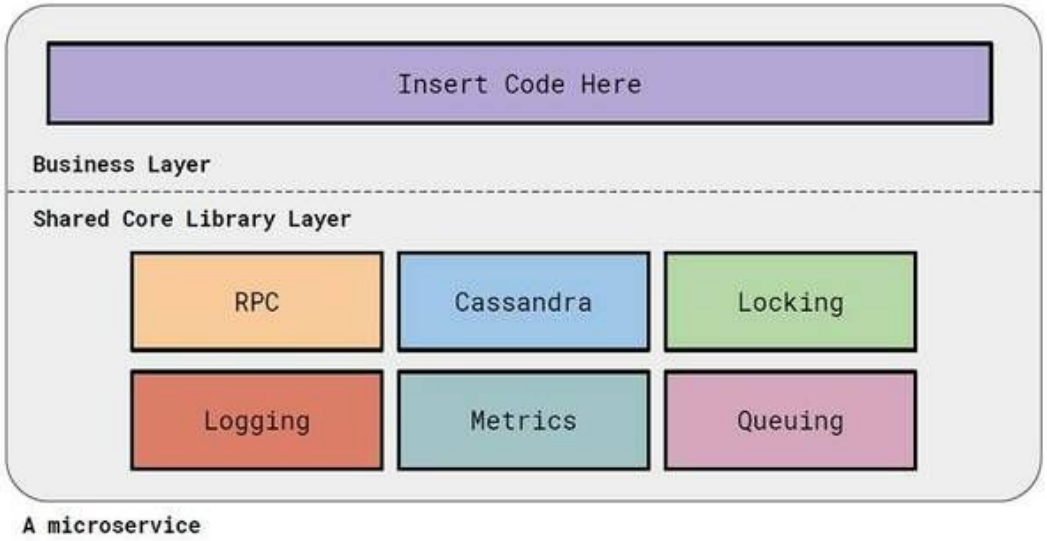
gotocon.com/dl/goto-amsterdam-2016/slides/RuslanMeshenberg_MicroservicesAtNetflixScaleFirstPrinciplesTradeoffsLessonsLearned.pdf

Microservices example - Monzo



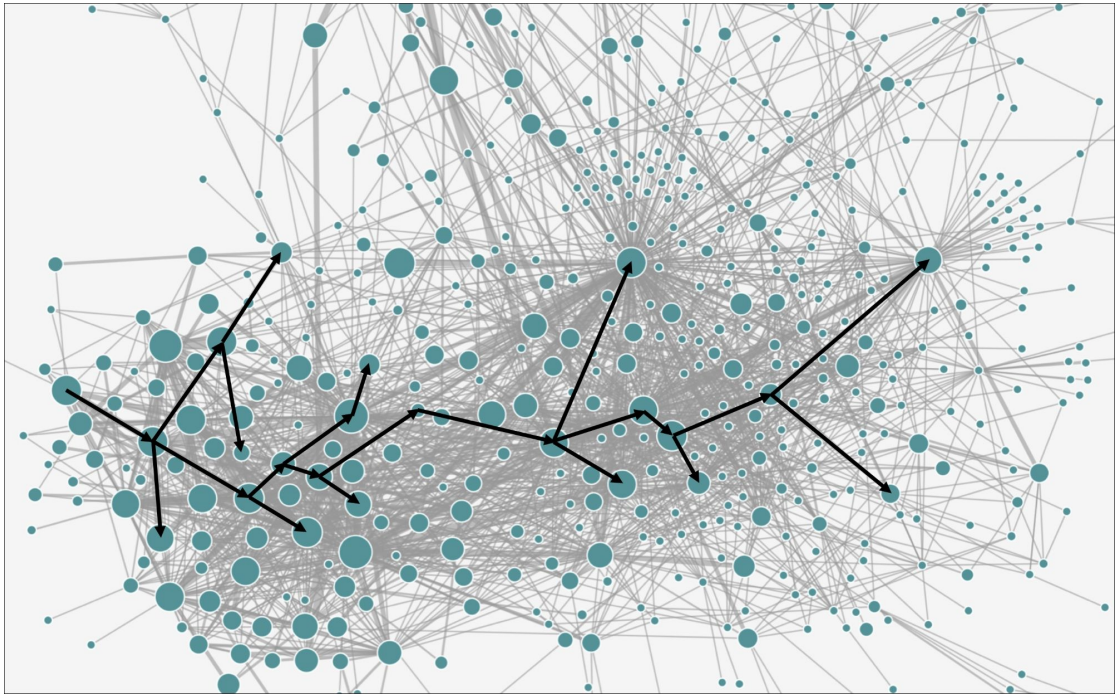
monzo.com/blog/we-built-network-isolation-for-1-500-services

Microservices example - Monzo



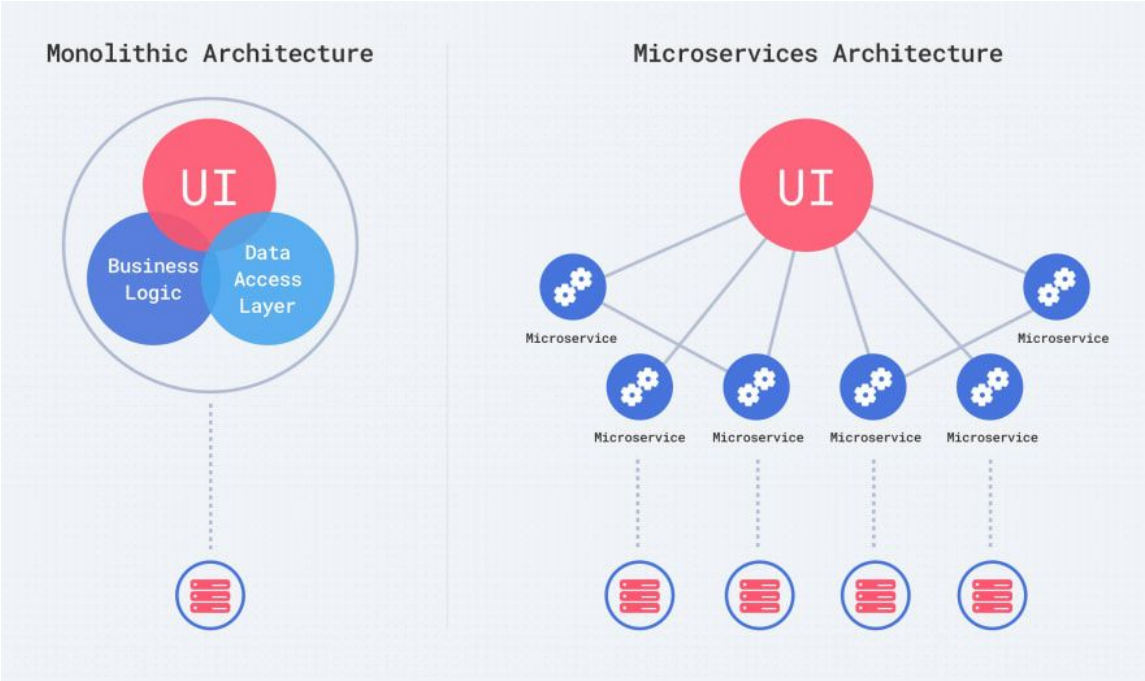
www.theregister.com/2020/03/09/monzo_microservices/

Microservices example - Uber



subscription.packtpub.com/book/networking_and_servers/9781788628464/1/ch01lvl1sec13/the-observability-challenge-of-microservices

Monolith VS Microservices



dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m

Monolith VS Microservices

Monolith

- Easy to comprehensive, develop, deploy, maintain.
- Better knowledge transfer

Microservices

- Team scalable
- Complex

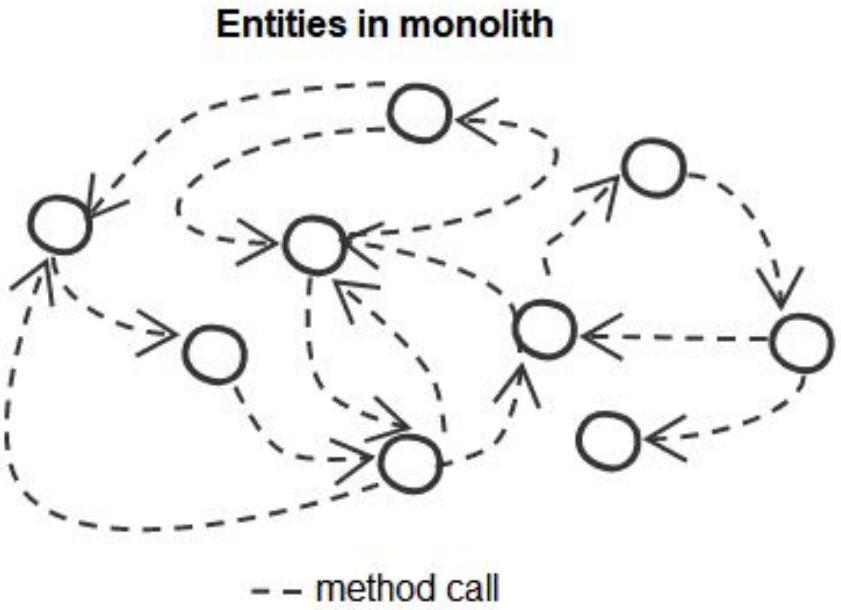
What about resource scaling? Isn’t that benefit of microservices?

- You can scale as much as you want in monolith too.

Why not microservices

- When it is a bad idea
- Unclear business/domain, premature abstraction is bad
- Startup
- Few people working together, using microservices will be too much of cognitive load
- Fast moving, everything change so fast
- Customer installed
- Complexity

Why not microservices

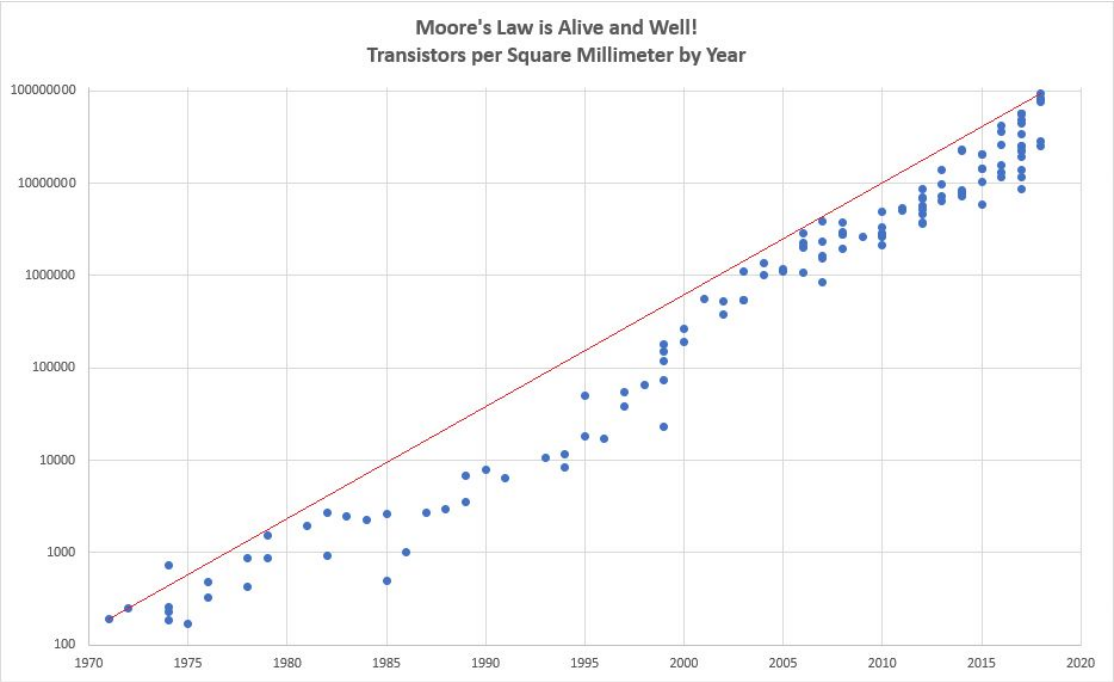


medium.com/transferwise-engineering/from-monolith-to-distributed-monolith-fd53d8dbbeba

Trade off

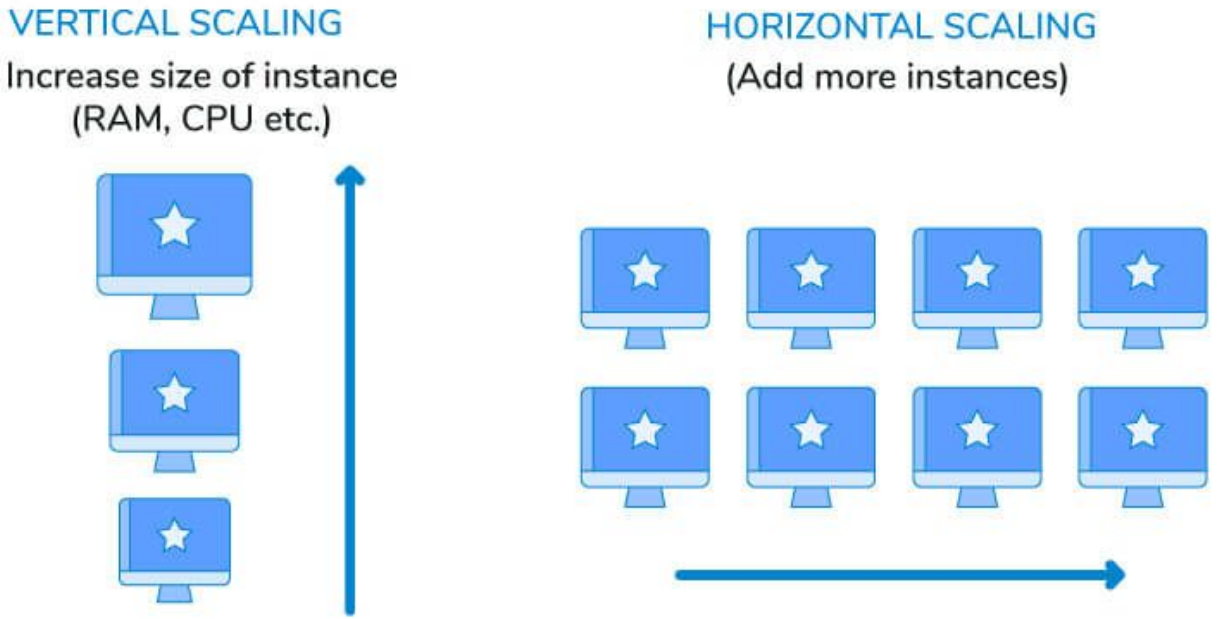
- Monitoring, logging, tracing, infrastructure, testing, deployment, etc.
- Team topologies
- Team cognitive load
- Complexity
- Organizational culture

History of computing - Moore’s Law



medium.com/predict/moores-law-is-alive-and-well-eaa49a450188

Vertical scale vs horizontal scale



www.redswitches.com/blog/difference-between-horizontal-vertical-scaling/

Distributed system

- also known as distributed computing.
- Multiple components located on different machines.
- Communicate and coordinate actions in order to appear as a single coherent system to the end-user.

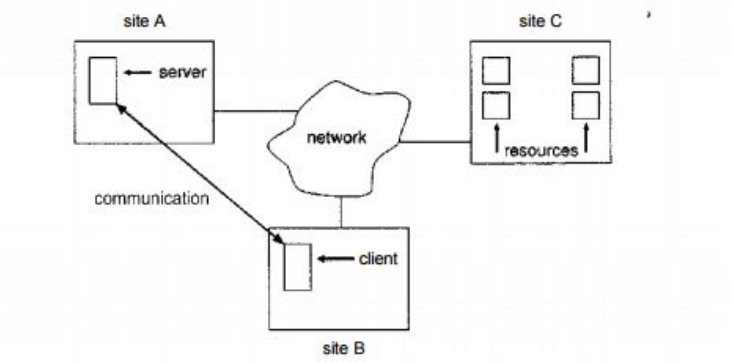
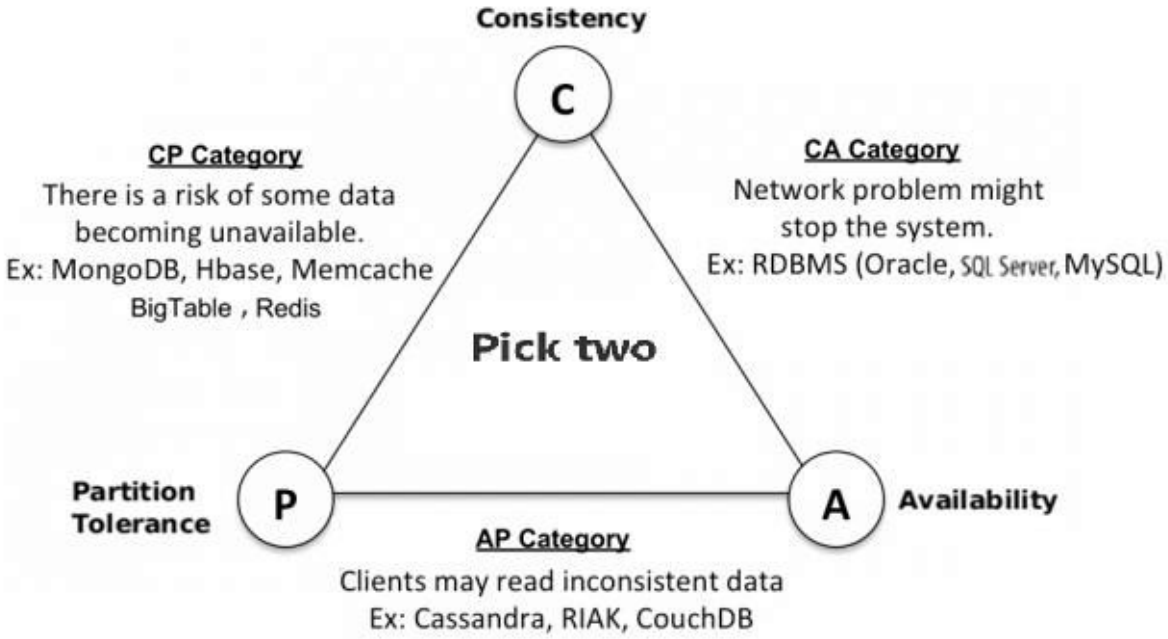


Figure 16.1 A distributed system.

padakuu.com/article/185-distributed-system-motivation

CAP theorem



medium.com/swlh/cap-theorem-in-distributed-systems-edd967e7bdf4

8 fallacies of distributed computing

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Part 2: Microservices, How?

What make a good service?

- Loose coupling
  - Change in one service should not require change to another
  - Less depend on other services
- High cohesion
  - What edit together should stay together
- Good team cognitive load

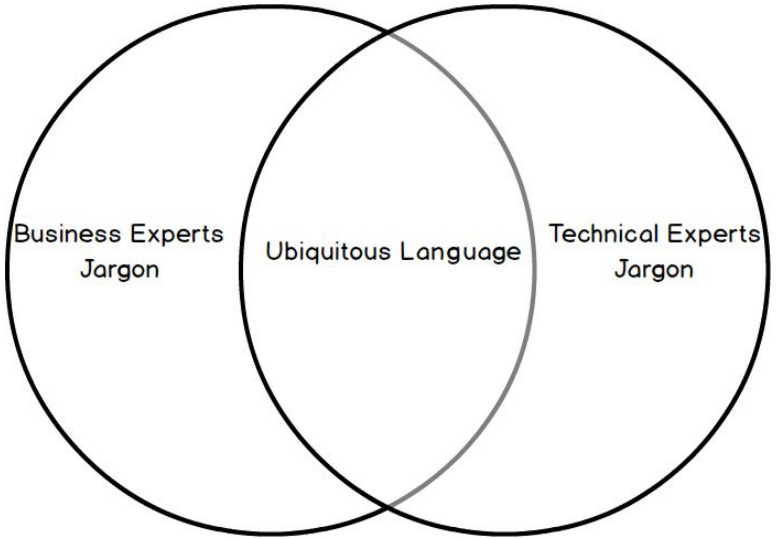
Domain driven design

- Approach on how to create systems that model real-world domains
- How to build microservices with thought process
- Bind data and intelligence together

thedomaindrivendesign.io/learning-map/

Ubiquitous language

- To share understand of the word between domain expert and technical



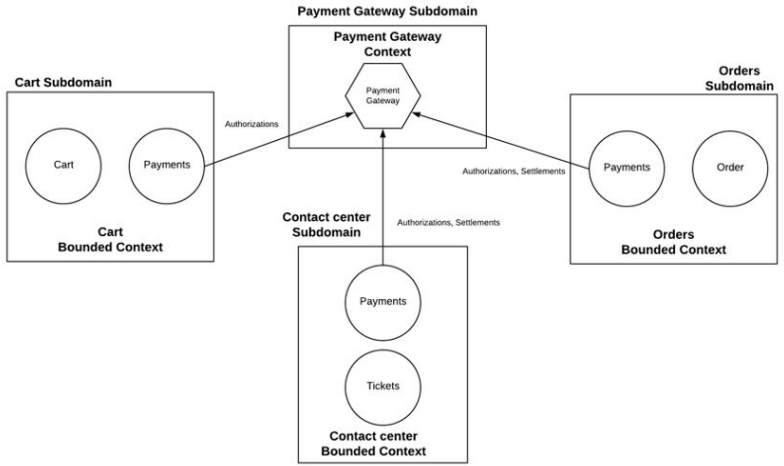
medium.com/the-coding-matrix/ddd-101-the-5-minute-tour-7a3037cf53b8

Strategic design

- Design system overview and boundaries

Bounded Contexts

- Specific responsibility enforced by explicit boundaries
- Boundary around parts of the services in term of business domain
- Single responsibility principle applied to your model



medium.com/walmartglobaltech/building-domain-driven-microservices-af688aa1b1b8

Subdomain

- Core
  - Key differentiator for business, most valuable part
- Supporting
  - Related to what business does but not differentiator
  - Implemented in-house or outsourced
- Generic
  - Not specific to business
  - Using off the shelf software

Context mapping

- How data flow from bounded contexts to others
- Relationship between bounded contexts can be vary
- Anti-corruption layer
  - Downstream implement layer to translate data to internal model
- Conformist
  - Adapt to upstream context, upstream does not change.
- Customer/Supplier
  - Customer ask supplier for their needs
- Open host
  - Open API, for other to use
- Shared Kernel
  - Sharing resources/component. Avoid if possible

Tactical design

- Take bounded context in strategic design and implement the software
- Entities
  - With unique identity, preserving history
- Value object
  - Immutable, only attributes, no identity
- Repositories
  - Retrieve objects from persistence
- Factories
  - Constructor for domain object to hide complexity

Tactical design

- Aggregates
  - Bind entities and value object
- Services
  - Manipulate business logic that doesn’t belong to a given object
- Domain event
  - Publish to communicate state of business changes to interested parties
- Command
  - Object send to domain for a state change

Event storming

- Collaborative exercise involving technical and domain expert
- Define a shared domain model together
- Steps
  - List scenario
  - Divide into subsystems (Core, Generic, Supporting)
  - List UI, Use case
  - Model data, flow
  - Event sourcing
  - Define CRUD, CQRS

Domain-Driven Design On a Page

DDD is a philosophy for developing software systems that encourages domain-thinking at each step of the software development lifecycle

Domain Discovery

Exploratory DDD

- Model as-is, to-be, and could-be states of the domain
- Model collaboratively and visually so that the whole team learns the domain and contributes to solution ideas
- Example: Big Picture EventStorming

Software Architecture

Strategic DDD

- Bounded Contexts: Split a large software system into specialised models aligned to areas of the domain
- Integrate bounded contexts using domain events
- Identify strategically-significant core domains

Software Design

Tactical DDD

- Create models in code which align to the team's shared understanding of the domain
- Use appropriate patterns in each context: entities aggregates, event sourcing etc.

twitter.com/ntcoding/status/1342805885224177666

DDD Doctrine

Principles and practices that are almost universally applicable in DDD

Make The Implicit Explicit
Avoid ambiguity and improve communication by making all relevant details visible.

Cultivate a Shared Language
Create a common language within each bounded context to foster improved collaboration.

Learning Never Stops
There is always more to learn about the domain so a learning mindset is essential

Explore Multiple Models
When the benefits of a better model are significant, don't stop at the first idea

Focus on Concrete Scenarios
Avoid creating beautiful designs that fail to solve the problem by challenging designs with concrete scenarios

Design is Evolutionary
Because learning never stops there is a constant stream of feedback which can be used to improve the design.

DDD Community Values

Beliefs that bind DDD practitioners

Embrace and Integrate With Ideas From Other Communities
There is a large overlap with DDD and other communities including BDD, UX, product management, and microservices. DDD is more beneficial when combined with ideas from outside the community.

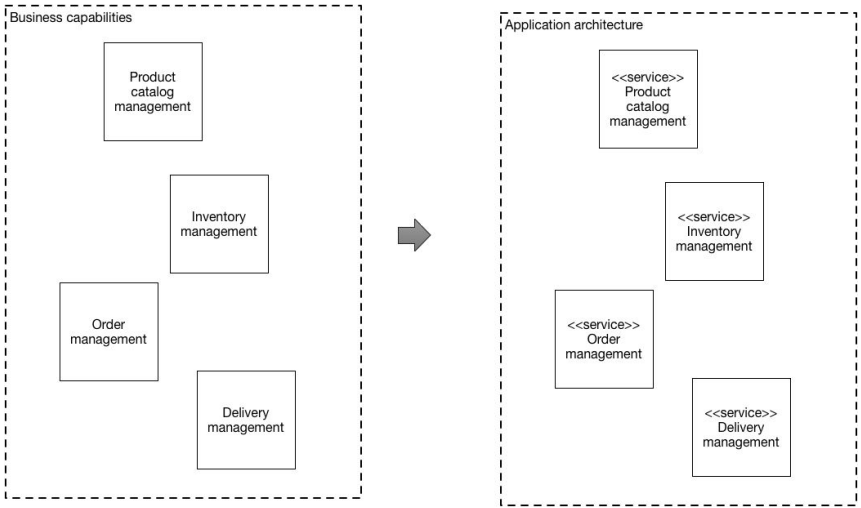
Vendor Independence & Accessible Learning
DDD is not about tools and frameworks. No ideas or techniques in the DDD community should be proprietary or hidden behind behind paywalls. The DDD community strives to make learning as inclusive and accessible as possible.

Decomposition application

- Decompose by business capability
- Decompose by subdomain
- Self-contained service
- Service per team

Decompose by business capabilities

- Mapping from business boundaries, User journey



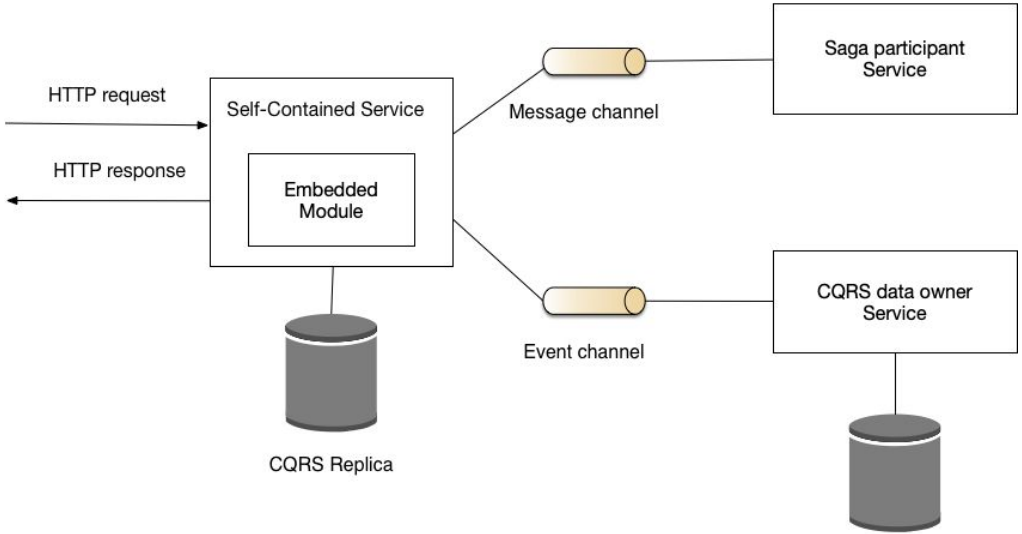
microservices.io/patterns/decomposition/decompose-by-business-capability.html

Decompose by subdomain

- Use Domain driven design

Self-contained service

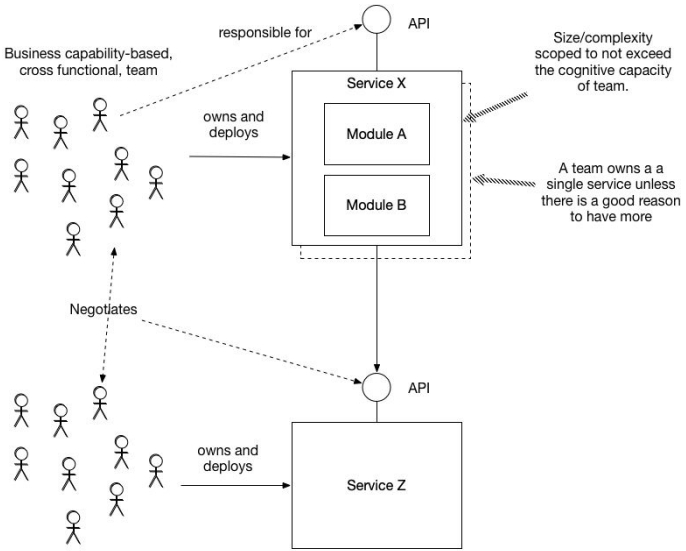
- Design a service that can respond to synchronous request
- Collaborate with other using CQRS or Saga



microservices.io/patterns/decomposition/self-contained-service.html

Service per team

- Each service owned by a team, sole responsibility for making changes



microservices.io/patterns/decomposition/service-per-team.html

Integration

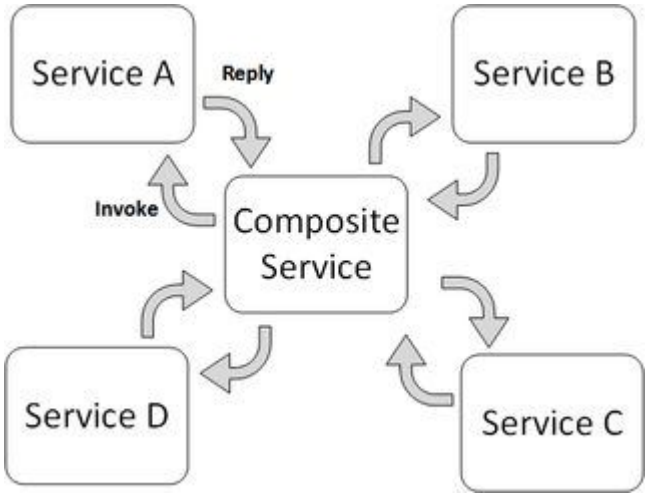
- Avoid breaking changes
- Technology Agnostic
- Simple for consumer
- Hide implementation details
- Interface and communication

Remote procedure calls

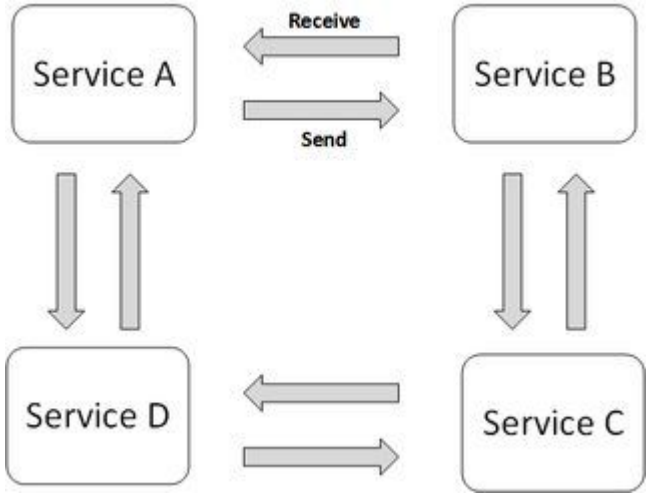
- HTTP
  - JSON
  - XML
  - Protobuf
- Rest and Restful
- gRPC

Orchestration and Choreography

- Managing business process that stretch across many services



Orchestration



Choreography

stackoverflow.com/questions/4127241/orchestration-vs-choreography

Share database?

- No
- Strong cohesion and loose coupling, we lose both things
- Share database makes it easy for services to share data, but does nothing about sharing behavior
- Changing in one service might affect others

Real world use case

- Talk about how real world use case of microservices

Part 3: Workshop 1

Using the link below

https://github.com/def4ultx/microservices-workshop

Part 4: From monolith to microservices

Migration

- Understand the goal why you want to use microservices
- Having microservices is not the goal
- Planning the migration
- Asking how or what to do to achieve that goal

3 Key question

- What you hoping to achieve?
- Have you consider alternative to use microservices?
- How you know if the transition is working?

Transition strategy

1. Ice cream scoop
    - Strangler pattern
2. Lego
    - No touch, just don’t add new feature
    - Creating new services
3. Nuclear option
    - Start fresh

- All strategy have trade off, make it align with your goal.

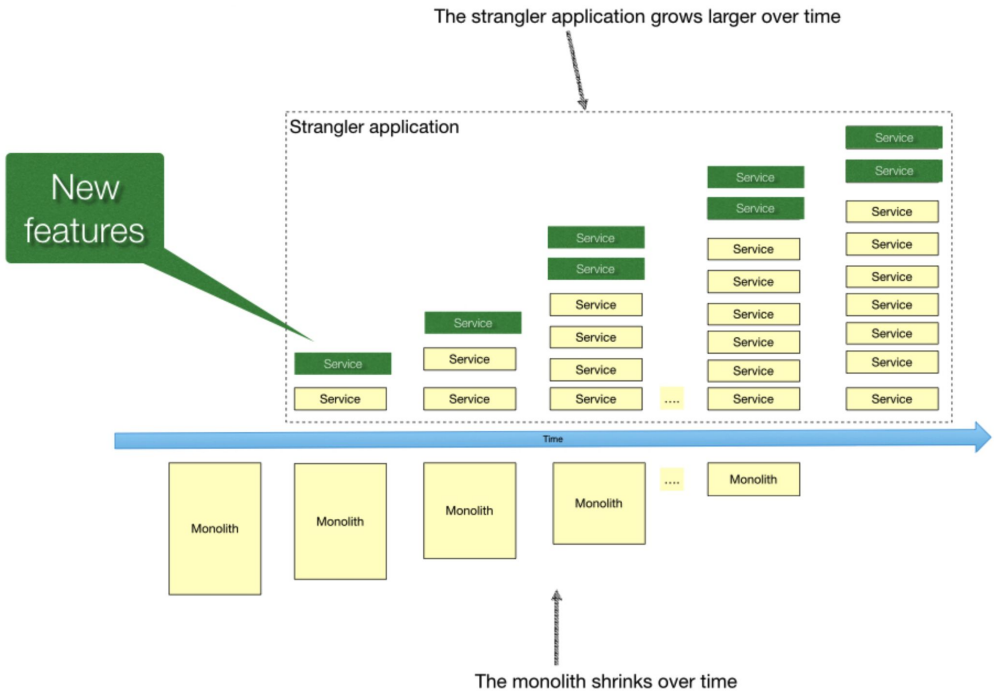
Splitting the monolith

- Should we change the existing system? Or able to?
- Refactor the monolith
- Is modular monolith an option
- Incremental rewrites or reimplementing
- Start from unrelated functionality of the system
- Decouple by business boundaries

Pattern - Strangler application

- Migrate from legacy monolith
- Incrementally develop new services around legacy services
- Take one part out, redirect the call
- Load balancer or branching needed
- Example - splitting monolith

Pattern - Strangler application

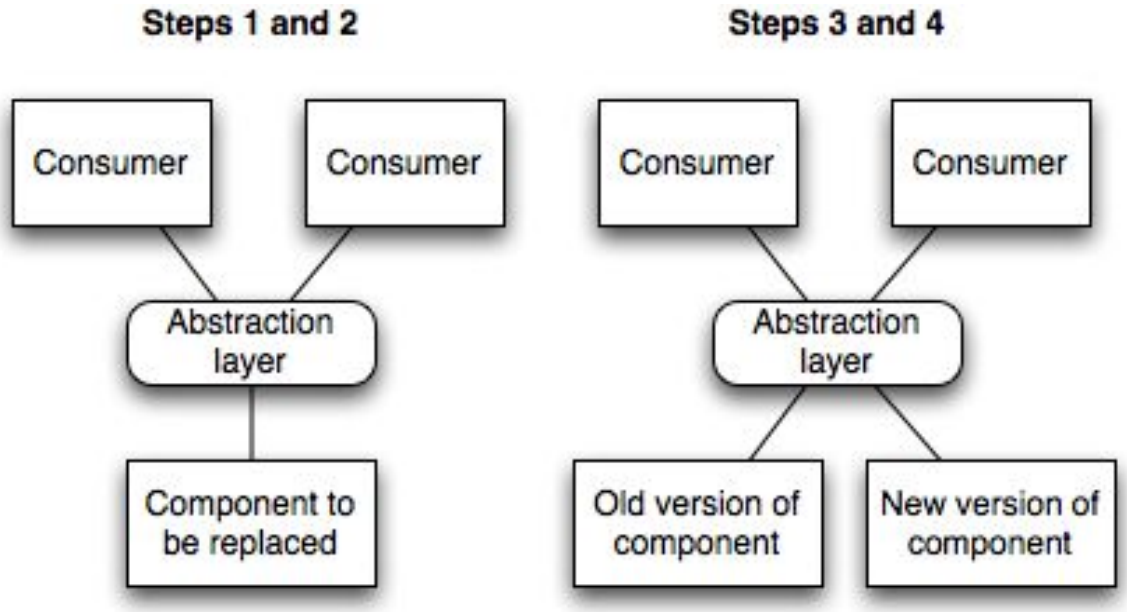


medium.com/swlh/cap-theorem-in-distributed-systems-edd967e7bdf4

Pattern - Branch by abstraction

- Like strangler pattern but at lower level of abstraction
- Example - Migrate to call new payment services
- Steps
  - Create abstraction
  - Update client to use new abstraction
  - Create new implementation that will call to new microservices
  - Switch over
  - Clean up

Pattern - Branch by abstraction



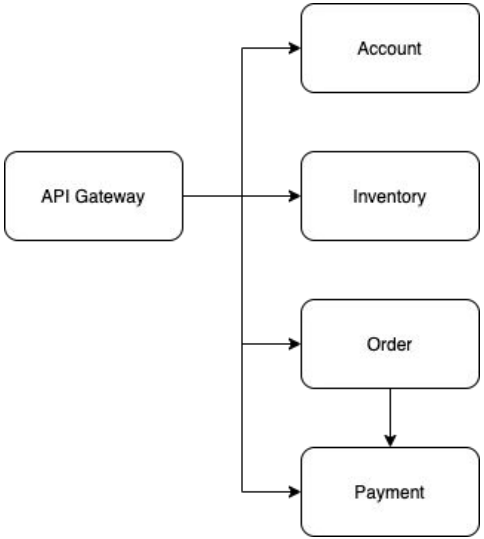
continuousdelivery.com/2011/05/make-large-scale-changes-incrementally-with-branch-by-abstraction

Pattern - Parallel run

- Many testing needed for transition
- Comparing between services.
- A/B Testing
- Traffic mirror
- Accuracy checker
- Canary/Feature flags

Mini workshop

- Let take Order services from previous workshop and split its payment to microservices

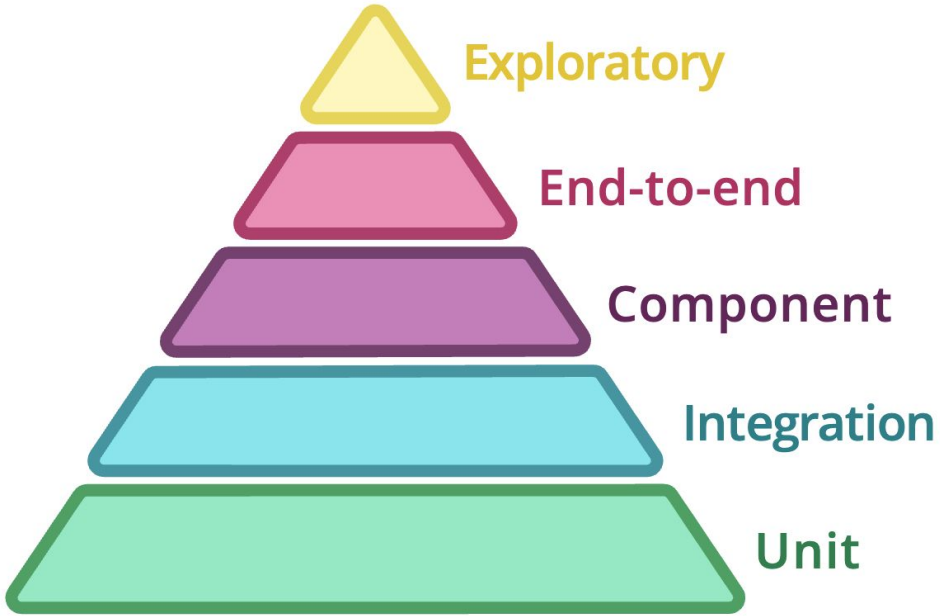


Part 5: Testing and deployment

How should we test in microservices?

- Testing in microservices can be hard
- Dependency problem, Component stability, missing data, etc.

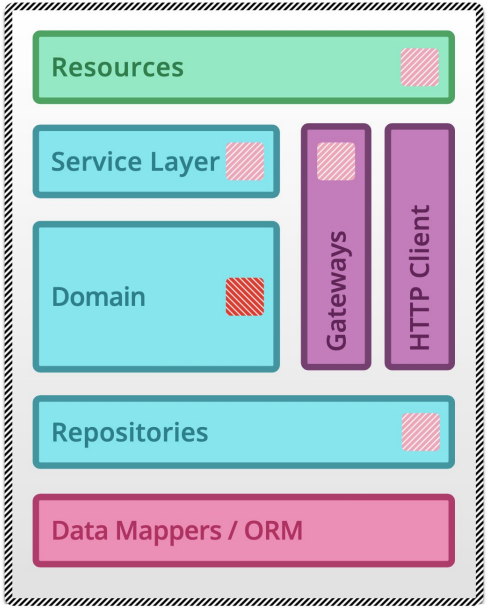
Testing pyramid



martinfowler.com/articles/microservice-testing

Unit test

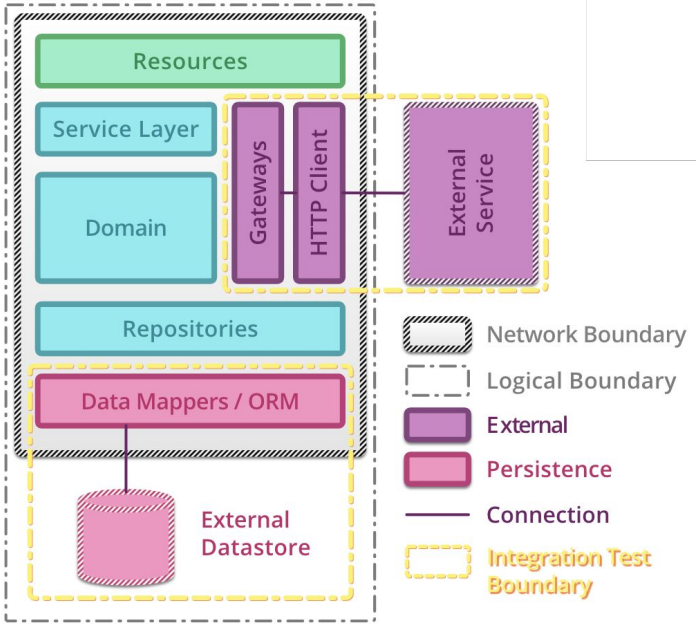
- Typically single function or method call, Small scope.
- Provide very fast feedback to developer.
- Catch most of the bugs.
- Does not guarantees about behaviour of the system



martinfowler.com/articles/microservice-testing

Integration test

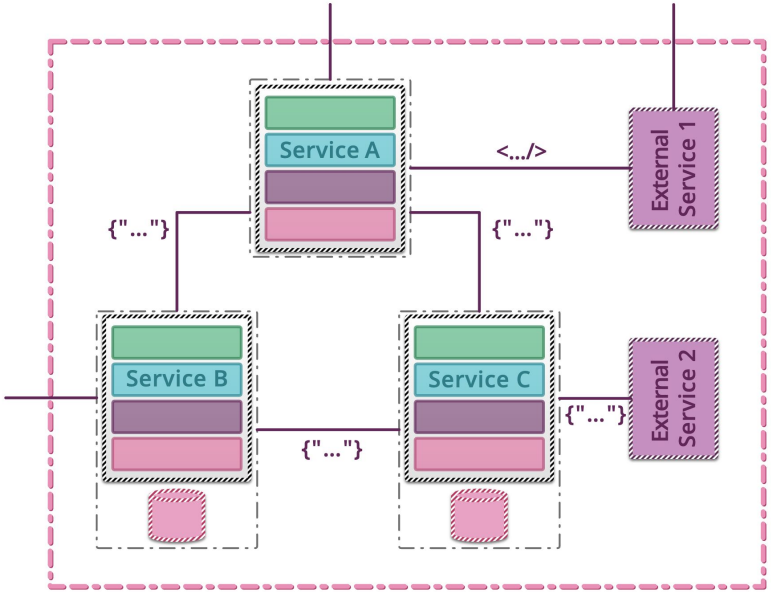
- Verify communication and interactions between components
- Tests with external components/data store
- Test between provider and consumer
- Should cover basic happy and error paths
- Not testing external components



martinfowler.com/articles/microservice-testing

End to End test

- Run against entire system.
- Should cover many scenario in the system
- Slow, very costly.
- User interface test, API test, etc.
- Many downsides.
  - Flaky test
  - Services version
  - Investigating times
  - A lot of moving parts can introduce failures
  - Who write the these tests if there multiple services by multiple teams
- Test user journeys



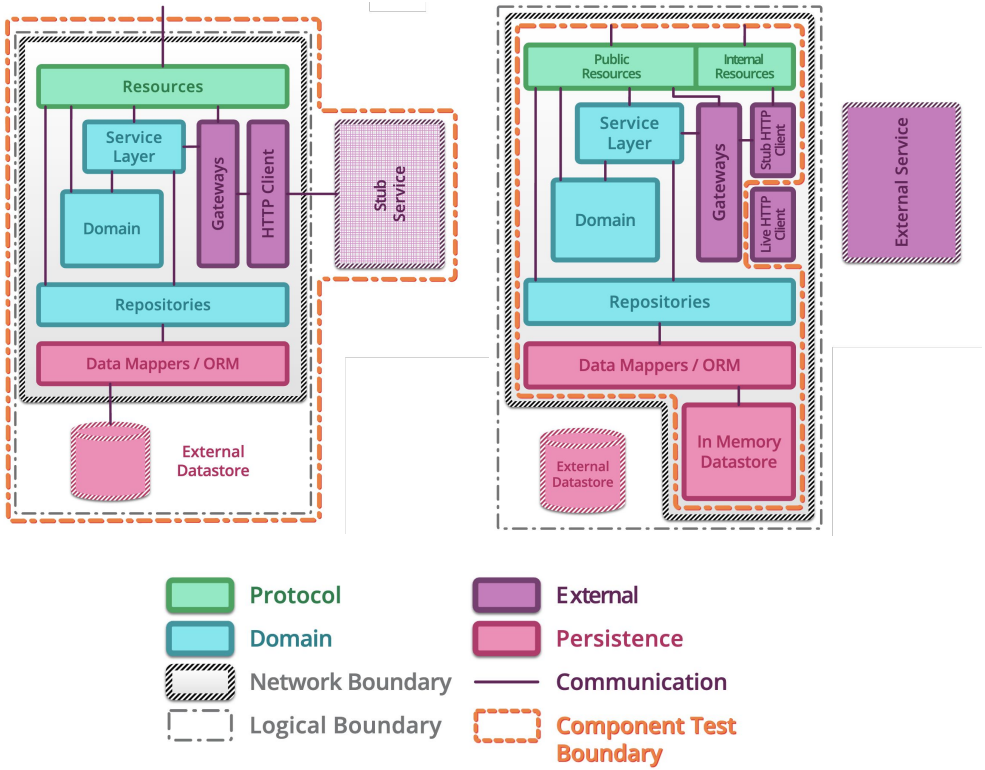
martinfowler.com/articles/microservice-testing

Component test

- Or services test in microservices architecture
- Mock/Stub testing
- Similar with integration test
- In-process or out of process test

martinfowler.com/articles/microservice-testing

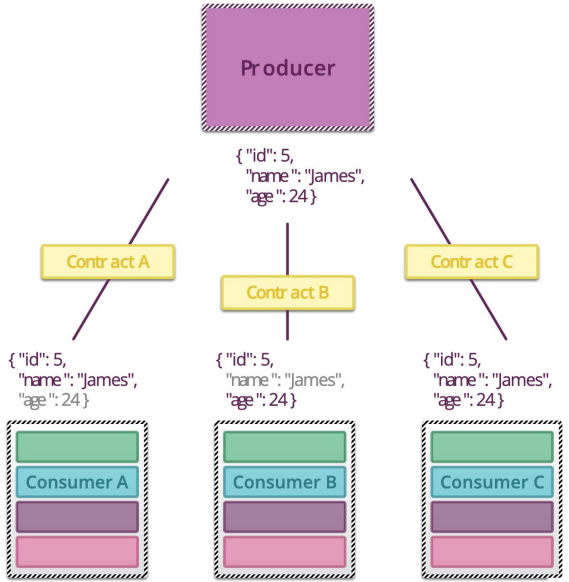
Component test



martinfowler.com/articles/microservice-testing

Contract test

- Test the contract between consumer and producer
- Ensure the message is in correct contract
- Record request/response contract in consumer side
- Share its contract with provider
- Provider use that artifact to run test
- Can verify only some part of the contract
- Example - pact.io

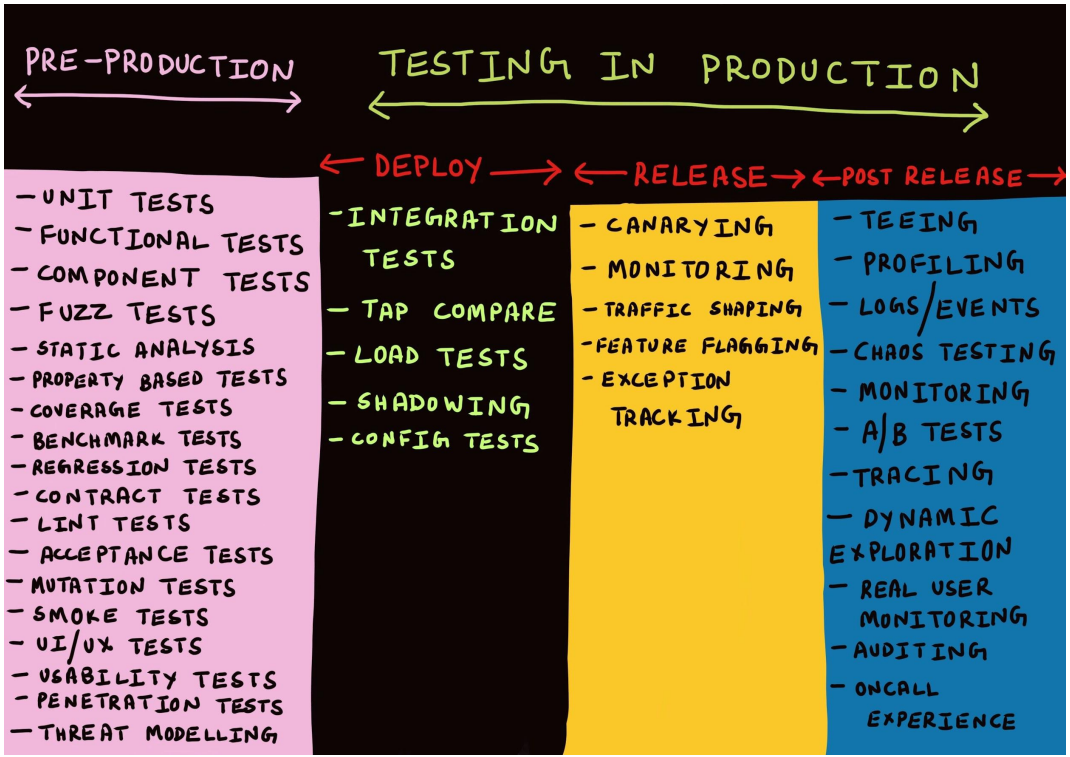


martinfowler.com/articles/microservice-testing

How many?

- There is an trade off between each type, balance them.
- As you go up the pyramid, slower the test run but it give more confident.
- As much as you want to make sure you feel safe. Keep stakeholder happy.
- Important things is the tests you have should ensure you about functionality
- Too much tests will cost some pain, eg Flaky test, time take to run.

And many more testing



copyconstruct.medium.com/testing-in-production-the-safe-way-18ca102d0ef1

Deployment

- The whole point of microservices is to enable team to develop, **deploy** and scale independently

Artifacts

- Result of things you build
- Can depend on different platform you use.
- Can be many format
- Container image, Binary, Jar, etc.

Environment

- We can deploy to different environment
- each environment serve different purpose

Configuration

- Services need configuration, eg. Database username and password.
- Ideally small amount
- Different environment also need different config
- Start from small config file, maybe different for each environment.

Deployment can take your prod down

- Capacity
- Bad version, Bugs
- Statefulness
- Redundancy

Deployment Strategy

- In-Place vs Replace
- All at once vs Rolling
- Blue-Green deployment
- Canary deployment

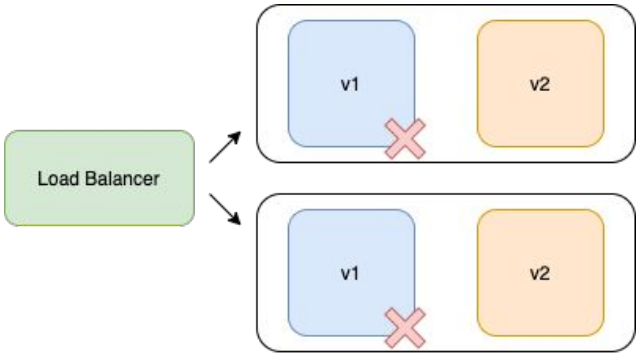
In-Place vs Replace

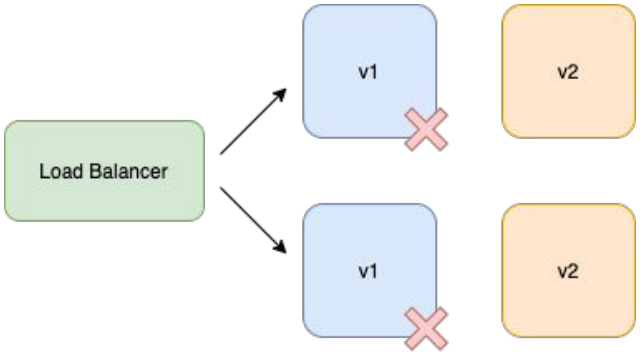
In-Place - replace old version in the server

- Fast
- Downtime during deployment

Replace - spin up new server

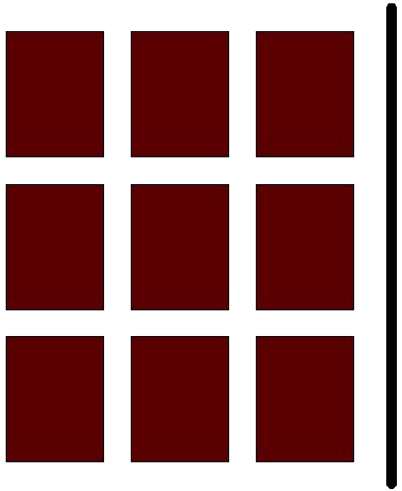
- Use load balancer
- Less risk
- Expensive in case of none cloud infrastructure





Rolling Deployment

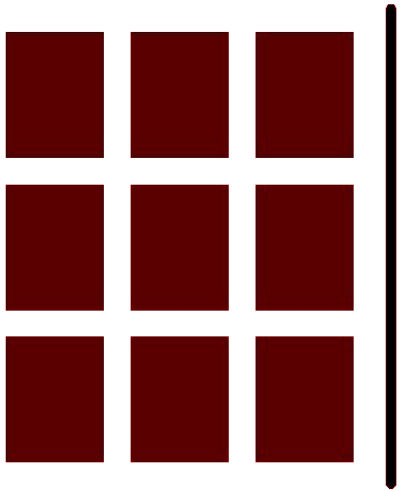
- Slower speed
- Capacity
- x% or x servers at a time
- Example 10 servers
  - 3 - 3 -3 - 1



opensource.com/article/17/5/colorful-deployments

All at once

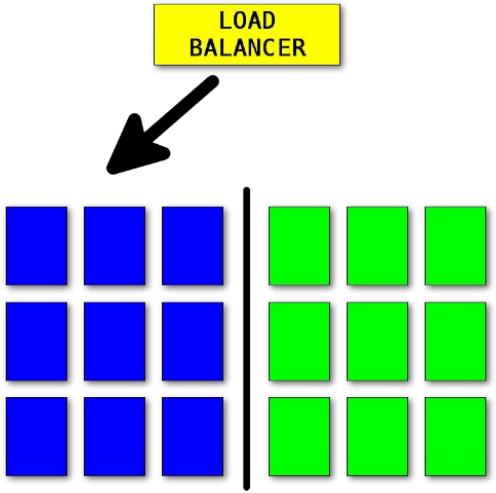
- Deploy to all



opensource.com/article/17/5/colorful-deployments

Blue-Green Deployment

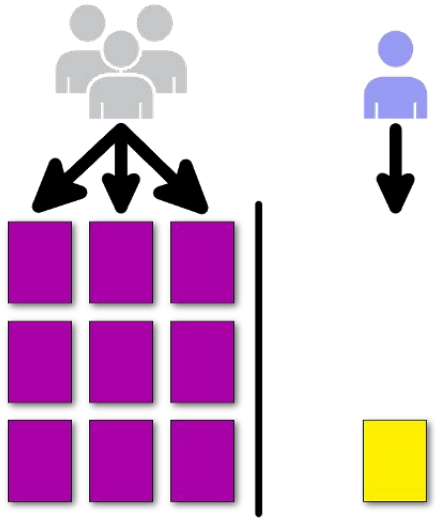
- Having 2 identical environments, load balancer to distribute traffic to appropriate environment



opensource.com/article/17/5/colorful-deployments

Canary Deployment

- Batch, Deploy to small set of users for some period of time



opensource.com/article/17/5/colorful-deployments

Deployment

- In the end we can use multiple deployment strategy for your need

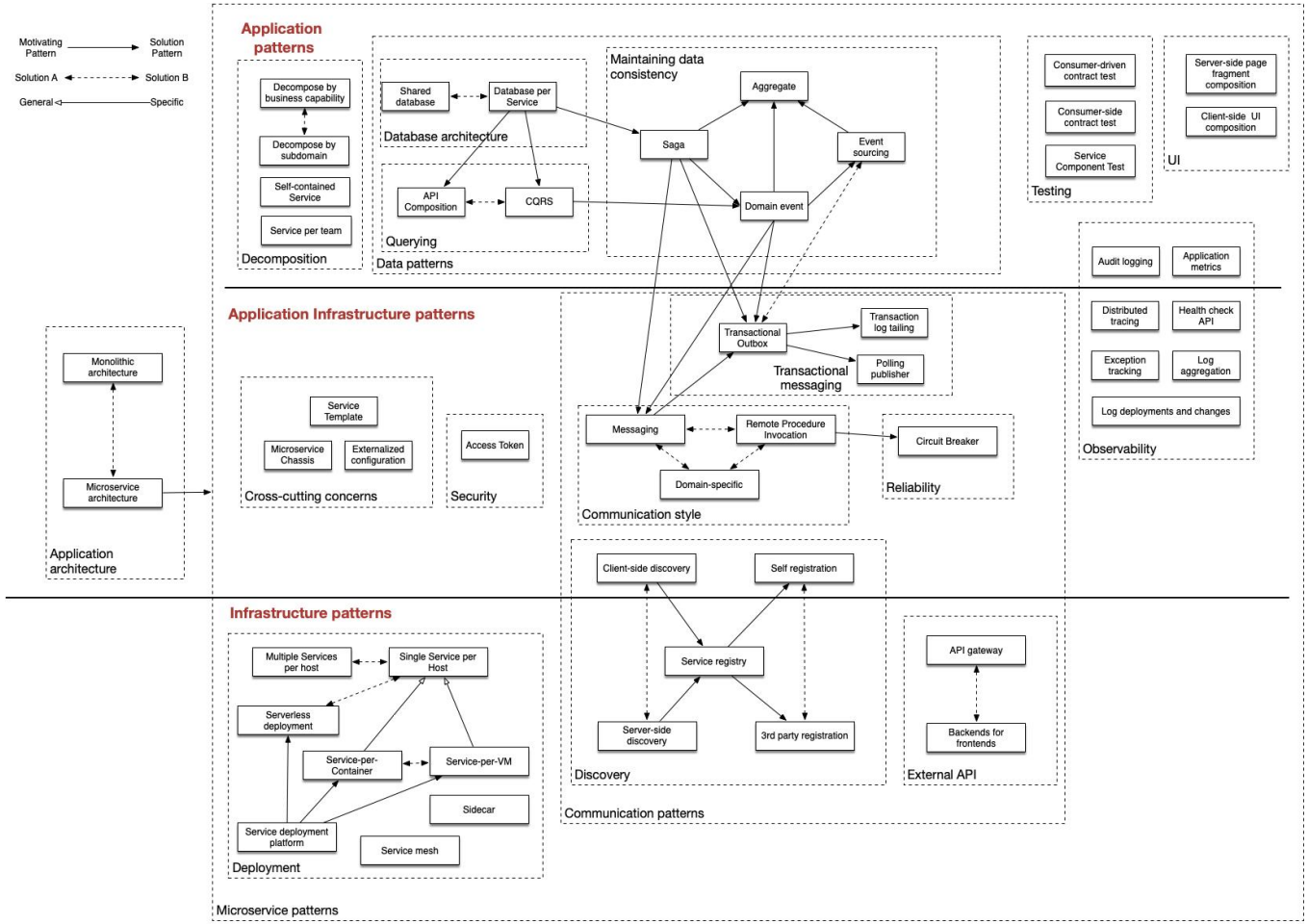
Deployment != Release

- We can decouple deployment from release
- Deployment - installing new version of software on production infrastructure
- Release - serving production traffic to new piece of code
- Safety

Continuous integration and Continuous Deployment

- the goal of CI is to keep everyone in sync with each other
- CD is about get change to customer and feedback loop from production readiness

Part 6: Deep dive into microservices pattern

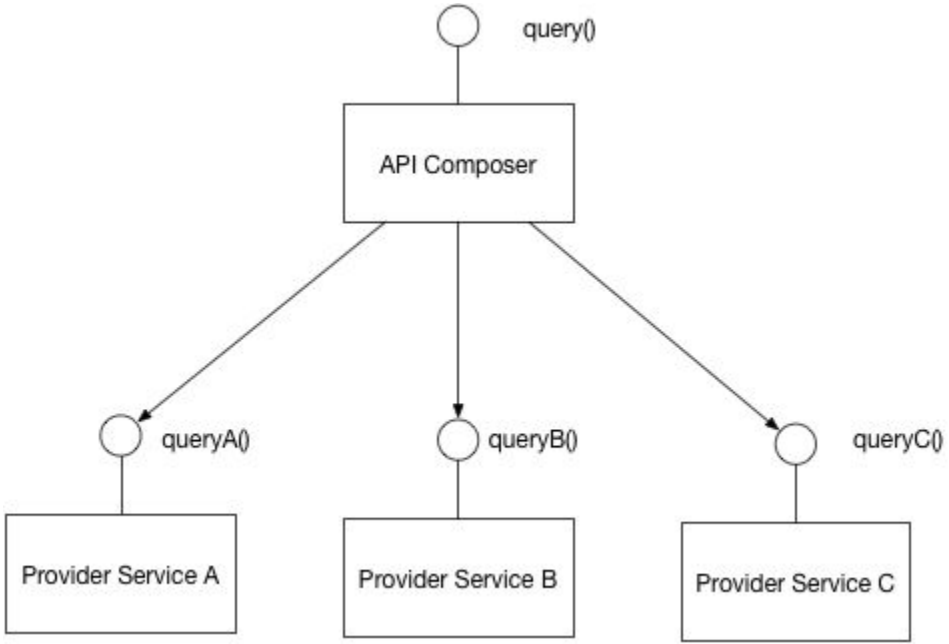


Load balancer

- Many algorithm
  - Round robin, weighted round robin, least connection, hash, ewma, etc.
- Client/Server side load balancer
- Example - Reverse proxy

API Composition

- API Gateway often does API composition



microservices.io/patterns/data/api-composition.html

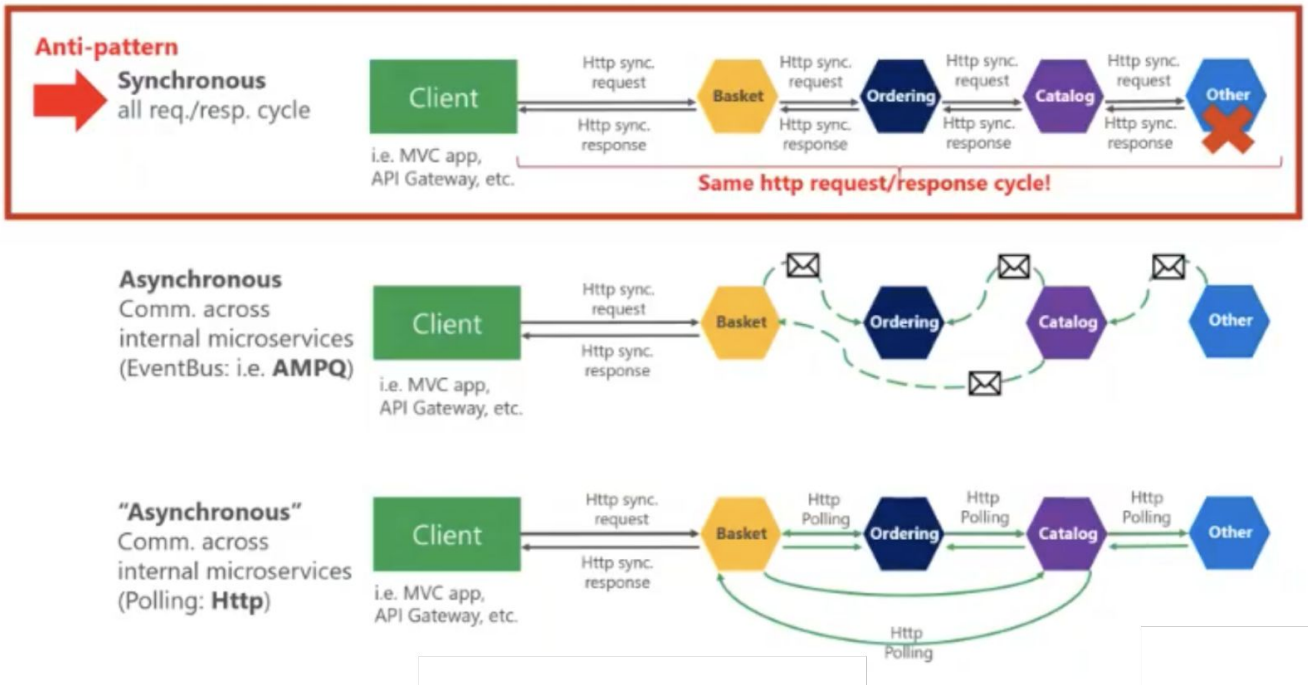
Async communication

- Do not wait for result communication model
- Message queue, Pub/Sub
- Kafka, RabbitMQ
- Notification, email, etc.

www.cloudamqp.com/blog/what-is-message-queuing.html

Communication

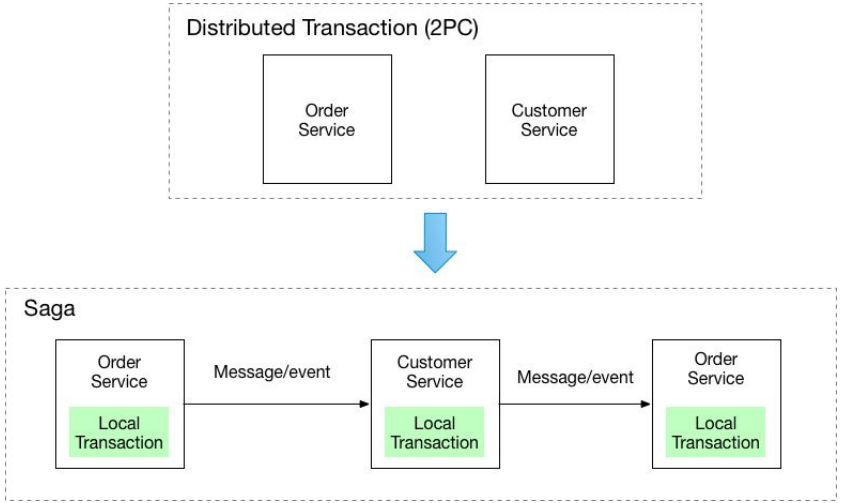
Synchronous vs. async communication across microservices



docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture

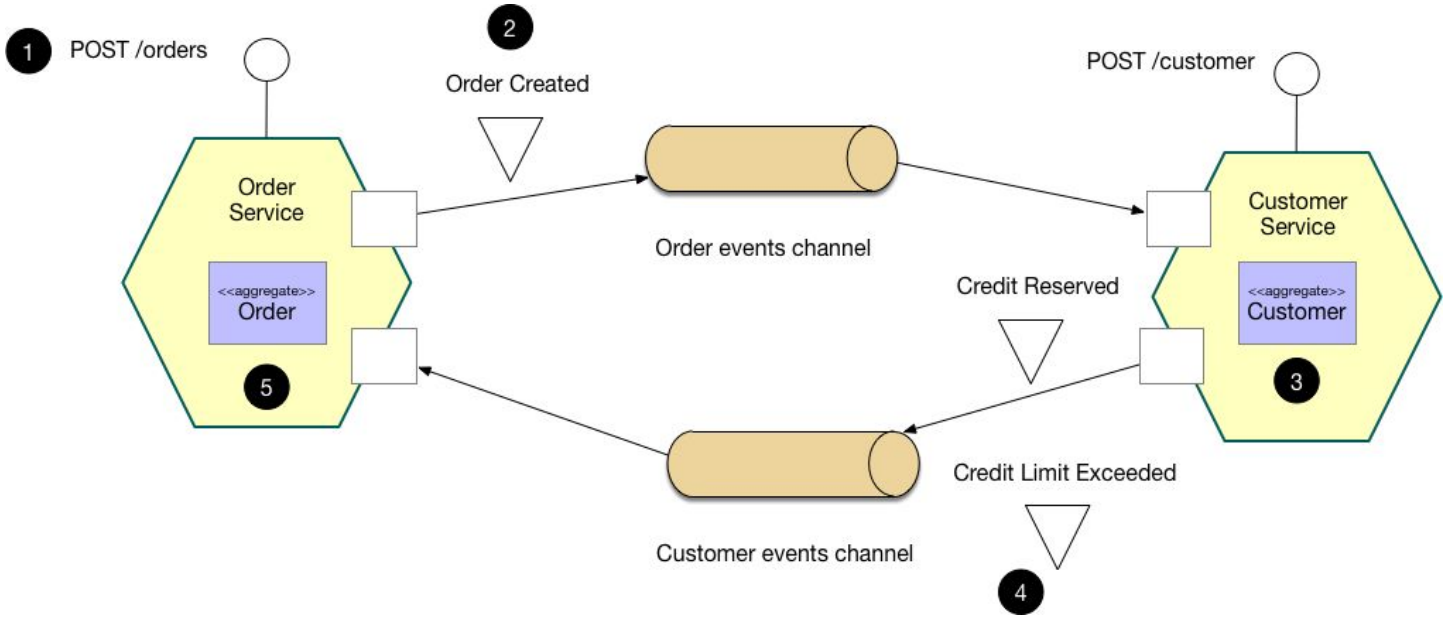
SAGA

- Transaction in microservices
- Local transaction update database and publish event to trigger next transaction
- Say no to distributed transaction (2PC)
- 2PC couple services together
- Split the data in first place



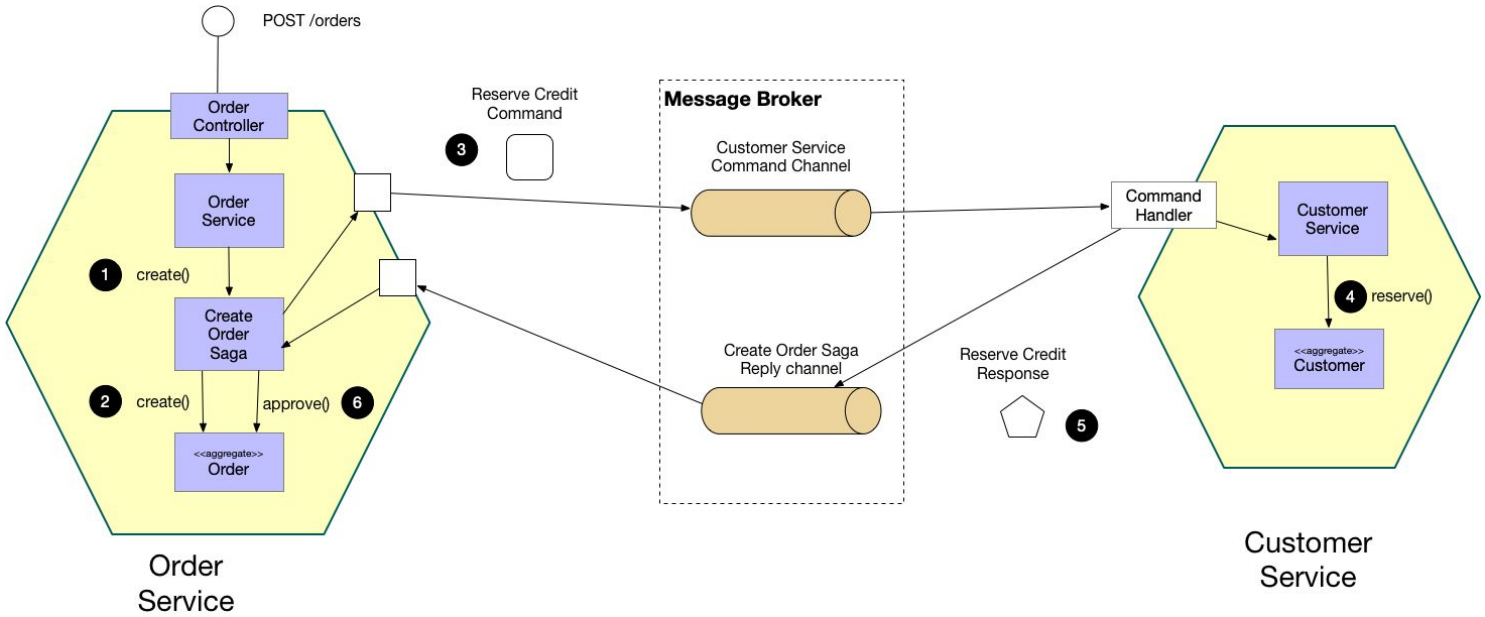
microservices.io/patterns/data/saga.html

Choreography



microservices.io/patterns/data/saga.html

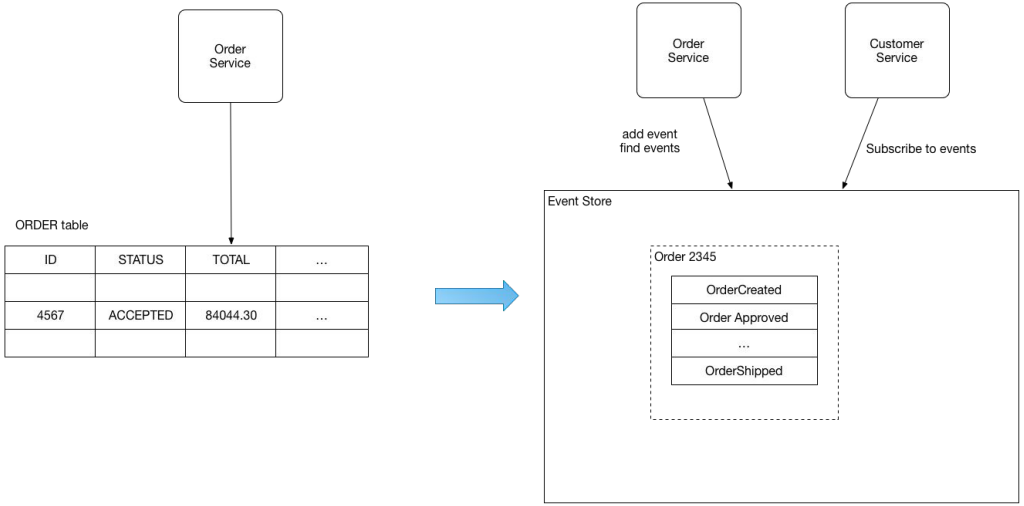
Orchestration



microservices.io/patterns/data/saga.html

Event sourcing

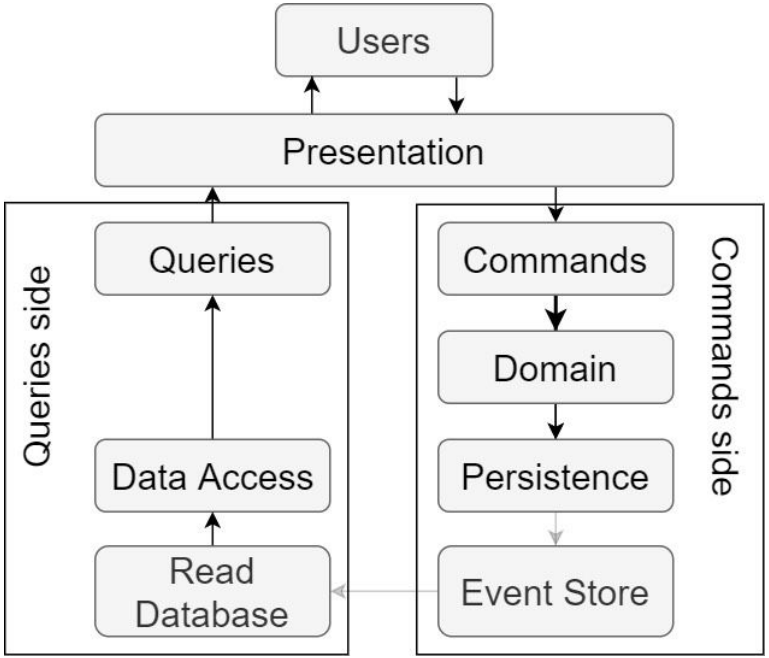
- Publish domain event across services
- Persist state of business entity to event store as a sequence of state-changing



microservices.io/patterns/data/event-sourcing.html

Command Query Responsibility Segregation

- Simple CRUD might be enough
- Single responsibility principle
- Separate Operation - Read/Write
- Different model for Read/Write
  - Easier to scale
- Eventually consistent views
- Can be used together with event sourcing
- Domain event can be publish across services



laptrinhx.com/3-cqrs-architectures-that-every-software-architect-should-know-3853992293/

Command Query Responsibility Segregation

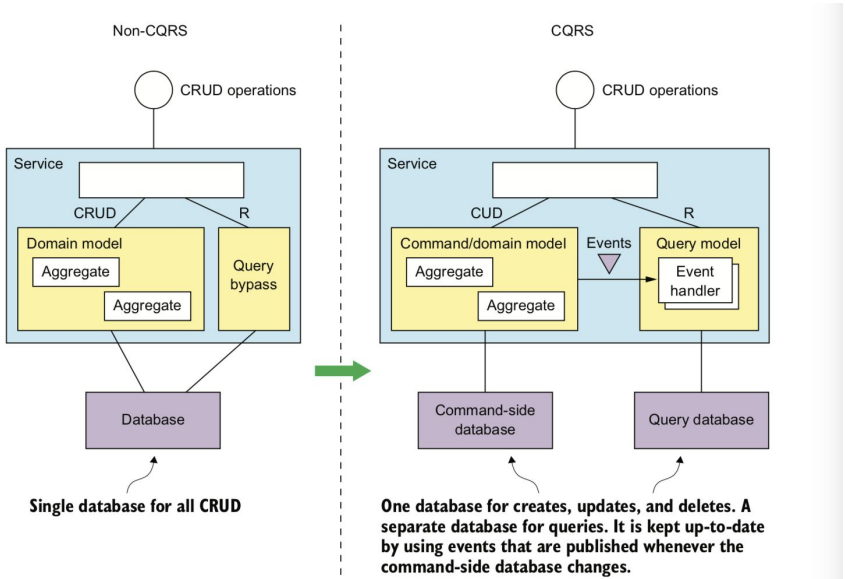


Figure 7.8 On the left is the non-CQRS version of the service, and on the right is the CQRS version. CQRS restructures a service into command-side and query-side modules, which have separate databases.

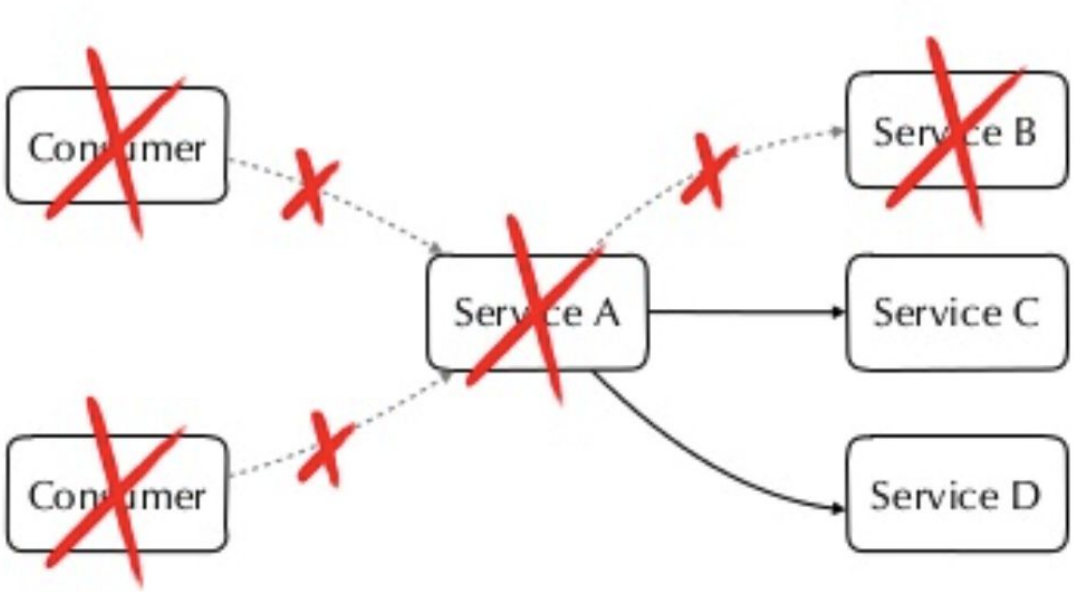
Service Discovery

- How service and discover another service
- Client/Server side
- DNS, Consul, Zookeeper, etc.

Failure in Microservices

- Failure is common in microservices
- High load, database down, Resources usage, scaling issue, bottleneck, etc.
- Latency, availability, durability of data.
- Build a resiliency system
- 6 nines does not matter if customer aren’t happy

Cascading Failure



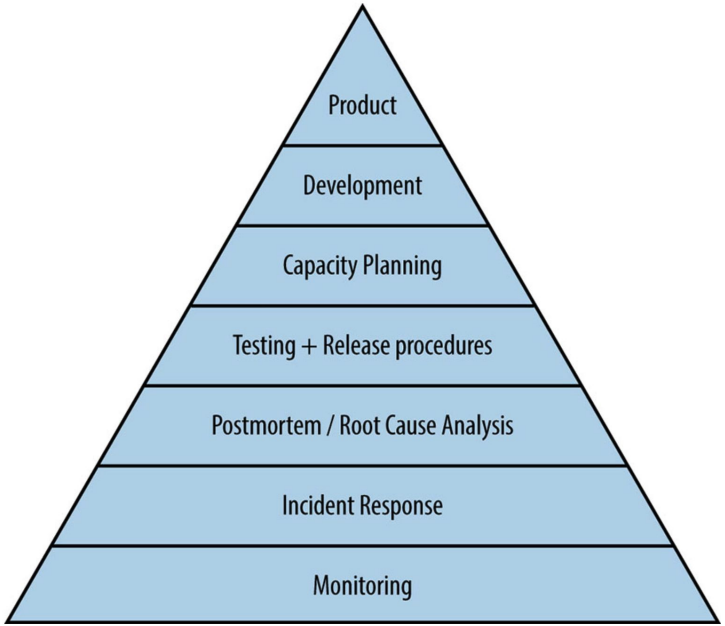
www.slideshare.net/welld/the-birth-and-evolution-of-a-microservices-architecture

Monitoring

- What’s the first thing you need to know?
- What the hell has gone wrong?
- SLI, SLO, SLA

https://opensource.com/article/17/5/colorful-deployments

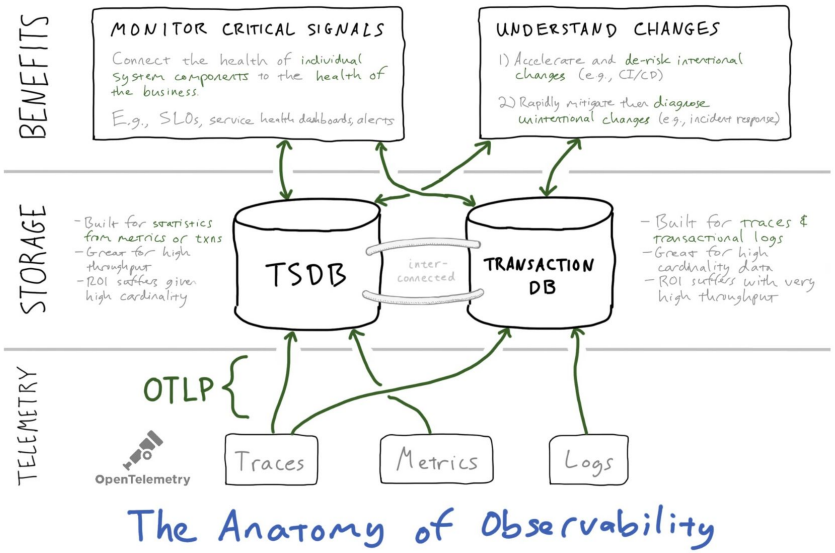
Service Reliability Hierarchy



sre.google/sre-book/part-III-practices/

3 Pillar of ~~observability~~ telemetry

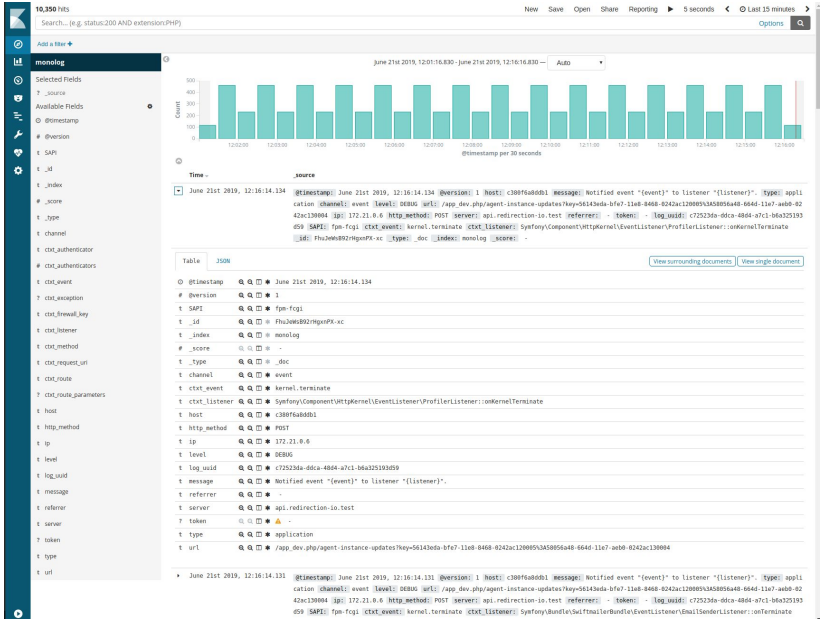
- Logging
- Metrics
- Distributed tracing



twitter.com/el_bhs/status/1349406398388400128/photo/1

Logging

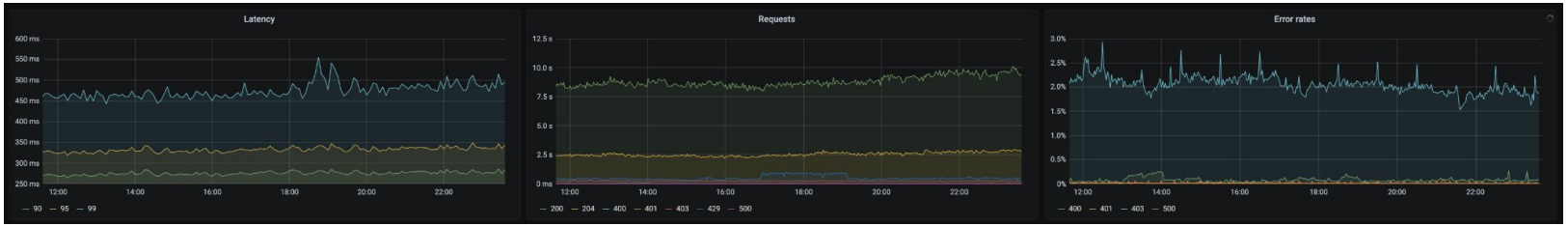
- Centralized logging
- Structured logging
- Log level



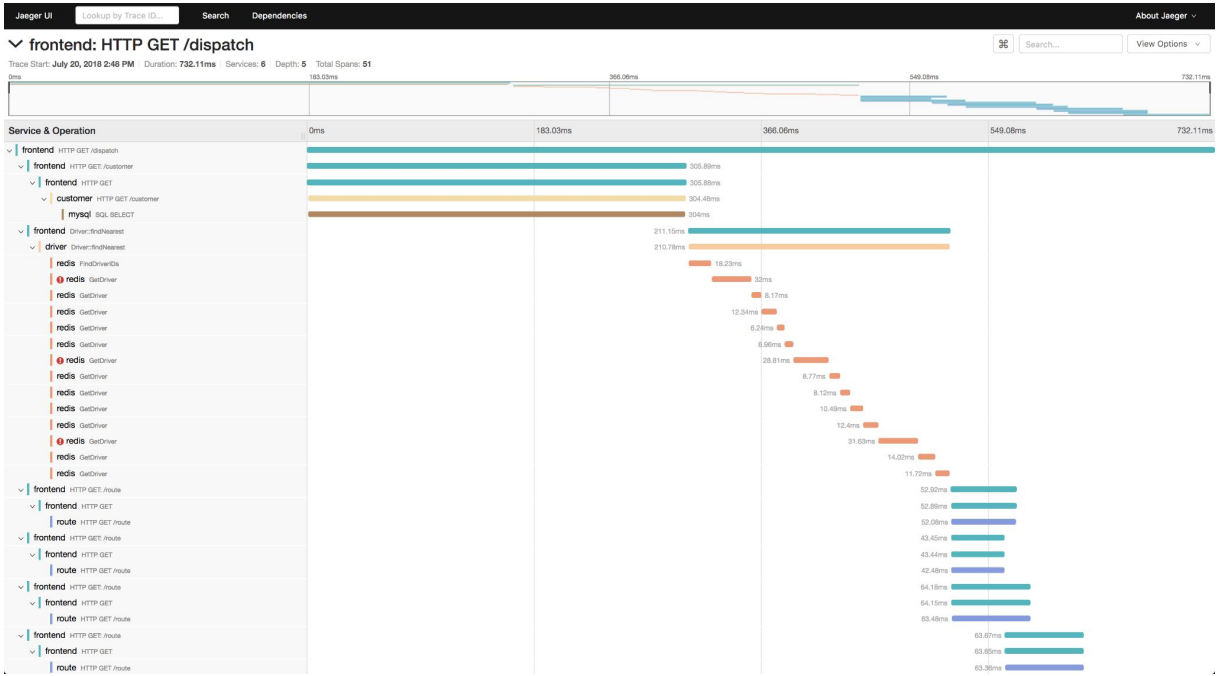
jolicode.com/blog/how-to-visualize-symfony-logs-in-dev-with-elasticsearch-and-kibana

Metrics

- Time series data
- Golden signal
  - Latency, Number of requests and Error rates



Distributed tracing



www.jaegertracing.io/docs/1.22/

Resiliency

- Timeout
- Circuit breaker
- Bulkhead
- Isolation between services
- Retry
  - Retry storm
  - Jitter and exponential backoff
  - Hedging/Speculative retry
- Fallback
- Health check

Scaling in microservices

- Make service stateless
  - Scaling stateless services is easier than stateful
- Idempotent
  - Outcome doesn’t change after first attempt
- Caching
  - In-memory, remote
- Splitting workloads
- Load balancing
- Worker, queue, Background job

Databases

- SQL
  - Relational
- NoSQL
  - Key/Value
  - Document
  - Column-oriented
  - Graph
- 1 database can have multiple type
- ACID or BASE
  - Atomic, Consistent, Isolated, Durable
  - Basically available, Soft state, Eventually consistent

Scaling Databases

- Choose suitable database
- CAP theorem
- Indexing
- Caching
- Read heavy/Write heavy
- CQRS
- Replication
- Sharding
    - Vertical/Horizontal
- Autoscaling

Example

- Kubernetes
- Netflix OSS

Part 7: Workshop 2

Using the link below

https://github.com/def4ultx/microservices-workshop

Part 8: Epilogue

There are always failure in distributed system

1. Understand your business
2. Choose the right tools
3. Know your tools
4. Avoid single point of failure
5. Scale out not scale up
6. Make the system resilience
7. Visualization is important
8. Have a plans when thing not go according to plans
9. Learn from mistakes
10. Just keep going

There is no right or wrong. Everything have a trade-off

- It always depend on the context you have
- Something look like anti-pattern might work well in different circumstances

Further readings

- Domain driven design - Eric Evans
- Building microservices - Sam Newman
- Designing Data-Intensive Application - Martin Kleppmann
- microservices.io
- docs.microsoft.com/en-us/azure/architecture/patterns/
- medium.com/a-technologists-pov/i-just-heard-that-monoliths-are-the-future-of-software-development-2190bf7f3c40

And much more

Happy learning

Thank you