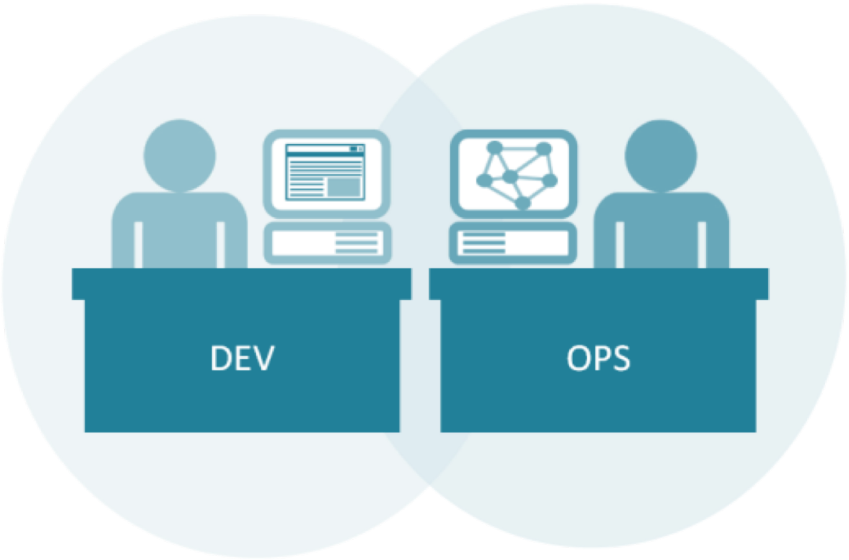


Microservices Engineering Boot Camp

DevOps Foundation

The journey is a “Team of Teams”

“First I needed to shift my focus from moving pieces on the board to shaping the ecosystem.”

– General S. McChrystal, *Team of Teams: New Rules of Engagement for a Complex World*

McChrystal (US Army General) understood that the **only way to scale the agility and speed** of a small team to a scope that could be employed in a hierarchical organization of many thousands of people was to employ radical transparency and **independent decision-making power that was decoupled** from the approval of the command structure. This “team of teams” approach reduced silos and flattened the organization, allowing much greater speed and adaptability.

DevOps Foundation
- Toyota Production System

8 Wasters

Your DevOps Journey:

- DevOps First way – Optimize Flow
- DevOps Second way – Amplify Feedback
- DevOps Third way – Continual Learning

Your DevOps Journey: Optimize Flow – First Way

- Infrastructure As Code
- Deployment Pipeline
- Automated Testing
- Continuous Integration
- Containerization/Microservices
- Architect for Low-Risk Releases

Your DevOps Journey: Amplify Feedback – Second Way

- Telemetry: Metrics, Monitoring and Alerting
- Use Telemetry to Anticipate Problems
- Feedback for Safe Deployment of Code
- Hypothesis-Driven Development
- Change Review and Coordination

Your DevOps Journey: Continual Learning – Third Way

- Learning culture
  - Blameless Postmortems
- Innovation culture
  - Rehearsing Failures
  - Knowledge Sharing
  - Reserve Time for Organizational Learning

Part 2:
Introduction to Microservices

Microservices is defined as a loosely-coupled, service-oriented architecture with bounded context.

Complication – Reasons and Types

- Context Switching
- Dependencies
- Multiple hand-offs
- Less Frequent Updates
- Slow Velocity
- App becomes complex

Situation: Enterprise team building an enterprise app

Lets imagine a medium sized enterprise app that has a team with roughly

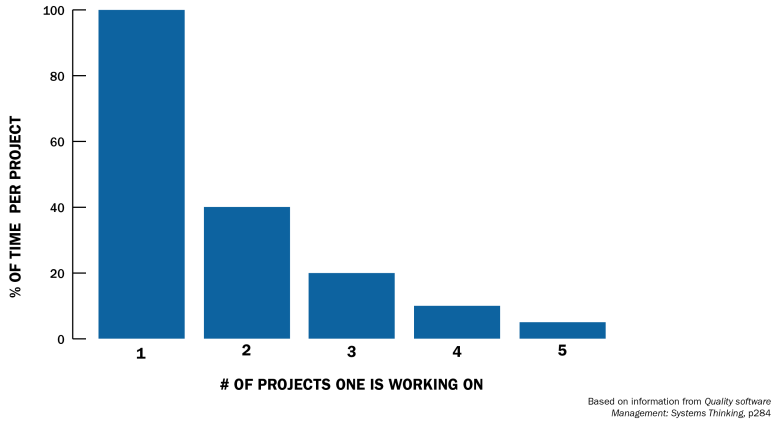
- **12 engineers**
- **2 project managers**
- **2 scrum masters**
- **3 database engineers**
- **8 UI engineers**
- **2 engineering managers**
- **1 UI engineering manager**
- **1 QA manager**
- **1 database engineering manager**
- **2 directors**
- **4 QA**

PROBLEM ???

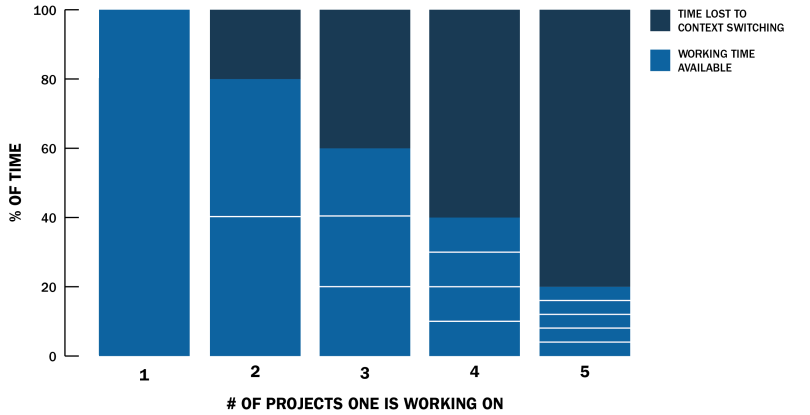
36 individuals total, with 1/3 partially allocated to this and **other projects**

Complication: Context Switching is Expensive

- According to the Software Engineering Institute at Carnegie Mellon University, **context switching** is **extremely expensive.**
- Switching projects results in rapidly diminishing results
- By implication, engineers manually tracking dependencies and switching to each manual task required for a specific service is similar in its impact on productivity
- Given that dependencies assignments are clustered with dev leads handling 3-7 each, productivity can be heavily impacted



Based on information from *Quality software Management: Systems Thinking*, p284



Based on information from *Quality software Management: Systems Thinking*, p284

- At 2 different tasks or projects, an engineer is losing 20% of their time to overhead. At 3 overhead jumps to 40%.
- Any value over 3 results in losing the majority of a resource's time to overhead and context switching
- Frequently, staff is assigned to multiple projects and jumps between meeting, reporting and development activities

Complication: Context Switching is Expensive

DevOps practices can help guard against some of the pitfalls of context switching, as well as alert the team when context switching is impacting product quality and team productivity. By leveraging continuous integration, any build failure will alert the team members when their contributions are impeding application or feature development. Likewise, automating the assignment of code reviews can insure code committed meets proper style and security standards.

According to Lewis and Fowler, a good way to overcome this hurdle – which resides among microservices and with external APIs – is through **domain-driven design**. DDD divides and re-bundles complex domains into multiple bounded contexts, which then maps out the relationship between them.

Complication: Dependencies

Typical app in a good case will have approx. 15 dependencies on other development groups

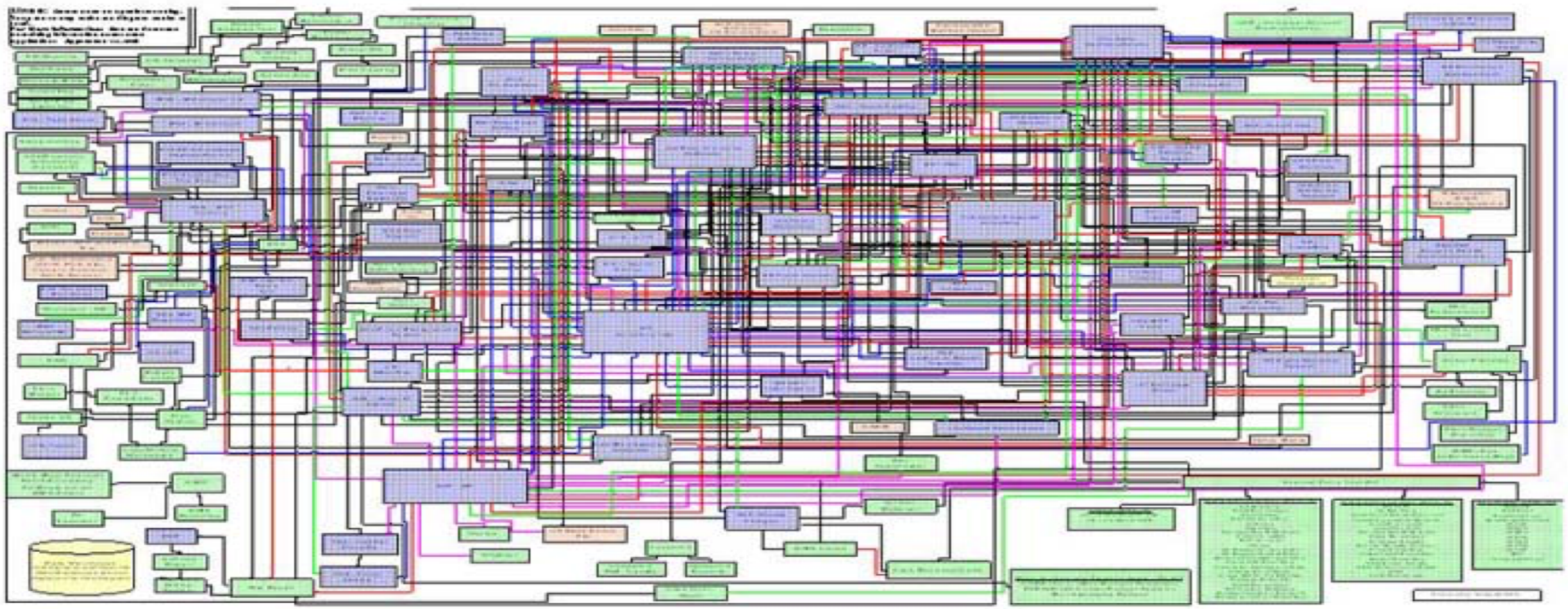
- Manual handoffs of data in file share drives
- Sharing of data tables
- Requisition of data center resources
- Approval from internal security groups
- Future creation of API's
  - Often private and undocumented

Complication

With no mechanism to reduce complexity, the application grows more complex

- Add dependencies
- Add features
- Add Security
- Add more complication ….

The Hairball



Complex App

- Deployment ??
- Scaling ??
- Updates ??

Complication: Deploying the complex app is hard

- Updates are less frequent
- Velocity slows
- Long QA cycles
- Multiple development teams must coordinate deployment and releases
- FEAR of new features

Complication: Scaling the complex app is hard

- We must scale the monolith, but it is highly coupled
- The best option is often to place it in a larger server (more CPU, memory)
- Typically very expensive to scale vertically
- Application becomes more vulnerable to outage and downtime

Complication: Updating the complex app is hard

- Long term commitment to technology choices
- Slow application startup times can drag down developer velocity as well
- Cost increases
- Perhaps fall behind newer competitors implementing on new technologies
  - In 1965, the average tenure of companies on the S&P 500 was 33 years. By 1990, it was 20 years. It's forecast to shrink to 14 years by 2026.
  - About 50 percent of the S&P 500 will be replaced over the next 10 years

Resolution: Microservice Architecture

Resolution: Create a Loosely Coupled API Service Platform, aka "Microservices"

- Decentralize planning and eliminate interdependencies

This will make it possible for a team of developers to conduct their planning and development in isolation, vastly improving time to market and cutting cost.

- A team of 7 planning without dependencies, with all the personnel required can reasonably plan for 10 weeks in a 1-2 days which would **result in agile planning event savings alone of 50%**

Skeptical? Consider a new team developing on Public Cloud (a Loosely Coupled API Service Platform)

(+) For example, a new development team on Cloud can submit API calls to create a VM, configure security routes, setup a firewall and create a database within minutes.

(-) The same operations within a common large enterprise often requires a complicated set of tickets and sophisticated connections to manage the hand-offs taking weeks to get started

Resolution: Microservices

Microservices enable Platforms

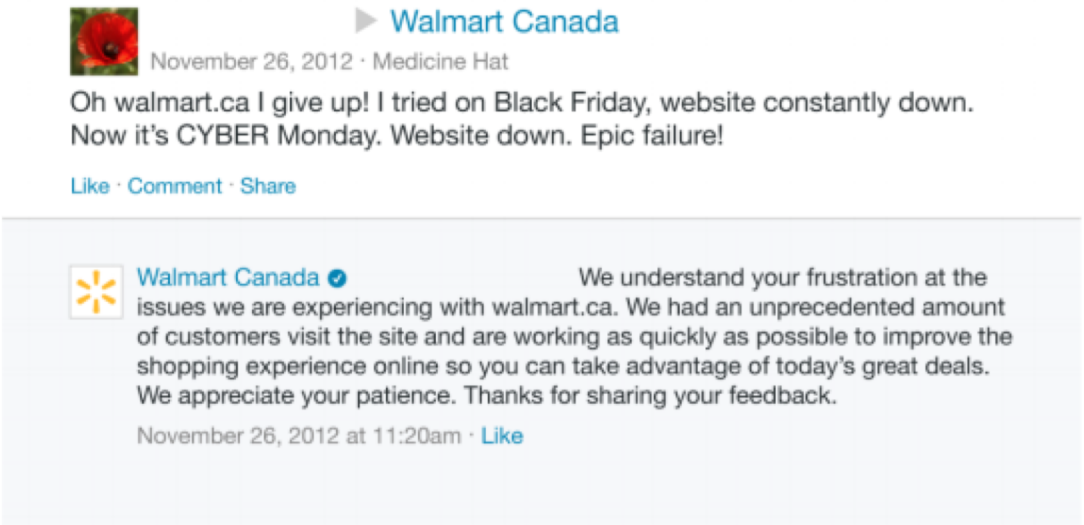
- A platform is an ecosystem of APIs that third parties may build functionality on top of.
- Notable Microservice Platform examples: Hootsuite, Gilt Group, SoundCloud, Amazon Web Services, Netflix, Spotify, Google Cloud, Box, and more!

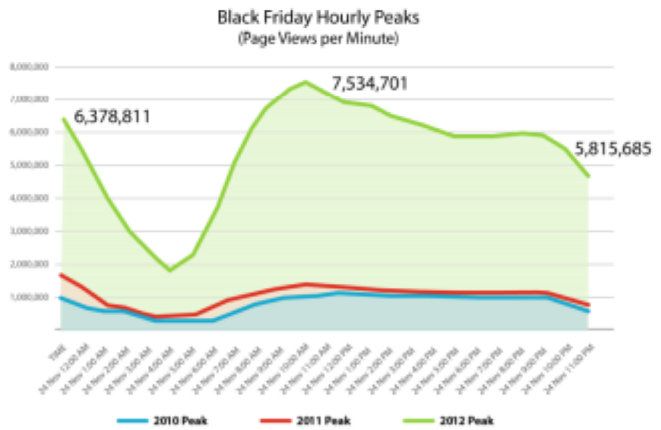
Walmart Canada Case Study

Walmart Canada: Situation

Designed in 2012, Walmart Canada's website **failed on Black Friday for two years in a row.**

- Traffic was increasing every year, keeping the team off balance
- Customer backlash, competitive pressures (i.e. Amazon) made this a top priority

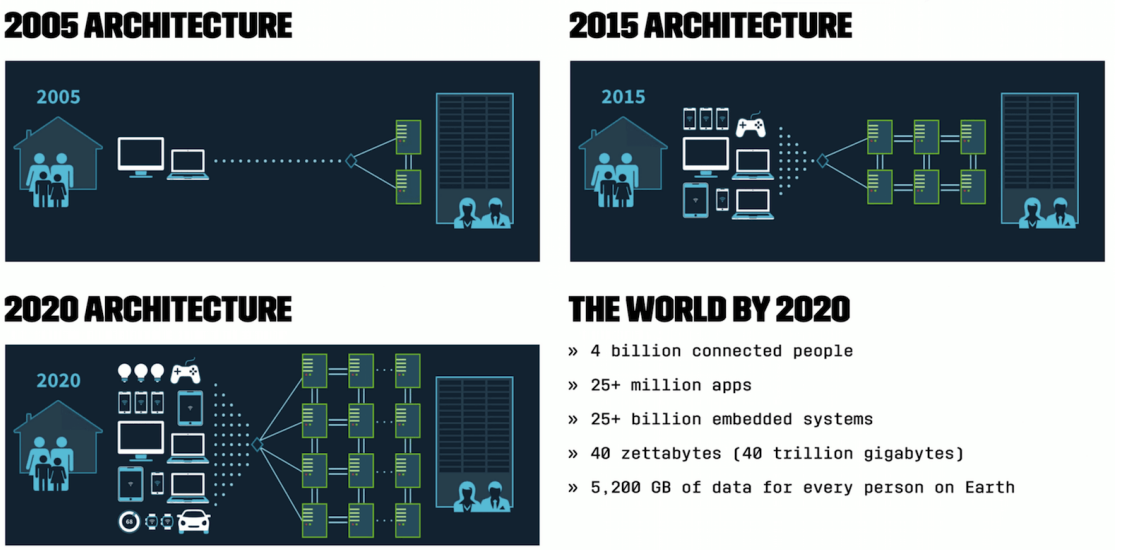




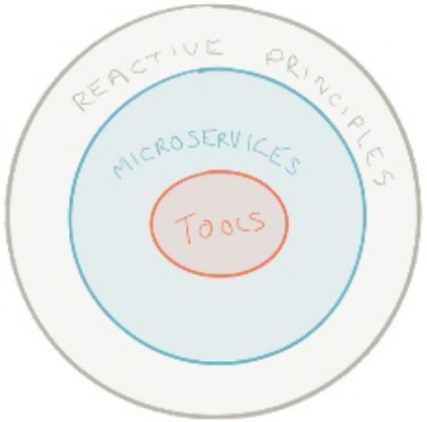
Walmart Canada: Goals

With traffic likely to continue to increase, goals were set

- Near 100% availability
- Consistent responsiveness under varying load conditions
- Predictable spikes of traffic e.g. Black Friday
- Less predictable spikes e.g. marketing campaign



Walmart Canada: Action



WHY, WHAT, HOW

Clip slid

Reactive (principles)

» responsive, resilient, elastic, message-driven

Microservices (strategy)

» bounded contexts (DDD), event sourcing, CQRS, eventual consistency

Tools (implementation)

» Typesafe Reactive Platform (Play, Akka, Spark)

Walmart Canada: Action

Re-architected solution

REACTIVE

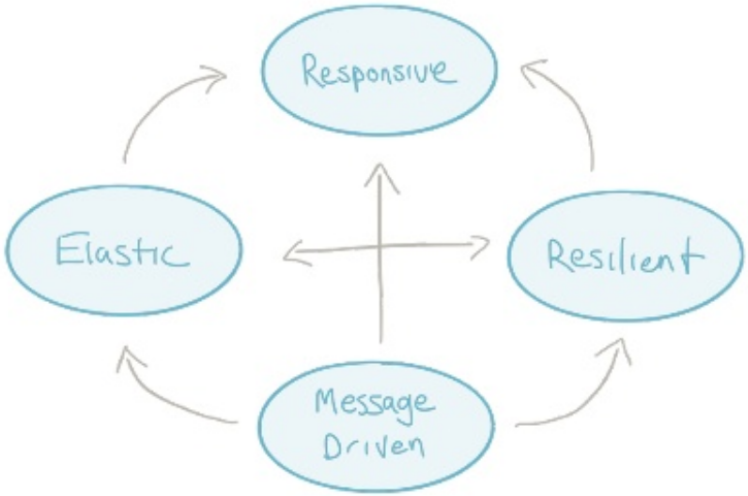
The ultimate maturity model
for microservices.

» Responsive

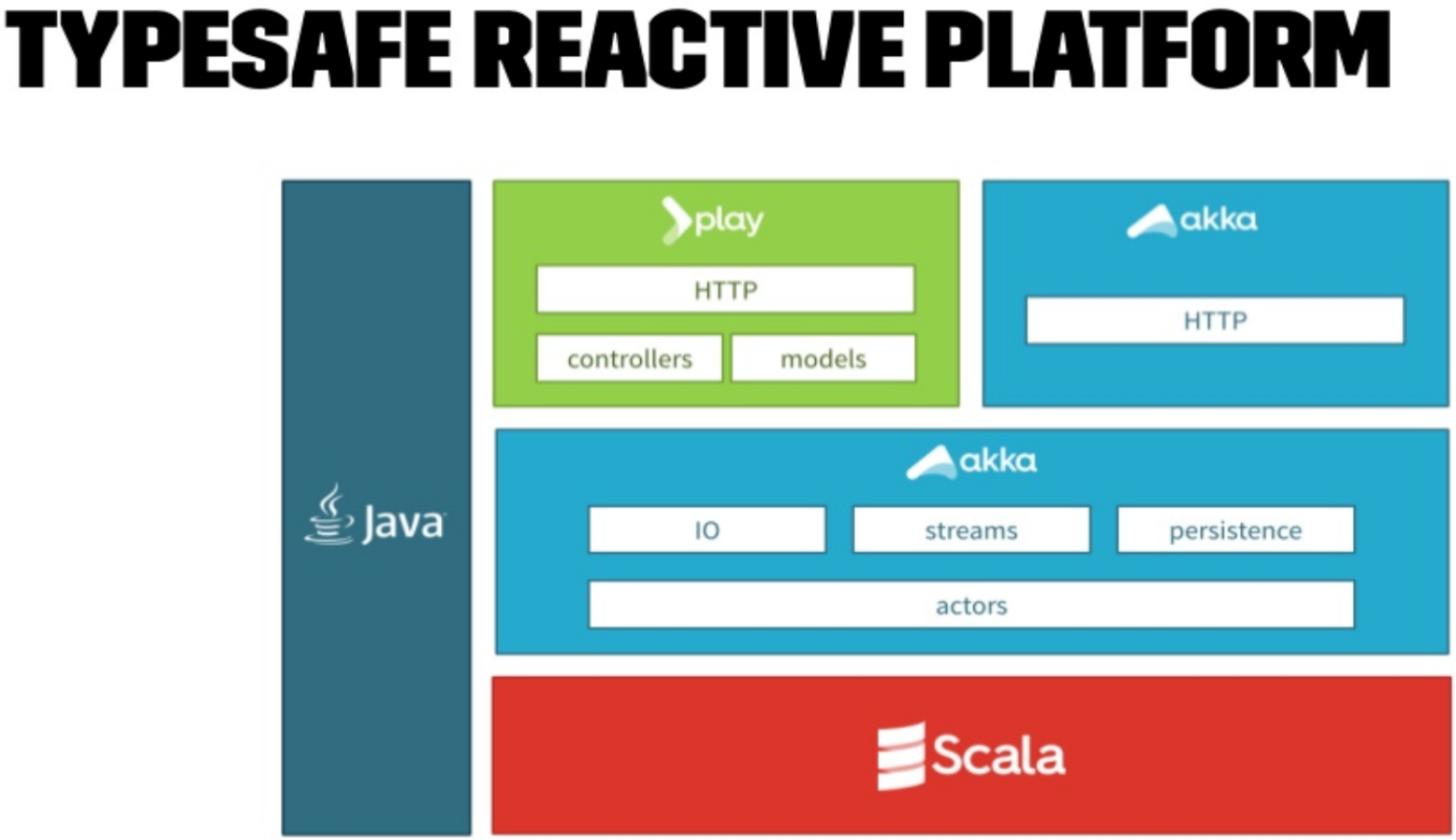
» Resilient

» Elastic

» Message-driven



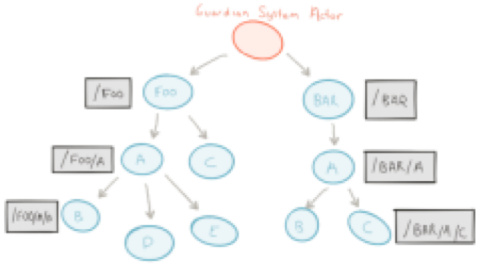
Walmart Canada: Action

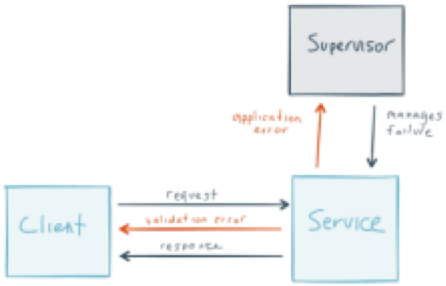


Walmart Canada: Action

Re-architected solution

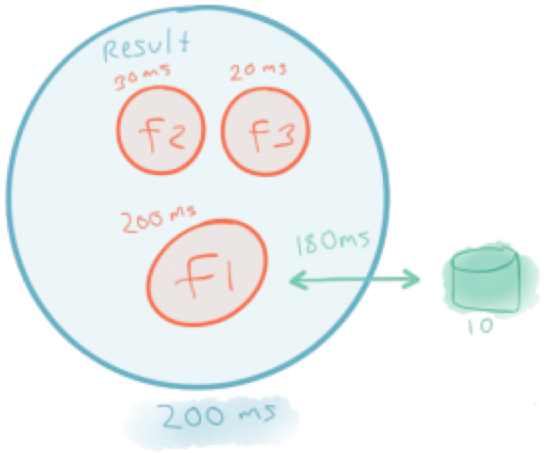
- **Message Driven**
  - Distribution
  - Location transparency
  - Isolation
- **Resilient**
  - Supervision
  - Dedicated separate error channel

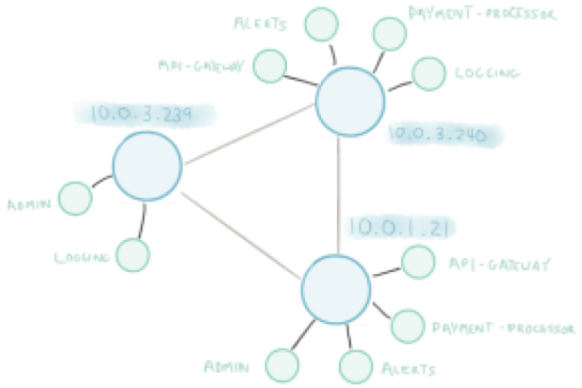




Walmart Canada: Action

- **Elastic**
  - **Scale up**
    - Async
    - Non-blocking
  - **Scale out**
    - Immutable data
    - Share nothing
- **Responsive**
  - Responsive to events, load failure, users
  - Distribution of data
  - Circuit Breakers





Walmart Canada: Result

Business Uplift

- Conversions up 20%
- Mobile orders up 98%
- No downtime on Black Friday or Boxing Day

Operational Savings

- Moved off expensive hardware
- Cheap virtual x86 servers
- 20-50% cost savings
- ~40% compute cycles

How to break an application into Microservices?

Based on existing system and natural architectural boundaries

- Database can be exposed as a Microservice for example
- Any slow process behind a queue can be considered a Microservice
- Language boundaries are an opportunity for Microservice creation

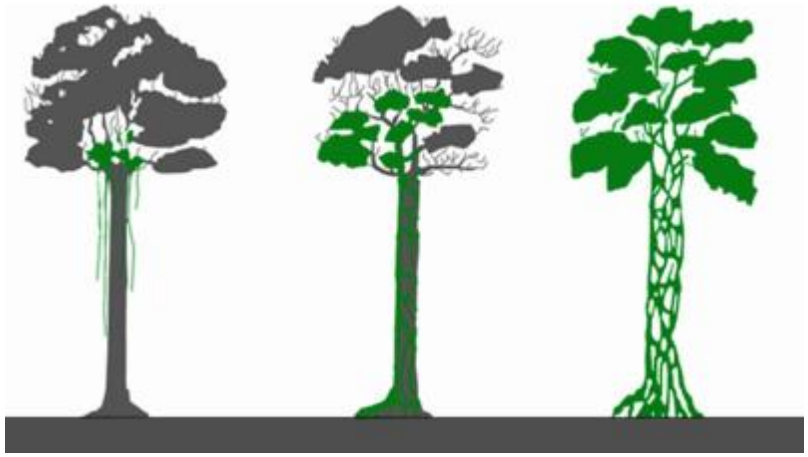
Strategy for legacy monoliths: The Strangler Pattern

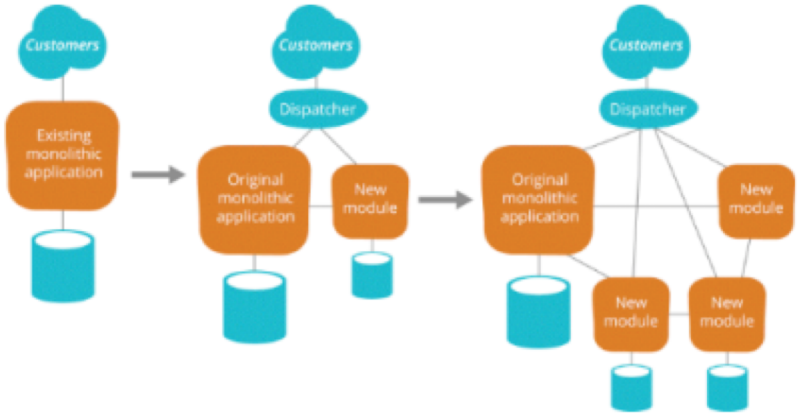
https://flic.kr/p/nMWBkR

Strategy for legacy monoliths: The Strangler Pattern

Gradually replace a legacy system by implementing new features of an existing system with those in one or several new systems

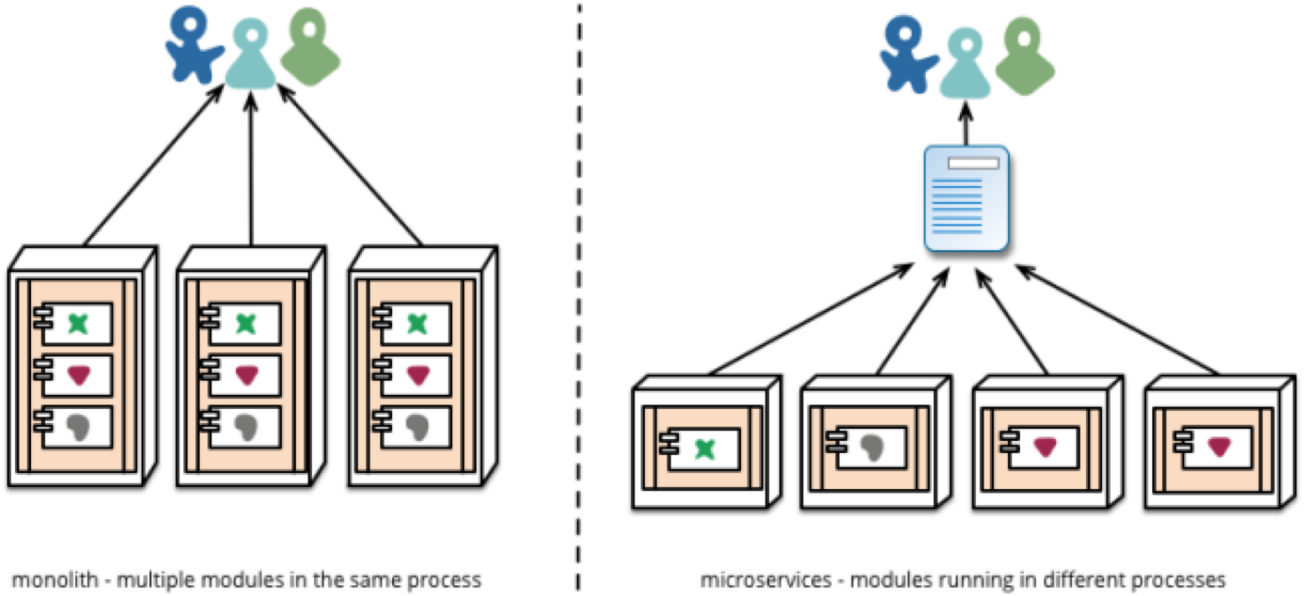
- Rather than trying to replace a large, expensive legacy system all at once you can iterate and improve in small amounts.
- This can be implemented for example by a load balancer in front the targeted functionality
- Over time, the old application is ‘strangled’ like a tree that has hosted a Strangler Fig.
- Focus on quick wins, needs that were not satisfied by existing software to deliver the most value quickly.





What is a Microservice?

- A service that can be independently deployed



monolith - multiple modules in the same process

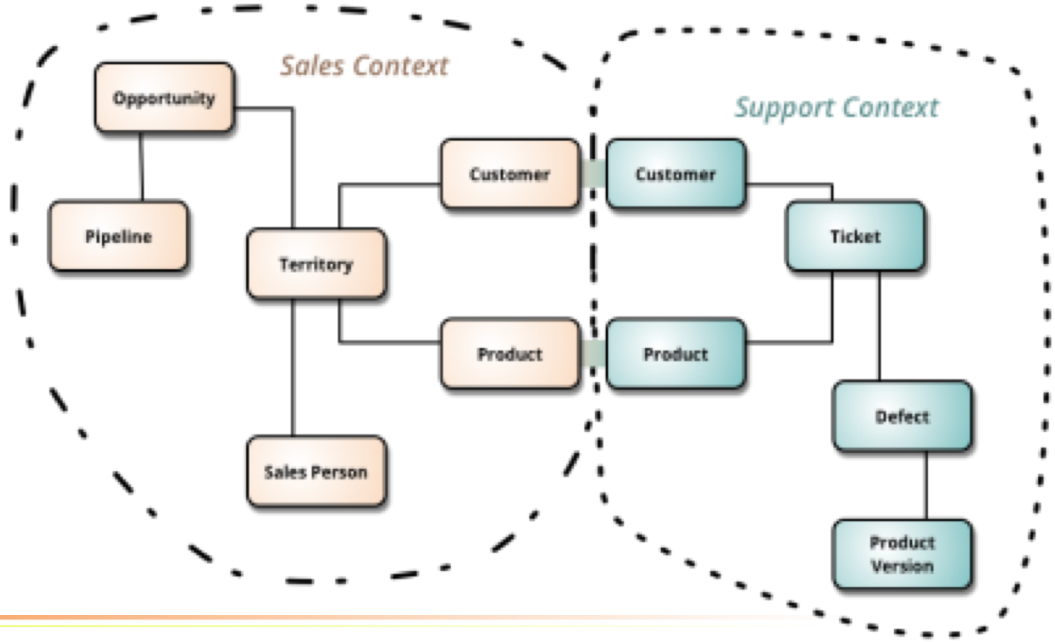
microservices - modules running in different processes

How big is a Microservice?

- A **single programmer** can design, implement, deploy and maintain a Microservice – Fred George
- Software that **fits in your head** – Dan North

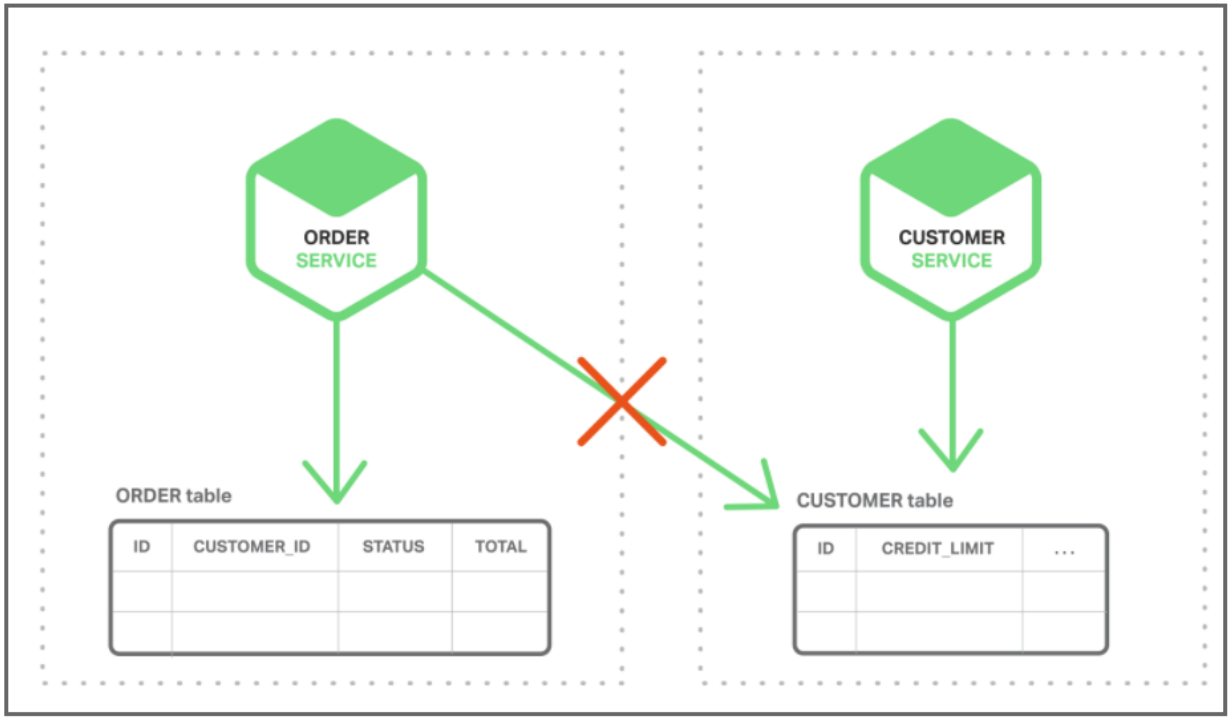
Where is the data for a Microservice?

- A single logical **database per service** – Chris Richardson
- A Microservice implements a single **Bounded Context**; Data model is driven by the domain model (DDD)
- **Event driven** data Management



Event Driven Data Management

- Implement business transactions that maintain consistency across multiple services
- Implement queries that retrieve data from multiple services

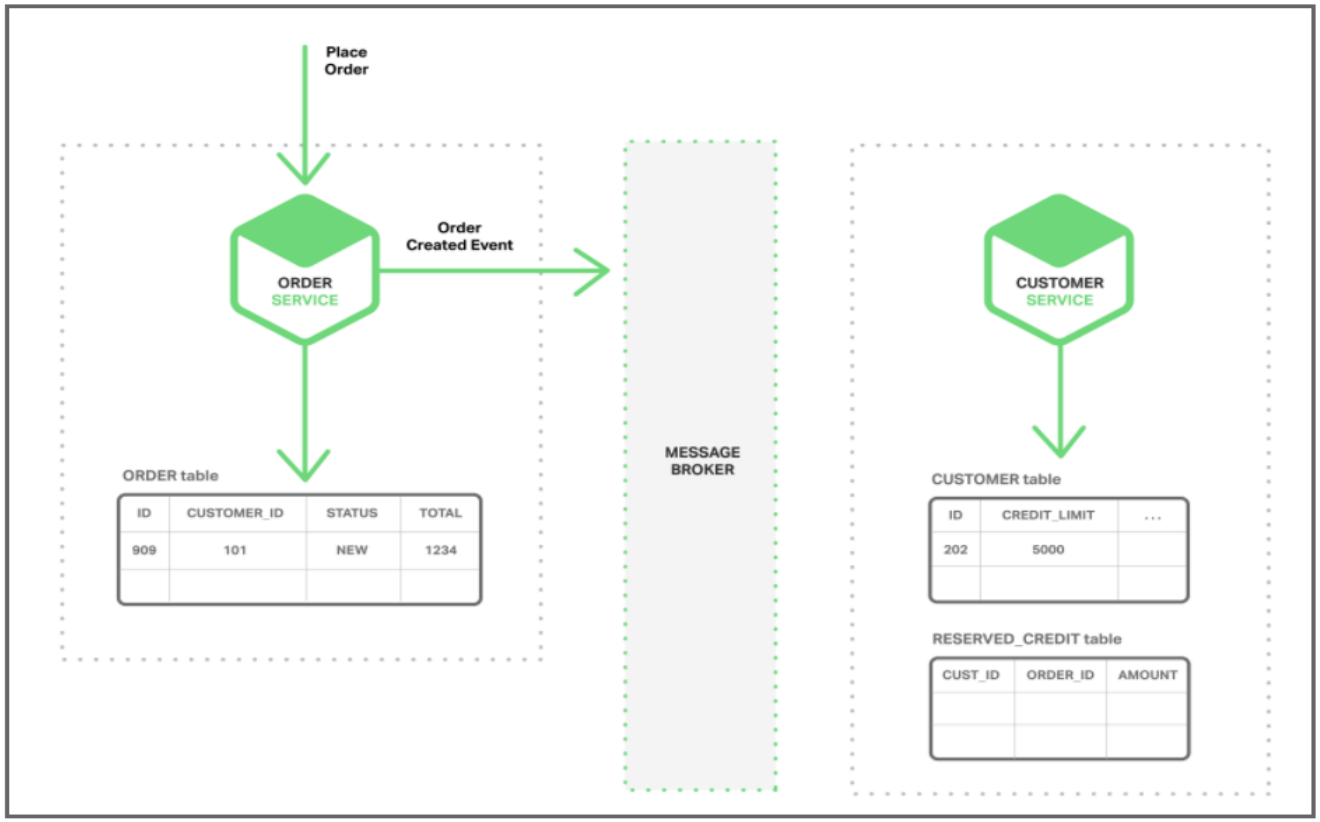


Event Driven Data Management

- Microservice publishes an **event** when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event it can update its own business entities, which might lead to more events being published.
  - use events to implement business transactions that span multiple services

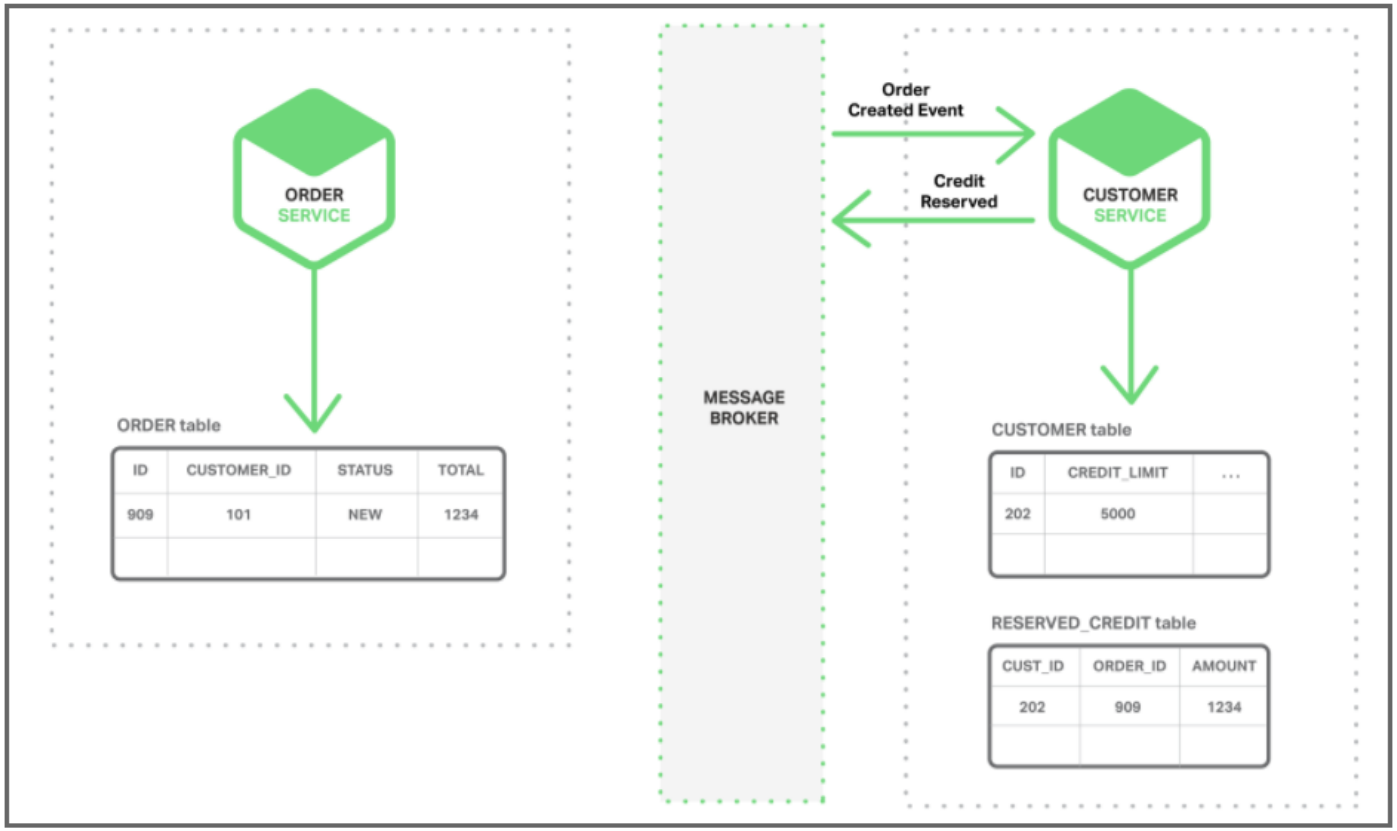
Event Driven Data Management

1. The Order Service creates an Order with status NEW and publishes an Order Created event.



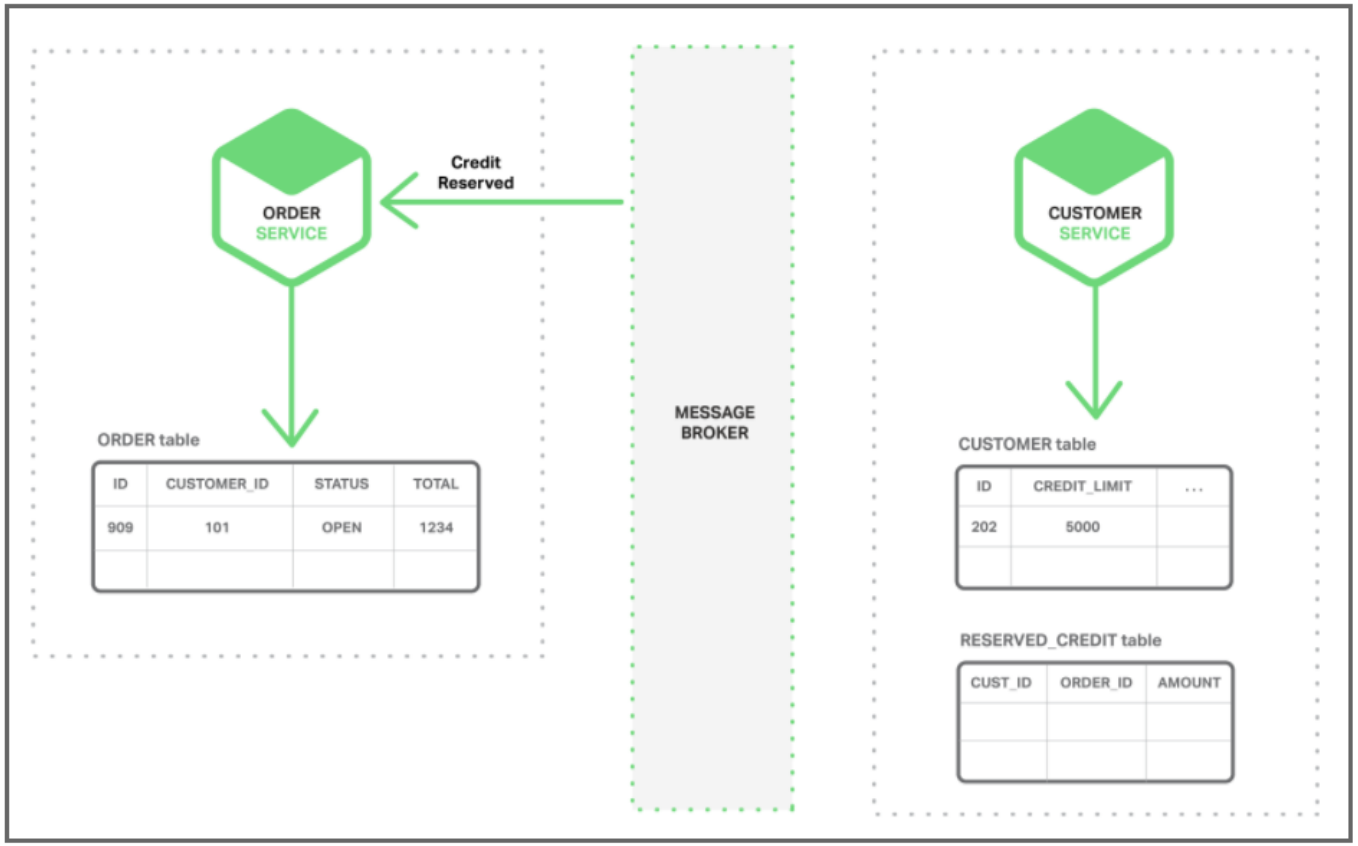
Event Driven Data Management

2. The Customer Service consumes the Order Created event, reserves credit for the order, and publishes a Credit Reserved event.



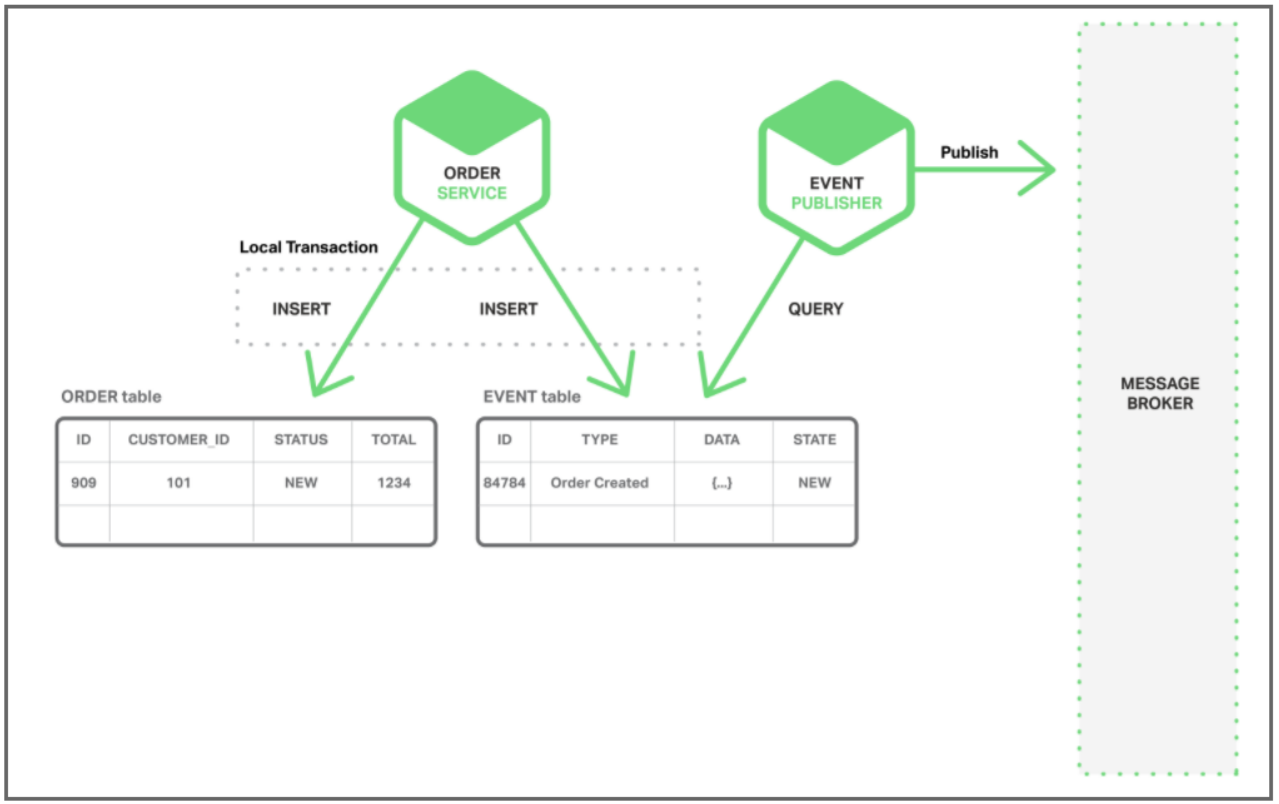
Event Driven Data Management

3. The Order Service consumes the Credit Reserved event, and changes the status of the order to OPEN.



Event Driven Data Management - Preserve Atomicity

EVENT table -- functions as a message queue, in the database that stores the state of the business entities. The application begins a (local) database transaction, updates the state of the business entities, inserts an event into the EVENT table, and commits the transaction. A separate application thread or process queries the EVENT table, publishes the events to the Message Broker, and then uses a local transaction to mark the events as published.



How to maintain Microservices?

- Addressable through a **service discovery** system – Chris Richardson
- Microservices **built & deployed independently. Stateless**, with state as backing services – 12Factor.net

** Service Discovery – Will discuss in “Patterns”

Microservice: advantages

- **Simple** services that are focused on doing one thing well
- Each service can be built **using** the **best** tool for the job
- Systems built this way are inherently **loosely coupled**
- Teams can deliver and **deploy independently** from each other
- Enable **continuous delivery** but allowing frequent releases while the rest of the system continues to be stable

Microservices

Solve organizational problems

- Teams blocked by or waiting on other teams
- Communication overhead
- Slow velocity

Cause Technical Problems

- Requires Devops, devs deploying and operating their services
- Need well defined business domains
- More distributed systems
- Need an orchestration layer

Self Service Mandate (Build a Platform)

1. All teams will expose their data and functionality through self service Api interfaces
2. No other forms of communication allowed, no tickets, direct linking, reads of data stores, no back doors, only service interface calls over the network
3. All services must be designed to be externalizable. The team must plan and design to be able to expose the api's to developers in the outside world, creating a platform.

The now-famous Jeff Bezos rant, leading to formation of Amazon Web Services
https://gist.github.com/chitchcock/1281611

DevOps = Optimize for Speed

"Therefore, broadly speaking, to achieve DevOps outcomes we need to reduce the effects of functional orientation ("optimizing for cost") and enable market orientation ("optimizing for speed")."

-Gene Kim

- This consists of having many small teams work independently, rapidly delivering customer value
- Teams are cross-functional and independent. Each team, consisting of 8 or fewer resources, deploys independently.
- Market oriented teams are responsible for feature development, testing, securing, deploying, and supporting service in production

MicroServices = Optimize for Speed

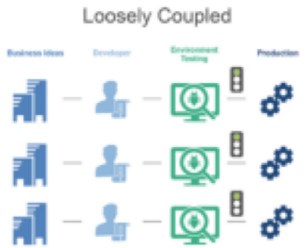
"Organizations with these types of service-oriented architectures, such as Google and Amazon, have incredible flexibility and scalability. These organizations have tens of thousands of developers where small teams can still be incredibly productive."

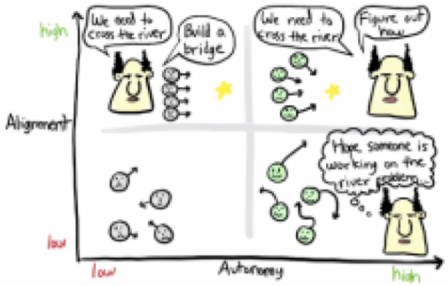
-Randy Shoup, former Engineering Director for Google App Engine

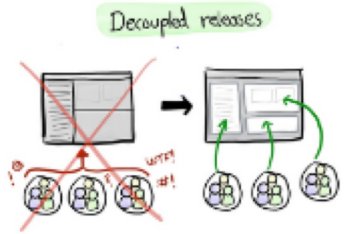
- This consists of having many small teams work independently, rapidly delivering customer value

Small Team Size

- Build platform with small teams of 6-8 engineer teams
- **Self Sufficient**: Each team completely owns one or more services and has everybody they need to develop functionality
- **Automate Interfaces:** Teams connect to each other through API's. Teams don't hire an army of program managers to manage dependencies.
- **Decentralized:** Teams plan, develop and deploy autonomously







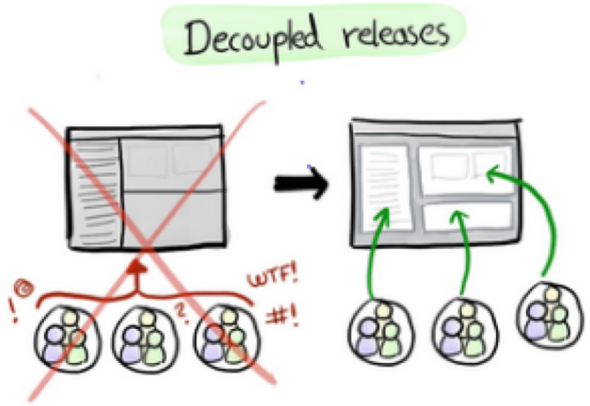
Starting and Scaling DevOps in the Enterprise by Gary Gruver
https://www.amazon.com/dp/B01M332BN2/ref=dp-kindle-redirect?_encoding=UTF8&btkr=1

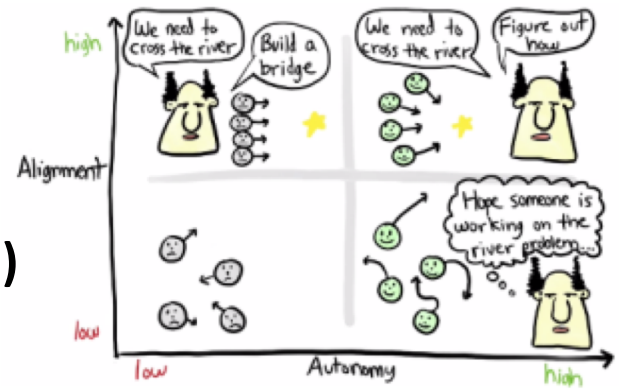
Why Small Team Size? Scaling the Organization

Autonomy

- Decentralizes power and creates autonomy which is critical to scaling the organization
- Each team can focus on the business metric they are responsible for and act autonomously to maximize the metric.

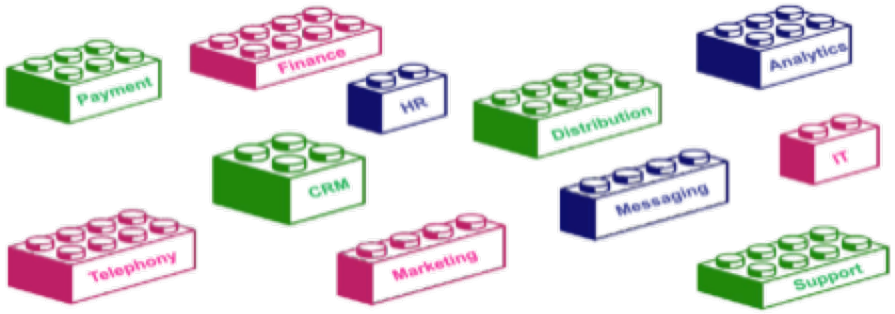
Amazon CTO Werner Vogels explained the advantages of this structure to Larry Dignan of *Baseline* in 2005 (see notes accompanying this slide)





Science behind why two-pizza team works http://blog.idonethis.com/two-pizza-team/
ITRevolution, Conways Law by Gene Kim http://itrevolution.com/conways-law/

Microservice building blocks



APIs provide the flexibility for businesses to grow by adopting new business models and enable accelerated development of new applications.

It is like picking different LEGO blocks to build a toy house.

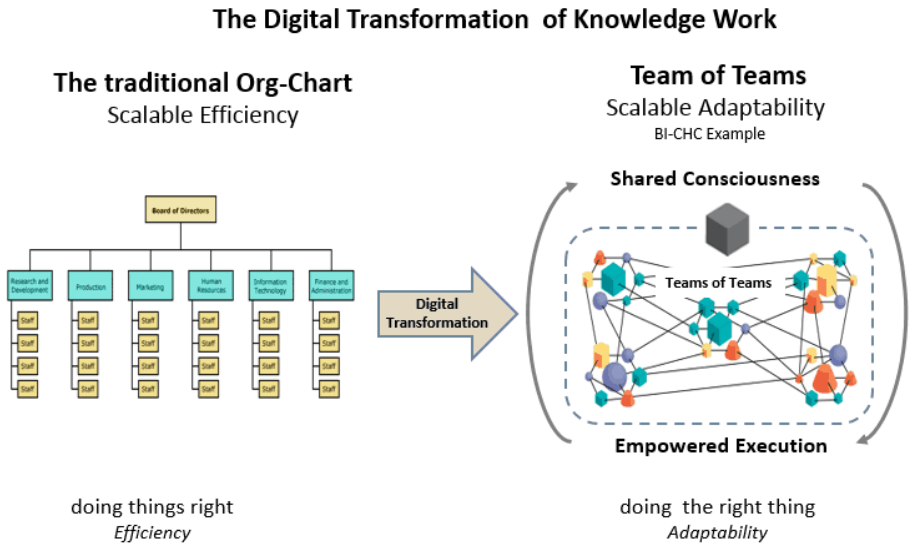
Resolution: Service Oriented Architecture

This concept aligns with General Stanley McChrystal's recent work Team of Teams.

- As McChrystal writes: "To defeat a network, we had to become a network."
- A network in this context is a collection of small cross-functional teams that have been empowered to self-organize, self-manage, and self-execute.
- Teams are enabled to become a 'network' by a Service Oriented Architecture (SOA) and have done so at various adopted of SOA such as AWS, Netflix et tal
- Cost, time to market, and adaptiveness are vastly improved

What is needed?

Changing Management Structures by moving from scalable Efficiency to scalable Adaptability in order to succeed



External Benchmarks

2016 IT Performance by Cluster

	High IT Performers	Medium IT Performers	Low IT Performers
Deployment frequency *For the primary application or service you work on, how often does your organization deploy code?*	On demand (multiple deploys per day)	Between once per week and once per month	Between once per month and once every 6 months
Lead time for changes *For the primary application or service you work on, what is your lead time for changes (i.e., how long does it take to go from code commit to code successfully running in production)?*	Less than one hour	Between one week and one month	Between one month and 6 months
Mean time to recover (MTTR) *For the primary application or service you work on, how long does it generally take to restore service when a service incident occurs (e.g., unplanned outage, service impairment)?*	Less than one hour	Less than one day	Less than one day*
Change failure rate *For the primary application or service you work on, what percentage of the changes either result in degraded service or subsequently require remediation (e.g., lead to service impairment, service outage, require a hotfix, rollback, fix forward, patch)?*	0-15%	31-45%	16-30%

* Low performers were lower on average (at a statistically significant level), but had the same median as the medium performers.

*** Source: Puppet Labs Report**

Competitive Advantage

High performing teams are breaking away from the pack and the status quo of even three years ago is a dangerous assumption currently.

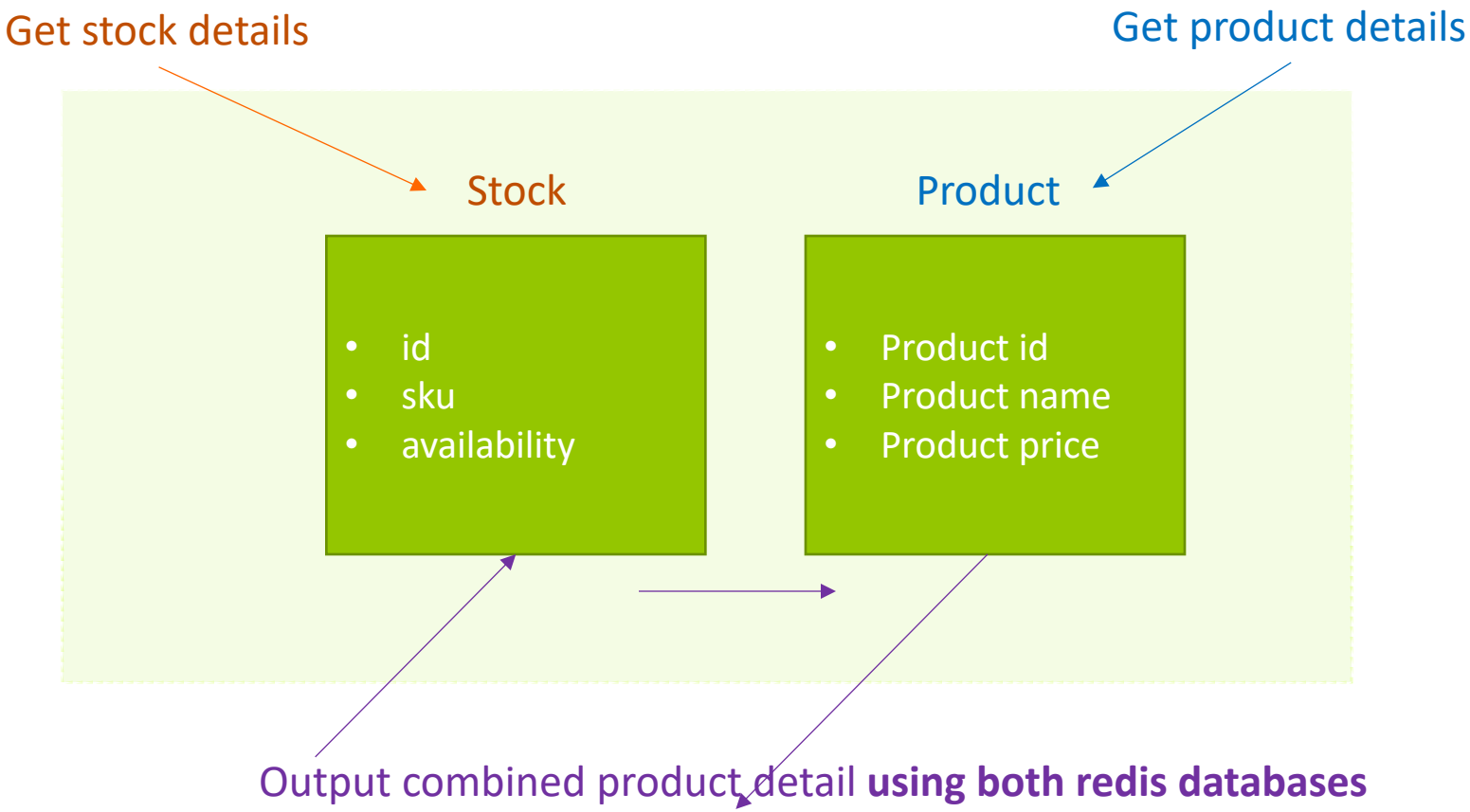
A team which had state of the

Microservices Patterns

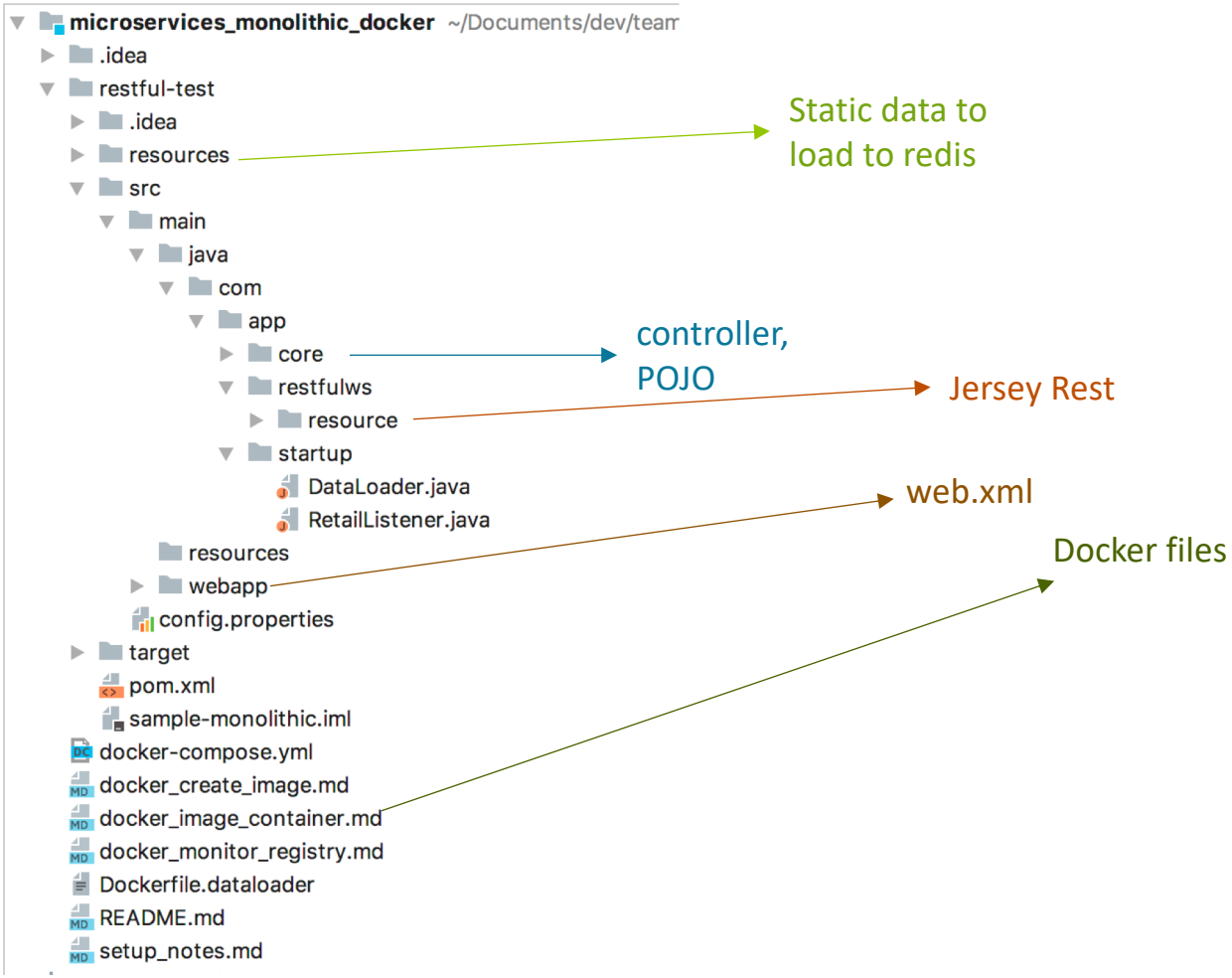
Before diving into Microservices Patterns, let’s walk through a Sample Monolithic Application.

Later we will convert the same application to Microservices and deploy in Kubernetes

Sample Monolithic App in Java



Project Structure



https://github.com/shekhar2010us/microservices_monolithic_docker/blob/master/README.md

Setup – Running Java Monolithic

- Login to virtual machine
- Install docker, maven and git
- Pull code
- Build code
- Build docker image
- Run docker-compose

https://github.com/shekhar2010us/microservices_monolithic_docker/blob/master/setup_notes.md

Writing Monolithic - Java

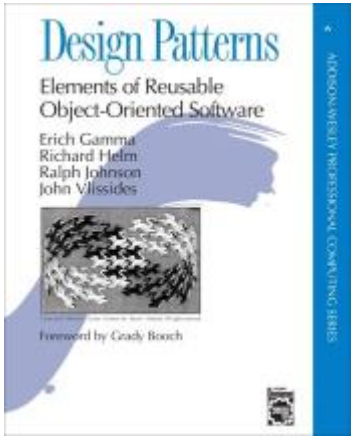
CLASSROOM WORK 101 (45 minutes)

1. **Maven Build and Run application**
2. **Redis data store**
3. **Jersey Rest API**
4. **Docker compose**

https://github.com/shekhar2010us/microservices_monolithic_docker

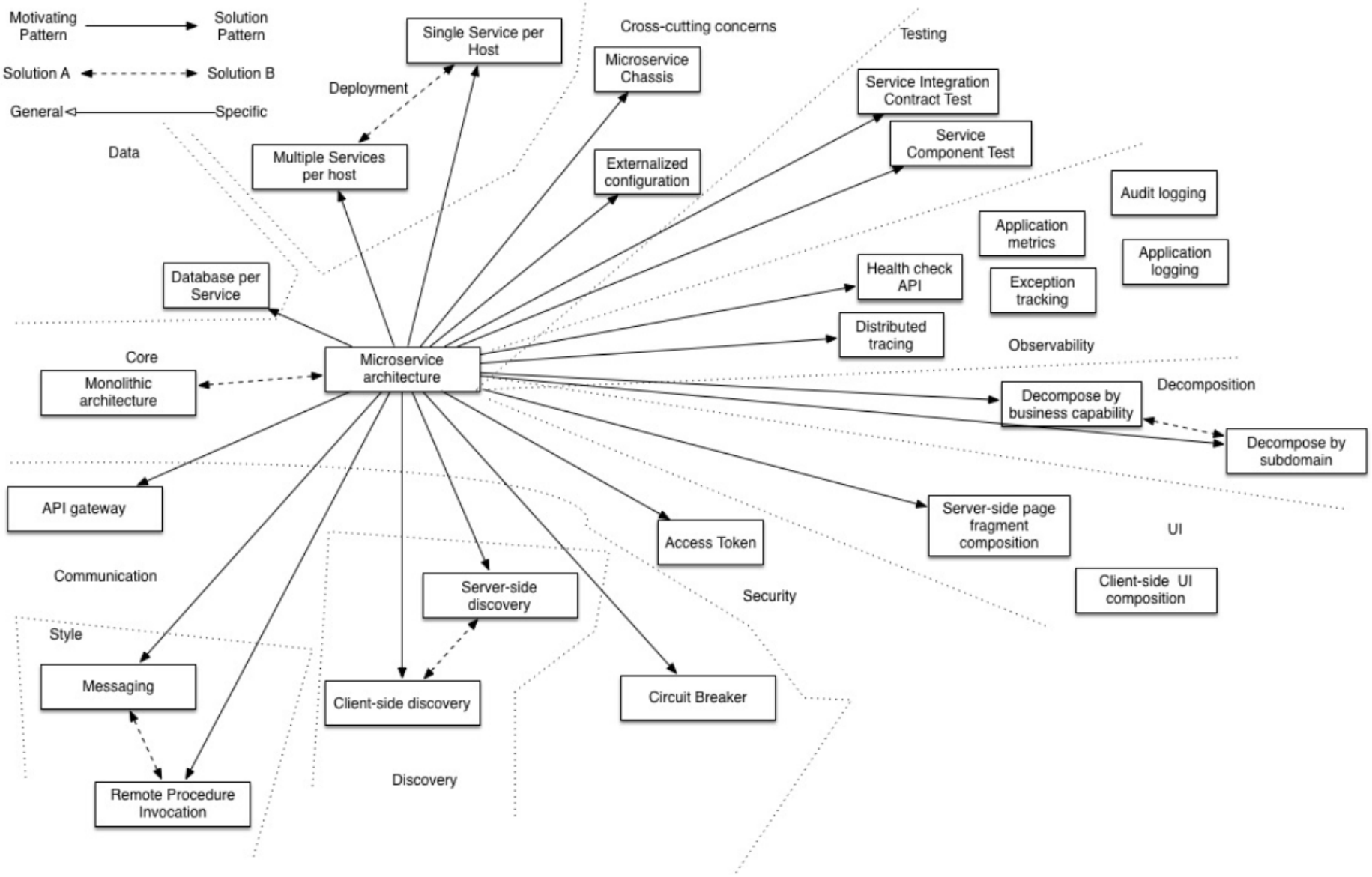
Patterns

What is a pattern?



Reusable **solution** to a **problem** occurring in a particular **context**

Microservice Architectures



Pattern - Categories

- **Data Management**
- Communication
- Deployment
- Discovery
- Reliability
- Observability
- Testing

Data Management Patterns

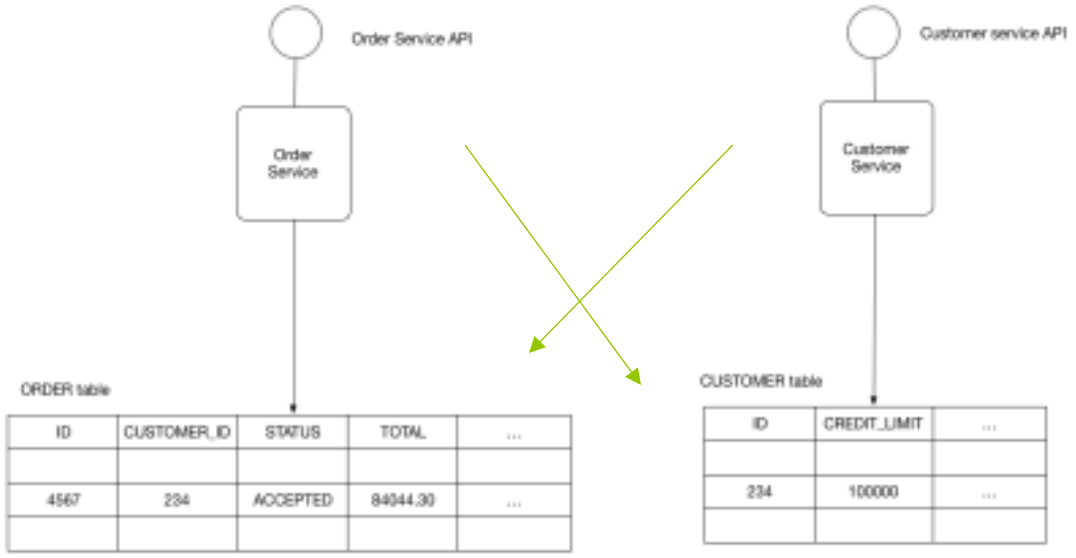
- Shared database
- Database per Service
- Saga
- Event Sourcing
- CQRS (Command Query Responsibility Segregation)
- API Composition

Shared Database

Shared Database
Use a (single) database shared by multiple services. Local ACID transactions.

Benefits:
- Straightforward ACID transactions to enforce data consistency
- Single database is simpler to operate

Drawbacks:
- Since all services access the same database, interference and transaction locks on the table.
- Single database might not satisfy the data storage and requirements of all services.



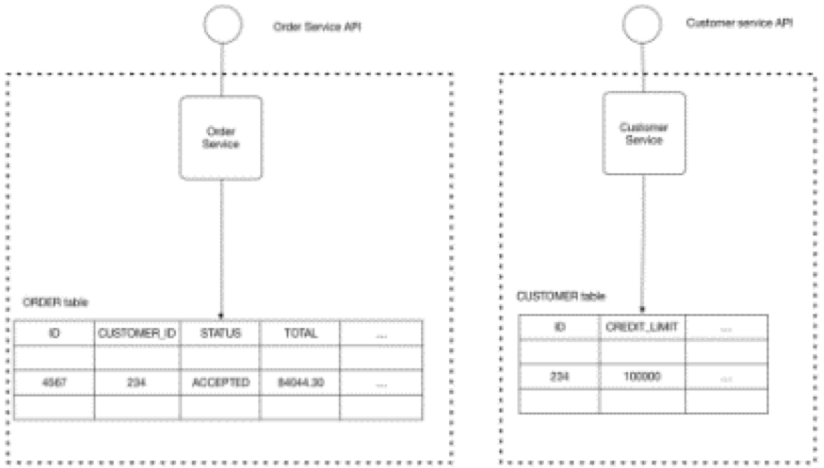
Database Per Service

Database Per Service
Every Microservice has its own persistent data and accessible only via its API
E.g. Private-tables-per-service; Schema-per-service; Database-server-per-service

Benefits:
- Ensures loose coupling
- Each service can use the type of database that is best suited to its needs. E.g., text searches could use ElasticSearch; social graph could use Neo4j.

Drawbacks:
- Transactions involving multiple services not straight forward

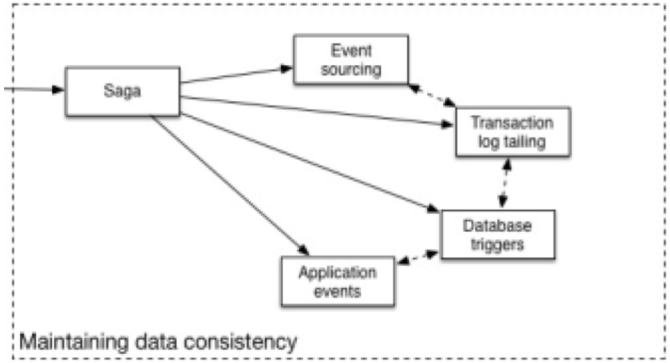
- ❑ For consistency; may need to implement capability to roll back (**Saga** pattern)
- ❑ Interaction is achieved by message/event brokers



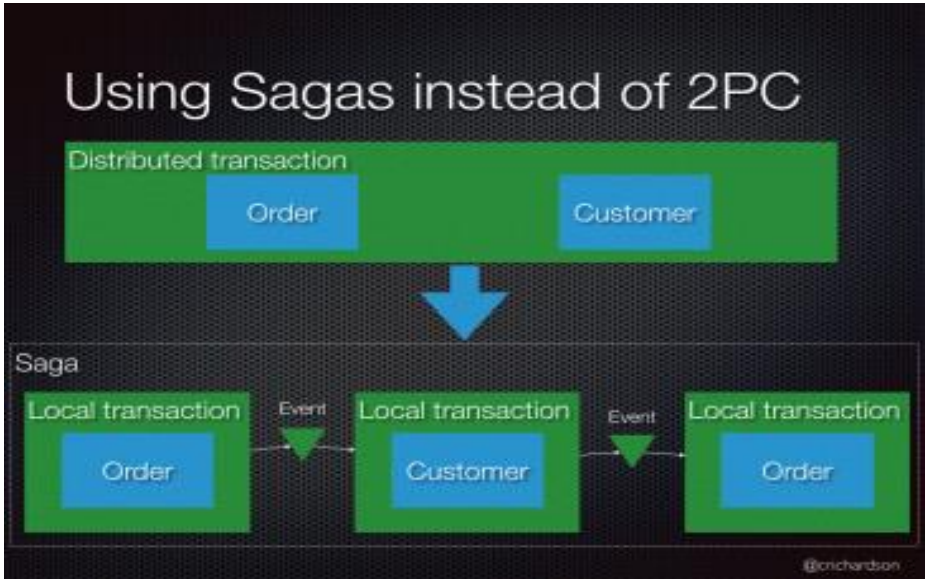
Saga Pattern – For Data Consistency

Saga Pattern

- Grouping transactions with the capability to undo changes if the transaction fails
- enables an application to maintain data consistency across multiple services



Maintaining data consistency



How to reliably/atomically publish events whenever state changes? Solution can be **Event Sourcing**

Event Sourcing

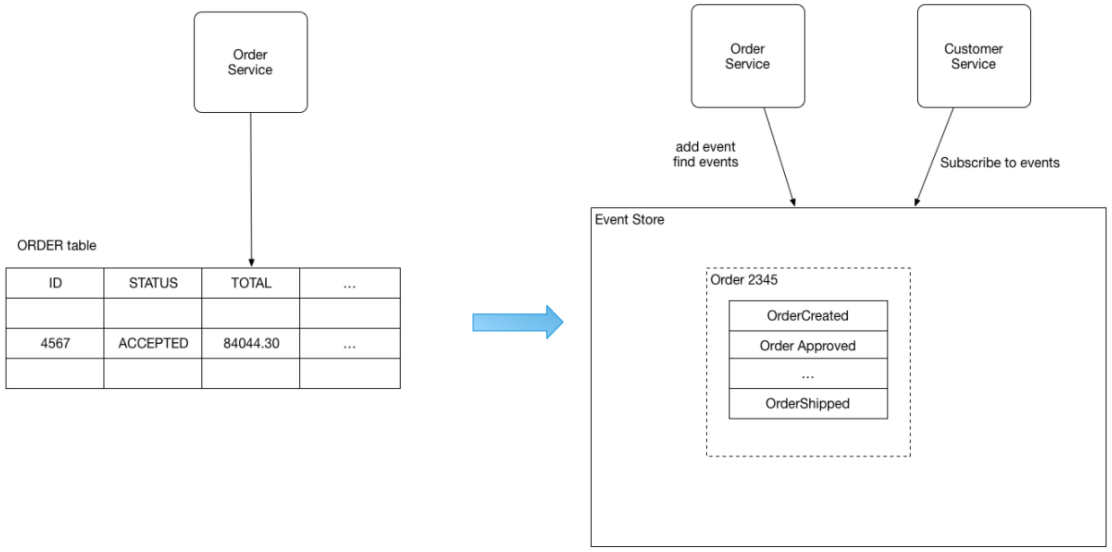
- Persists the state of a business entity as a sequence of state-changing events – in Event Store
- The store has APIs for adding, retrieving an entity's events and subscribe to events.
- The store is like a message broker; A new event is delivered to all interested subscribers.
- E.g. the Order is saved as sequence of events, and customerService subscribe to those events

Benefits:

- reliably publish events whenever state changes

Drawbacks:

- Difficult to design (unconventional)
- The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities - complex and inefficient --- **use CQRS to query**

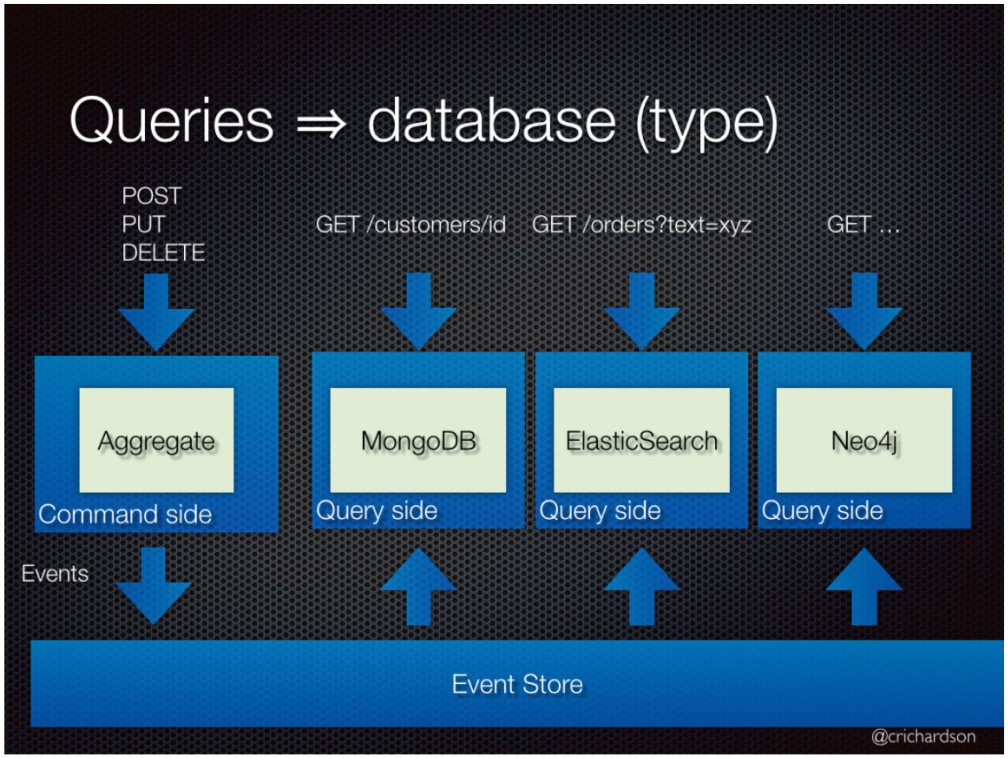


CQRS – Command Query Responsibility Segregation

Split the application into two parts:

- **Command-side**: handles create, update, and delete requests and emits events when data changes.
- **Query-side:** handles queries by executing them against one or more materialized views that are kept up to date by subscribing to the stream of events emitted when data changes.

A Command cannot return data and
a Query cannot change the data.



Use Case

- Get data from multiple different Aggregates
- Application has an imbalance in responsibility (read/writes)

How to

- Two models talking to same date store (applications that require synchronous processing and immediate results.)
- Two models talking to different data store (Eventual Consistency)

API Composition

Perform queries in Microservices architecture

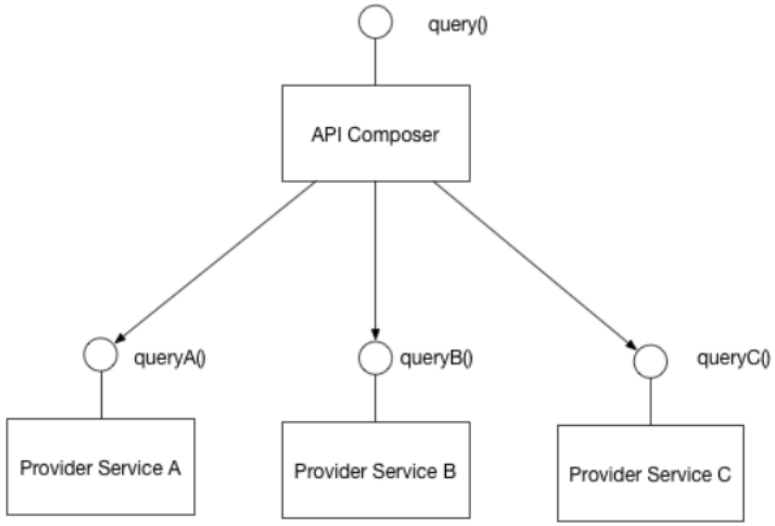
- Implement a query by defining an *API Composer*, which invoking the services that own the data and performs an in-memory join of the results.

Benefits:

- It a simple way to query data in a Microservice architecture

Drawbacks:

- Some queries would result in inefficient, in-memory joins of large datasets.



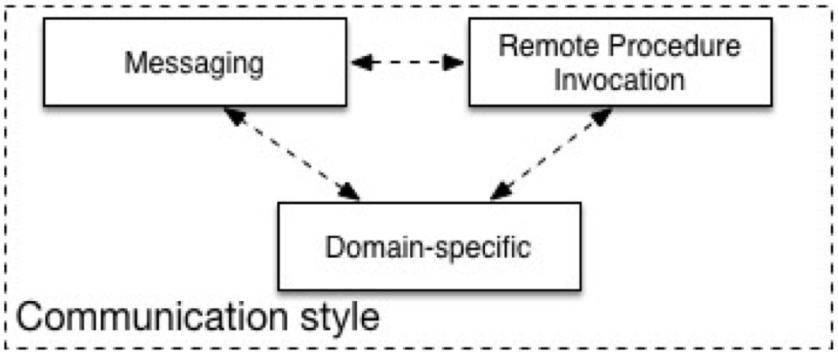
Pattern - Categories

- **Data Management**
- **Communication**
- Deployment
- Discovery
- Reliability
- Observability
- Testing

Communication Style

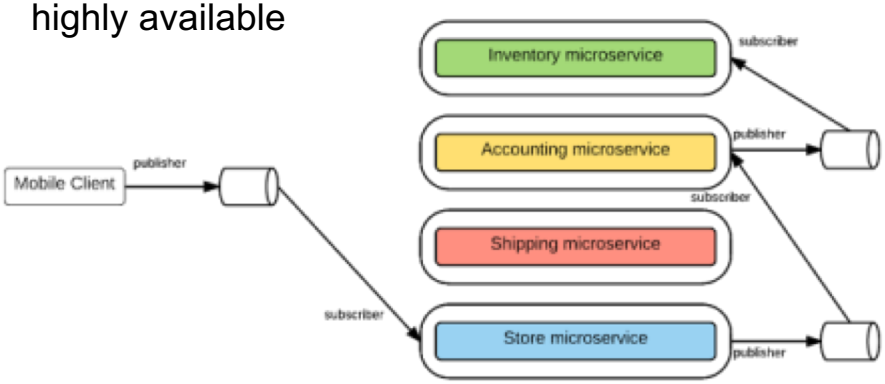
Messaging

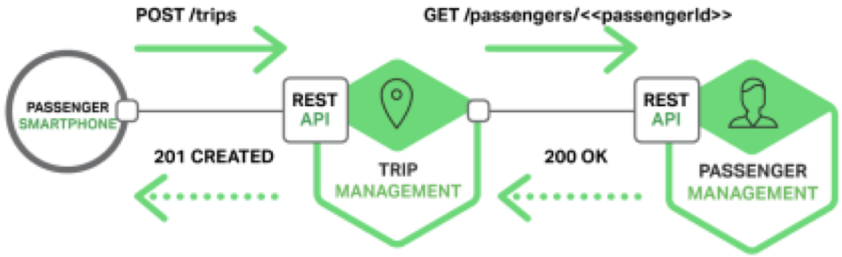
- Services communicate by sending asynchronous messages
- Decouples client & service
- Must also install and maintain a broker (Kafka, RabbitMQ, etc)
- Complex and should be highly available



Remote Procedure Invocation

- Simple and familiar
- Both client and service must be available
- Only simple request and reply. Notifications, publish/subscribe, async communication not supported
- E.g. Rest, Thrift





Pattern - Categories

- **Data Management**
- **Communication**
- **Deployment**
- Discovery
- Reliability
- Observability
- Testing

Deployment Patterns

- Service instance per host
- Multiple Service instance per host
- Service instance per VM
- Service instance per container
- Server-less

Service instance per host

Deploy each single service instance on its own host

Benefits:

Services instances are isolated from one another

There is no possibility of conflicting resource requirements or dependency versions

A service instance can only consume at most the resources of a single host

It is straightforward to monitor, manage, and redeploy each service instance

Drawbacks:

Potentially less efficient resource utilization

Multiple Service instance per host

Run multiple instances of different services on a host (Physical or Virtual machine).

- Deploy each service instance as a JVM process. For example, a Tomcat or Jetty instances per service instance.
- Deploy multiple service instances in the same JVM. For example, as web applications or OSGI bundles.

Benefits:

- More efficient resource utilization

Drawbacks:

- Risk of conflicting resource requirements
- Risk of conflicting dependency versions
- Difficult to limit the resources consumed by a service instance
- If multiple services instances are deployed in the same process then its difficult to monitor the resource consumption of each service instance. Its also impossible to isolate each instance

Service instance per VM

Package the service as a virtual machine image and deploy each service instance as a separate VM

Benefits:

- Straightforward to scale the service by increasing the number of instances. Amazon Autoscaling Groups can even do this automatically based on load.
- The VM encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in exactly the same way.
- Each service instance is isolated
- A VM imposes limits on the CPU and memory consumed by a service instance

Drawbacks:

- Building a VM image is slow and time consuming

Service instance per Container

Package the service as a (Docker) container image and deploy each service instance as a container

E.g. Kubernetes, DC/OS, Marathon/Mesos, Docker Swarm

Benefits:

- Easy to scale up and down a service by changing the number of container instances.
- The container encapsulates the details of the technology used to build the service.
- Each service instance is isolated
- A container imposes limits on the CPU and memory consumed by a service instance
- Containers are **extremely fast to build and start**. E.g. 100x faster to package an application as a Docker container than it is to package it as an AMI.

Drawbacks:

- Less mature than VMs

Server-less

Deployment infrastructure hides the concept of servers and takes the service's code & desired performance characteristics and runs the service.

The deployment infrastructure is a utility operated by a public cloud provider. It typically uses either containers or virtual machines to isolate the services.
E.g. AWS Lambda, Google Cloud Functions, Azure functions

Benefits:

- Eliminates the need to spend time on the heavy lifting of managing low-level infrastructure.
- Extremely elastic. It automatically scales your services to handle the load.
- You pay for each request rather than provisioning what might be under utilized VMs or containers.

Drawbacks:

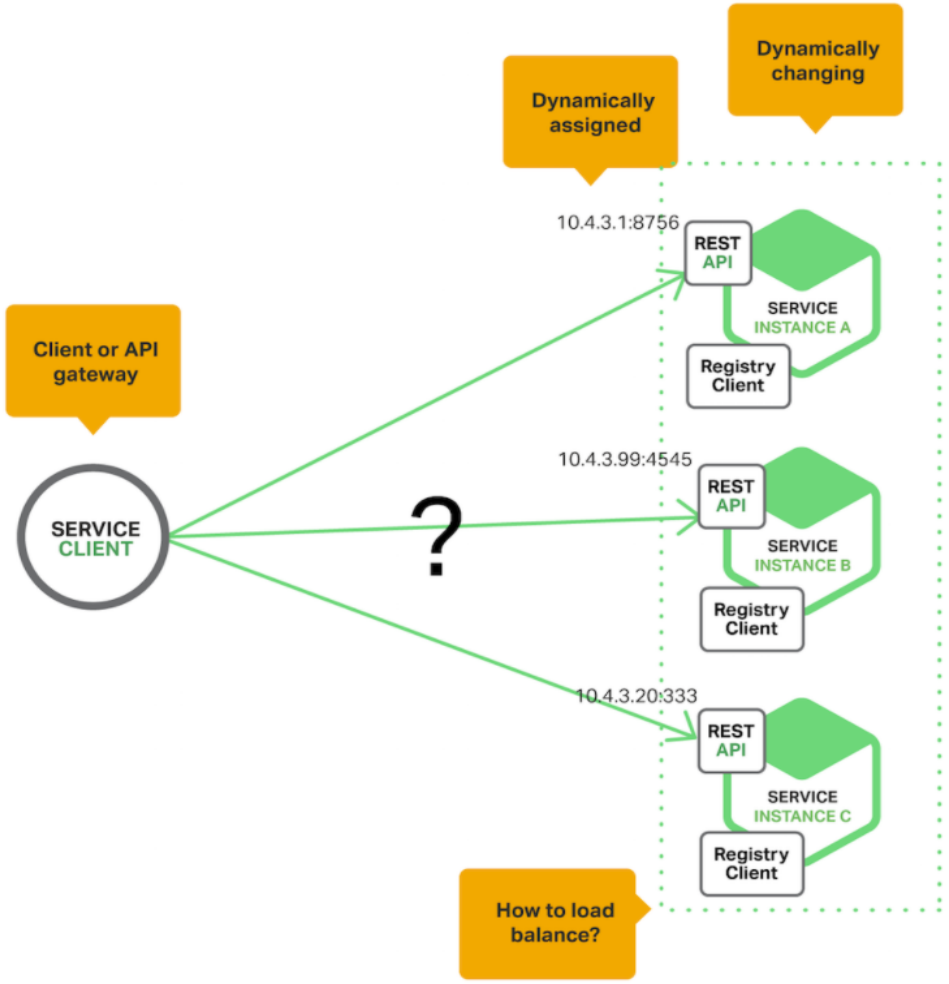
- Has far more constraints than a VM/Container-based infrastructure. E.g., language support.
- Only suitable for deploying stateless applications, not for long running stateful or message broker.
- Risk of high latency - the time it takes for the infrastructure to provision an instance and for the function to initialize might result in significant latency.
- Can only react to increases in load, cannot proactively pre-provision capacity.

Pattern - Categories

- **Data Management**
- **Communication**
- **Deployment**
- **Discovery**
- Reliability
- Observability
- Testing

Service Discovery

- Service instances have **dynamically assigned network locations**. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, the client code needs to use a more elaborate **service discovery mechanism**.



Service Registry

- Key of **service discovery**
  - Database containing the network locations of service instances.
  - Needs to be highly available and up to date.
  - Clients can cache network locations obtained from the service registry -- that information eventually becomes out of date
  - A service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

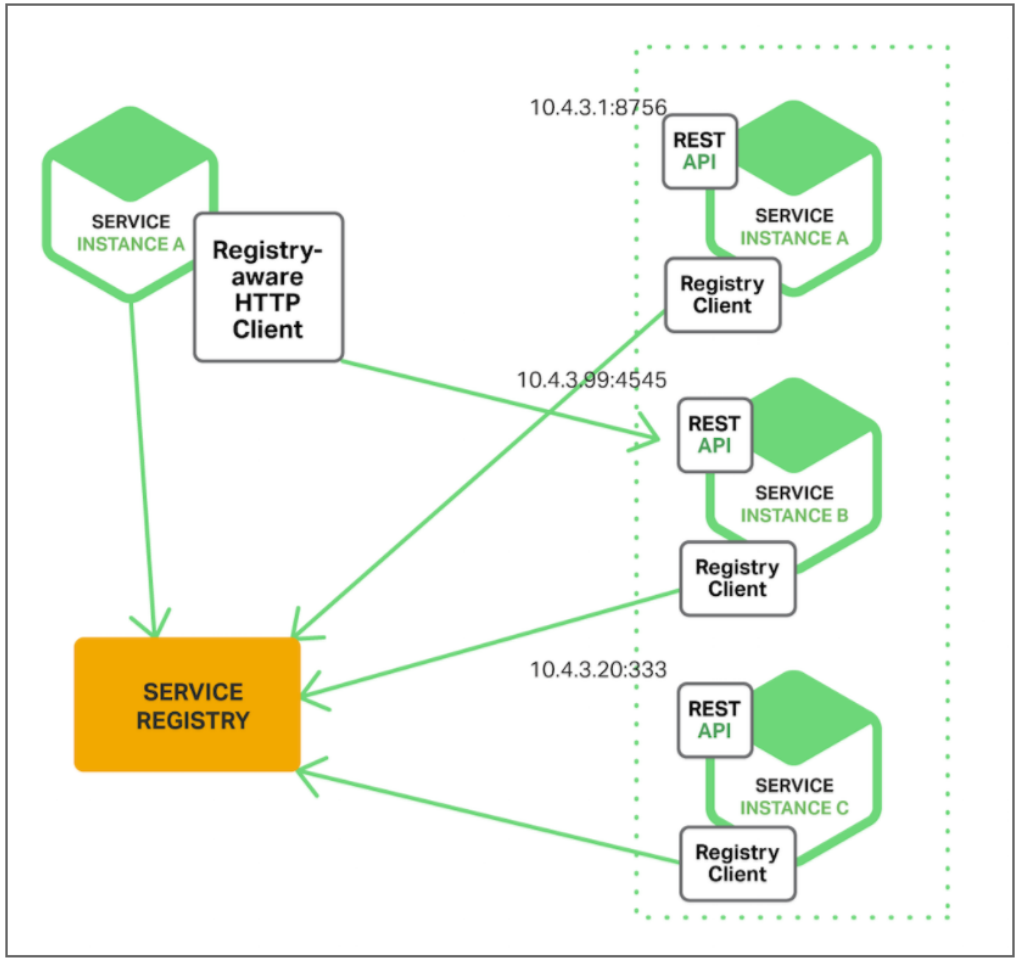
  E.g. Netflix Eureka, consul, Zookeeper, etcd

  https://github.com/hashicorp/consul-template -- dynamically reconfigure NGINX nginx.conf

Client-side Service Discovery Pattern

The client queries a **service registry**, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

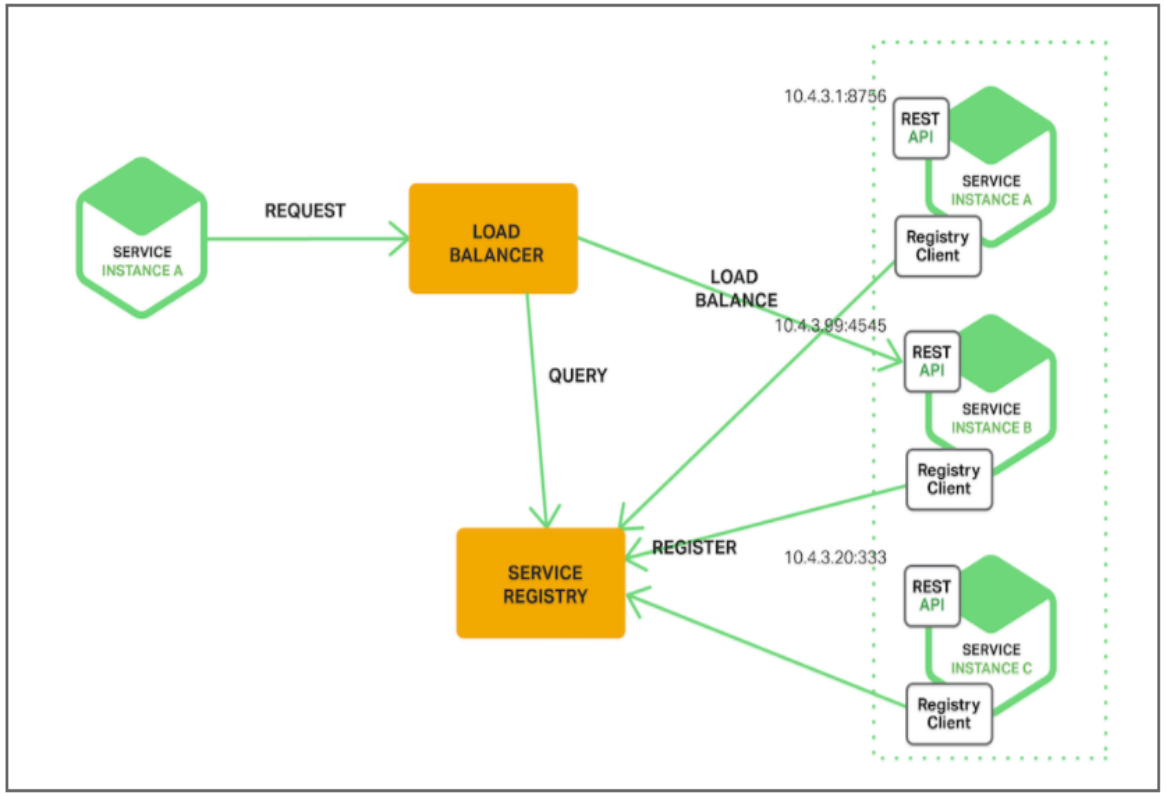
E.g. Netflix OSS



Server-side Service Discovery Pattern

The client makes a request to a service via a load balancer. The load balancer queries the **service registry** and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

E.g. Kubernetes, Mesos, AWS Elastic LoadBalancer



API Gateway

Single entry point for all clients. The API gateway handles requests in one of two ways:

- Proxied/routed to the appropriate service
- Fanned out to multiple service

The API gateway can expose a different API for each client. E.g., **Netflix API** gateway runs client-specific adapter code

Benefits:

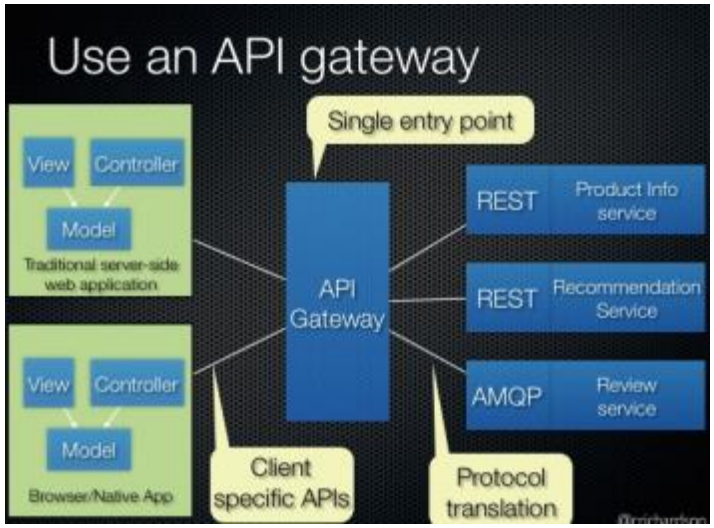
- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client

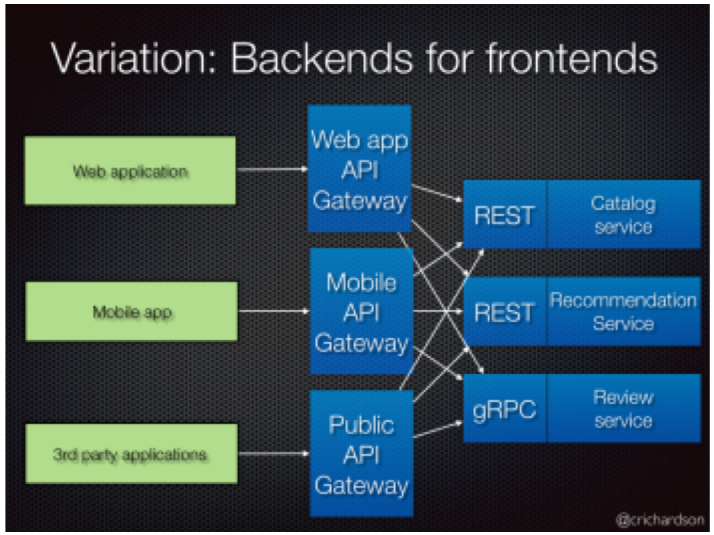
Drawbacks:

Increased complexity - another moving part to develop, deployed and managed

Increased response time due to the additional network hop through the API gateway.

E.g. Netty





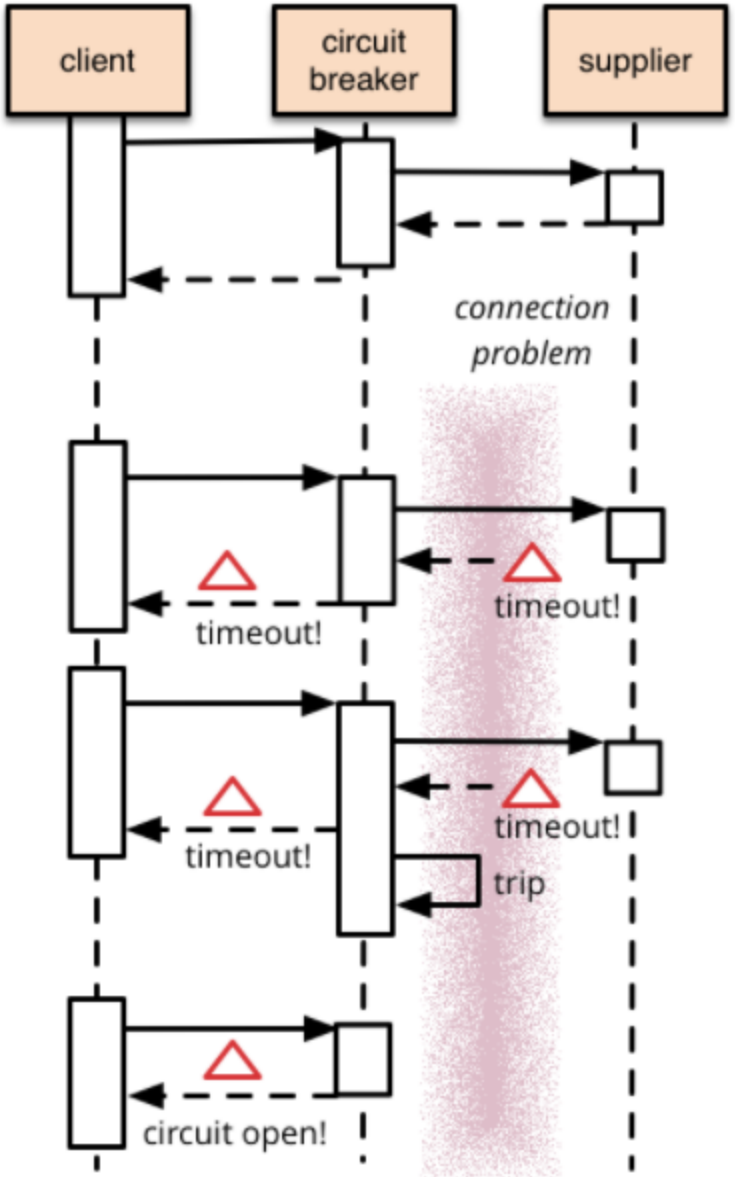
Pattern - Categories

- **Data Management**
- **Communication**
- **Deployment**
- **Discovery**
- **Reliability**
- Observability
- Testing

Circuit Breaker

Reason:
Remote calls can fail, reach timeout due to unresponsive supplier.
Increased number of calls may lead to cascading failures across multiple systems

What is it:
Wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all.



Pattern - Categories

- **Data Management**
- **Communication**
- **Deployment**
- **Discovery**
- **Reliability**
- **Observability**
- Testing

Observability Patterns

- Application Metrics
- Health check API
- Distributed Tracing
- Exception Tracking
- Log aggregation
- Audit Logging

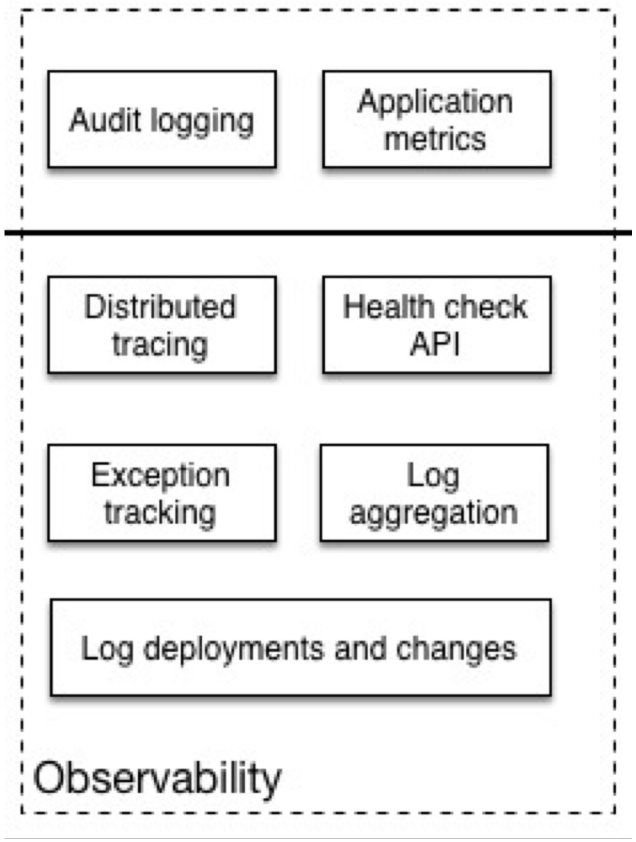
Observability (Metrics)

Health Check API

A health check client - a monitoring service - periodically invokes the endpoint to check the health of the service instance

- Ideally, you have a service that can ensure your service is active and accepting requests, a 'Health Check'

Health check examples included Consul, Kubernetes, and various libraries such as Sprint Boot, Gokit, etc.



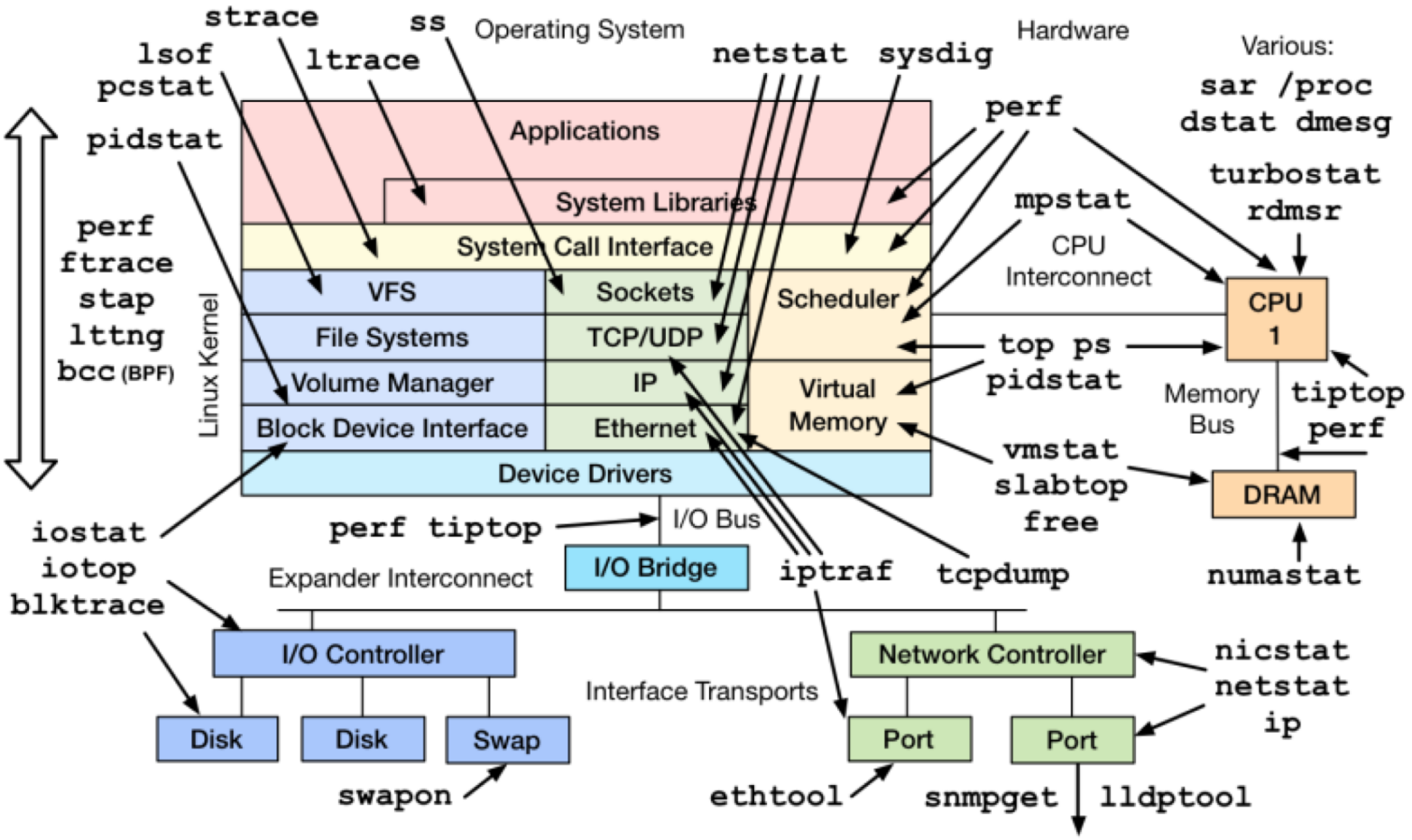
Application Metrics

- Need to understand the behavior of an application so performance and other problems can be proactively managed
- Typically come in two models:
  - Push – service pushes metrics (typically an agent) from the service
  - Pull – Service pulls metrics from the service (typically a dashboard)
- Examples include Prometheus, AWS Cloud Watch, Sysdig, etc

Observability Metrics

- **USE** method (Brendan Gregg)
- For every resource, check
  - **Utilization**: average time that the resource was busy servicing work
    - Example: One disk is running at 90%
  - **Saturation:** the degree to which the resource has extra work which it can't service, often queued
    - Example: The CPUs have an average run queue length of four
  - **Error count** (rate): the count of error events
    - Example: This network interface has had fifty late collisions
- Useful for resources such as queues, CPU's, memory, interconnects, etc.

Linux Performance Observability Tools



http://www.brendangregg.com/linuxperf.html 2017

Observability Metrics

RED method (Tom Wilkie)

For every service, check that

- **Request count** (rate)
- **Error count** (rate)
- **Duration**

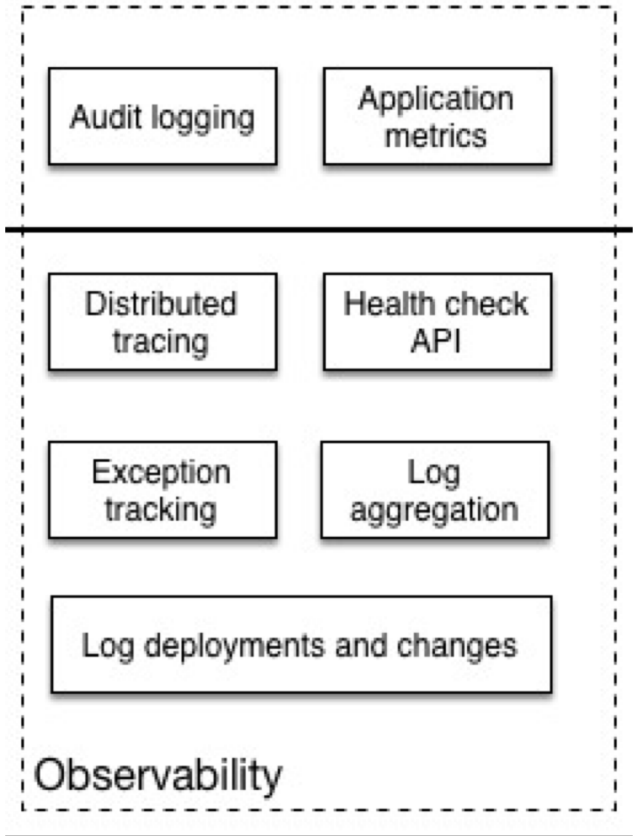
Note:

Observability = Logging + Monitoring + Tracing + Visualization

All good instrumentation libraries (Prometheus, Graphite, etc.) have at least three main primitives

- Counter: records events that happen such as incoming requests, bad requests, errors, etc.
- Gauge: records things that fluctuate over time such as the size of a thread pool
- Histogram: records observations of scalar quantities of events such as request durations

Observability (Tracing)



Distributed Tracing

- It becomes quite important to understand what broke and why

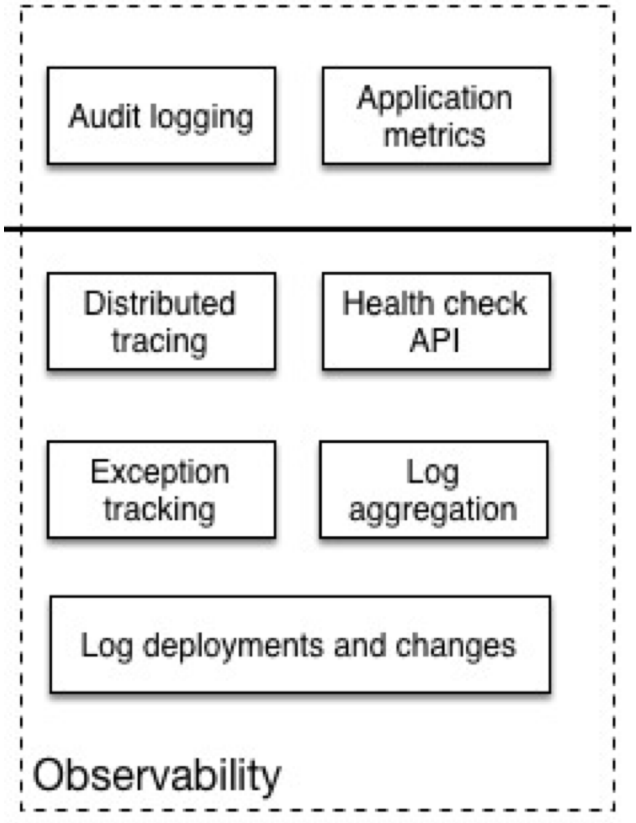
Common Distributed Tracing functionality

- Assign each external request a unique ID
- Pass ID to all services involved with handling the request
- Include id in all log messages
- Record information (start/end time) about requests and operations performed

Observability (Exceptions and Logging)

Exception Tracking

- It is crucial to save errors and the corresponding stack trace
- Ideally, exceptions are de-duplicated and recorded for further aggregation and investigation



Log deployments and changes

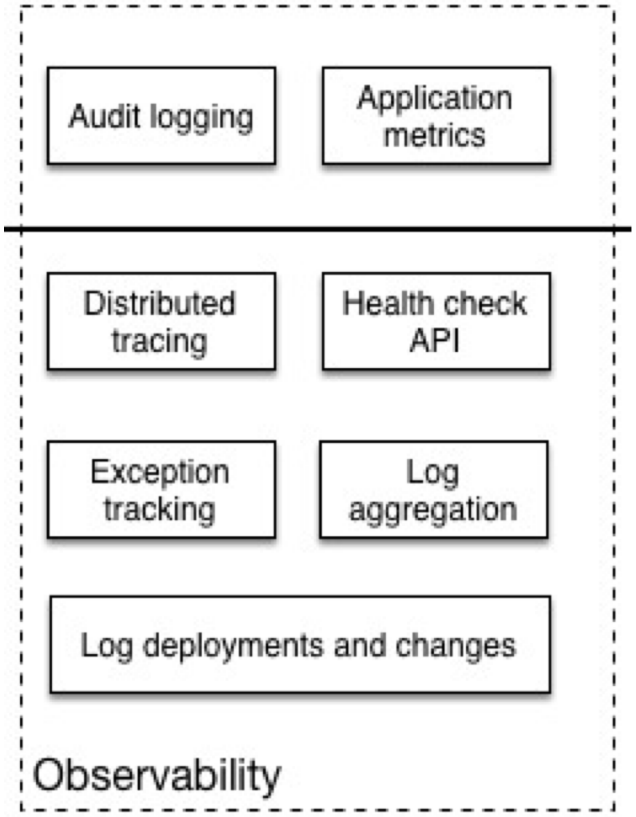
- Track changes as they are opportunities for failure
- For example, tracking deployments and changes so they can be easily correlated with issues later for faster resolution

*Etsy graphing error rates against new code deployments

Observability (Aggregation and Audit)

Log Aggregation

- Requests span multiple instances over multiple machines. It is important to gather information in a standardized format
- Log files contain errors, warnings, and debug information
- It is important to aggregate, search and analyze such logs
- Popular solutions for this include the ELK stack (Elastic Search, Logstash and Kibana) and AWS Cloud Watch



Audit Logging

- It is vital to know actions a user has performed.
- For security and compliance, this is often a must have
- Average time to detect intrusions is 98 days for financial services, 197 days for retail, it should/can be much lower

Logging best practices

- Log output as a set of JSON key/value pairs instead of pure text
- Easier for computer to test and aggregate
- Easier to make rules and query
- Comprehensive

- Do this:
  - “Timestamp: 2017-09-01 1:25”, ”caller”:”main.go:5”, “transport”:”HTTP”,”addr”:”8080”
- Not this
  - “2017-09-01 1:25 main.go 5 HTTP 8080”

Pattern - Categories

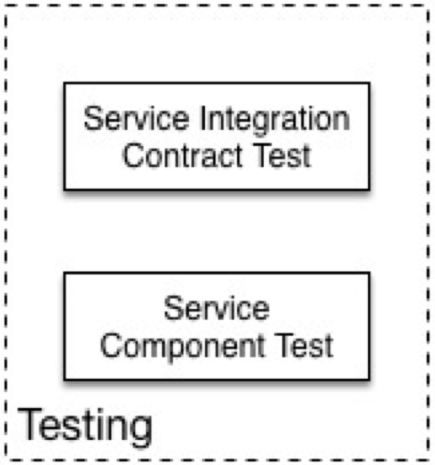
- Data Management
- Communication
- Deployment
- Discovery
- Observability
- Testing

Service Integration/Component Testing

Service Integration Contract Tests

- End-to-end testing is slow and expensive
- Test the APIs (functionality) provided by services

- **Black box testing**



Service Component Tests

- Test the components of the service
- A test suite that tests a service in isolation using test doubles for any services that it invokes.

- **White box testing**

End-To-End tests Vs Unit Test

End-to-end tests: time taking and expensive

- The dev team might have to spend weeks finding all the issues breaking the end-to-end test
- They would be better served with a large base of unit test that find bugs quickly

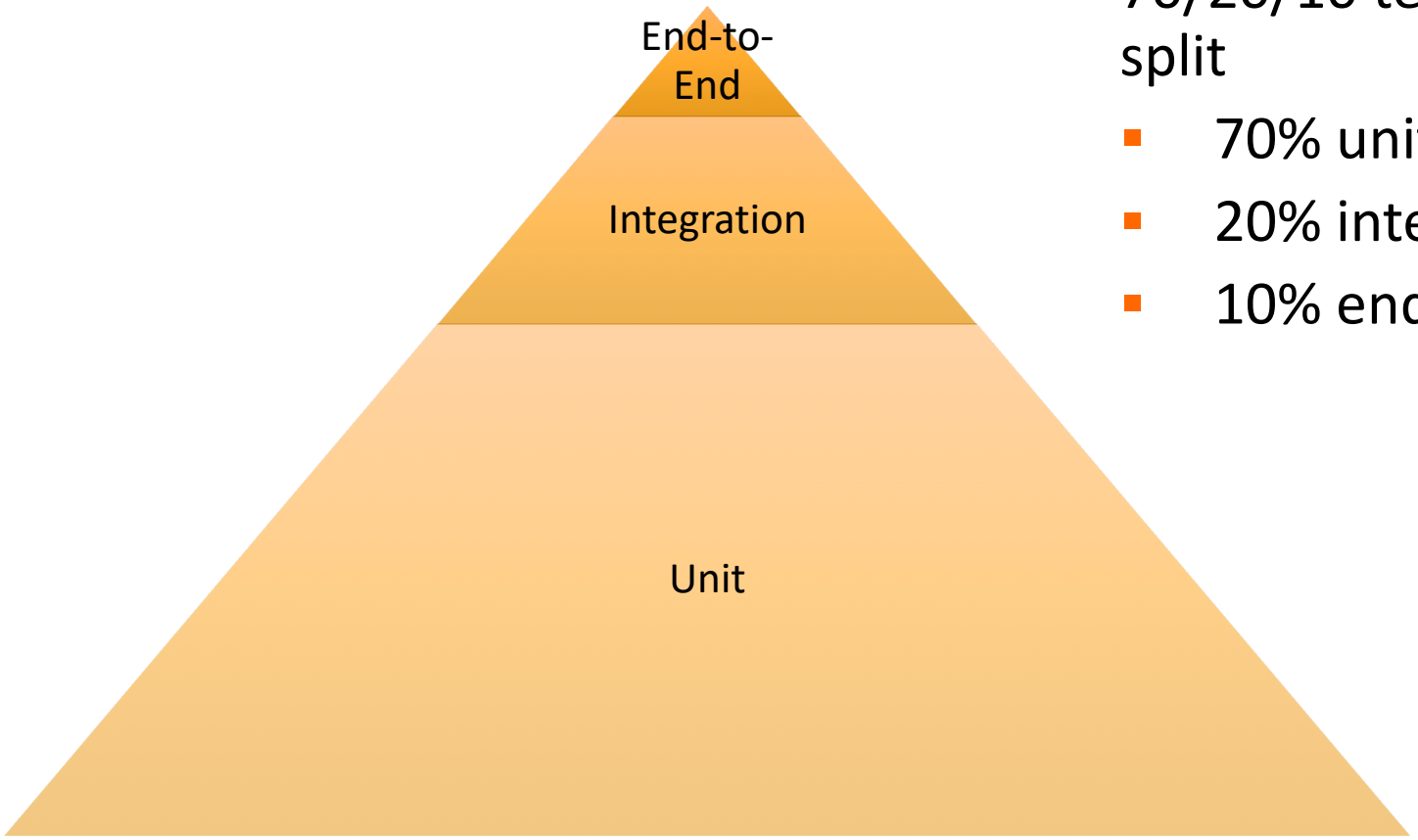
- Slow
- No isolated failures
- Simulates a real user

Unit Tests: Fast and cheap

- Unit tests should form the majority of tests as they quickly identify bugs and represent a fast feedback loop
- When a Unit test fails, the function and area that fails is much narrower in scope than broad end-to-end tests

- Fast
- Isolated Failures
- But do not simulates a real user

Testing Pyramid



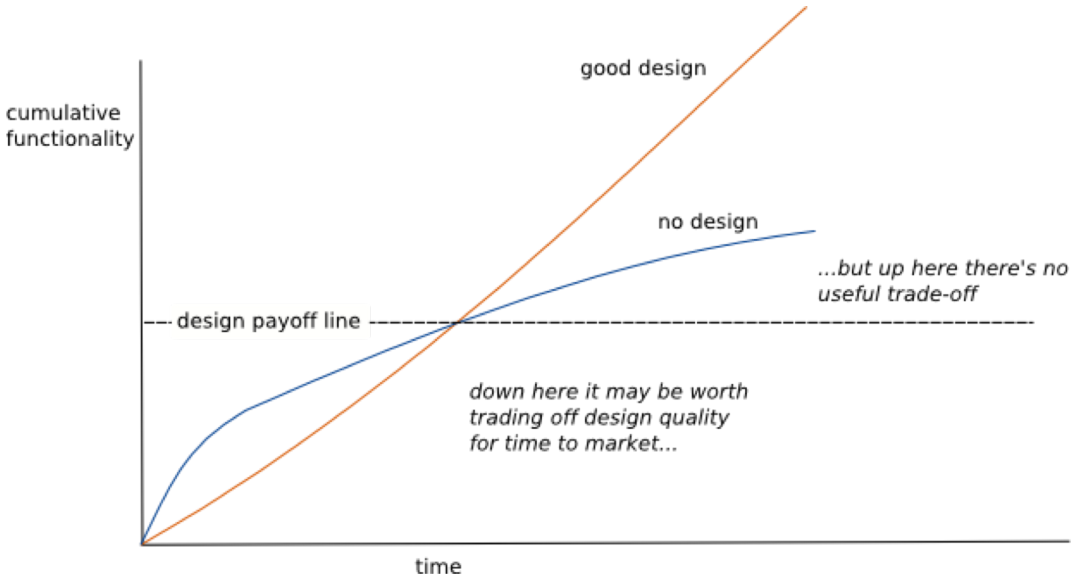
Google recommends a 70/20/10 test composition split

- 70% unit tests
- 20% integration tests
- 10% end-to-end tests

Technical Debt Payback

Addressing **Technical Debt** makes an **assumption** that the team will get a **payoff** on their **investment**

- Have more unit tests, helps in continuous deployment, helps in fast tracing of errors and building a resilient product – This reduces technical debt
- The interest from technical debt can be used to reduce even more technical debt (compounding)

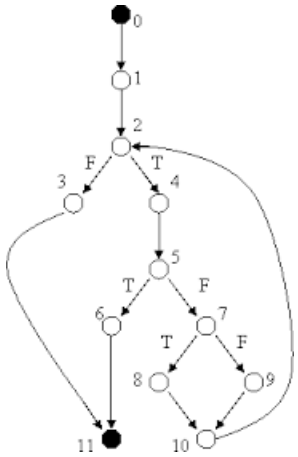


Static Code Metrics

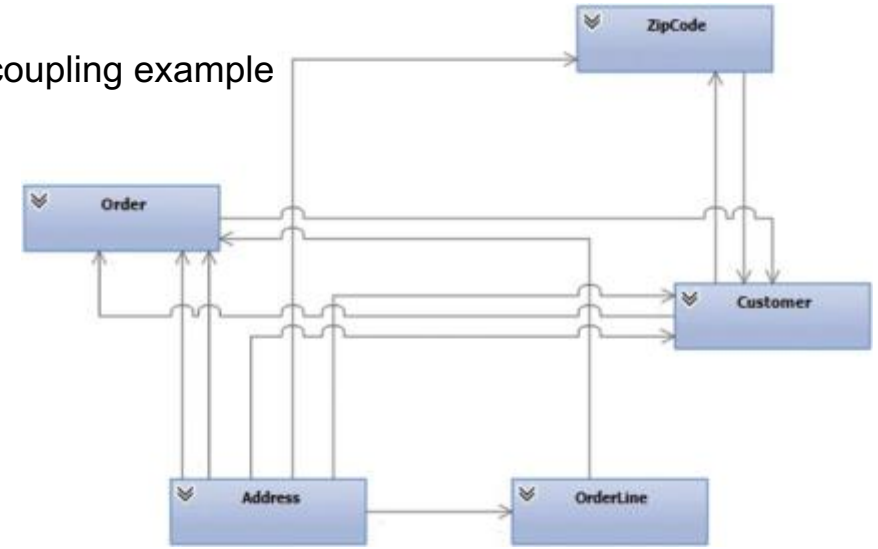
Coding Techniques to Improve Architecture Agility

- Code Complexity – recommend about 20 lines per method
- Cyclomatic Complexity (# of linearly independent paths) – recommend keep it low
- Coupling (measure of how many classes a single class uses), keep it low
- Inheritance, no deeper than 6.

Cyclomatic Complexity



Class coupling example



EXTERNAL BENCHMARKS (Velocity & Stability Metrics)

2016 IT Performance by Cluster

	High IT Performers	Medium IT Performers	Low IT Performers
Deployment frequency *For the primary application or service you work on, how often does your organization deploy code?*	On demand (multiple deploys per day)	Between once per week and once per month	Between once per month and once every 6 months
Lead time for changes *For the primary application or service you work on, what is your lead time for changes (i.e., how long does it take to go from code commit to code successfully running in production)?*	Less than one hour	Between one week and one month	Between one month and 6 months
Mean time to recover (MTTR) *For the primary application or service you work on, how long does it generally take to restore service when a service incident occurs (e.g., unplanned outage, service impairment)?*	Less than one hour	Less than one day	Less than one day*
Change failure rate *For the primary application or service you work on, what percentage of the changes either result in degraded service or subsequently require remediation (e.g., lead to service impairment, service outage, require a hotfix, rollback, fix forward, patch)?*	0-15%	31-45%	16-30%

* Low performers were lower on average (at a statistically significant level), but had the same median as the medium performers.

*** Source: Puppet Labs Report**

Competitive Advantage

High performing teams are breaking away from the pack and the status quo of even three years ago is a dangerous assumption currently.

A team which had state of the

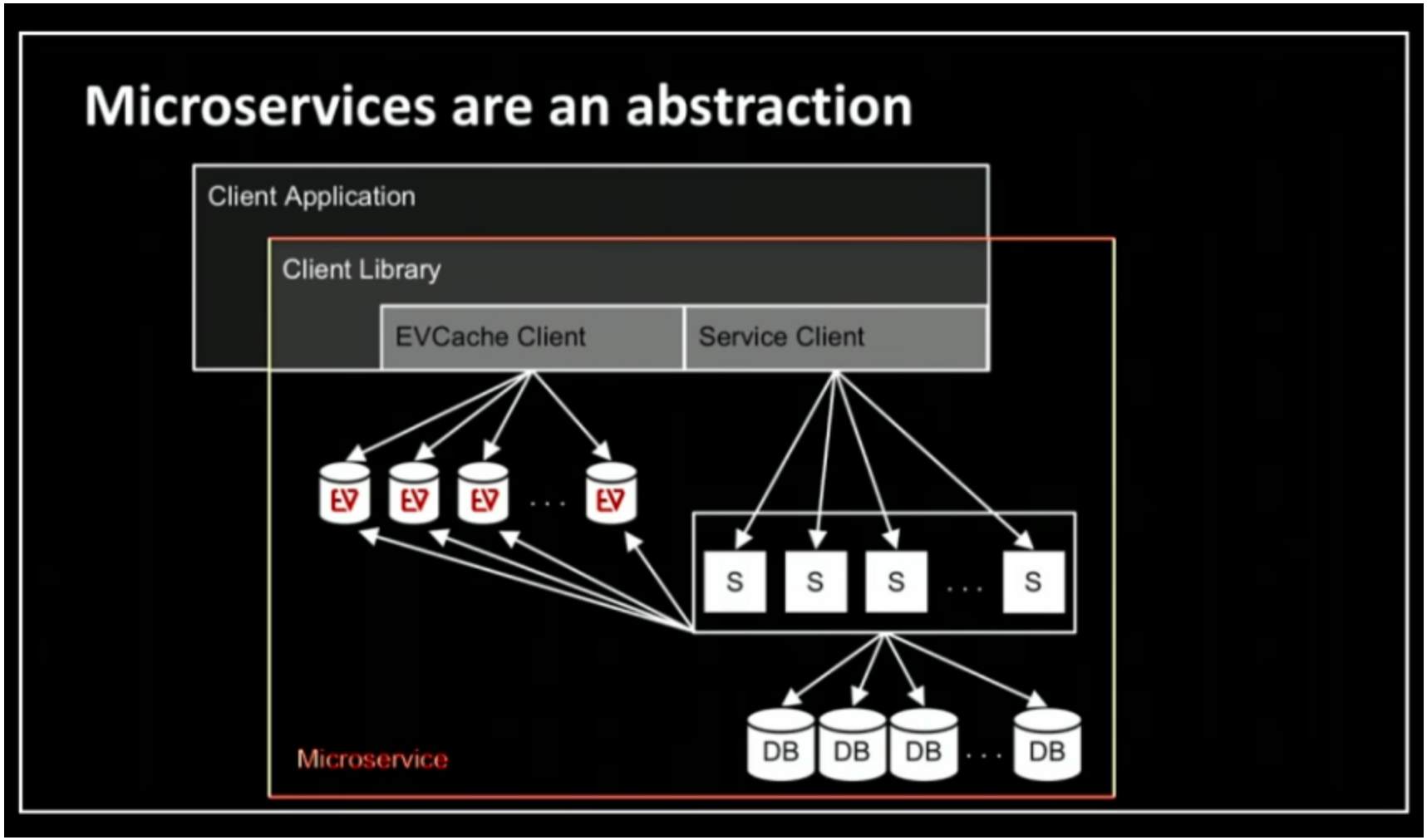
Microservices - Definition of Done

- Can be deployed, replaced, and upgraded independently (Services are autonomous, loosely coupled)
- Published service interface (Services are discoverable)
- Bounded Context (Specific responsibility enforced by explicit boundaries)
- Externally accessible API
  - No other forms of communication allowed, no tickets, direct linking, reads of data stores, no back doors, only service interface calls over the network

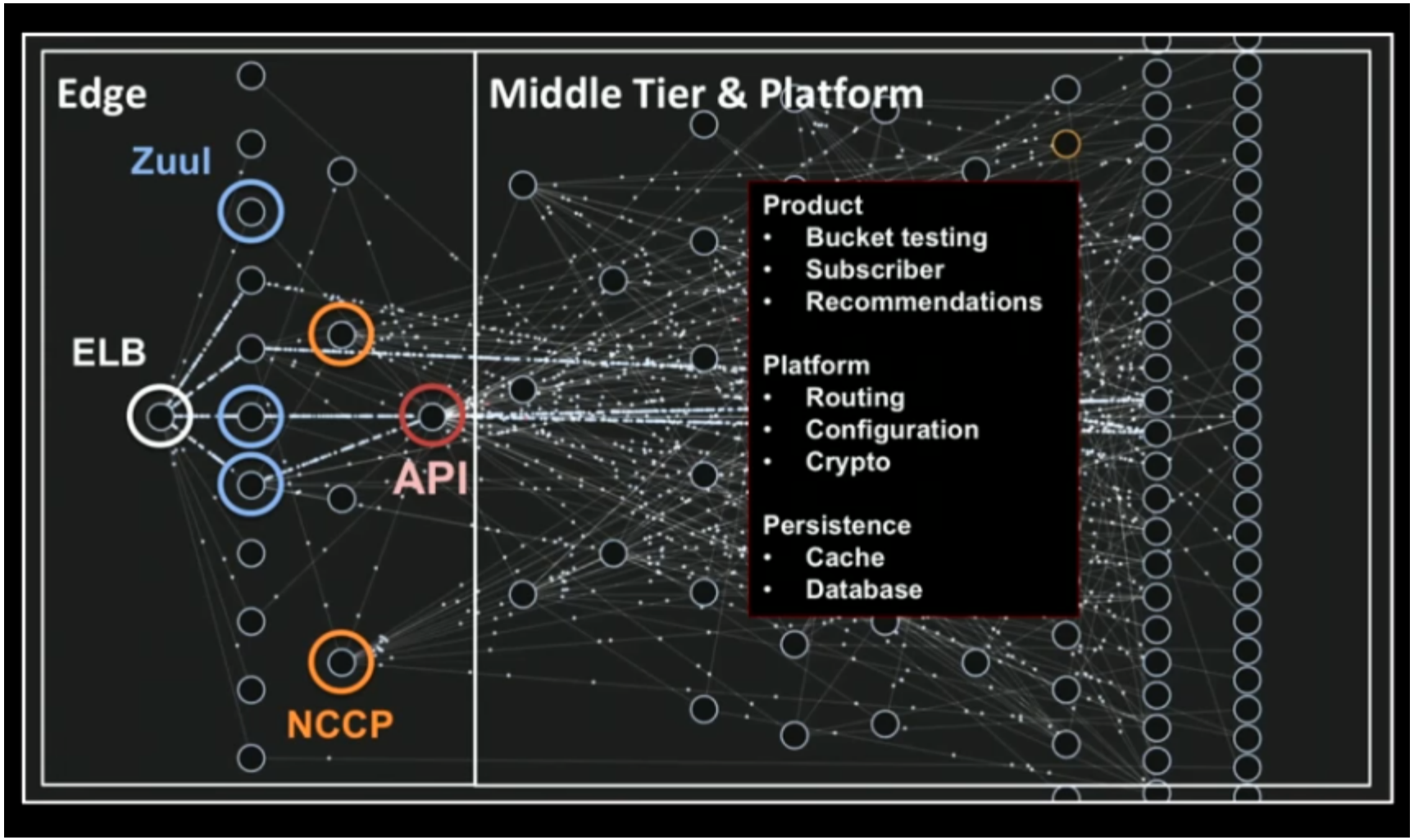
- **Microservices in Netflix**
  - **By Josh Evans (Director of Operational Engg.)**
  - **At Qcon, San Francisco, 2016**

https://www.youtube.com/watch?v=CZ3wIuvmHeM

Microservices in Netflix



Microservices in Netflix



Microservices in Netflix

- ELB
- Dependencies
- Eventual consistency, CAP
- Circuit breakers
- Workload partitioning
- Auto-scaling
- Multi-region failure strategy

End of Microservices concept – Part II