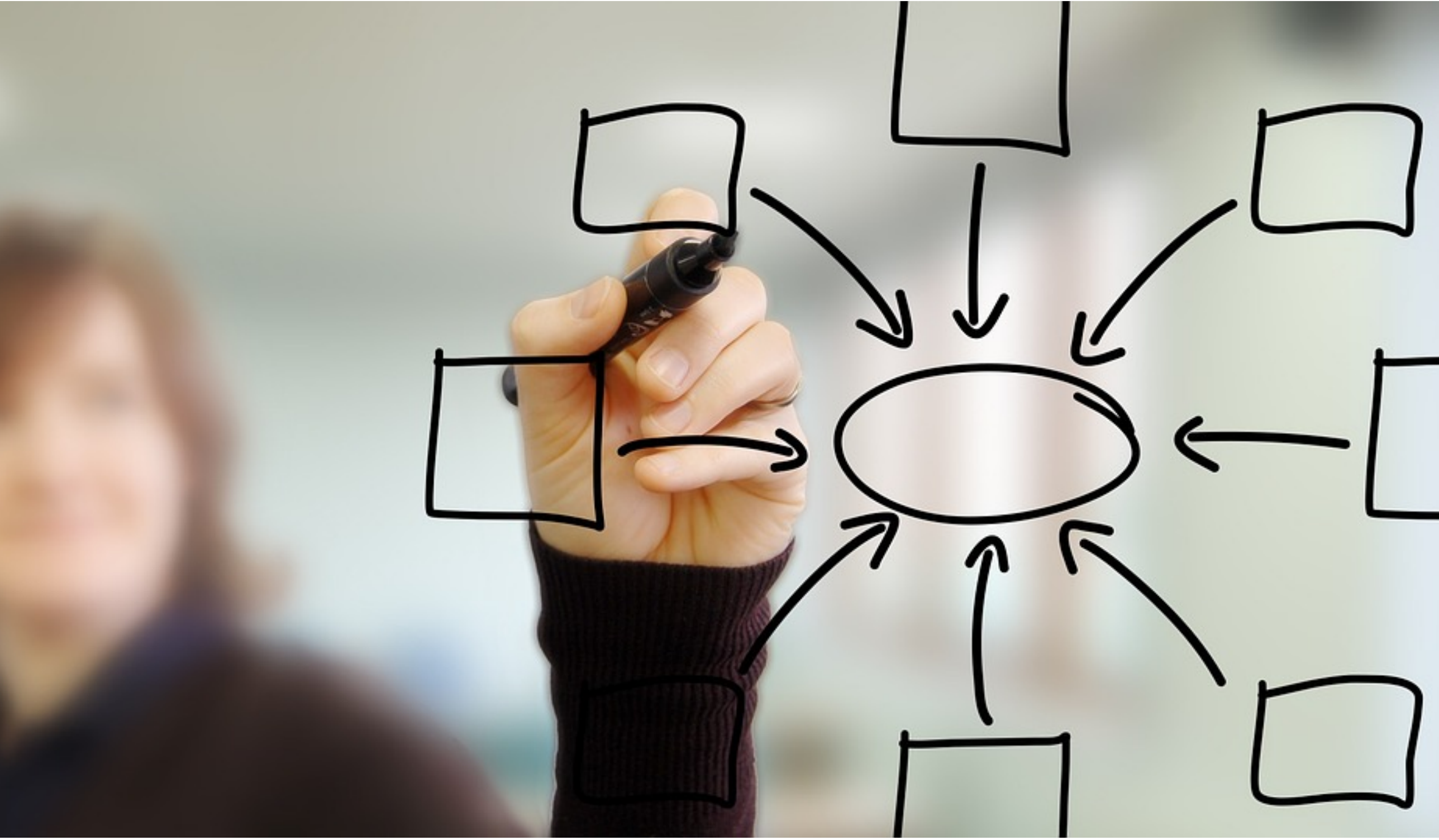


Microservices Architectures

Sébastien Mosser

01.10.2018

pictures: sxc.hu

@MELANIECEBULA / SEPT 2017 / FUTURESTACK

From Monolith to Microservices:

How to Scale Your Architecture

ARAF KARSH HAMID

Co-Founder / CTO

MetaMagic Global Inc., NJ, USA

@arafkarsh

arafkarsh

$$\sum_{b=1}^{n} f(b) = \sqrt[3]{desi^3r^2e}\ 3D$$

Microservices Architecture

World Agile Testing & Automation

Novotel Techpark, Bangalore, June 22, 2018

https://1point21gws.com/testingsummit/bangalore/

ARAF KARSH HAMID

Co-Founder / CTO

MetaMagic Global Inc., NJ, USA

@arafkarsh

arafkarsh

Micro Services Architecture

Part 1 : Infrastructure Comparison &

Design Styles (DDD, Event Sourcing / CQRS, Functional Reactive Programming)

Araf Karsh Hamid – Co Founder / CTO, MetaMagic Global Inc., NJ, USA

*A **Micro Service** will have its own Code Pipeline for build and deployment functionalities and it's scope will be defined by the Bounded Context focusing on the Business Capabilities and the interoperability between Micro Services will be achieved using message based communication.*

Domain Driven Design

The AirBnB Case Study

“Monorail”

OUR MONOLITHIC RUBY ON RAILS APPLICATION

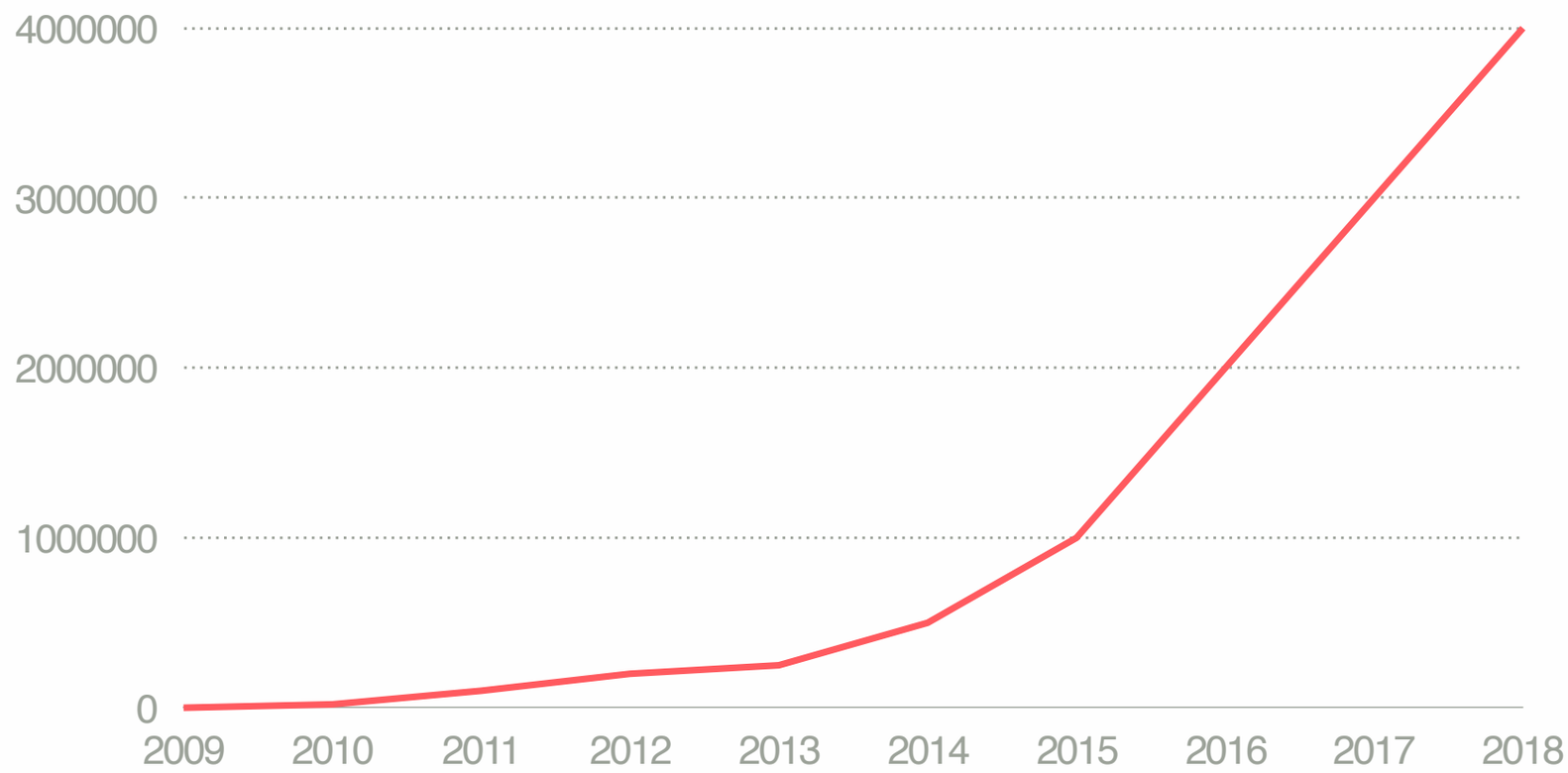
1 GitHub repo

1 SRE

Sysops

When it starts to become a problem

MONORAIL LOC



Democratic deploys

All developers are expected to “own” their features from implementation to production.

Each developer is expected to:

- deploy their changes
- monitor their changes
- roll back, abort, or revert if necessary

Scaling Deploy Process

- builds, tests, and deploys must be fast and reliable
- ping authors when they need to take action
- automatic (and fast) reverts
- merge queue
- automatic rollbacks for new exceptions

Why Not Microservices?

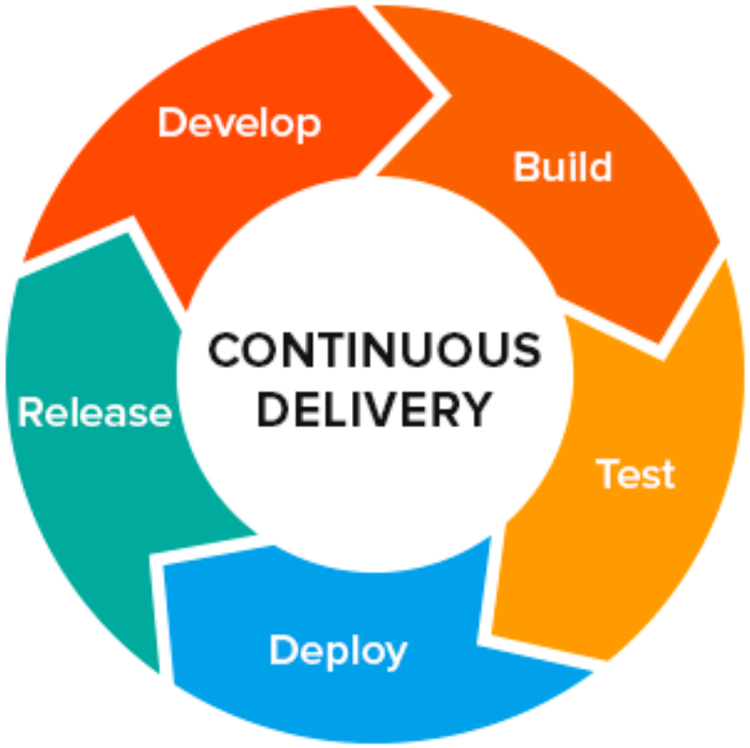
- Distributed systems are hard!
- Naming things is hard!
- Having to support different languages / frameworks

Why Microservices?

- Independent deployment
- Modular structure / Ownership
- Secure / Performant
- Upgrading dependencies

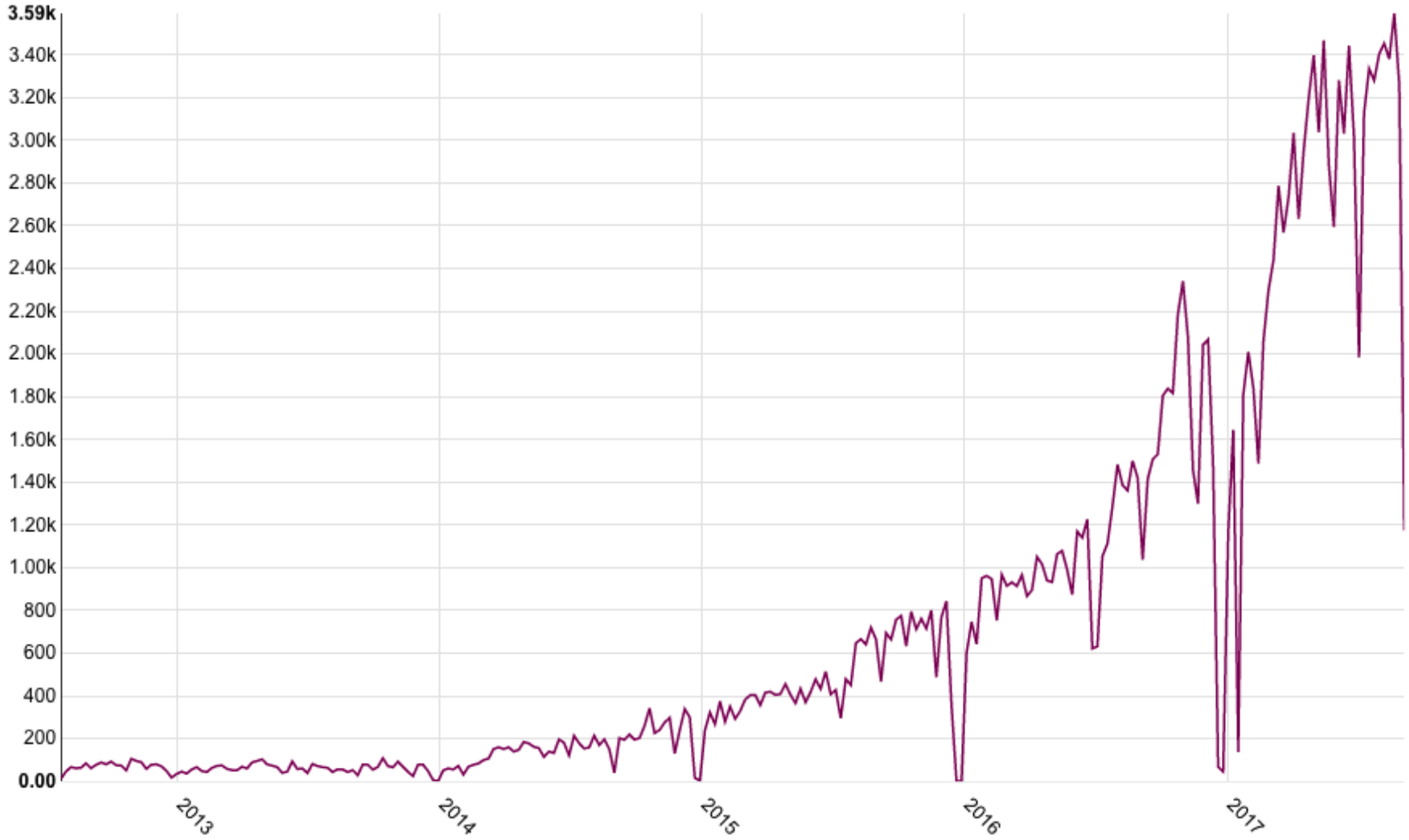
Why Microservices?

SCALING CONTINUOUS DELIVERY



Why Microservices?

Deploys per week (all apps, all environments)



Why Microservices?

75,000

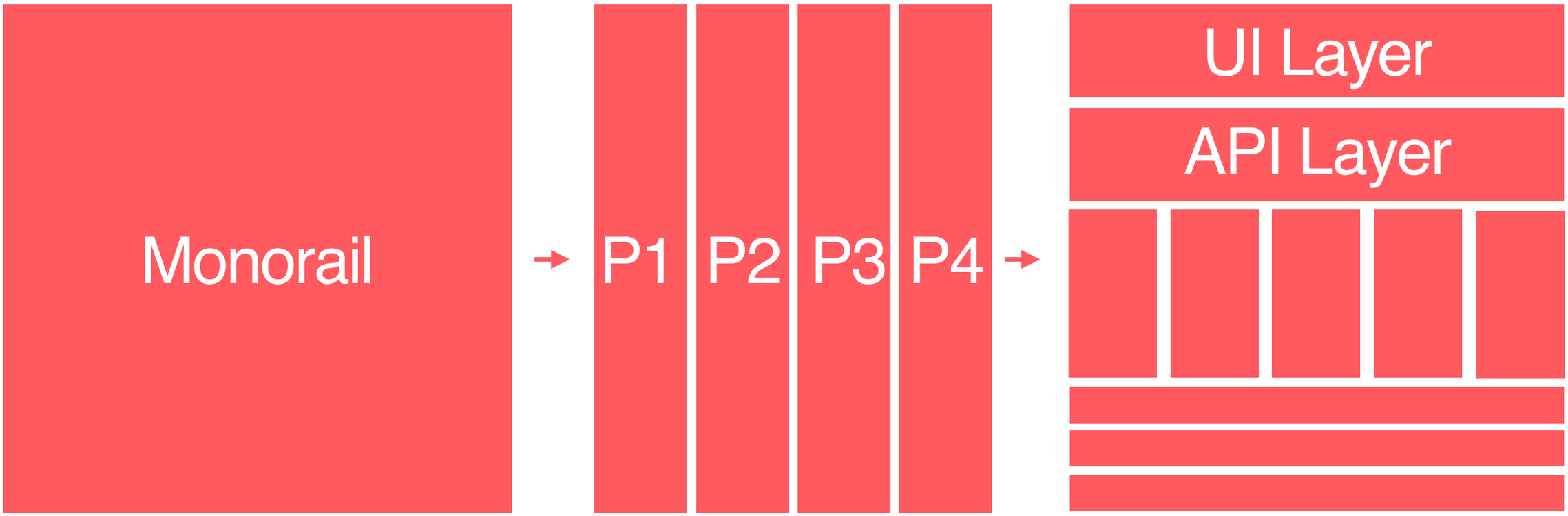
production deploys

per year

Services

900	300	4k
developers	services	deploys per week

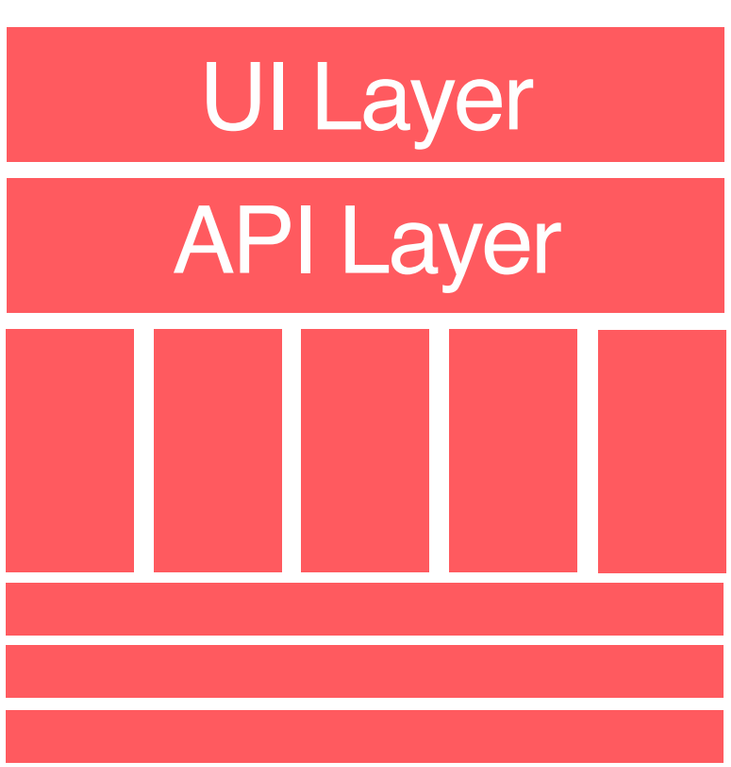
Splitting the Monolith

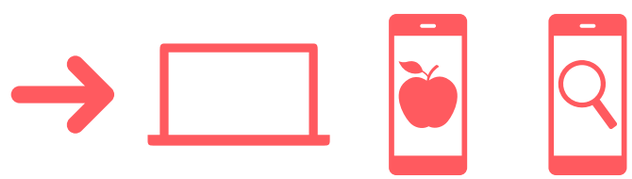


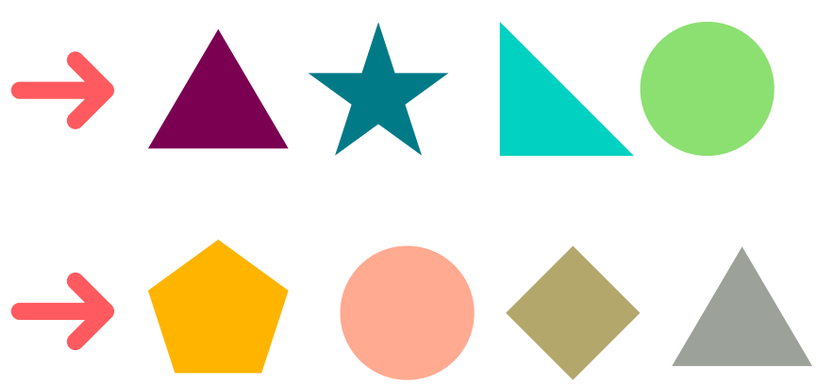
Vertical concerns

Horizontal concerns

Splitting the Monolith







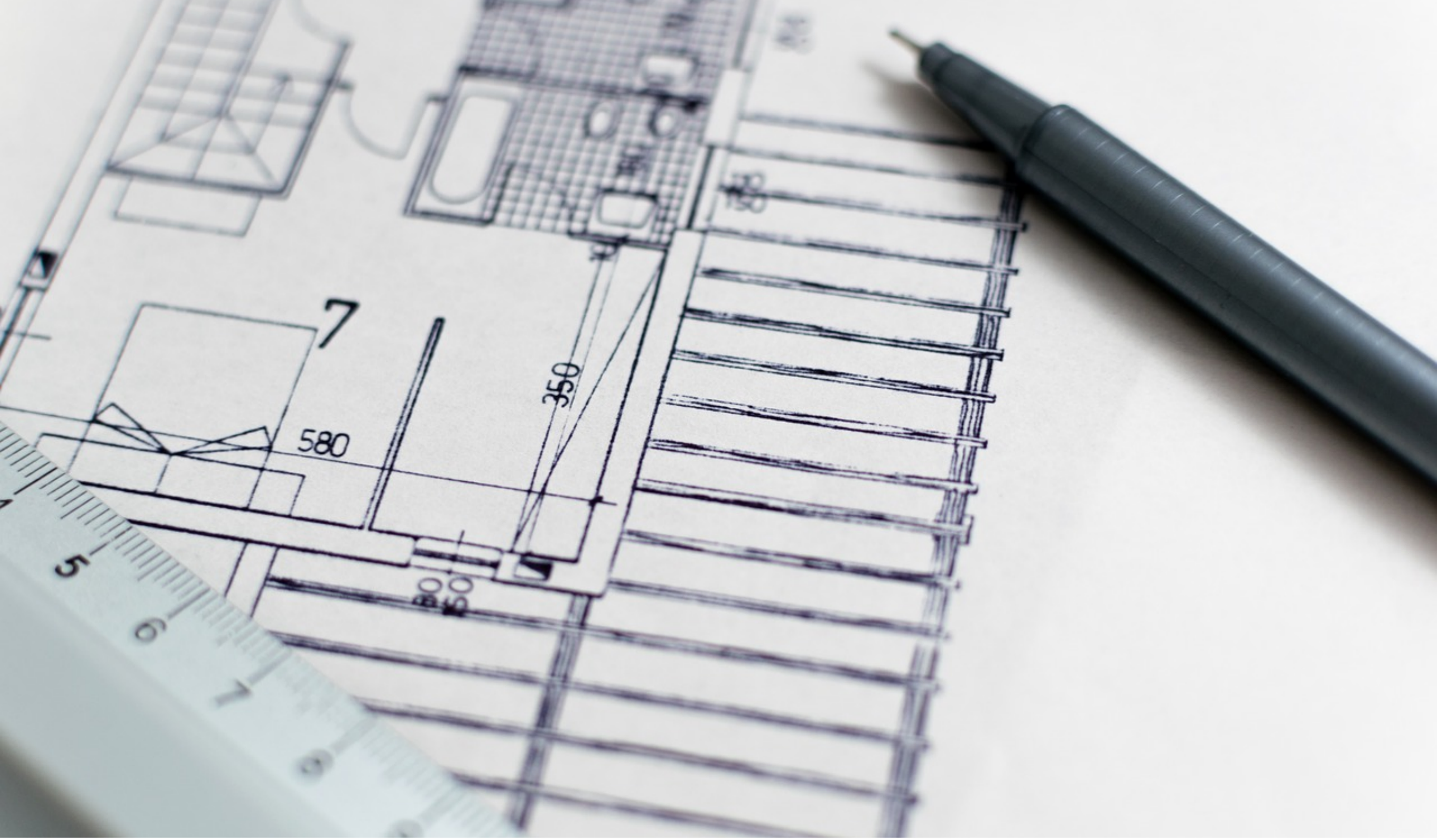
Vertical concerns

Horizontal concerns

The important part

10 TAKEAWAYS

1. Monolith first
2. Devops culture
3. Configuration as code
4. Monitoring and Alerting
5. Continuous delivery
6. Automate what you can
7. Your first service is going to be bad
8. Services own their own data
9. Break apart the Monolith
10. Standardize service creation



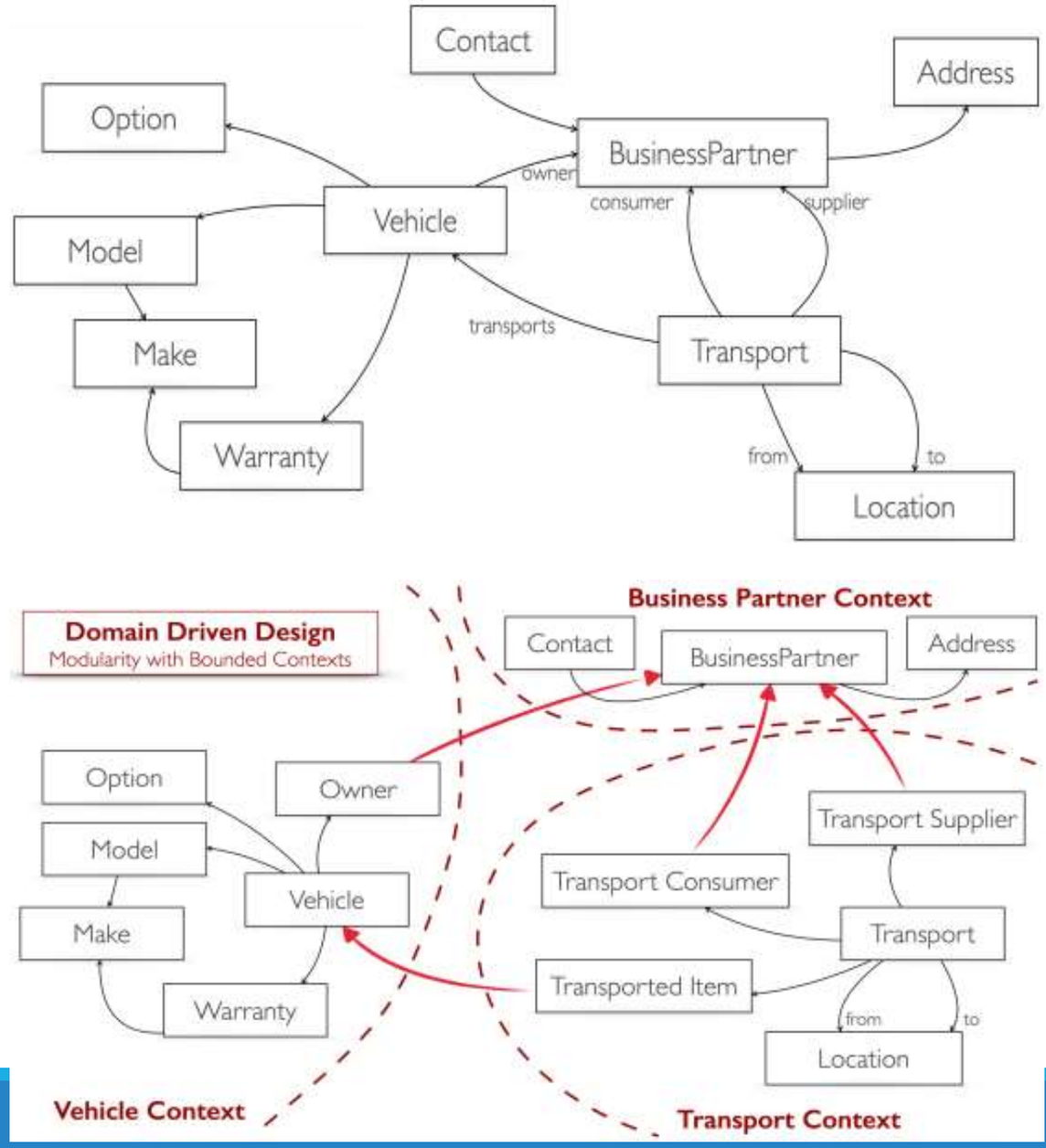
Elements of Domain-driven design

Understanding Requirement Analysis using DDD

Bounded Context

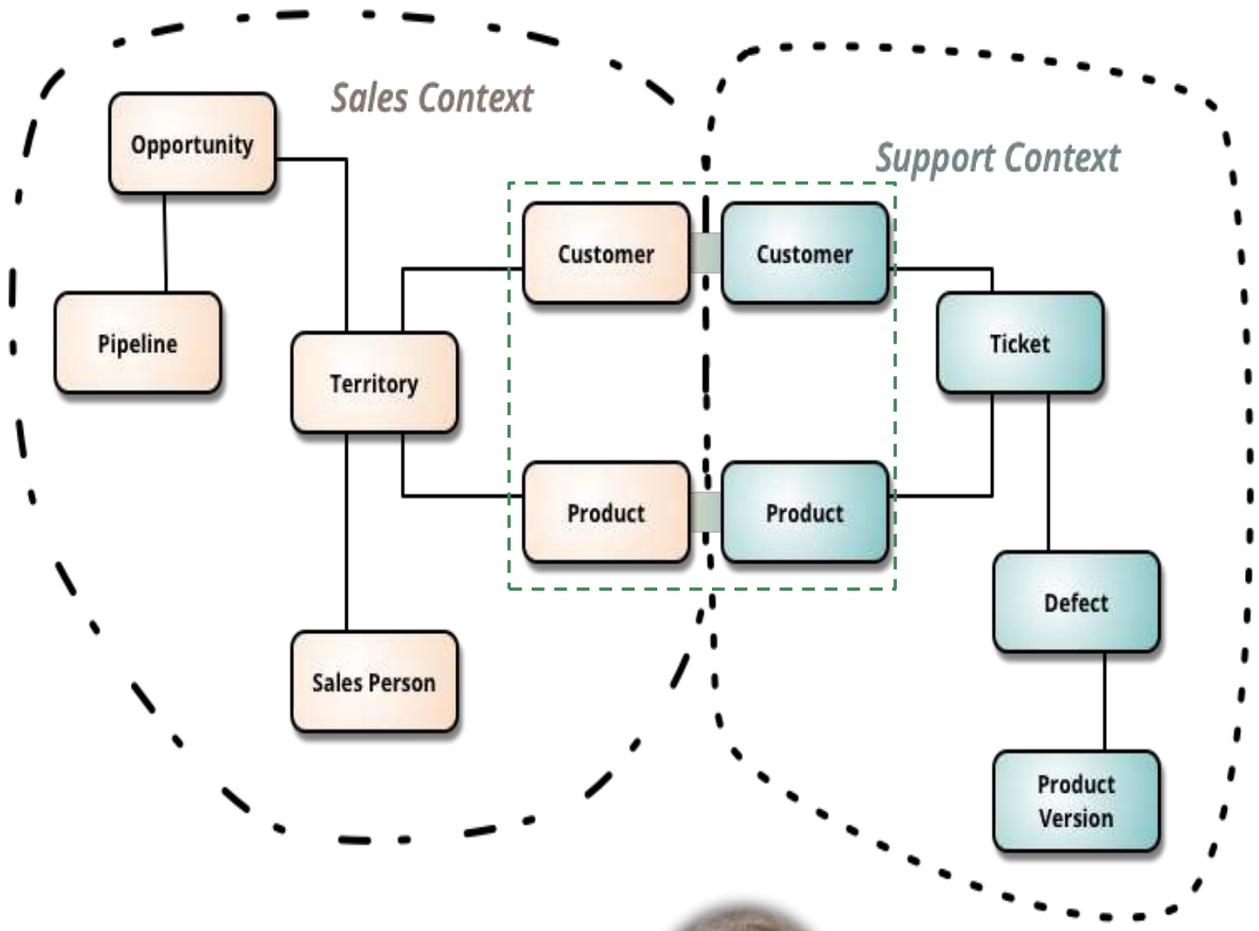
Areas of the domain treated independently

Discovered as you assess requirements and build language



DDD : Understanding Bounded Context

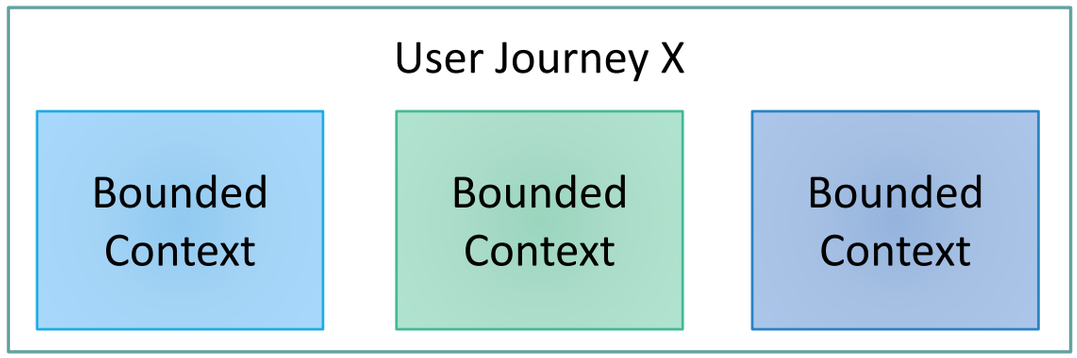
- DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.
- Bounded Contexts have both unrelated concepts
  - Such as a support ticket only existing in a customer support context
  - But also share concepts such as products and customers.
- Different contexts may have completely different models of common concepts with mechanisms to map between these polysemic concepts for integration.



Source: BoundedContext By Martin Fowler : http://martinfowler.com/bliki/BoundedContext.html

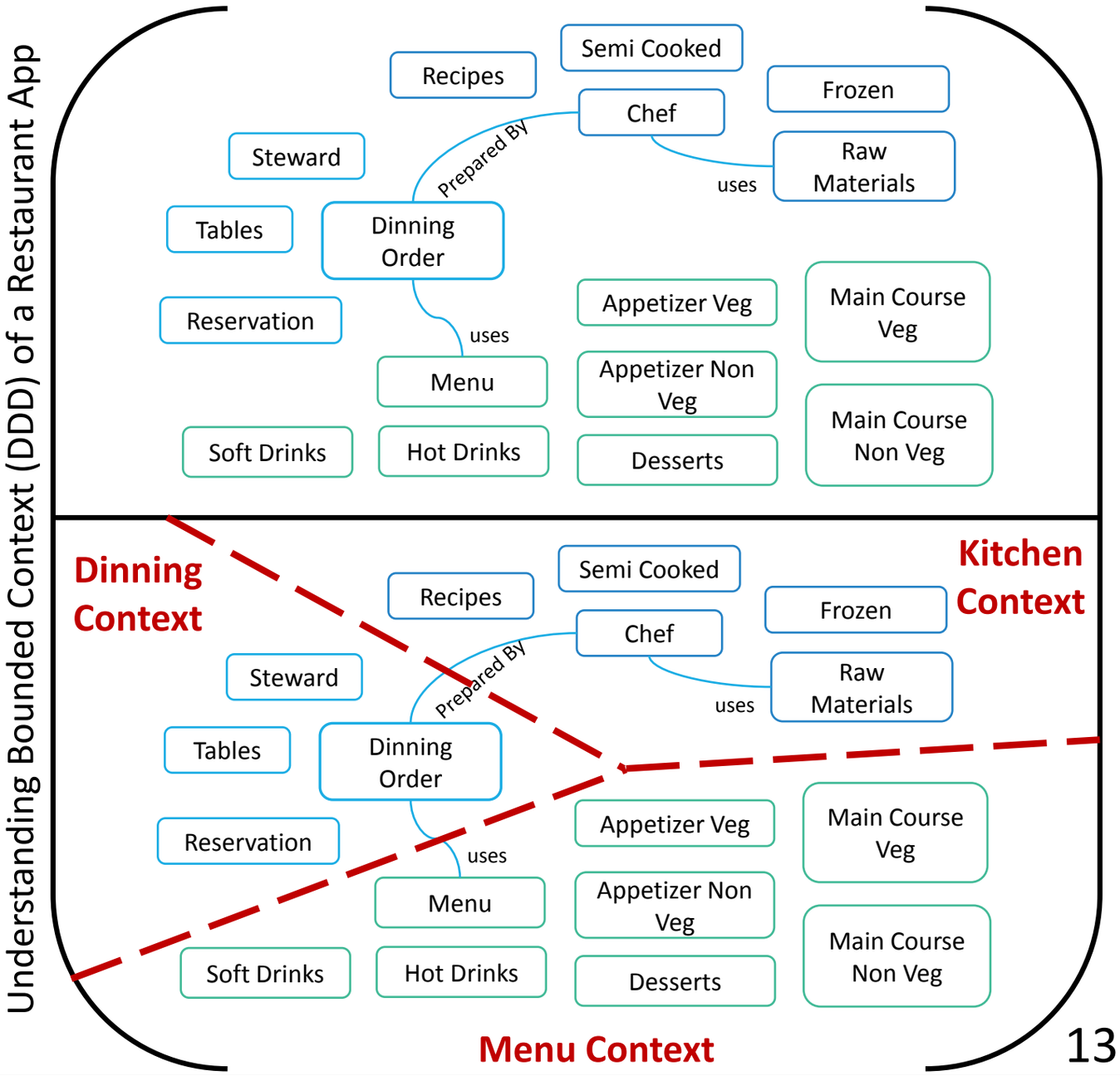
DDD: Bounded Context – Strategic Design

An App User's Journey can run across multiple Bounded Context / Micro Services.



Areas of the domain treated independently

Discovered as you assess requirements and build language



Source: Domain-Driven Design Reference by Eric Evans

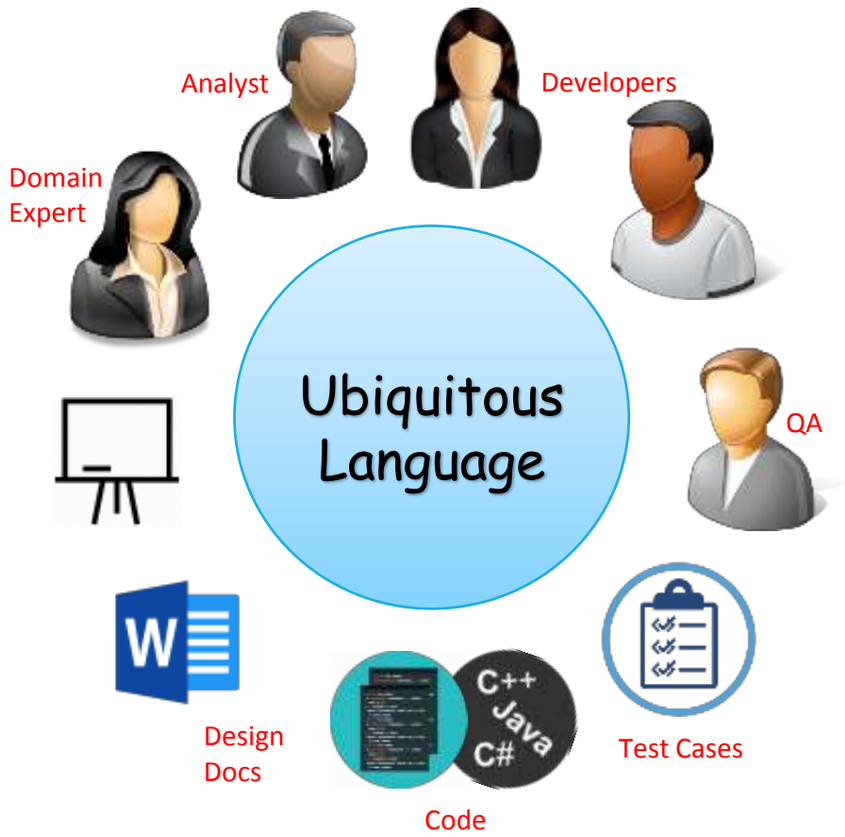
DDD: Ubiquitous Language: Strategic Design

Ubiquitous Language → Vocabulary shared by all involved parties → Used in all forms of spoken / written communication

Restaurant Context – Food Item :

Eg. Food Item (Navrathnakurma) can have different meaning or properties depends on the context.

- In the Menu Context it's a Veg Dish.
- In the Kitchen Context it's is recipe.
- And in the Dining Context it will have more info related to user feed back etc.



Role-Feature-Reason Matrix

As an Restaurant Owner
I want to know who my Customers are
So that I can serve them better

BDD Construct

Given	Customer John Doe exists
When	Customer orders food
Then	Assign customer preferences as Veg or Non Veg customer

BDD – Behavior Driven Development

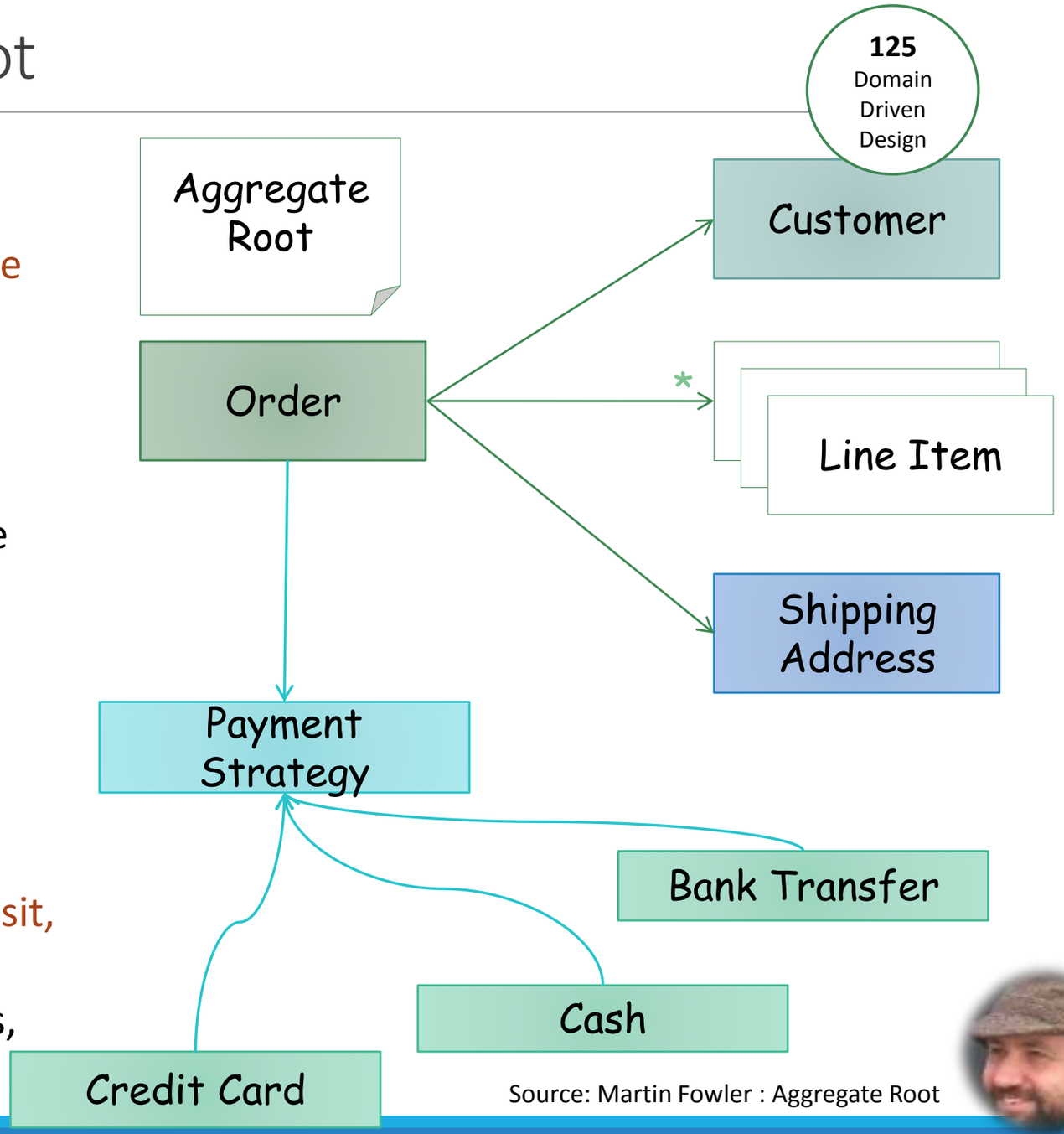
Understanding Aggregate Root

- An aggregate will have one of its component objects be the aggregate root. Any references from outside the aggregate should only go to the aggregate root. The root can thus ensure the integrity of the aggregate as a whole.

- Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates. Transactions should not cross aggregate boundaries.

- Aggregates are sometimes confused with collection classes (lists, maps, etc.).

- Aggregates are domain concepts (order, clinic visit, playlist), while collections are generic. An aggregate will often contain multiple collections, together with simple fields.

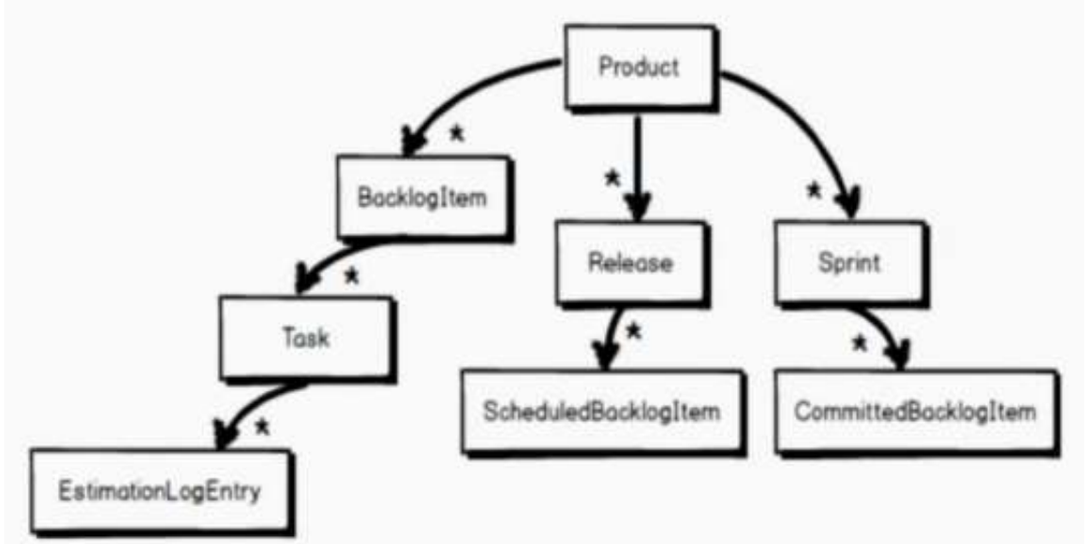


Source: Martin Fowler : Aggregate Root

2

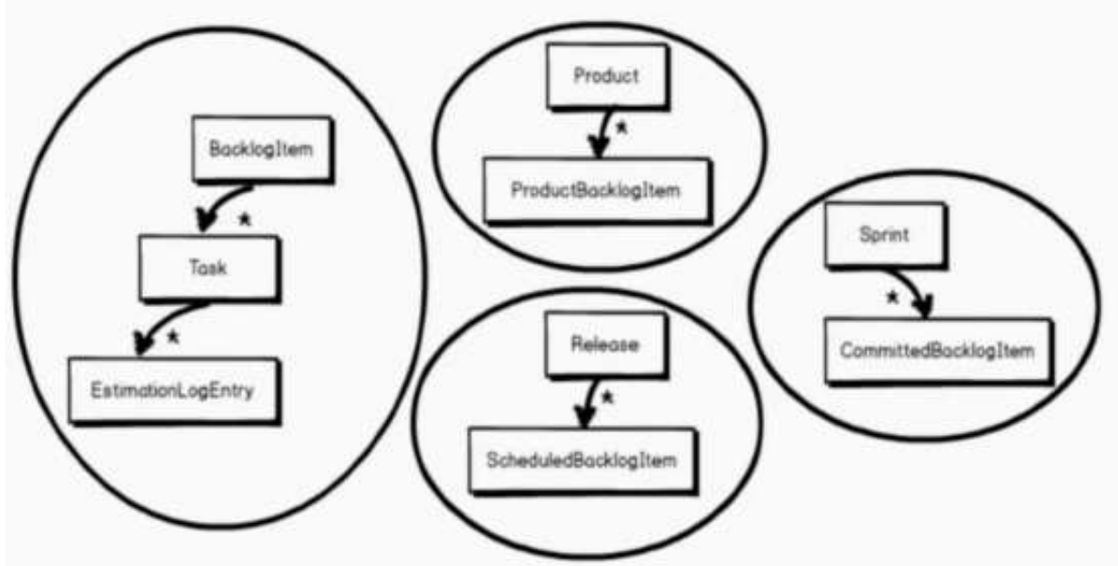
Designing and Fine Tuning Aggregate Root

Aggregate Root - #1



Super Dense Single Aggregate Root
Results in Transaction concurrency issues.

Aggregate Root - #2



Super Dense Aggregate Root is split into 4 different smaller Aggregate Root in the 2nd Iteration.

Working on different design models helps the developers to come up with best possible design.

Source : Effective Aggregate Design Part 1/2/3 : Vaughn Vernon
http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf

Anemic Domain Model : Anti Pattern

- There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have.
- The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters.
- The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together.
- The anemic domain model is really just a procedural style design, exactly the kind of thing that object bigots like me (and Eric) have been fighting since our early days in Smalltalk.

```java
package com.fusionfire.examples.commons.utils;

import java.util.ArrayList;

public class AnemicUser {

    private String name;

    private boolean isUserLocked;

    private ArrayList<String> addresses;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isUserLocked() {
        return isUserLocked;
    }

    public void setUserLocked(boolean isUserLocked) {
        this.isUserLocked = isUserLocked;
    }

    public ArrayList<String> getAddresses() {
        return addresses;
    }

    public void setAddresses(ArrayList<String> addresses) {
        this.addresses = addresses;
    }
}
```

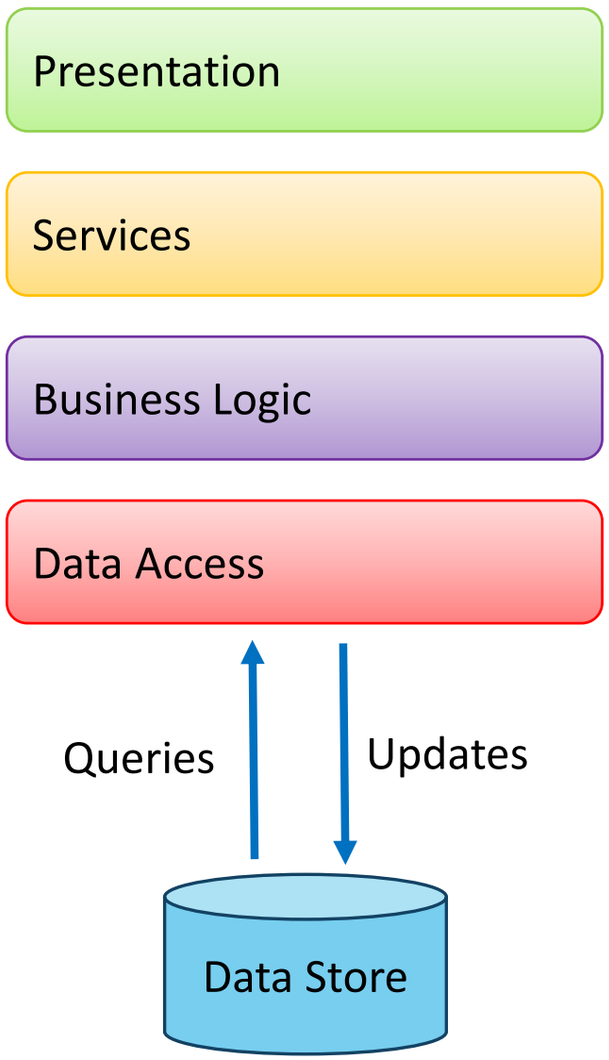
- lockUser()
- unlockUser()
- addAddress(String address)
- removeAddress(String address)

Source: Anemic Domain Model By Martin Fowler : http://martinfowler.com/bliki/AnemicDomainModel.html

CRUD and CQRS

Source: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
https://blogs.msdn.microsoft.com/maarten_mullender/2004/07/23/crud-only-when-you-can-afford-it-revisite

Traditional CRUD Architecture



CRUD Disadvantages

- **A mismatch between the read and write** representations of the data.

- It **risks data contention when records are locked in the data store** in a collaborative domain, where multiple actors operate in parallel on the same set of data. These risks increase as the complexity and throughput of the system grows.

- It can make **managing security and permissions more complex** because each entity is subject to both read and write operations, which might expose data in the wrong context.

2

Event Sourcing Intro

Standard CRUD Operations – Customer Profile – Aggregate Root

Profile Created | Title Updated | New Address added | Notes Removed

Time T1 | T2 | T3 | T4

Event Sourcing and Derived Aggregate Root

Commands

1. Create Profile
2. Update Title
3. Add Address
4. Delete Notes

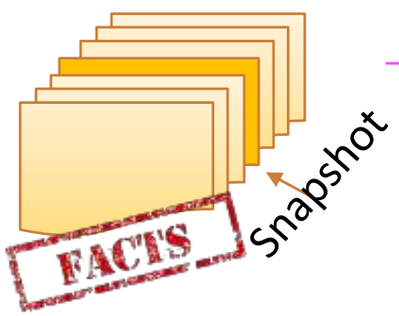
2

Events

1. Profile Created Event
2. Title Updated Event
3. Address Added Event
4. Notes Deleted Event

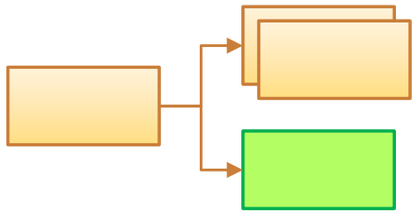
3

Event store



Single Source of Truth

Derived



Current State of the Customer Profile

4

Greg Young

2

Restaurant Dining – Event Sourcing and CQRS

Processes 1

When people arrive at the Restaurant and take a table, a Table is opened. They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served. Table is closed then the bill is prepared.

Customer Journey thru Dinning Processes

Commands 2

- Add Drinks
- Add Food
- Update Food

ES Aggregate 4

- Dinning Order
- Billable Order

Food Menu	Dining	Kitchen	Order	Payment
	Open Table	Remove Soda	Serve Drinks	**Prepare Bill**
	Add Juice	Add Food 1	Prepare Food	Process Payment
	Add Soda	Add Food 2	Serve Food	
	Add Appetizer 1	Place Order		
	Add Appetizer 2	**Close Table**		

Microservices

Events 3

- Drinks Added
- Food Added
- Food Updated
- Food Discontinued

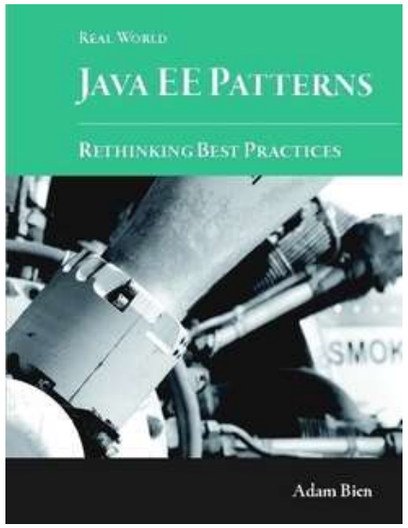
Dining		Kitchen	Order	Payment
Table Opened	Remove Soda	Juice Served	Bill Prepared	Payment Approved
Juice Added	Food 1 Added	Soda Served	Payment Processed	Payment Declined
Soda Added	Food 2 Added	Appetizer Served		Cash Paid
Appetizer 1 Added	Order Placed	Food Prepared		
Appetizer 2 Added	Table Closed	Food Served		

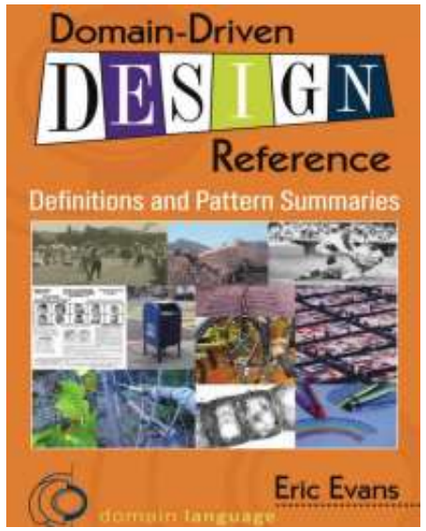
The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Summary – Event Sourcing and CQRS

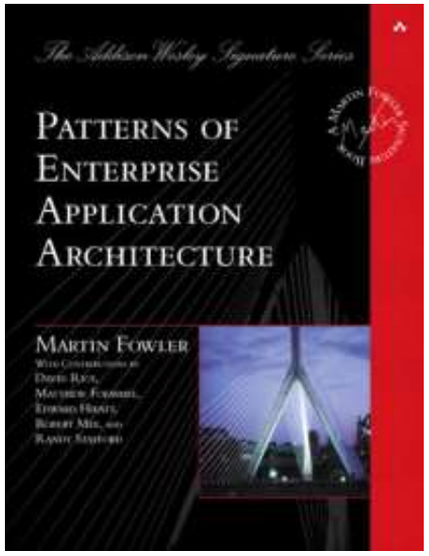
1. Immutable Events
2. Events represents the state change in Aggregate Root
3. Aggregates are Derived from a Collection of Events.
4. Separate Read and Write Models
5. Commands (originated from user or systems) creates Events.
6. Commands and Queries are always separated and possibly reads and writes using different data models.

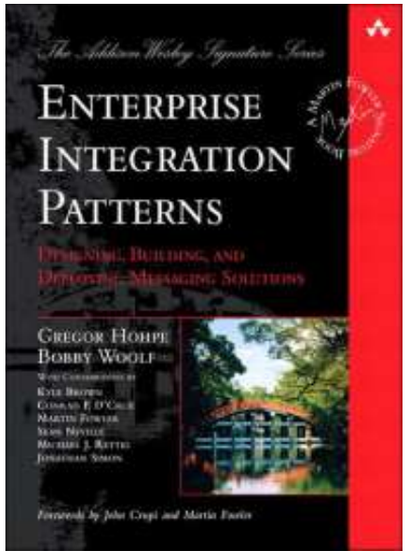
Design Patterns – Holy Grail of Developers

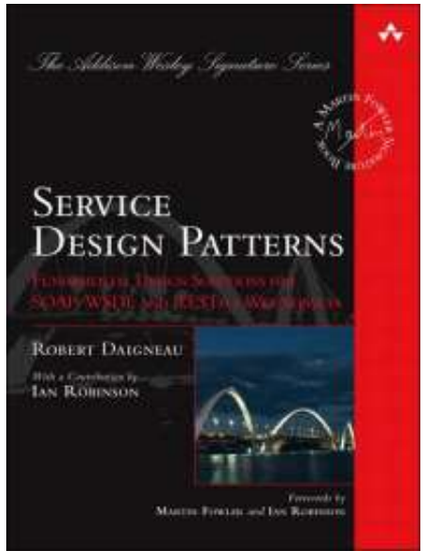


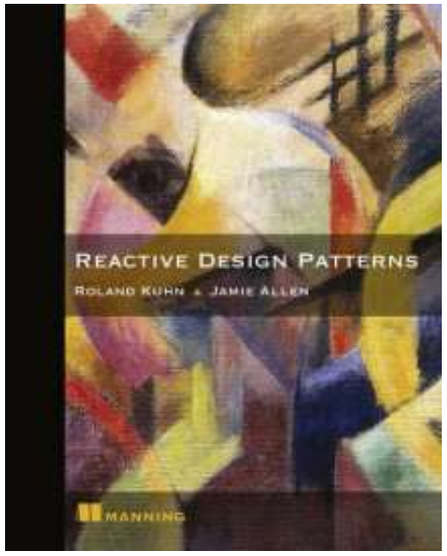


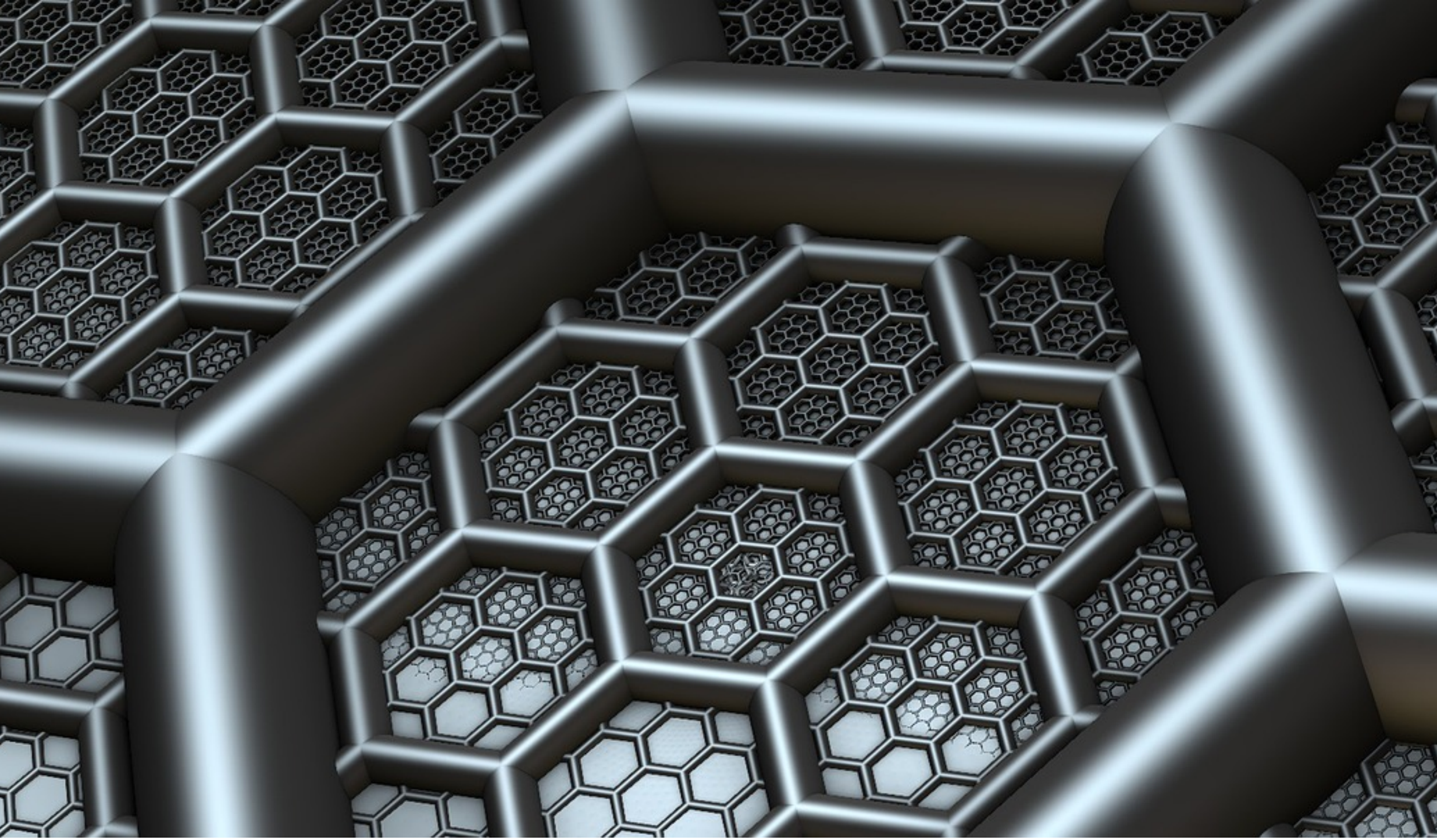
Design Patterns are solutions to general problems that software developers faced during software development.











Microservice Architecture (and hexagonal ones)

Pioneers in Microservices Implementation

NETFLIX

Spotify®

ebay

amazon web services

airbnb

UBER

NIKE

twitter

New Entrants

Walmart

verizon

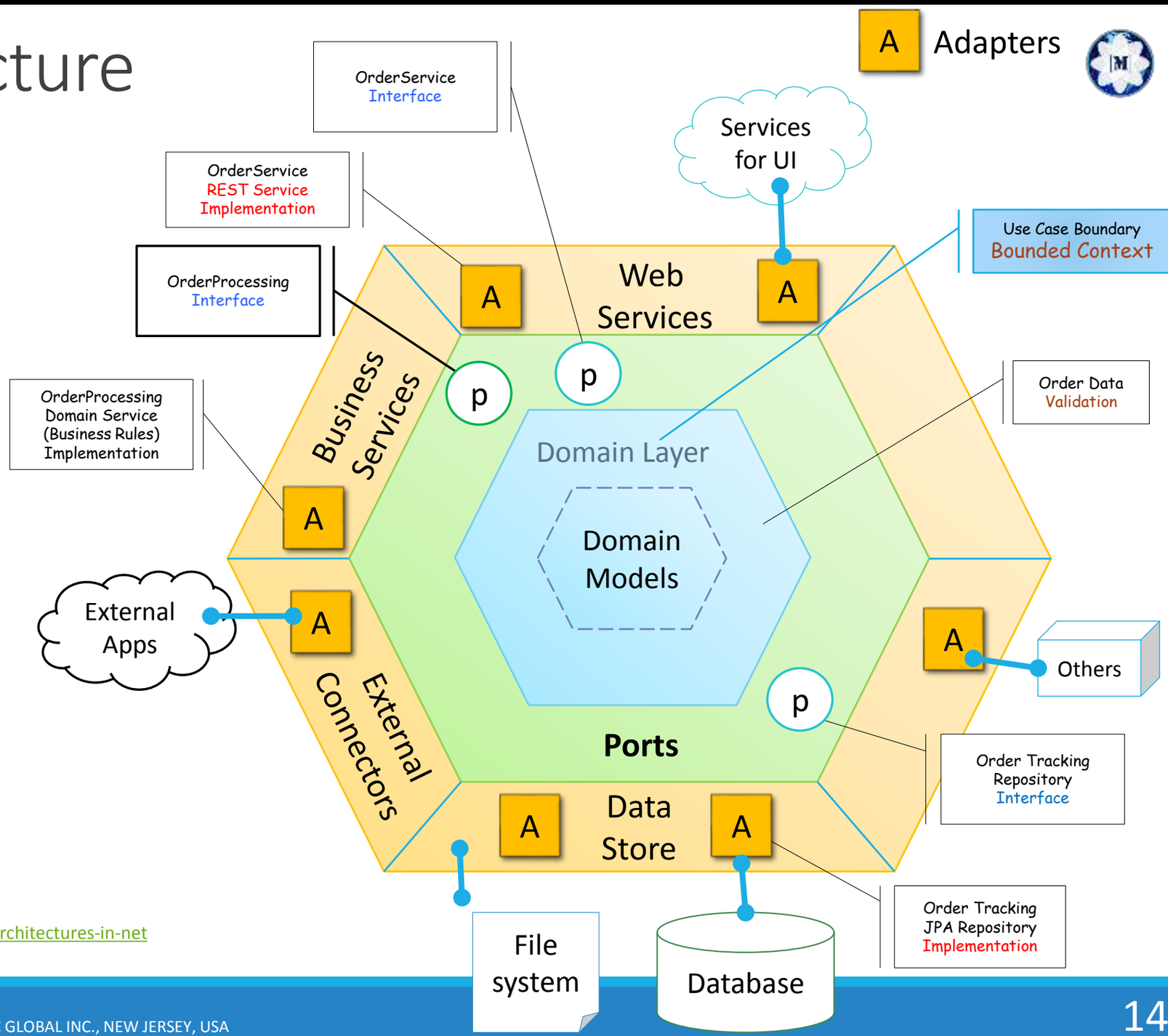
Hexagonal Architecture

Ports & Adapters

The layer between the **Adapter** and the **Domain** is identified as the **Ports** layer. The Domain is inside the port, adapters for external entities are on the outside of the port.

The notion of a “port” invokes the OS idea that any device that adheres to a known protocol can be plugged into a port. Similarly many adapters may use the Ports.

- Reduces Technical Debt
- Dependency Injection
- Auto Wiring



Source : http://alistair.cockburn.us/Hexagonal+architecture
https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net

Micro Services Characteristics

By James Lewis and Martin Fowler

Components via Services

Organized around **Business Capabilities**

Products NOT Projects

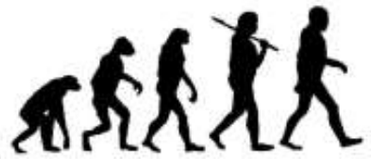
Smart Endpoints & Dumb Pipes

Decentralized Governance & Data Management

Infrastructure **Automation**

Design for **Failure**

Evolutionary Design



The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
Alan Kay, 1998 email to the Squeak-dev list

Modularity ... is to a technological economy what the division of labor is to a manufacturing one.
W. Brian Arthur,
author of *e Nature of Technology*

We can scale our operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration. —
Werner Vogels, CTO, Amazon Web Services

Pros and Cons

Pros

1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

Cons

1. Adds Complexity
2. Skillset shortage
3. Confusion on getting the right size
4. Team need to manage end-to-end of the Service (From UI to Backend to Running in Production).

Microservices Architecture

Infrastructure Architecture

- API Gateway, Service Discovery
- Event Bus / Streams
- Service Mesh

Software Design

- Domain Driven Design
- Event Sourcing & CQRS
- Functional Reactive Programming

Monolithic vs. Micro Services Example

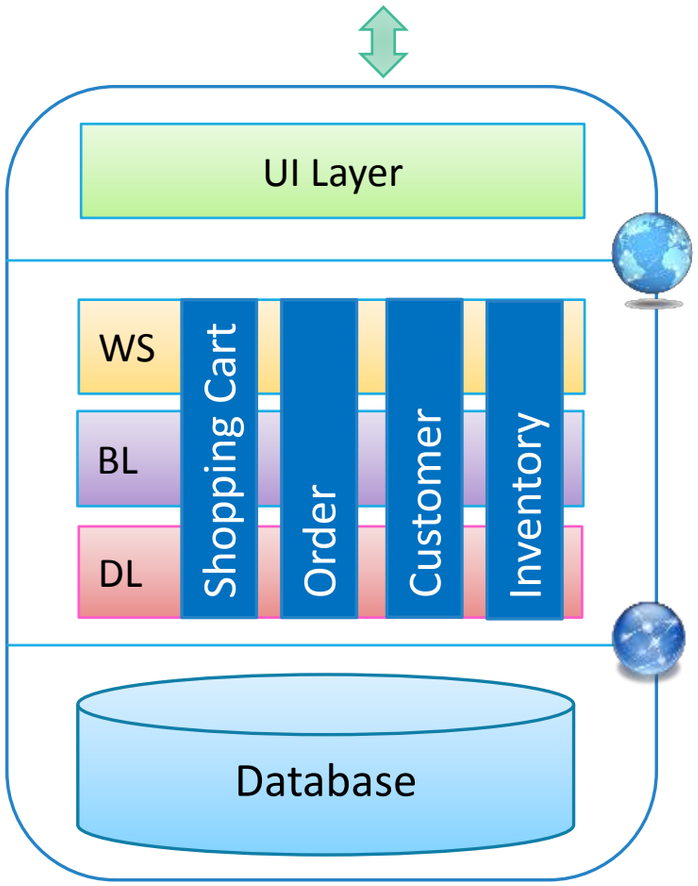




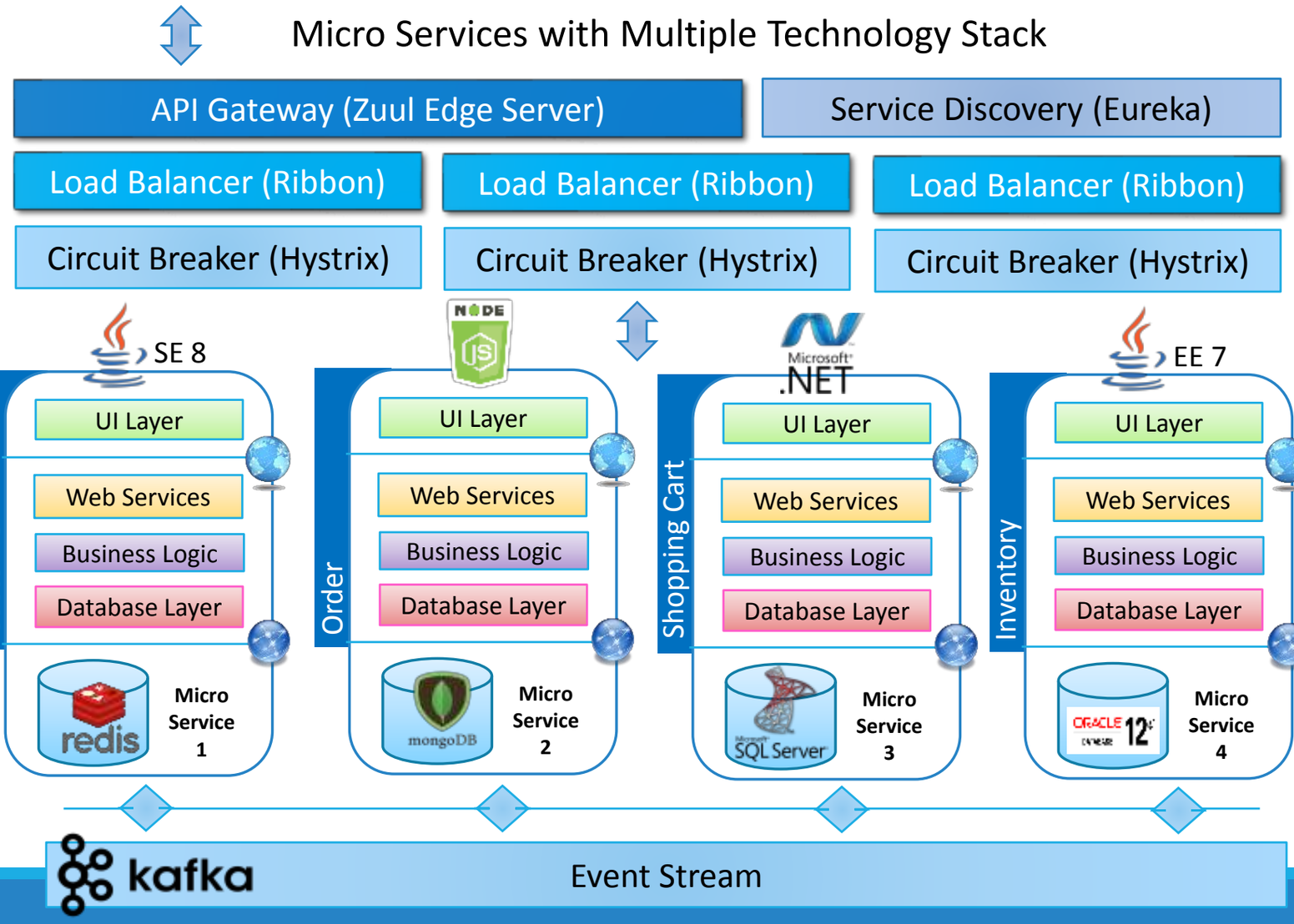

Existing aPaaS vendors creates Monolithic Apps.

This 3 tier model is obsolete now.

Source: Gartner Market Guide for Application Platforms Nov 23, 2016

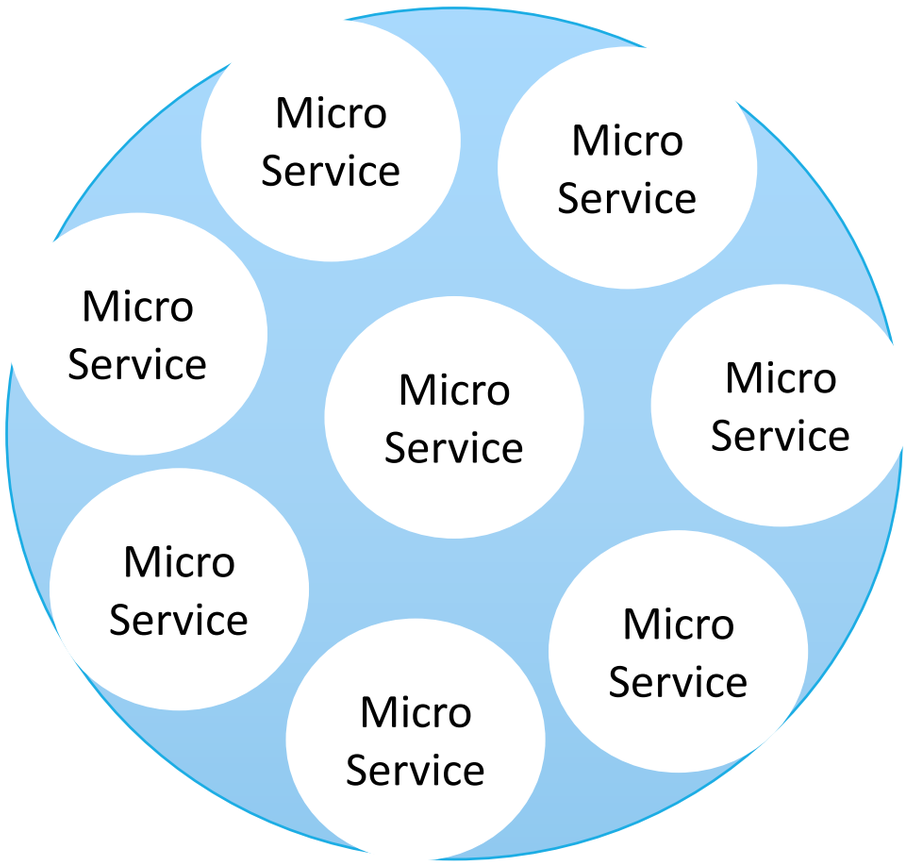


Traditional Monolithic App using Single Technology Stack



SOA vs. Micro Services Example

Summary – Micro Services Intro



Key Features

1. Small in size
2. Messaging–enabled
3. Bounded by contexts
4. Autonomously developed
5. Independently deployable
6. Decentralized
7. Language–agnostic
8. Built and released with automated processes

Benefits

1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

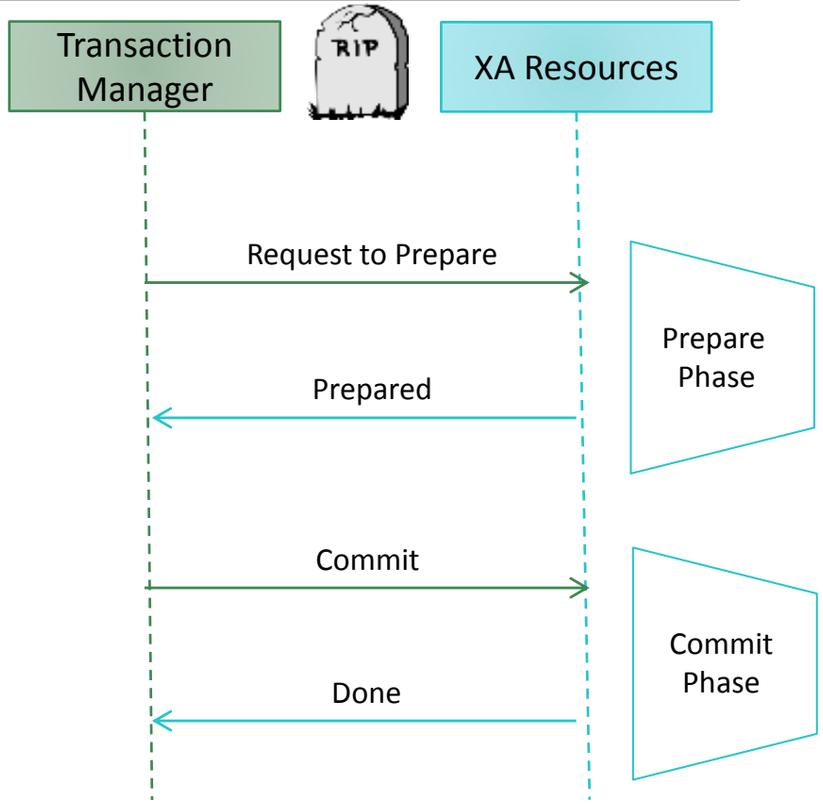
Martin Fowler – Micro Services Architecture
https://martinfowler.com/articles/microservices.html
Dzone – SOA vs Micro Services : https://dzone.com/articles/microservices-vs-soa-2

Distributed Transactions : 2 Phase Commit

2 PC or not 2 PC, Wherefore Art Thou XA?

How does 2PC impact scalability?

- Transactions are committed in two phases.
- This involves communicating with every database (XA Resources) involved to determine if the transaction will commit in the first phase.
- During the second phase each database is asked to complete the commit.
- While all of this coordination is going on, locks in all of the data sources are being held.
- ***The longer duration locks create the risk of higher contention.***
- *Additionally,* ***the two phases require more database processing time than a single phase commit.***
- **The result is lower overall TPS in the system.**



Solution : Resilient System

- Event Based
- Design for failure
- Asynchronous Recovery
- Make all operations idempotent.
- Each DB operation is a 1 PC

Source : Pat Helland (Amazon) : Life Beyond Distributed Transactions Distributed Computing : http://dancres.github.io/Pages/

Handling Invariants – Monolithic to Micro Services

In a typical Monolithic App Customer Credit Limit info and the order processing is part of the same App. Following is a typical pseudo code.

In Micro Services world with Event Sourcing, it's a distributed environment. The order is cancelled if the Credit is NOT available. If the Payment Processing is failed then the Credit Reserved is cancelled.

Monolithic 2 Phase Commit

Begin Transaction

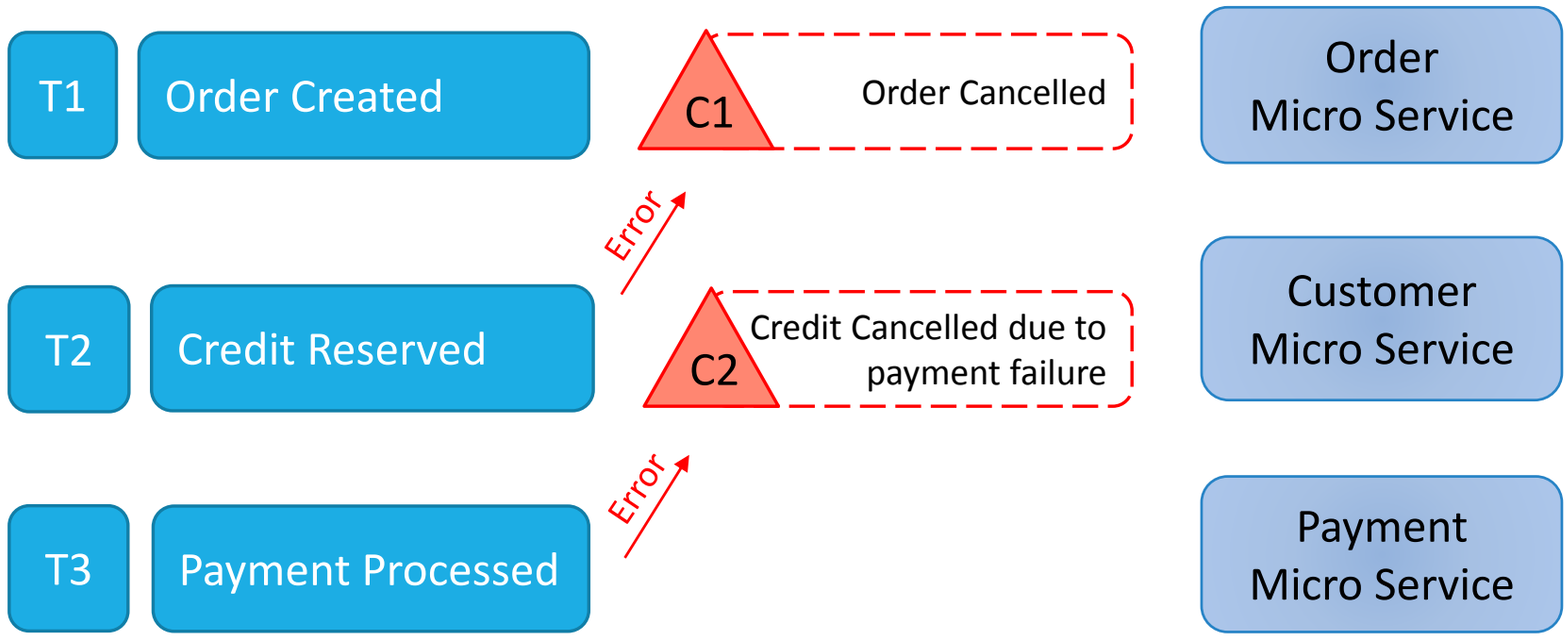
If Order Value <= Available Credit

Process Order

Process Payments

End Transaction

https://en.wikipedia.org/wiki/Invariant_(computer_science)



Scalability Best Practices : Lessons from ebay

	Best Practices	Highlights
#1	Partition By Function	• Decouple the Unrelated Functionalities. • Selling functionality is served by one set of applications, bidding by another, search by yet another. • 16,000 App Servers in 220 different pools • 1000 logical databases, 400 physical hosts
#2	Split Horizontally	• Break the workload into manageable units. • eBay's interactions are stateless by design • All App Servers are treated equal and none retains any transactional state • Data Partitioning based on specific requirements
#3	Avoid Distributed Transactions	• 2 Phase Commit is a pessimistic approach comes with a big COST • **CAP Theorem** (Consistency, Availability, Partition Tolerance). Apply any two at any point in time. • @ eBay No Distributed Transactions of any kind and NO 2 Phase Commit.
#4	Decouple Functions Asynchronously	• If Component A calls component B synchronously, then they are tightly coupled. For such systems to scale A you need to scale B also. • If Asynchronous A can move forward irrespective of the state of B • SEDA (Staged Event Driven Architecture)
#5	Move Processing to Asynchronous Flow	• Move as much processing towards Asynchronous side • Anything that can wait should wait
#6	Virtualize at All Levels	• Virtualize everything. eBay created their on O/R layer for abstraction
#7	Cache Appropriately	• Cache Slow changing, read-mostly data, meta data, configuration and static data.

Source: http://www.infoq.com/articles/ebay-scalability-best-practices

24 June 2018

Summary

1. Highly Scalable & Resilient Architecture
2. Technology Agnostic
3. Easy to Deploy
4. SAGA for Distributed Transaction
5. Faster Go To Market

In a Micro Service Architecture,

The **services tend to get simpler**, but the **architecture tends to get more complex**.

That **complexity** is often managed with **Tooling, Automation, and Process.**

Microservices Testing Strategy

Unit Testing

A unit test exercises the smallest piece of testable software in the application to determine whether it behaves as expected.

Integration Testing

An integration test verifies the communication paths and interactions between components to detect interface defects

Component Testing

A component test limits the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces and using test doubles to isolate the code under test from other components.

Contract Testing

An integration contract test is a test at the boundary of an external service verifying that it meets the contract expected by a consuming service.

End 2 End Testing

An end-to-end test verifies that a system meets external requirements and achieves its goals, testing the entire system, from end to end

Source: https://martinfowler.com/articles/microservice-testing/#agenda

When to use it?

In the beginning:

- You don't need it
- It will slow you down

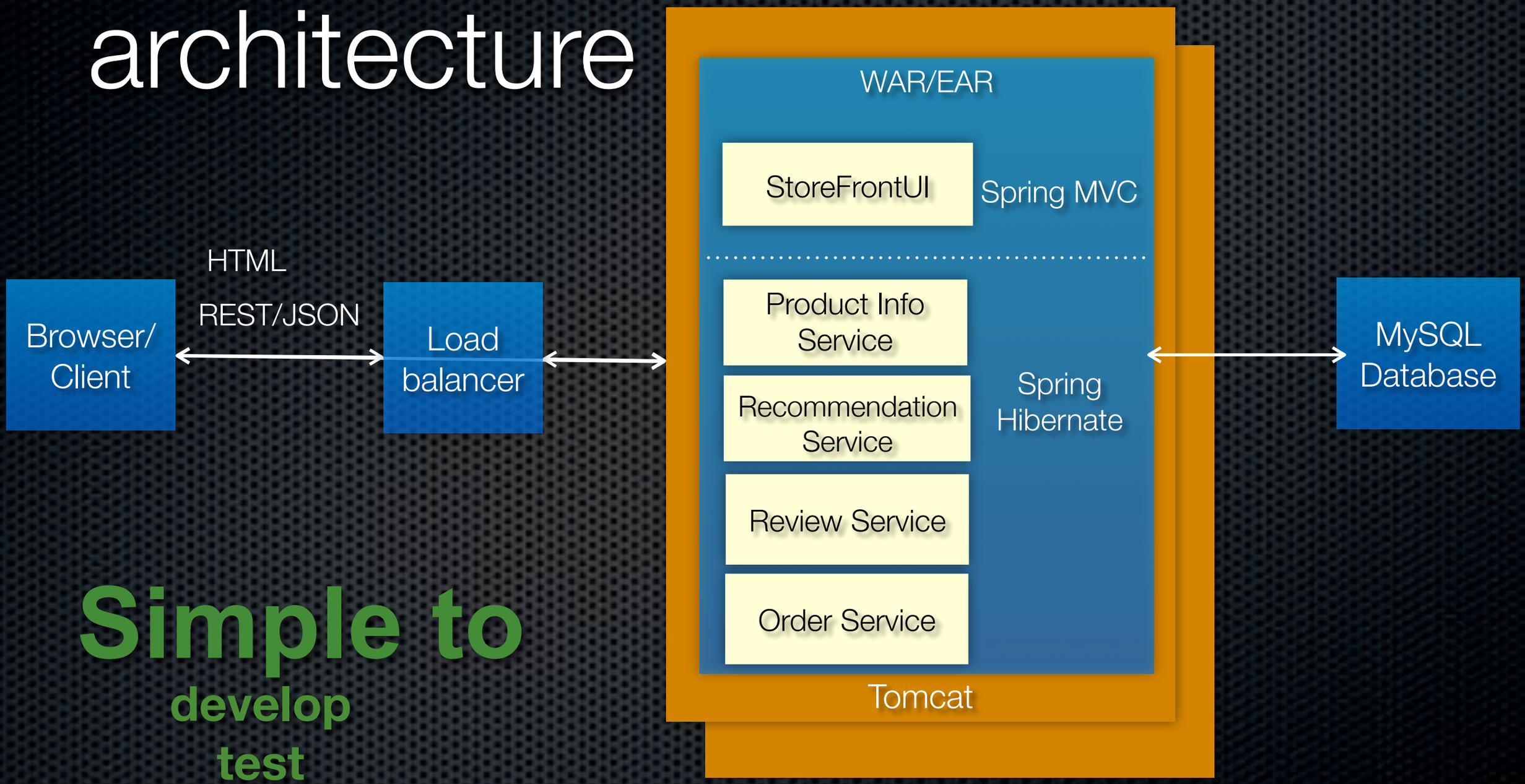
Later on:

- You need it
- Refactoring is painful

@crichardson

Industrial Feedback

Traditional application architecture



Simple to

develop
test
deploy
scale

Obstacle to frequent deployments

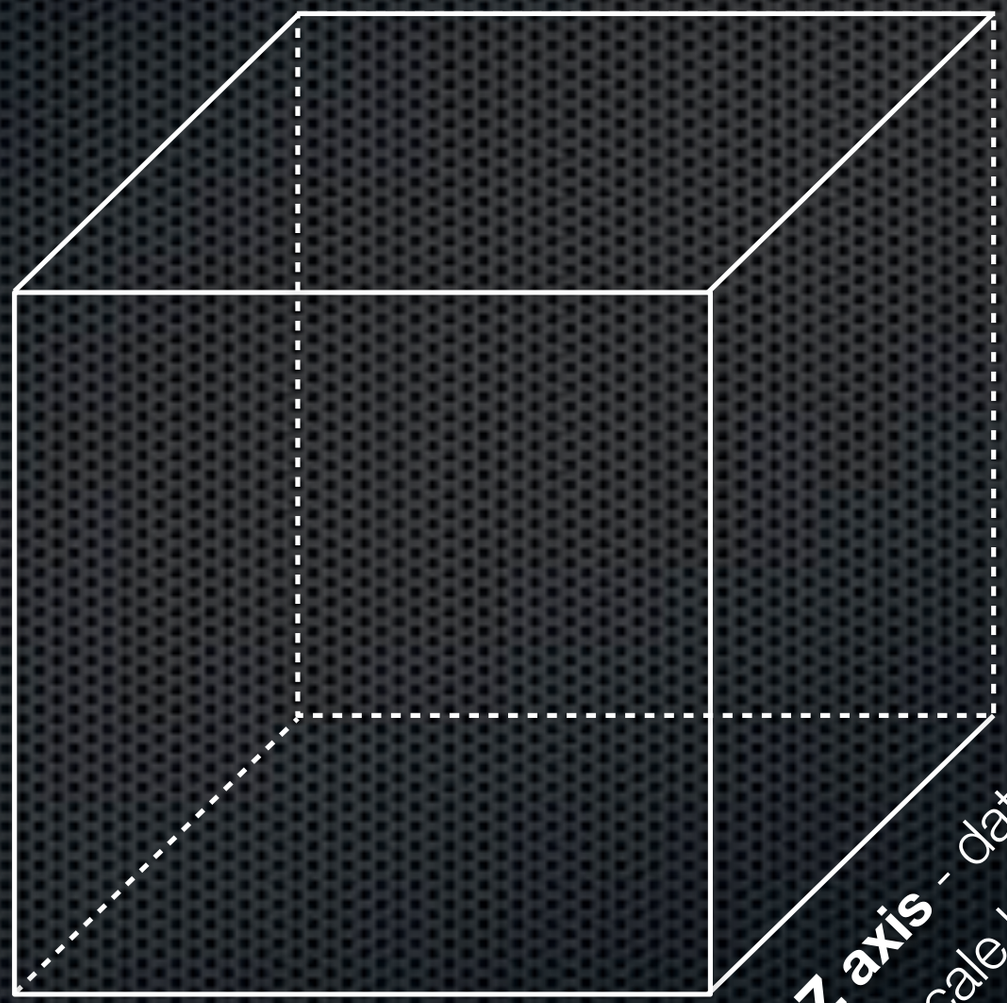
- Need to redeploy everything to change one component
- Interrupts long running background (e.g. Quartz) jobs
- Increases risk of failure

- Updates will happen less often - really long QA cycles
- e.g. Makes A/B testing UI really difficult

The scale cube

Y axis - functional decomposition

Scale by splitting different things

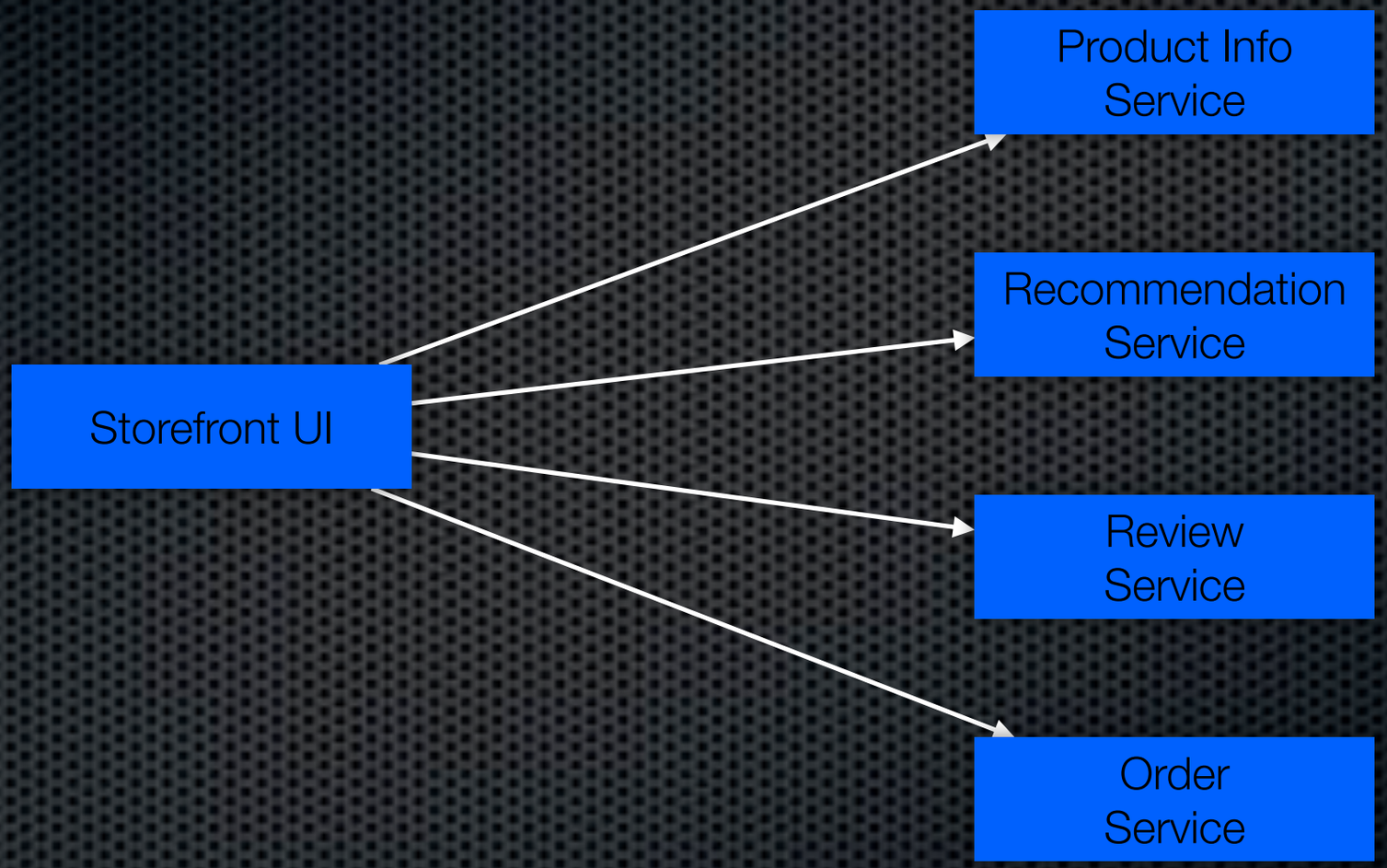


Z axis - data partitioning
Scale by splitting similar things

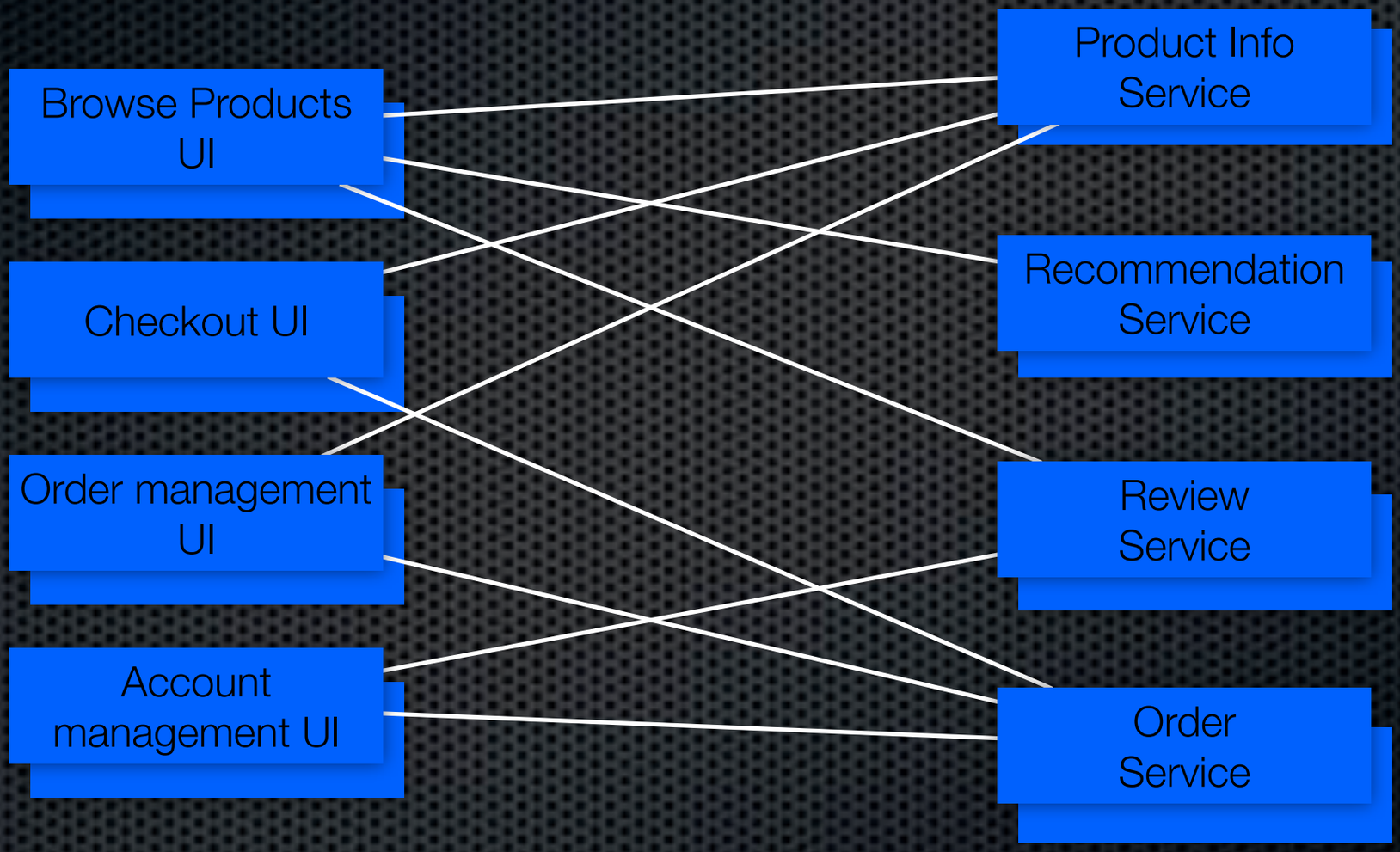
X axis
- horizontal duplication

Y-axis scaling - application level

Y-axis scaling - application level



Y-axis scaling - application level



Apply X-axis and Z-axis scaling to each service independently

Smaller, simpler apps

- Easier to understand and develop
- Less jar/classpath hell - who needs OSGI?
- Faster to build and deploy
- Reduced startup time - important for GAE

Scales development:
develop, deploy and scale
each service independently

Improves fault isolation

Eliminates long-term commitment to a single technology stack

Modular, polyglot, multi-framework applications

Complexity

Complexity of developing a distributed system

http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

Multiple databases
&
Transaction management

e.g. Fun with eventual consistency
Come to my 11.30 am talk

Developing and deploying features that span multiple services requires careful coordination

?