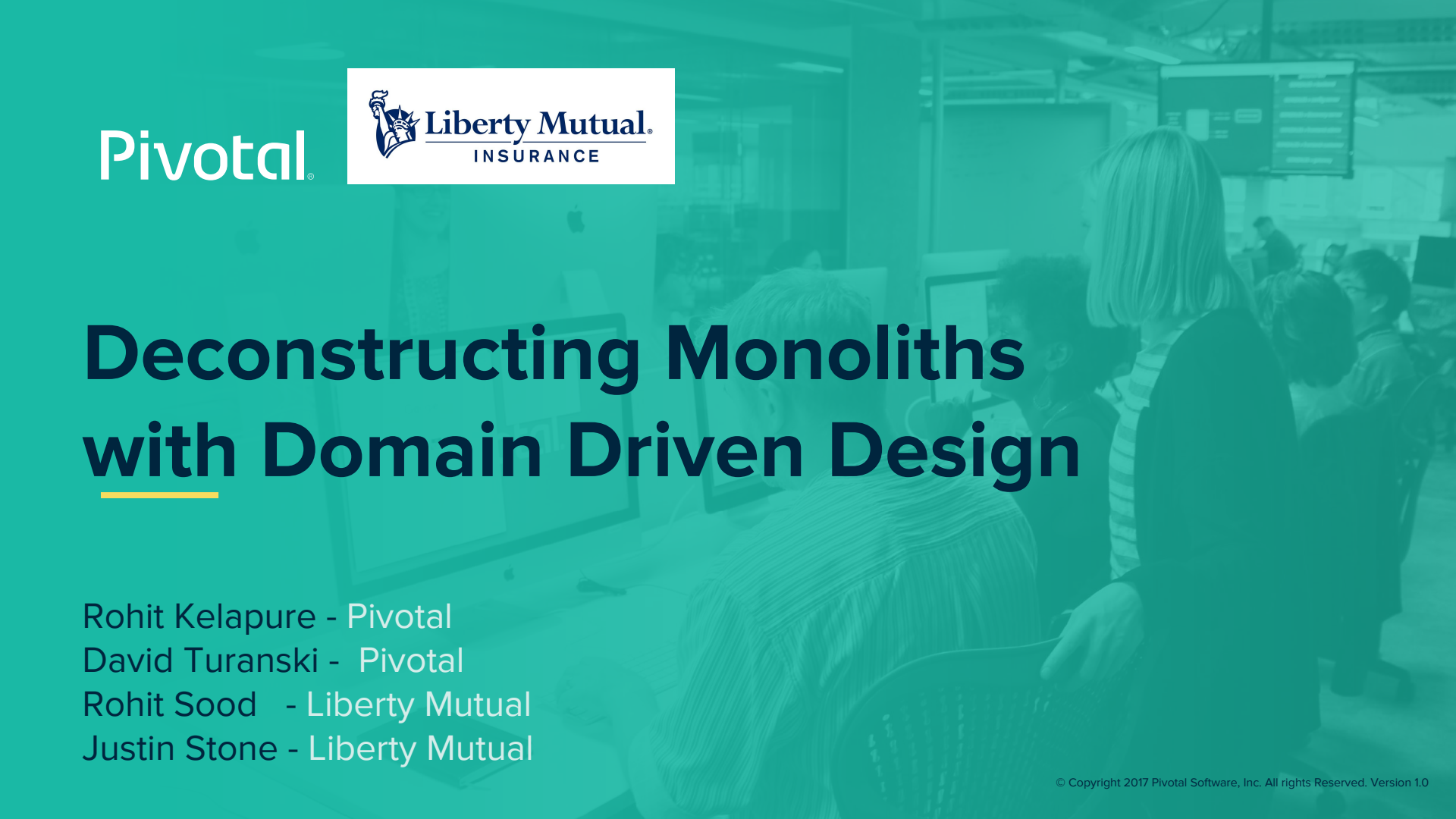


Pivotal

Liberty Mutual INSURANCE

Deconstructing Monoliths with Domain Driven Design

Rohit Kelapure - Pivotal
David Turanski - Pivotal
Rohit Sood - Liberty Mutual
Justin Stone - Liberty Mutual

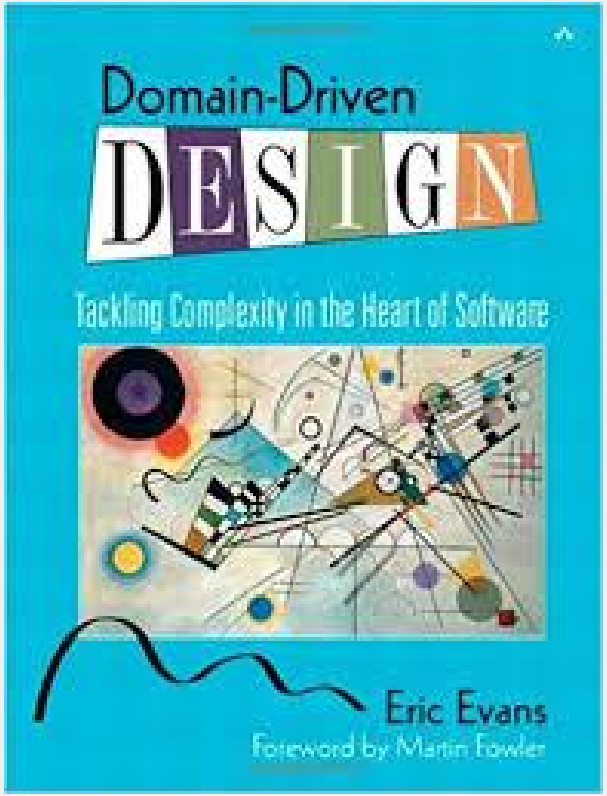
Part 1
DDD, CQRS, EVENT SOURCING

Domain-Driven Design

"...the real breakthrough of object design comes when the code expresses the concepts of a model."

"Anyone responsible for changing code must learn to express a model through the code. Every developer must be involved in some level of discussion about the model and have contact with **domain experts**."

- Eric Evans, Domain-Driven Design



Domain Driven Design

“In the end the code is the model and the model is the code.” - Vaughn Vernon **,** Domain-Driven Design Distilled

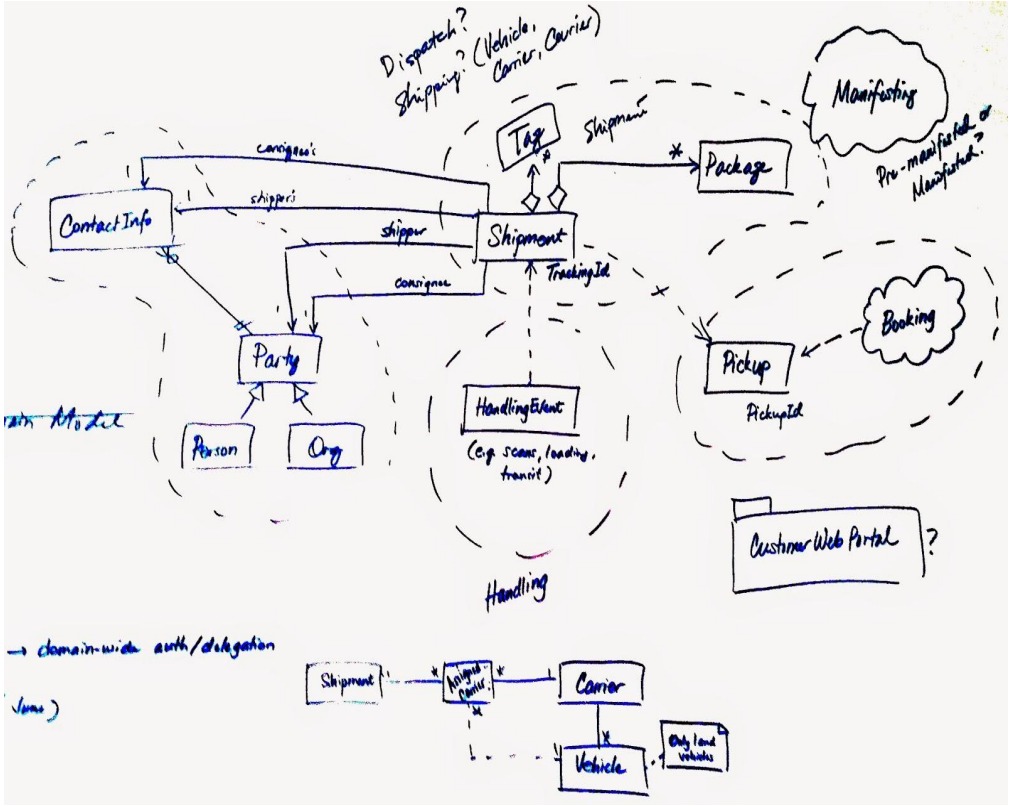
As opposed to:

1. Business Overview

1.1 Requirement

This interface is used to send Sched CGO messages from SHIP-PRO to ODS. SHIP-PRO will send the SyncSCHD_CGO BO to MBS and MBS will push the data to ODS as SCHD_CGO_WRAPPER BO.

This interface will also be added to the AMS view for monitoring.



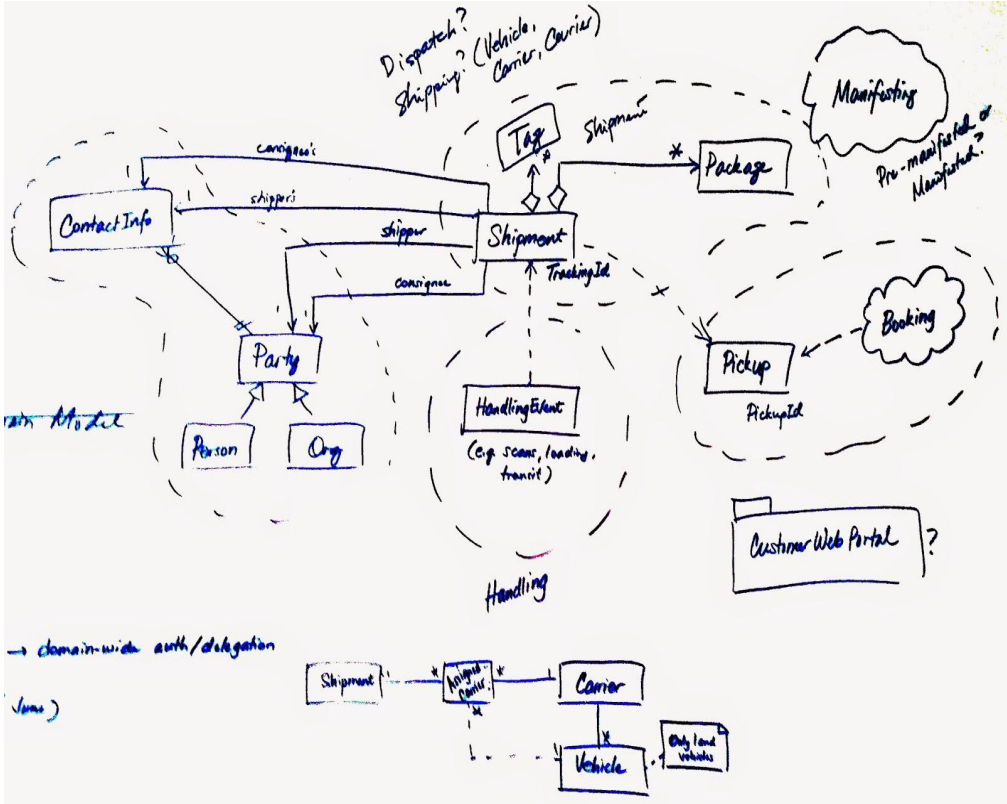
Entities, Value Objects, and Services

Entity - something with continuity and identity (Customer)

Value Object - an attribute that describes the state of something else (Address)

Service - Actions or operations that don't belong in an Entity or VO

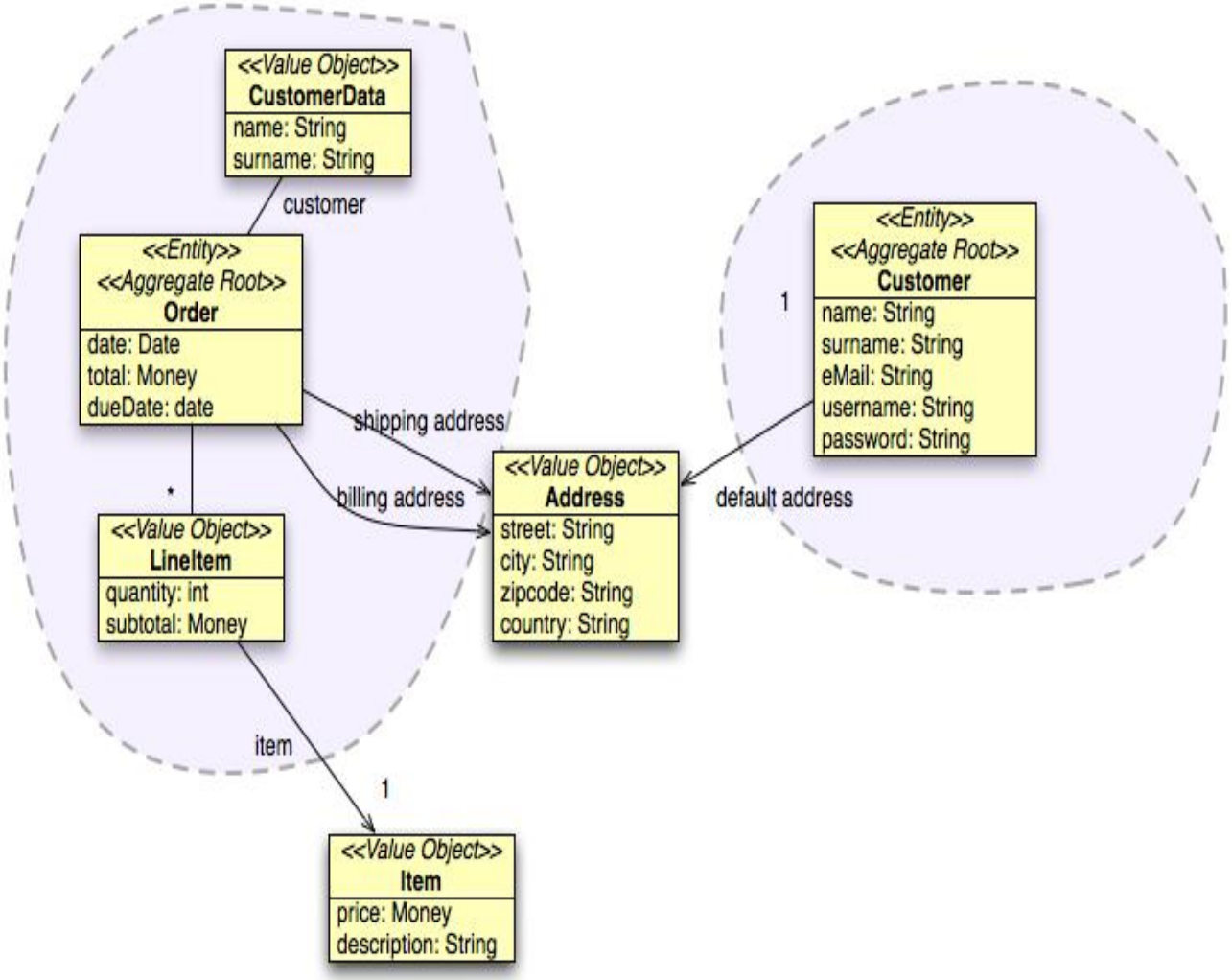
- Domain service - Funds transfer
- Technical service - Notification



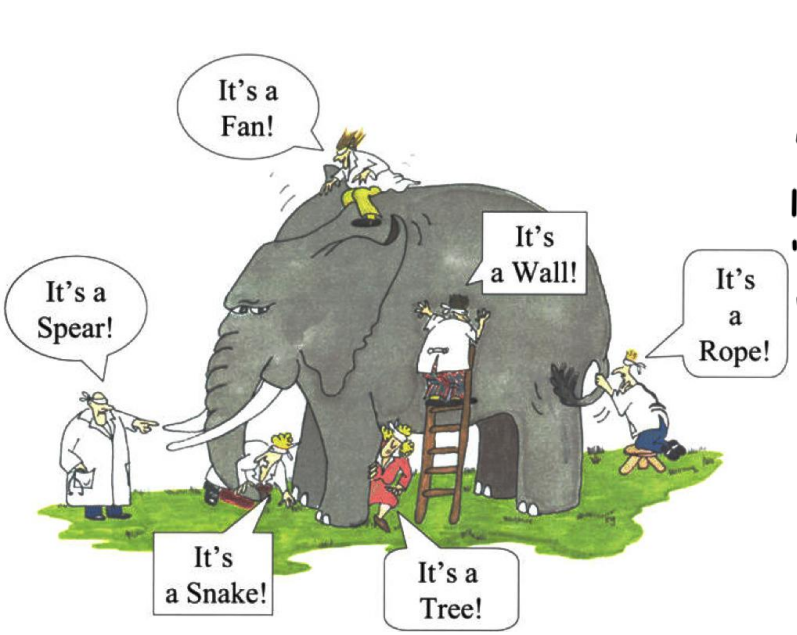
Aggregates

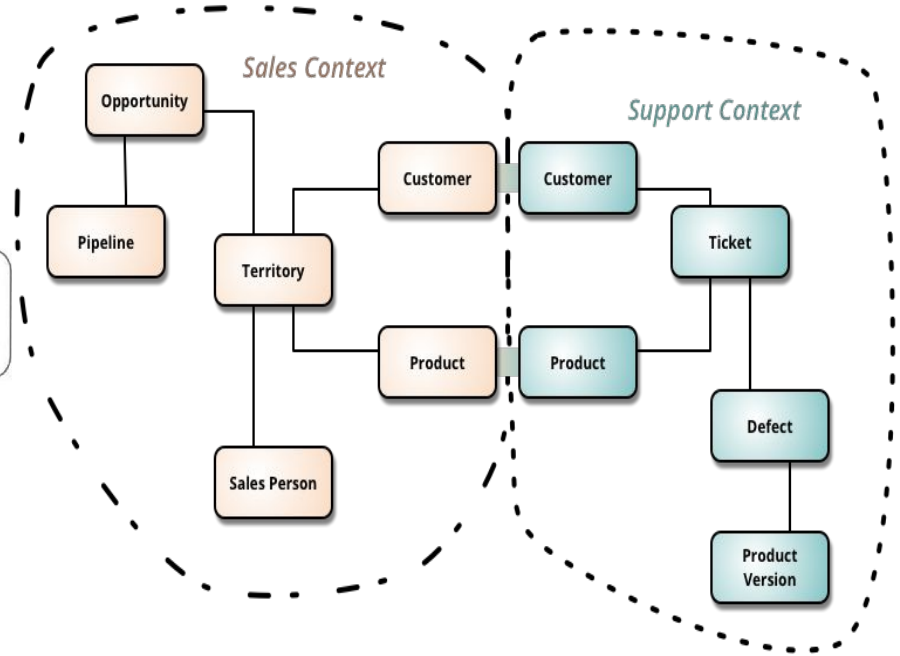
AGGREGATES mark off the scope within which invariants have to be maintained at every stage of the lifecycle

- Transactional Boundary
- Public access only via the Aggregate Root



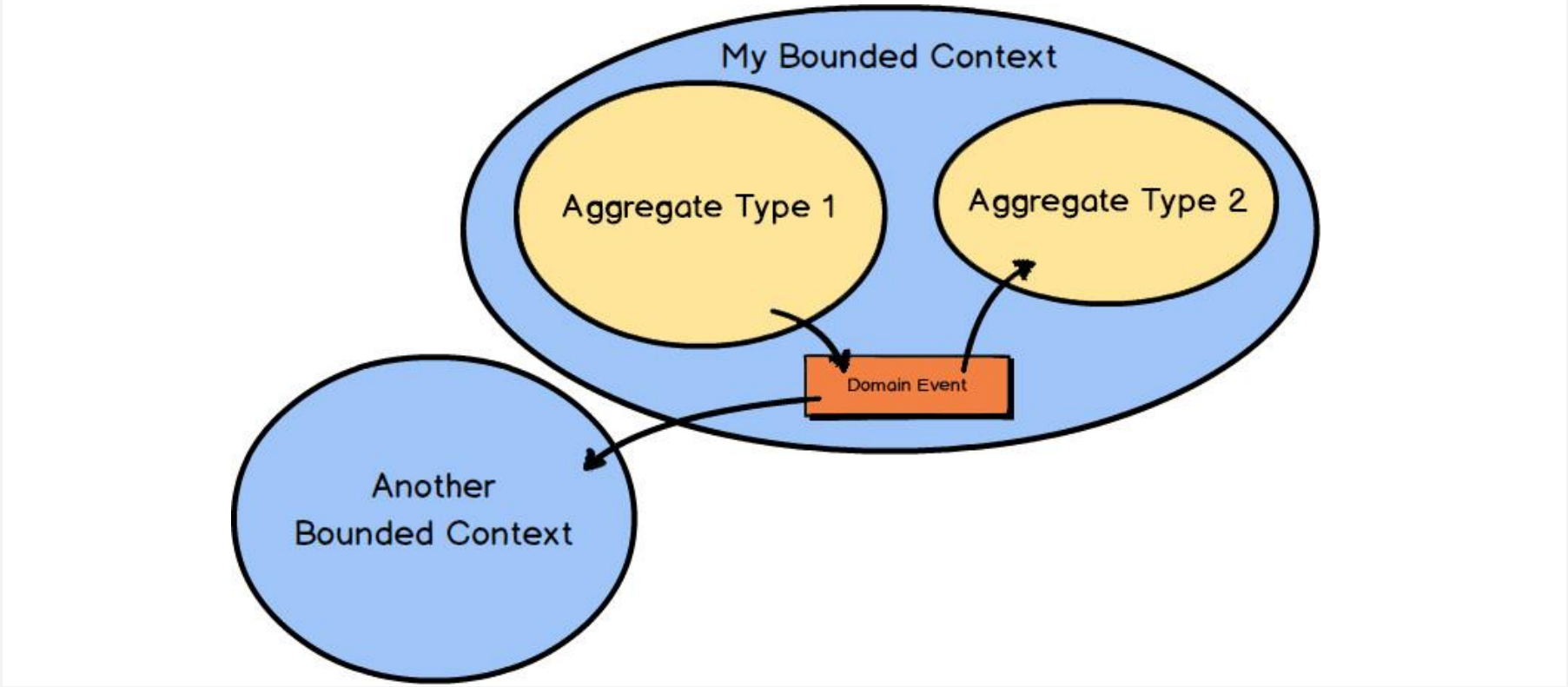
Bounded Context



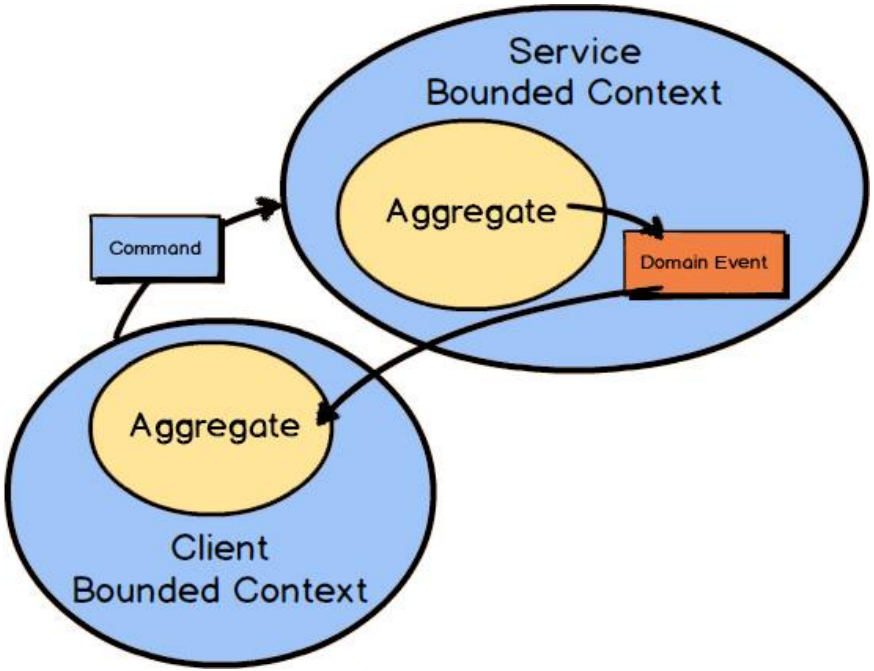


"Generally speaking, there is a correspondence of one team per BOUNDED CONTEXT. One team can maintain multiple BOUNDED CONTEXTS, but it is hard … for multiple teams to work on one together." - Evans

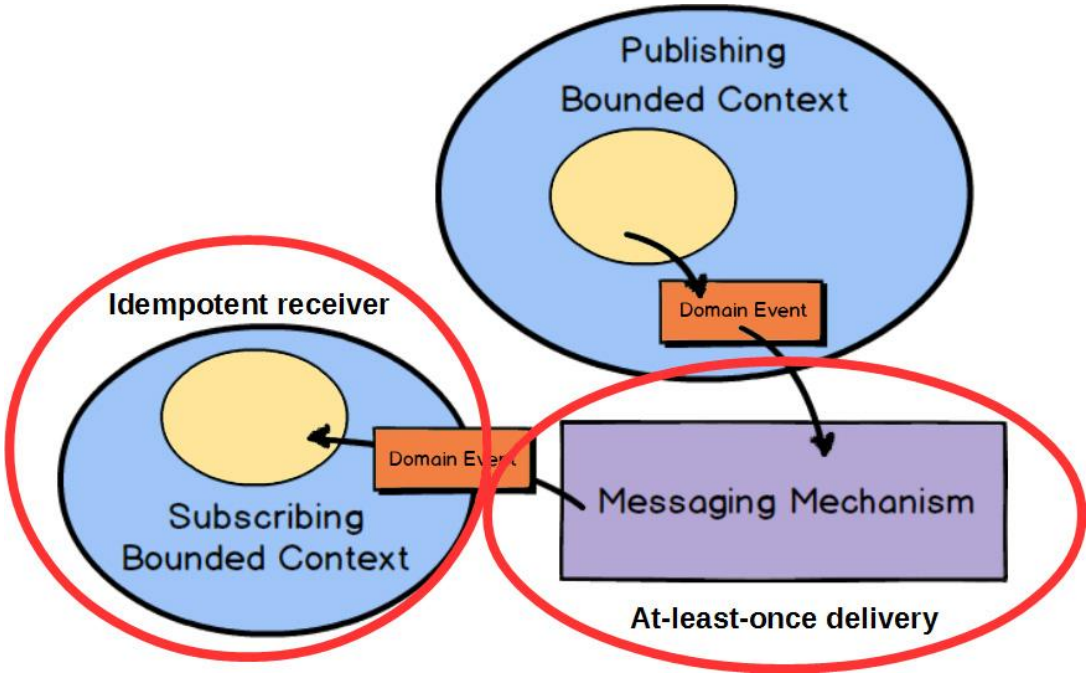
Event Driven Architecture



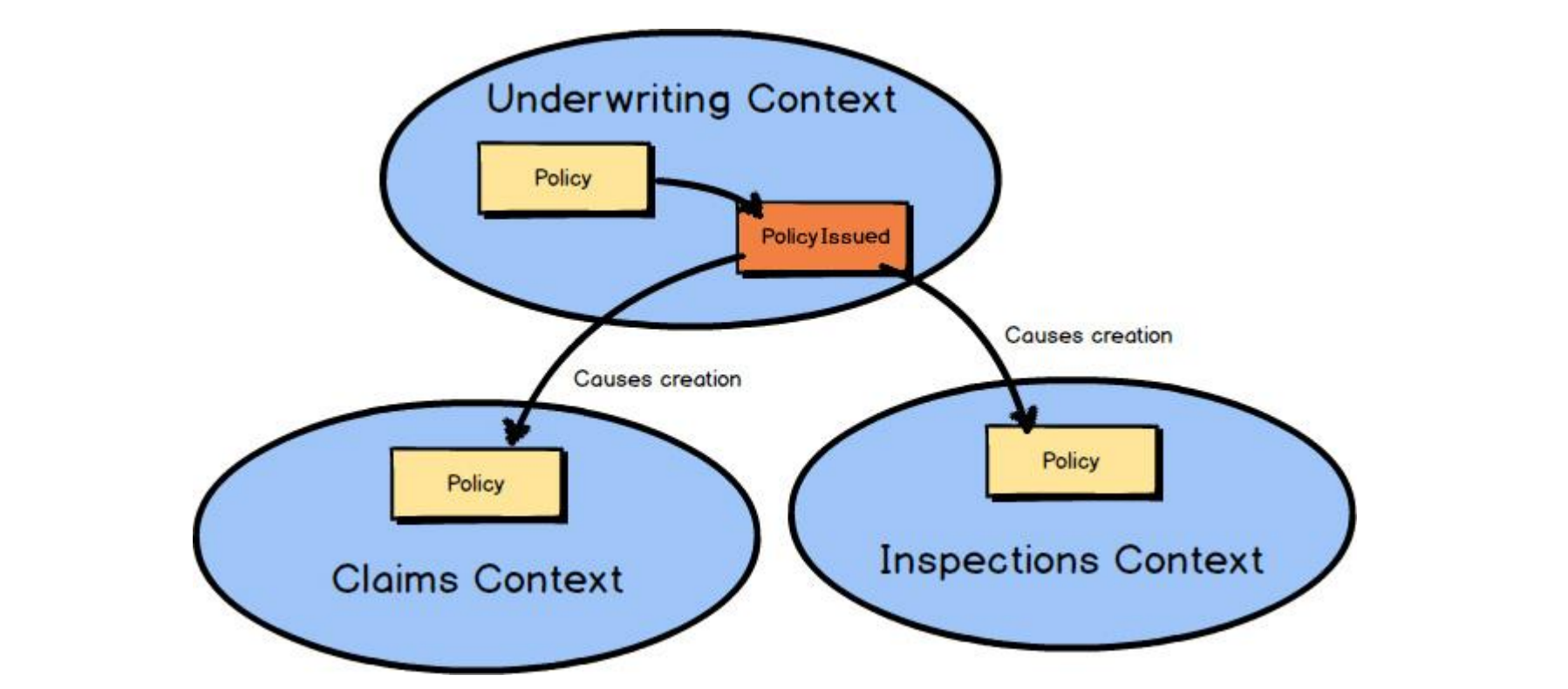
Command Triggers Events



Messaging

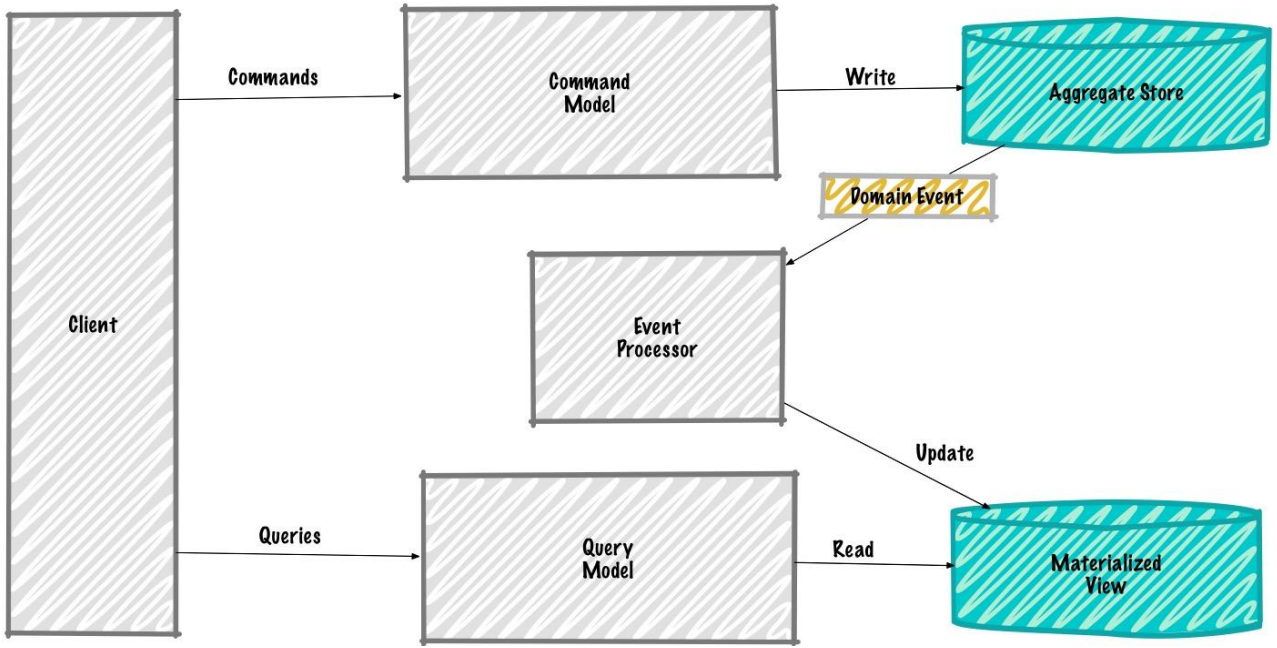

Publishing
Bounded Context
Domain Event
Idempotent receiver
Subscribing
Bounded Context
Messaging Mechanism
At-least-once delivery

Publish-Subscribe

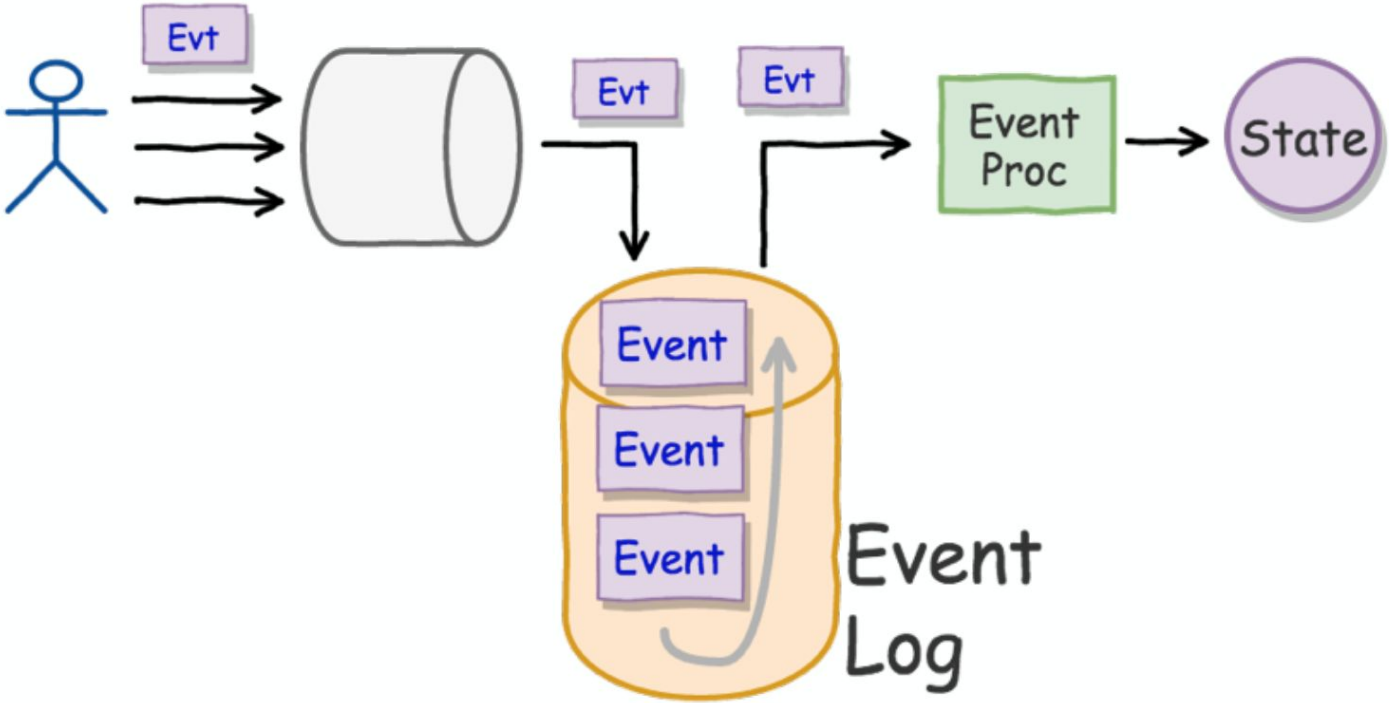


Eventual Consistency

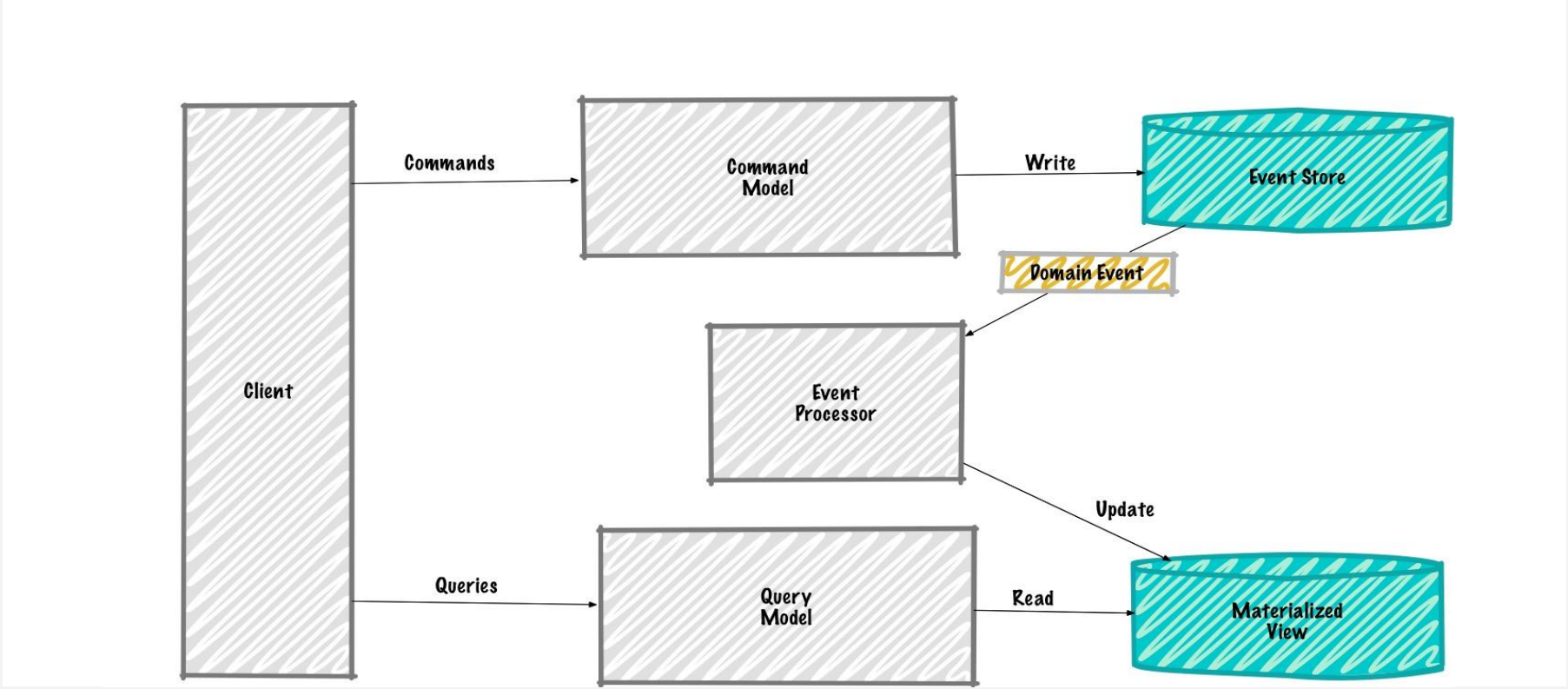
CQRS


Client
Commands
Command Model
Write
Aggregate Store
Domain Event
Event Processor
Update
Queries
Query Model
Read
Materialized View

Event Sourcing



CQRS + ES


Client
Commands
Command Model
Write
Event Store
Domain Event
Event Processor
Update
Queries
Query Model
Read
Materialized View

Resources

SpringOne Platform Sessions

- Reactive DDD, Vaughn Vernon - Tue
- State or Events, J. Pillmon, K. Bestani - Tue
- Refactoring to System of Systems, O. Gierke - Next up

Event Sourcing Frameworks

- Axon Framework - http://www.axonframework.org/
- Eventuate - http://eventuate.io/

Event Store

- https://eventstore.org/

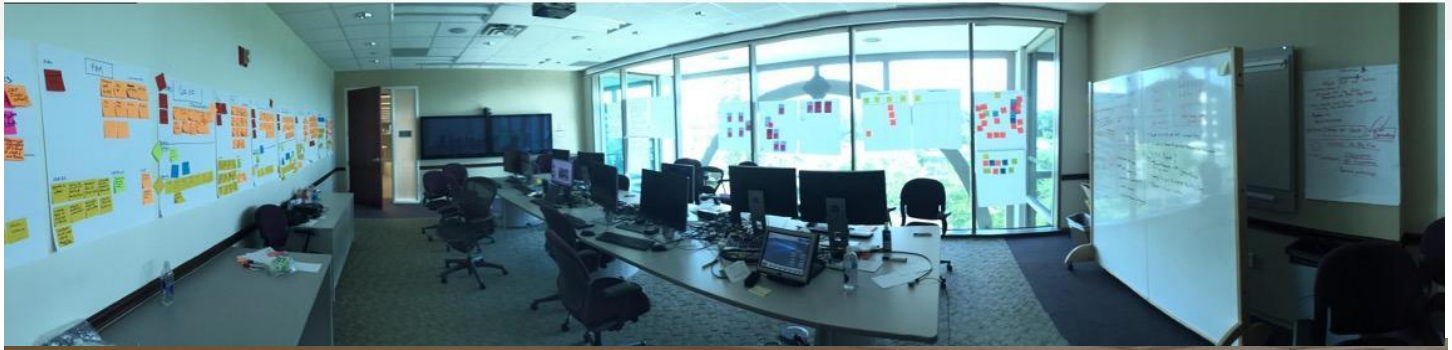
Further Reading

- Domain-Driven Design, Eric Evans
- Implementing Domain-Driven Design, Vaughn Vernon
- Domain-Driven Design Distilled, Vaughn Vernon
- Building Microservices, Sam Newman
- https://dturanski.wordpress.com/2017/03/26/spring-cloud-stream-for-event-driven-architectures/ , David Turanski

Modernizing Legacy workloads

Deconstructing The Monolith

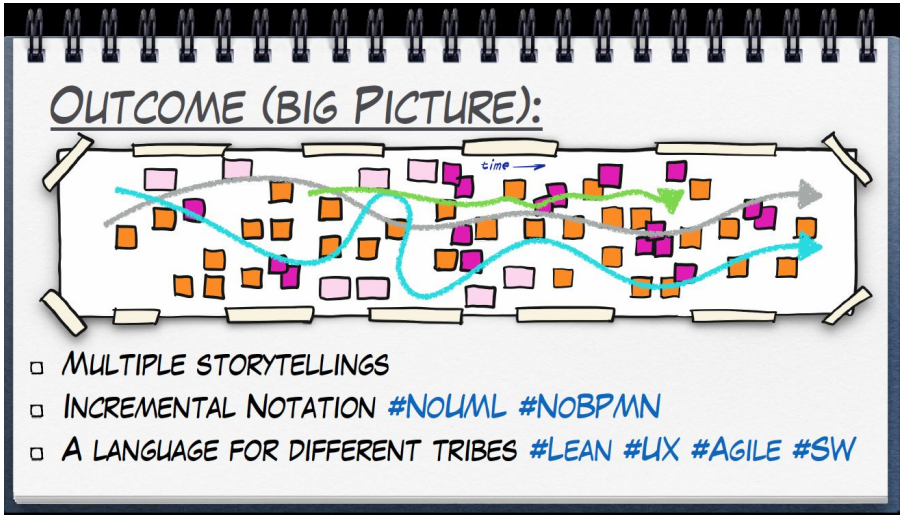
Break a Complex Domain with Event Storming



ES - Enables Cross Perspective Communication

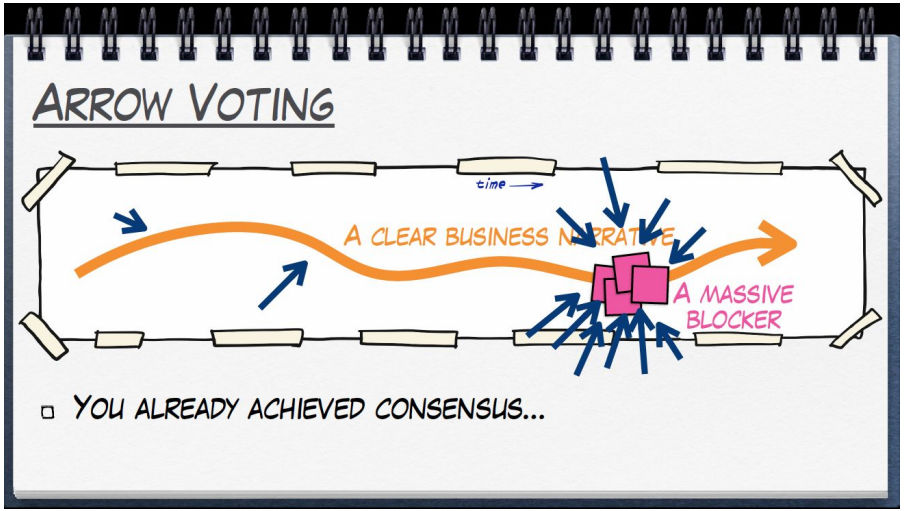
Situational Awareness

Making sense of a complex domain



Clarity

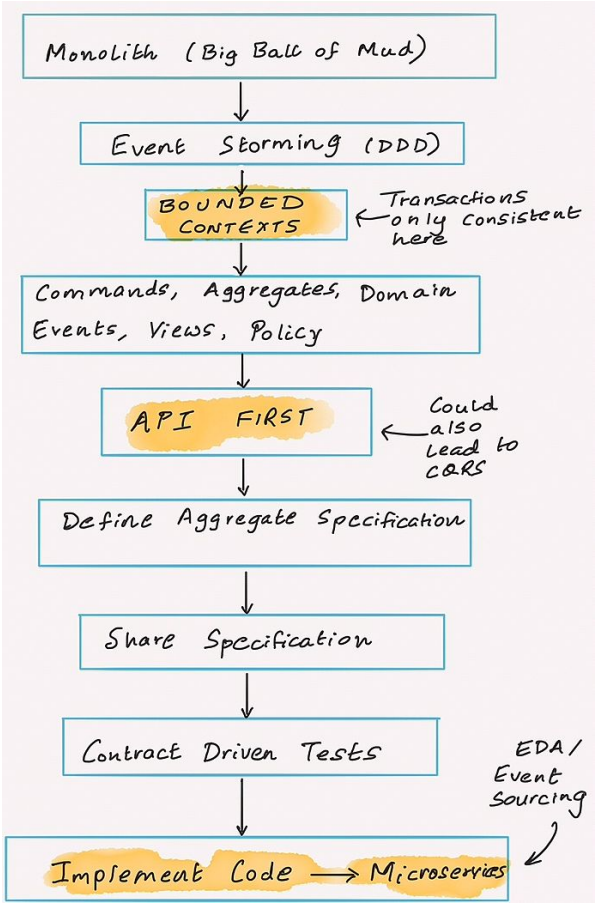
Identify the Right Constraint



Business Domain Model derived from Business Events

Deconstructing a Monolith

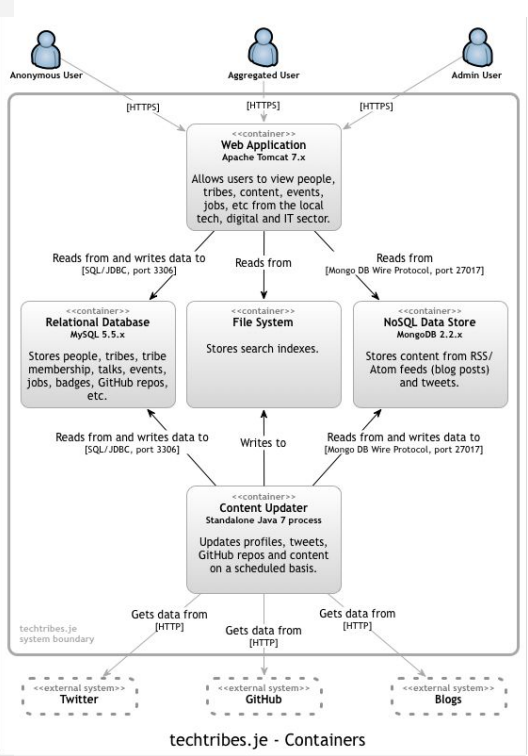
- Define Objectives and Key Results (OKRs)
- Event Storm the app and identify the core domain and key constraints
- Pick horizontal sequences of events in the core domain identified during ES
- Use C4 diagrams to identify a vertical slice of components that realize the flow of events
- Identify a decomposition strategy
- Perform SNAP analysis on each of the components of the vertical slice
- Generate a backlog of user stories and tie it back to the OKRs
- User story mapping to map user stories to MVPs

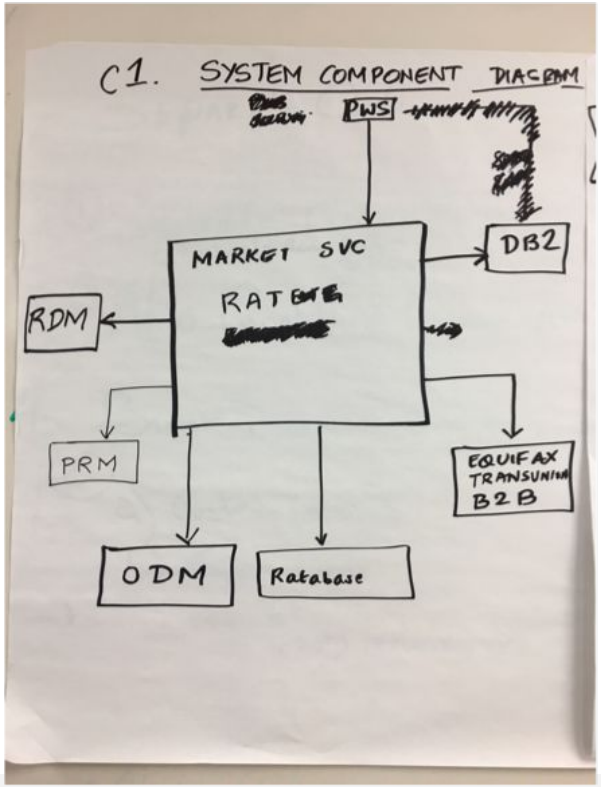


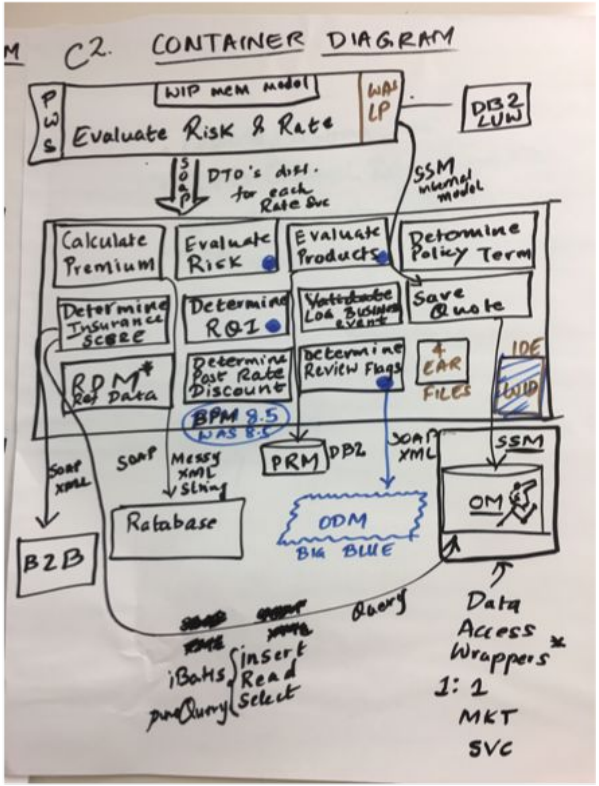
Selection of a Vertical Slice : Pick a Core Domain

Pivotal

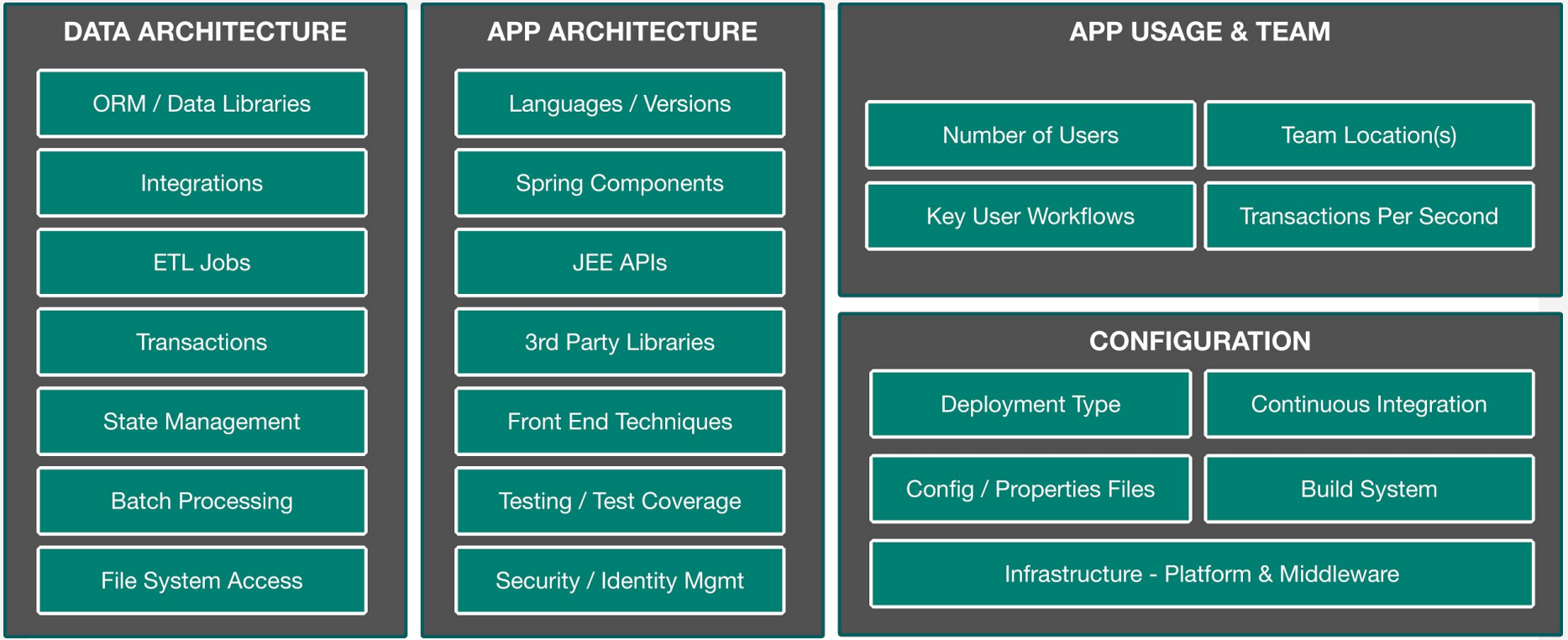
C4 Architecture Diagrams



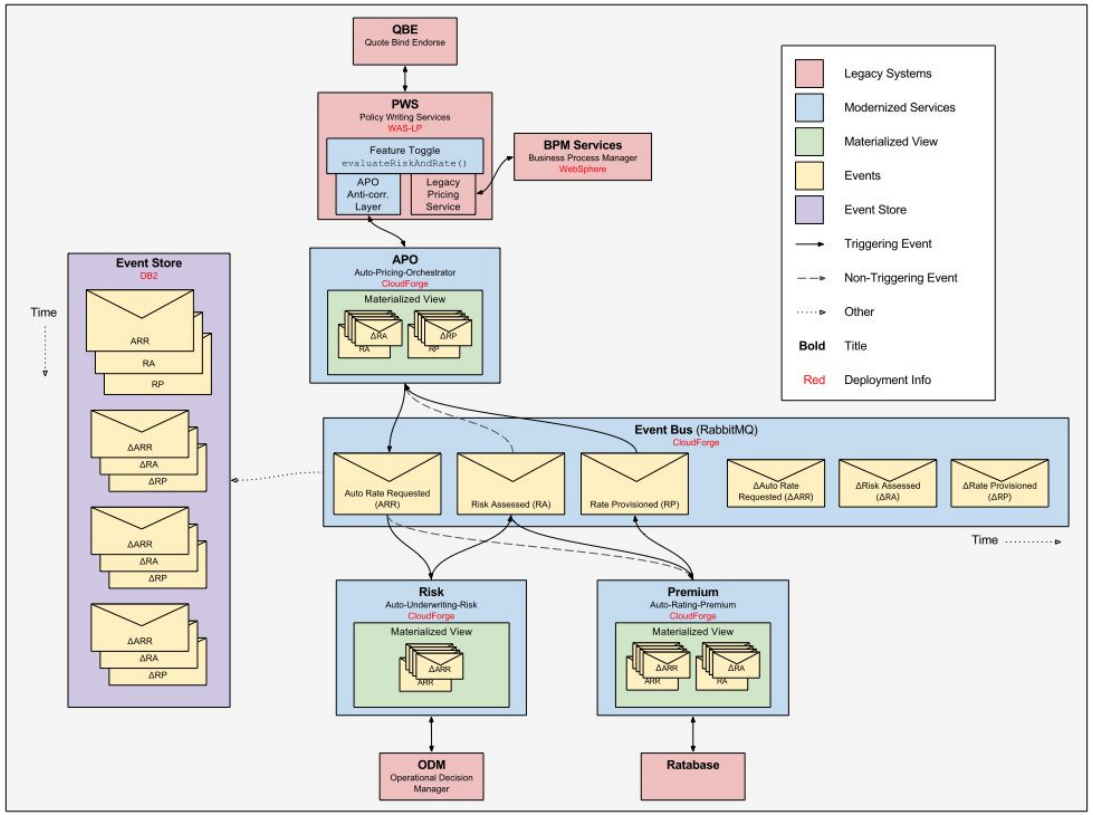


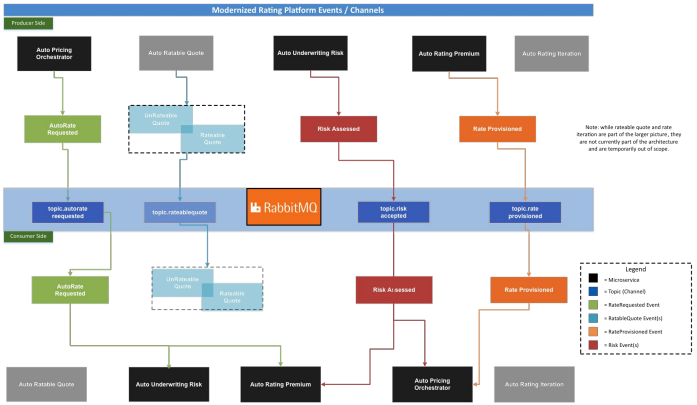


SNAP Analysis



1M LOC Insurance Domain Monolith





Pivotal

What is your Decomposition Strategy ?

Bounded Contexts

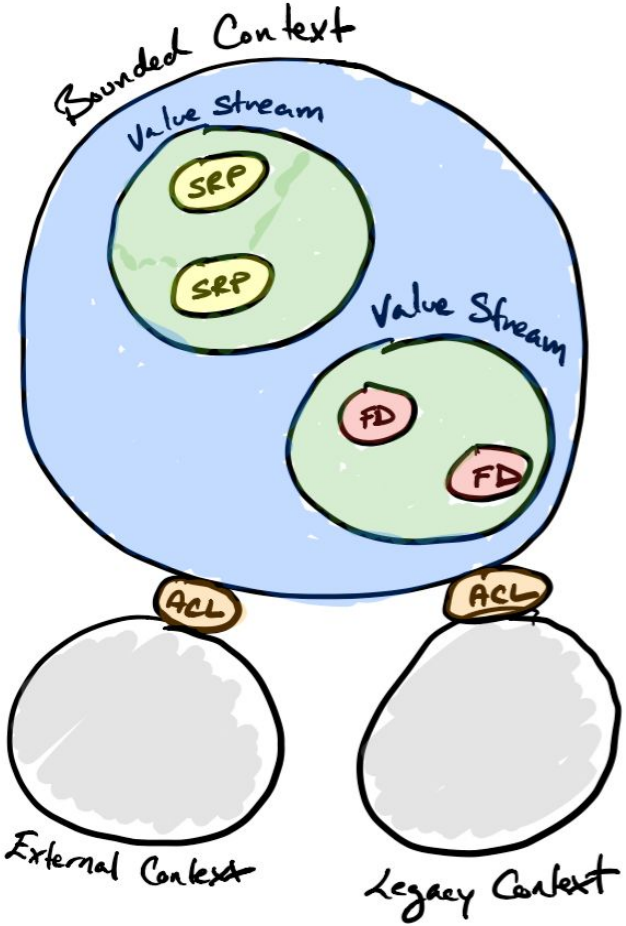
Value Streams

Failure Domains

Anti-Corruption Layers

Single Responsibility Principle

Kitchen Sink



Event Thinking

Reduce temporal coupling

- Instrument Monolith with Events
- Break Things apart with events
- Use domain language for events
- Build things on the outside
- Strangler Pattern
- Minimally Invasive
- Cannot rely on ordering of messages or zero message loss

Strive for autonomy with messaging

- Messaging schemas can introduce coupling
- Sagas handle Eventual Consistency
- What operations are idempotent ?
- Handle duplicate messages with Outbox pattern
- Messaging schemas can introduce coupling
- Change in mindset
- Avoid Kafka as an ESB anti-pattern
- Spring Application Events or CDI Events

Many Meanings of Event Driven Architecture (EDA)

Event Notification

- Send event messages to notify other systems of a change in its domain
- Event carries ID and link back to the sender for more info
- Decouples sender from receiver
- No Statement of overall behavior

Event Carried State Transfer

- Update clients in a way that they don't need to contact the source system
- Receiver has to maintaining all the state
- Decoupling
- Reduced load on supplier
- Replicated Data
- Eventual Consistency

Event Sourcing

- Every state change of the system is recorded as an event
- Event store becomes the source of truth, and the system state is derived from it
- Audit, Debugging, Historic State, Alternative State, Memory Image
- Unfamiliar, External Systems, Event Schema, Identifiers, Asynchrony, Versioning

Life Beyond Distributed Transactions

How To Deal With 2PC XA Global Transactions in the Cloud

Pat Helland

http://queue.acm.org/detail.cfm?id=3025012

Pivotal

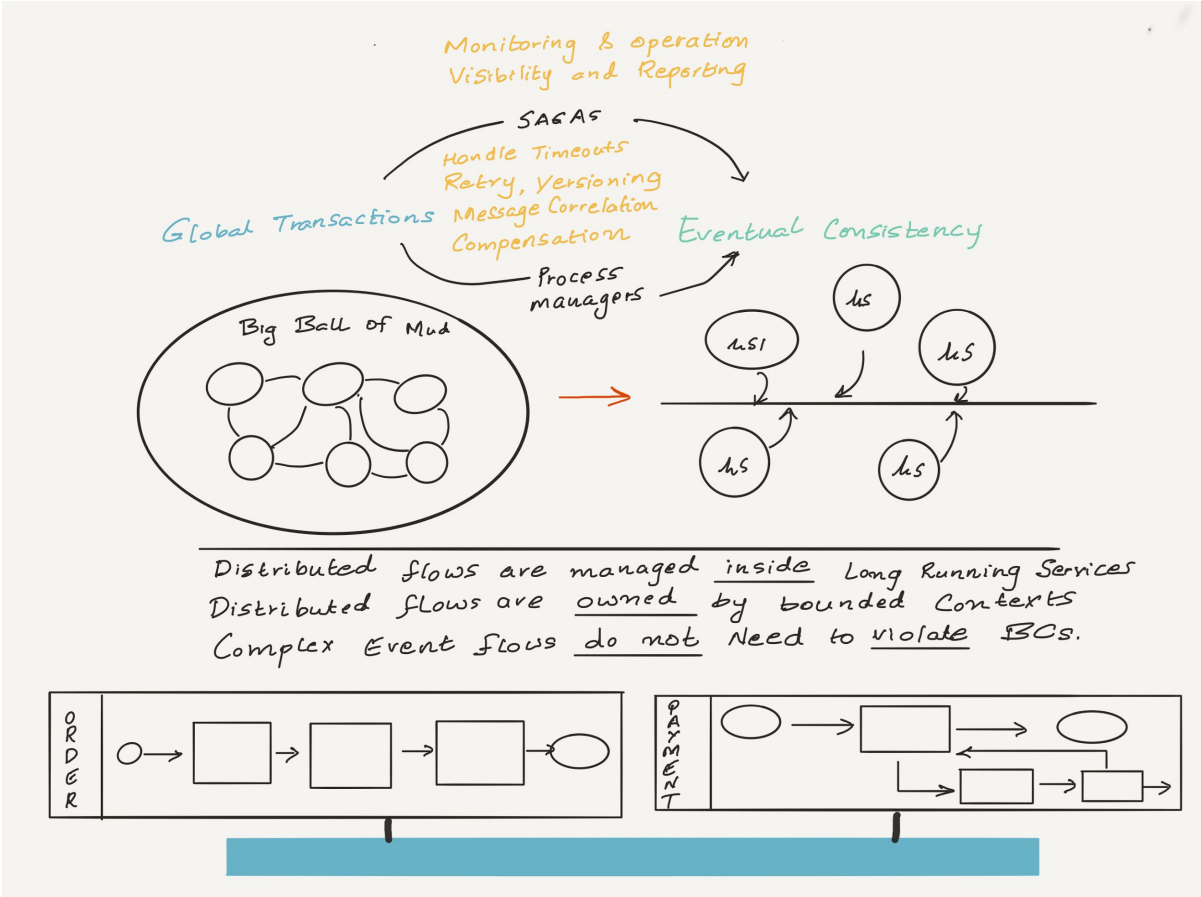
In a system that cannot count on distributed transactions, the management of uncertainty must be implemented in the business logic.

The uncertainty of the outcome is held in the business semantics rather than in the record lock. This is simply workflow.

We can’t use distributed transaction so we need to use workflow.

Demons Lie Here

Complex Event Flows



Should you Event Source or Not ?

Should I invest in the stock market now ?

Every team is at liberty to select their own patterns

Is there tooling available to guide you

Do you have developers with the aptitude and ability

Does data need to be persisted temporally?

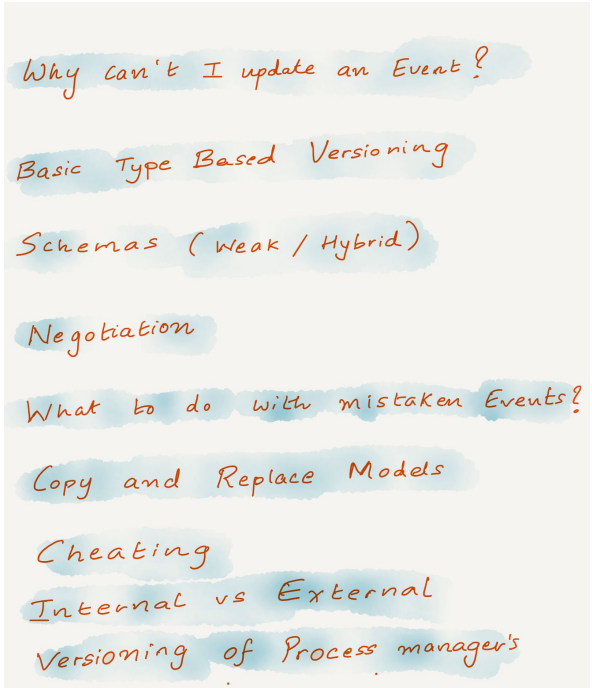
Is an audit log necessary ?

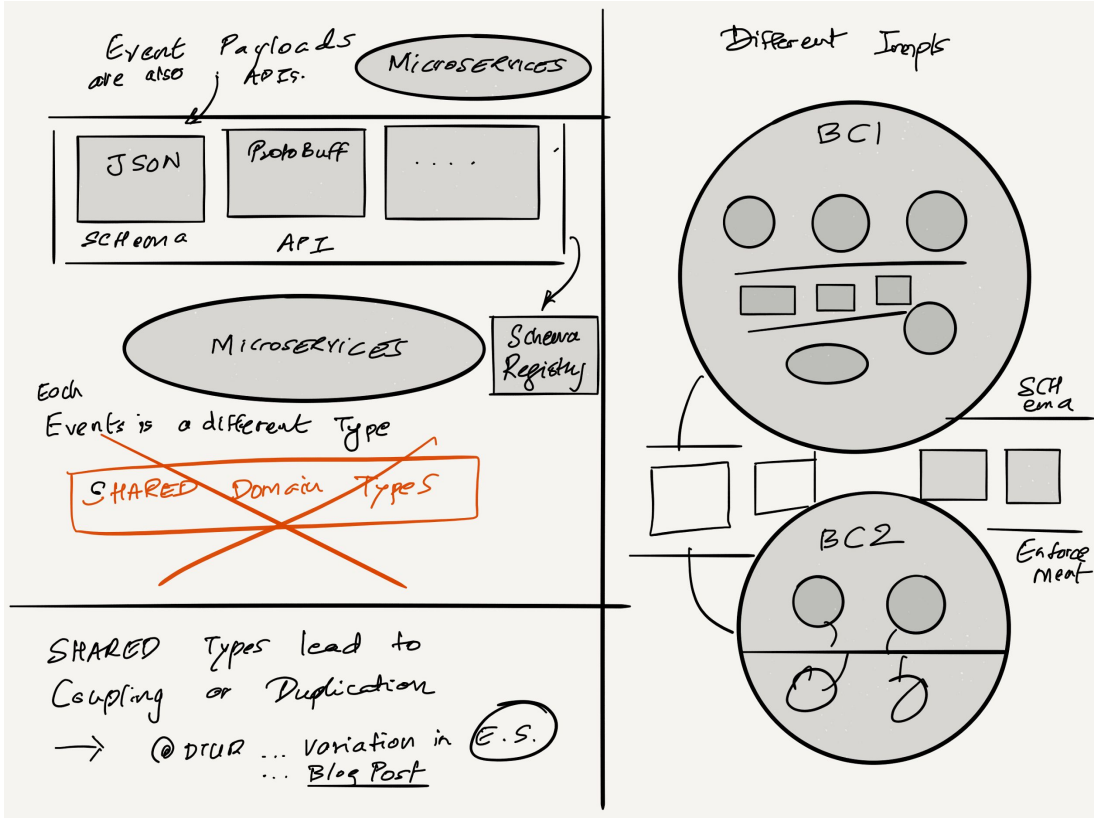
What is the overall system architecture ?

Event Driven Architecture Challenges

Versioning in an Event Sourced System

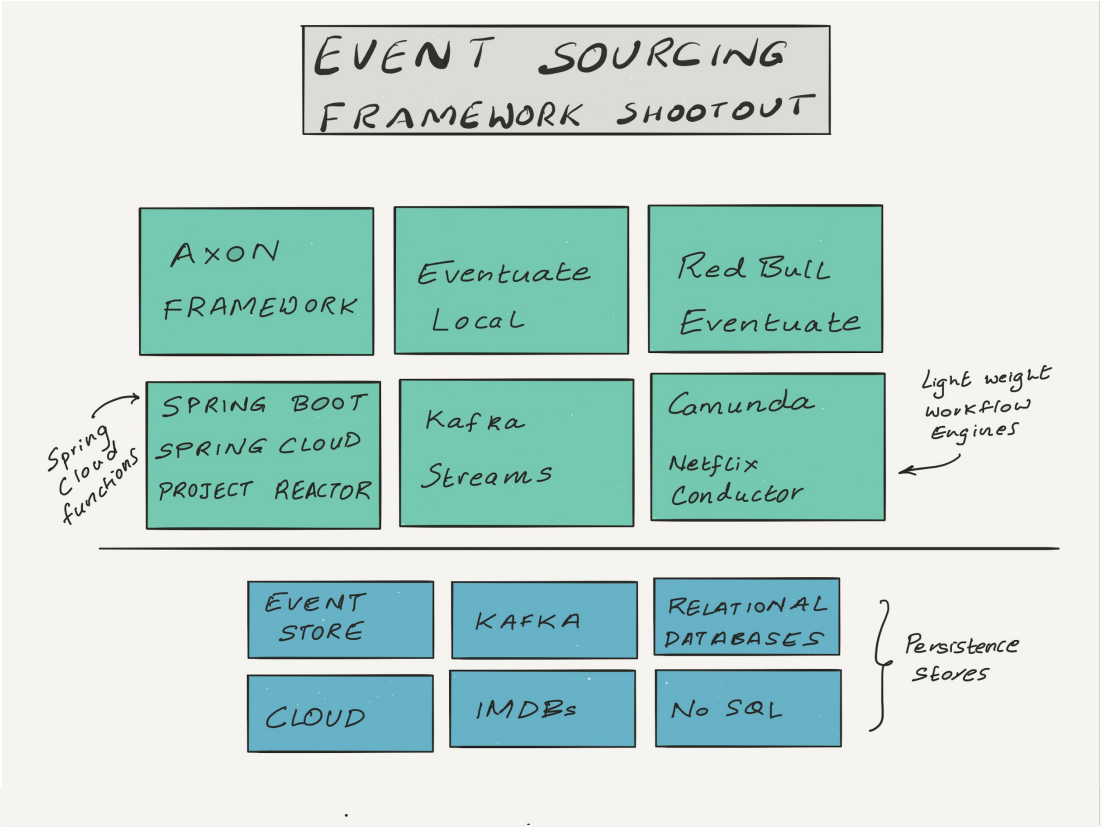
Greg Young https://leanpub.com/esversioning





Pivotal

Event Sourcing Framework Shootout



Lessons Learned From Real World Modernizations

Too many fine bounded contexts.

Correlation strategy and Lock Strategy get complicated

Complex work flows require Sagas & Process Managers

Microservices - one per aggregate ?

DIY Event Sourcing systems are hard. Too much time spent in toil

Leverage Consumer driven Contracts to establish clear schemas across Bounded Contexts for messages

Chose the right style of Event Driven Architecture

Delay making frameworks

Powerpoint is dead long live the Stickies

Spring is missing an event sourcing and event persistence framework

Event Sourcing is hard without a supporting event persistence framework or library

Pick a framework based on the guidance on the next slide before beginning

Challenge preconceived architectural decisions

Set goals with measurable metrics.

Be wary of the Hourglass effect

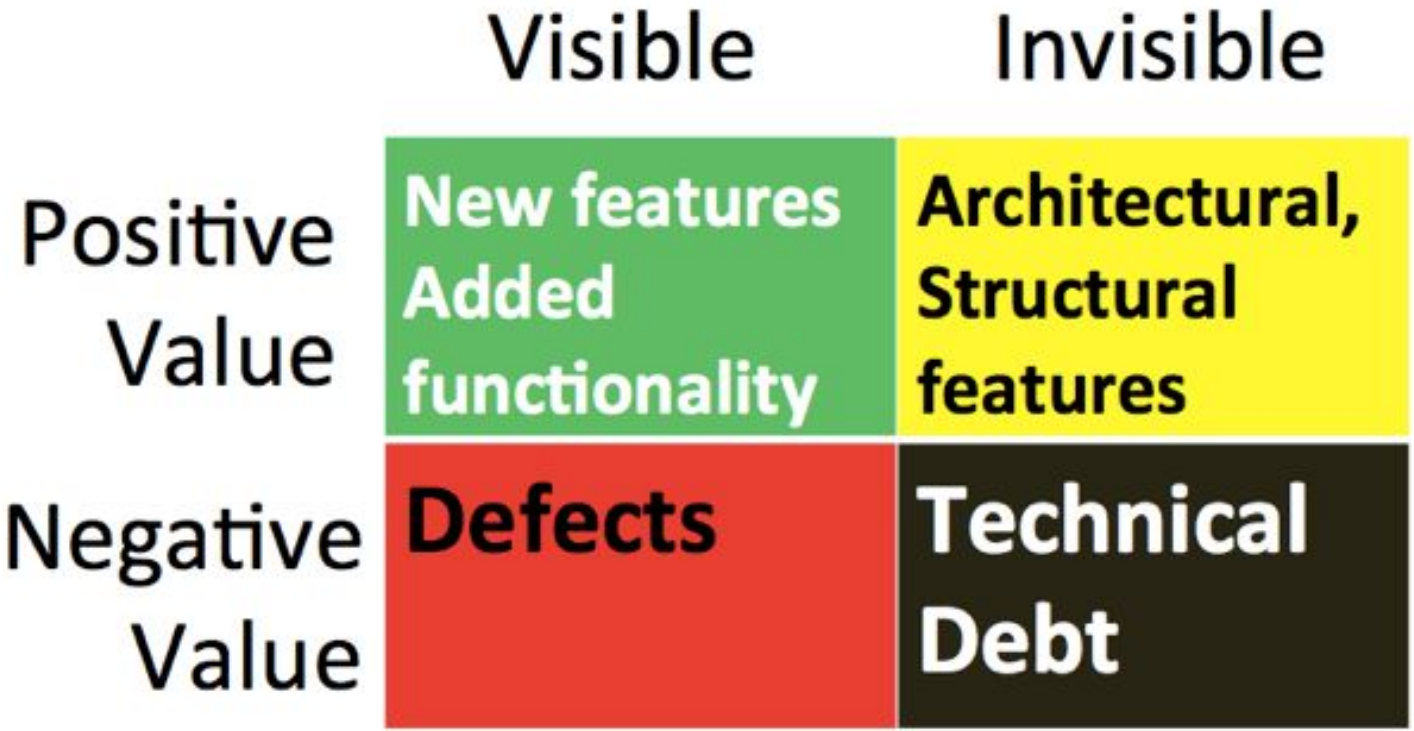
Part 3
Real World Experiences

Our Experiences with...

- Technical Debt
- Monoliths
- Entropy
- Event Storming
- Breaking Down Monoliths for real
- MSA w/DDD to API-First
- 12 factors & beyond

Pivotal

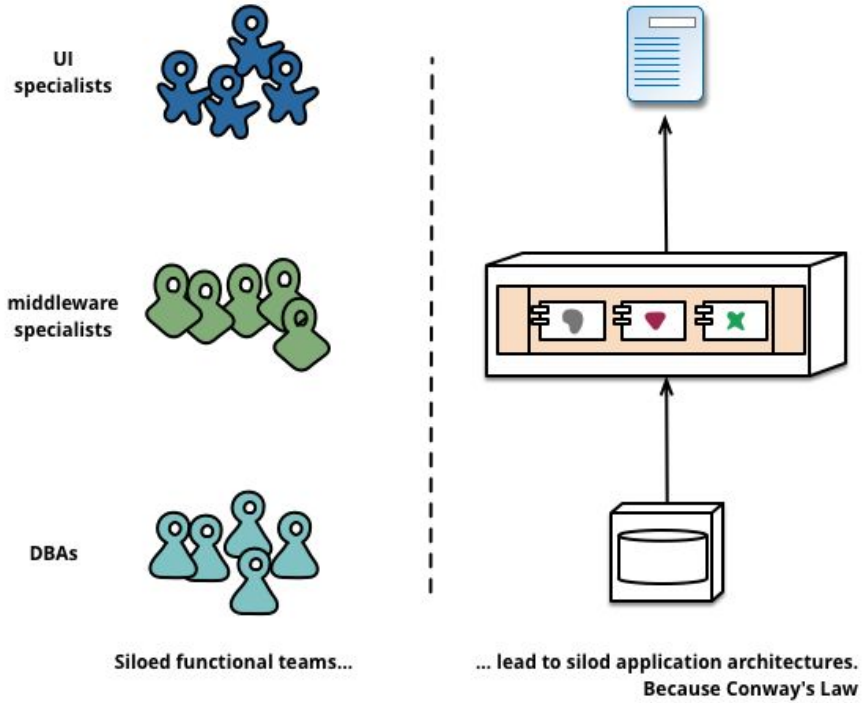
Technical Debt is Invisible but Real



Source: https://philippe.kruchten.com/

Conway's Law holds true

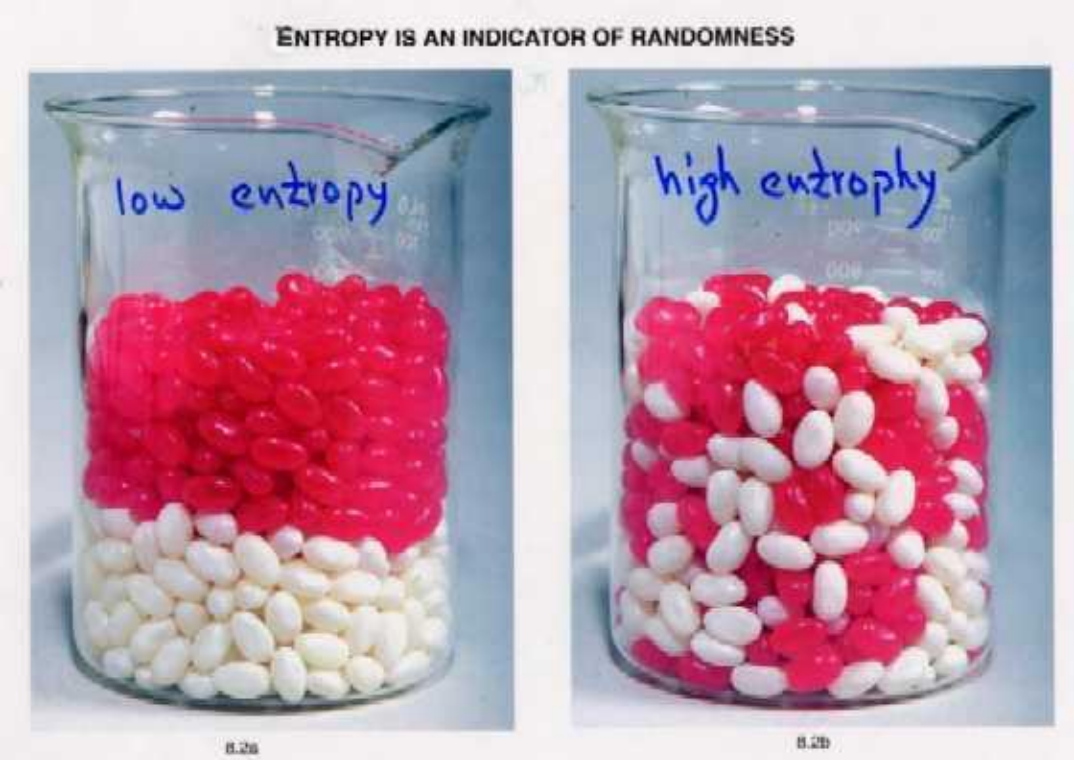
"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."
- Conway's Law



Source: http://melconway.com/Home/Conways_Law.html

Source: https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture

Entropy always tends to increase within monoliths



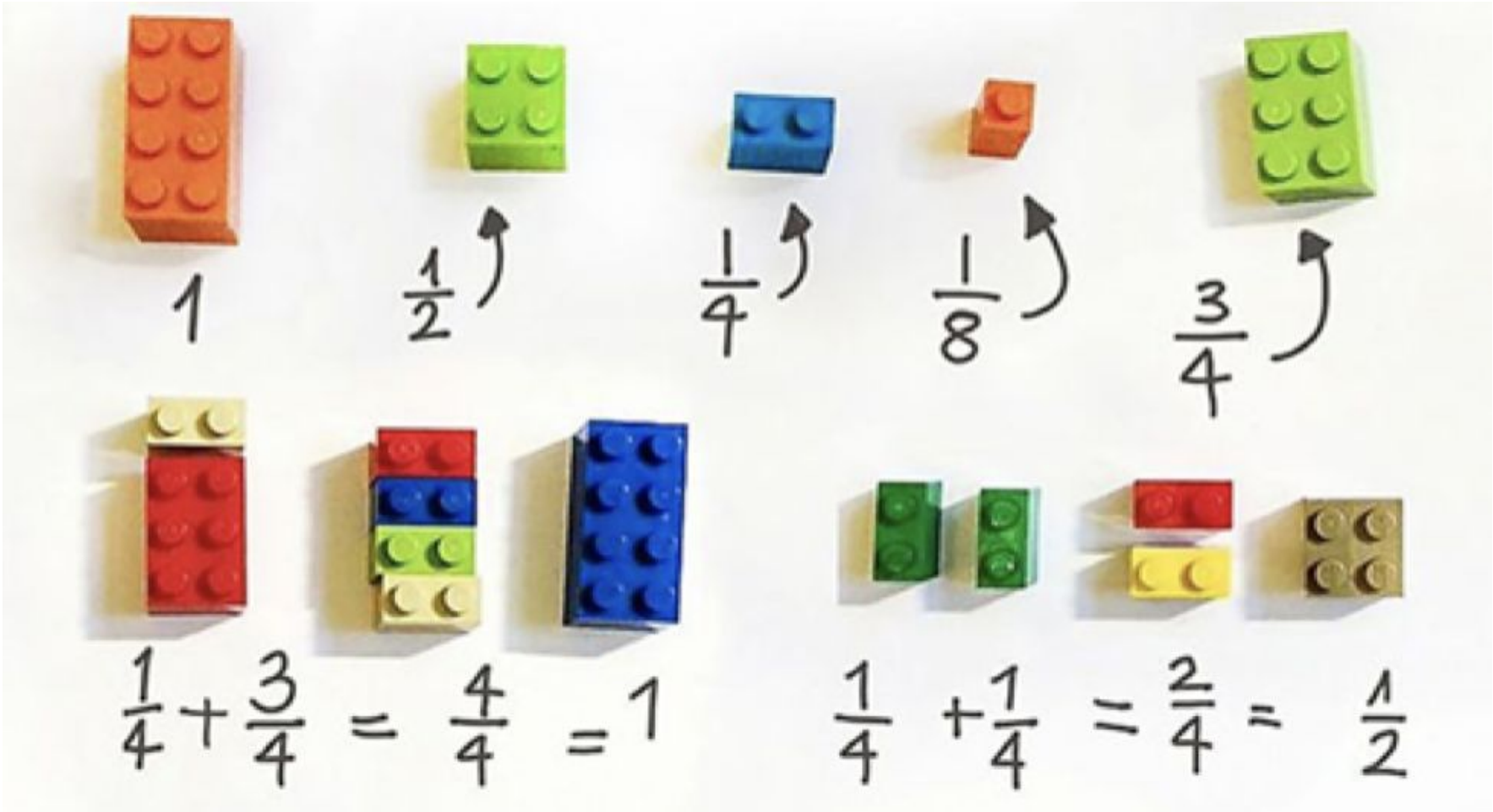
Source: http://facstaff.gpc.edu/~mkim/C1211&1212Lec/C1212_Lecture.htm

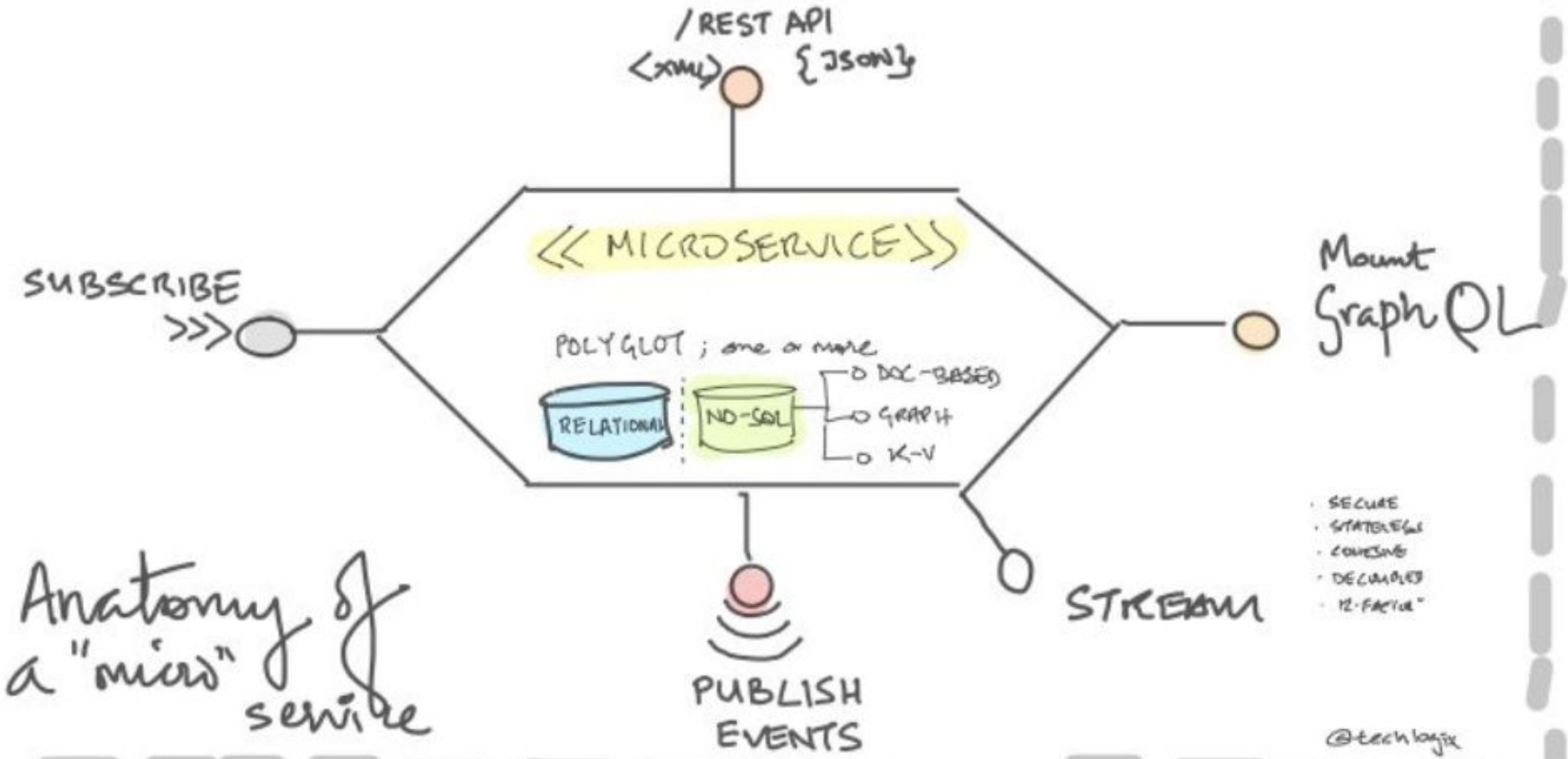
Microservices

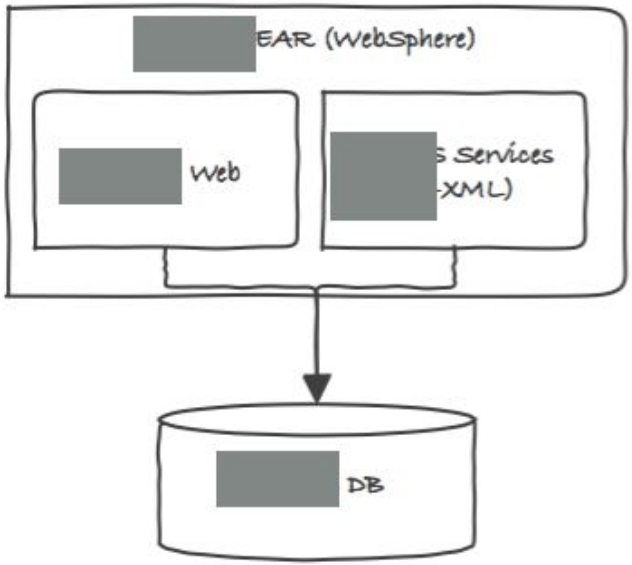
"In short, the micro-service **architectural style** is an approach to **developing a single application** as a suite of **small services**, each running in its **own process** and **communicating** with lightweight mechanisms, often an HTTP resource **API**.

These services are built around **business capabilities** and **independently deployable** by fully **automated deployment** machinery. There is a bare **minimum of centralized management** of these services, which may be written in **different programming languages** and use **different data storage technologies**."

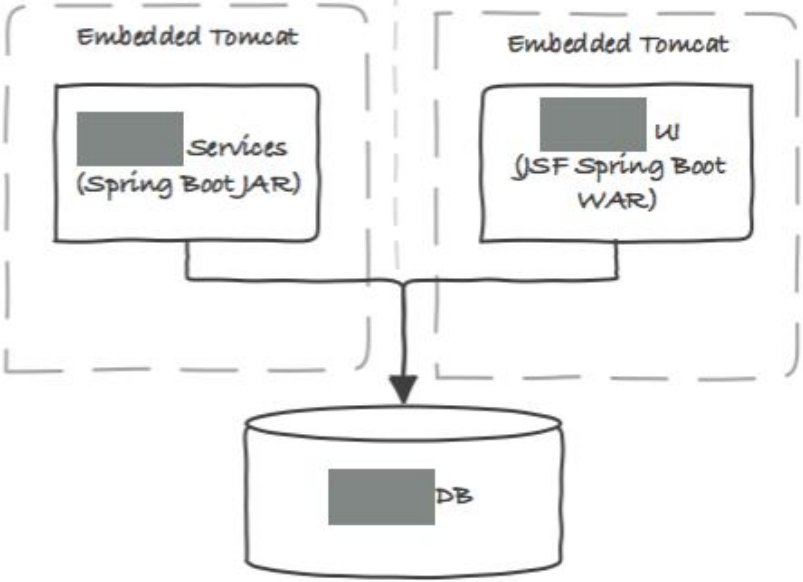
-Fowler, Lewis et al


1
1/2
1/4
1/8
3/4
1/4 + 3/4 = 4/4 = 1
1/4 + 1/4 = 2/4 = 1/2


/REST API
<xml>
{JSON}
<< MICROSERVICE >>
SUBSCRIBE
>>>
POLYGLOT ; one or more
RELATIONAL
NO-SQL
DOC-BASED
GRAPH
K-V
Mount
GraphQL
STREAM
PUBLISH
EVENTS
Anatomy of
a "micro"
service
@techlogix



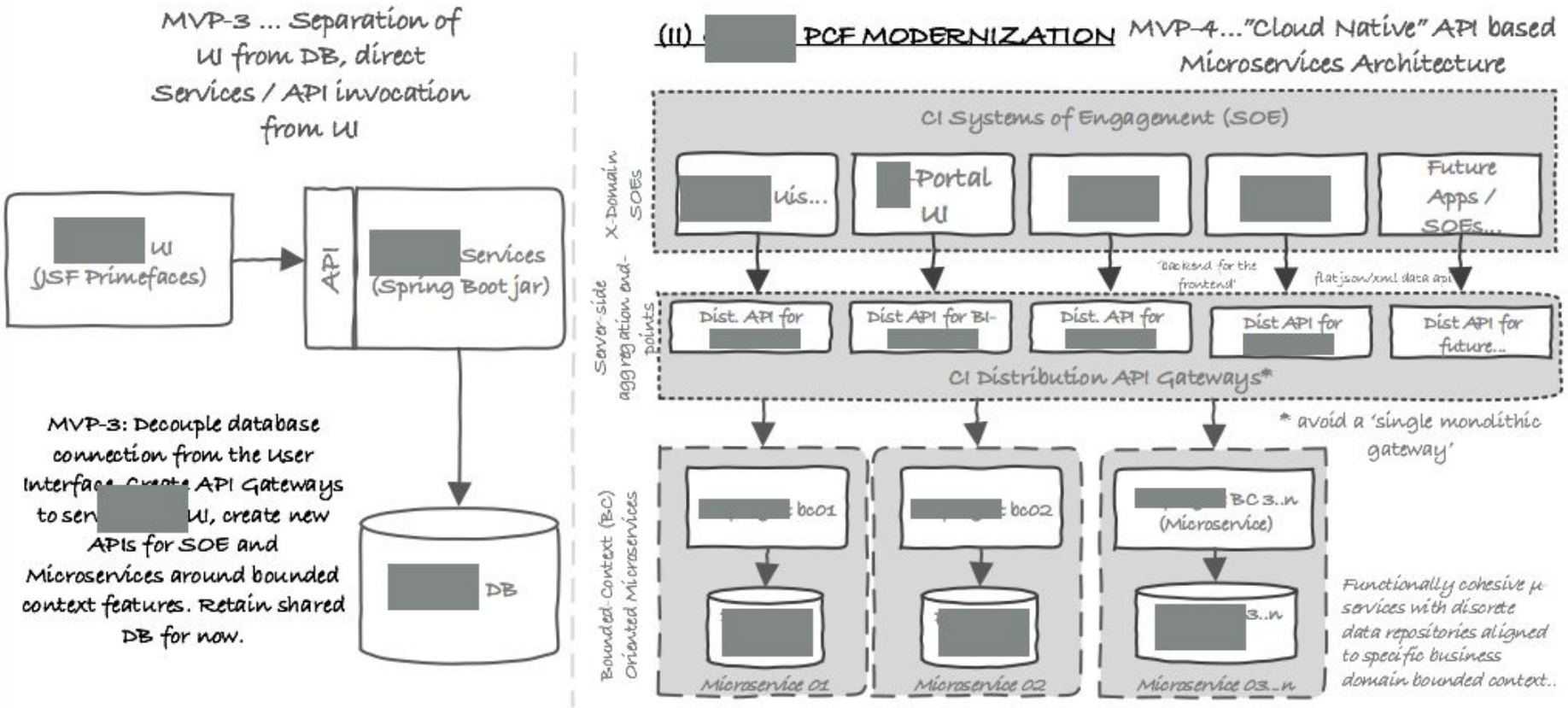
Starting with a monolithic architecture -WAS EAR with a Shared DB between a Web App & Services within a monolith.



Breaking down the monolith application architecture. First step: Split EAR into 2 deployments, one WAR and a jar into separate build pipelines. Keep DB as-is for replatforming. Each split is an MVP that runs separately.

Logical Separation

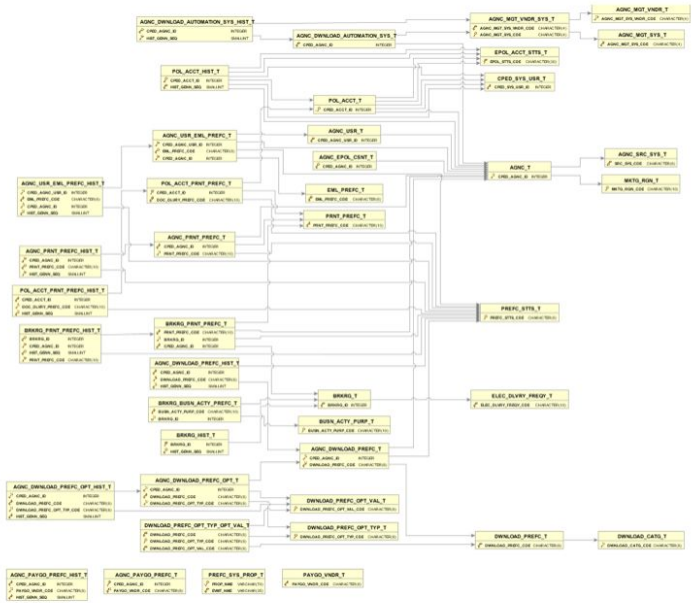
Physical Separation



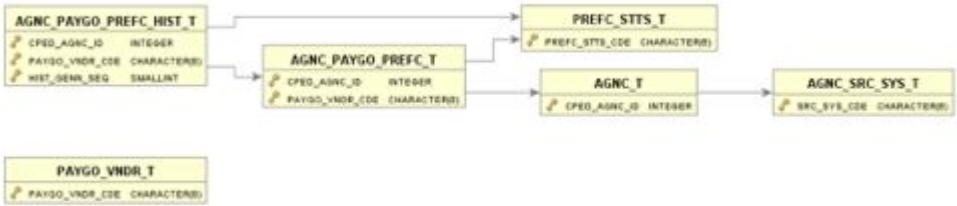
Decoupled UI

Microservices

Pivotal



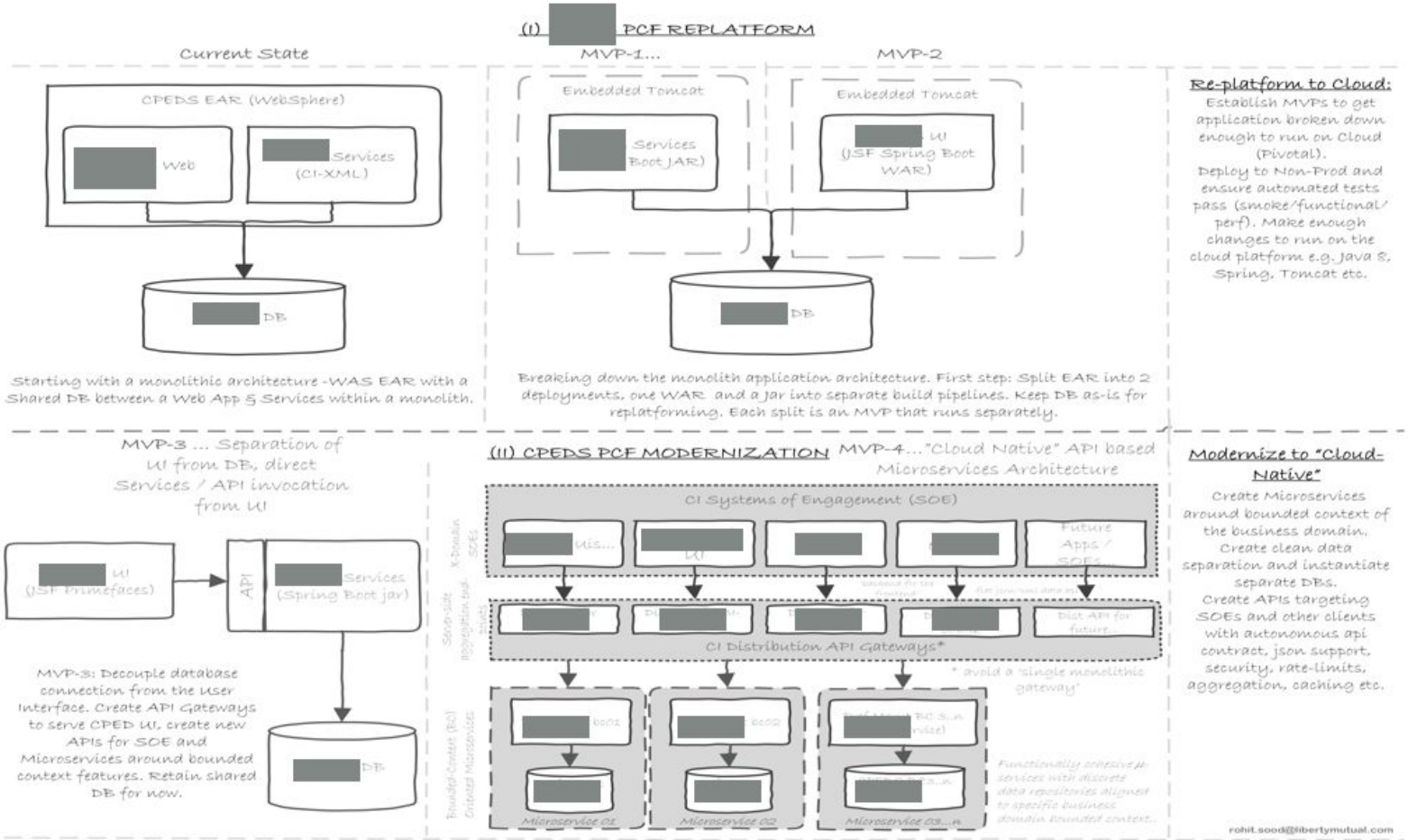
Before



After *

* one bc agg

Pivotal

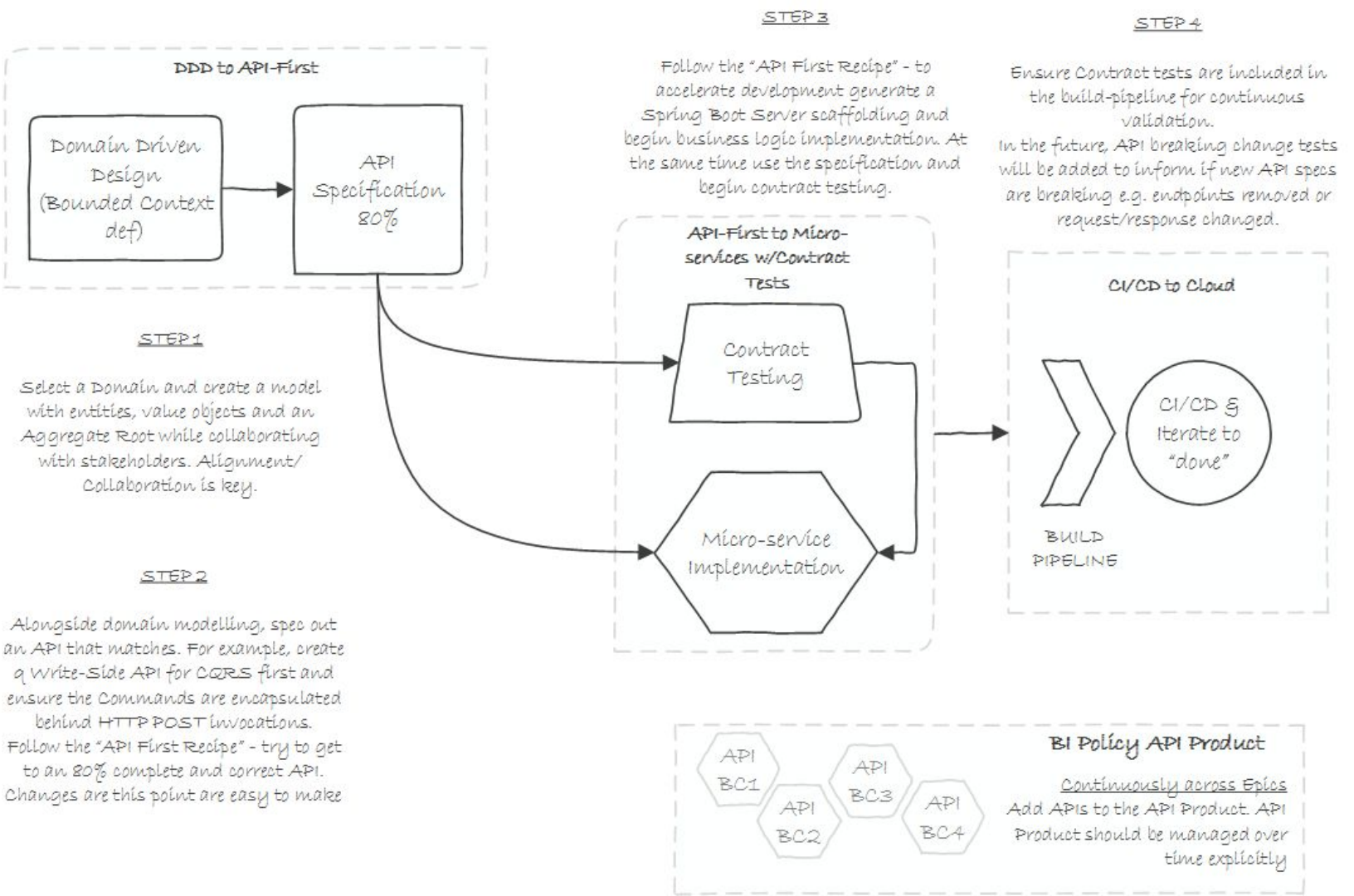

(I) [redacted] PCF REPLATFORM
Current State
MVP-1...
MVP-2
CPEDS EAR (WebSphere)
Web
Services (CI-XML)
DB
Embedded Tomcat
Services Boot JAR)
Embedded Tomcat
UI (JSF Spring Boot WAR)
DB
Re-platform to Cloud:
Establish MVPs to get application broken down enough to run on Cloud (Pivotal).
Deploy to Non-Prod and ensure automated tests pass (smoke/functional/perf). Make enough changes to run on the cloud platform e.g. Java 8, Spring, Tomcat etc.
Starting with a monolithic architecture -WAS EAR with a Shared DB between a Web App & Services within a monolith.
Breaking down the monolith application architecture. First step: Split EAR into 2 deployments, one WAR and a jar into separate build pipelines. Keep DB as-is for replatforming. Each split is an MVP that runs separately.
MVP-3 ... Separation of UI from DB, direct Services / API invocation from UI
UI (JSF Primefaces)
API
Services (Spring Boot jar)
DB
MVP-3: Decouple database connection from the User Interface. Create API Gateways to serve CPED UI, create new APIs for SOE and Microservices around bounded context features. Retain shared DB for now.
(II) CPEDS PCF MODERNIZATION
MVP-4..."Cloud Native" API based Microservices Architecture
CI Systems of Engagement (SOE)
UIs...
Future Apps / SOEs...
CI Distribution API Gateways*
Dist API for future
* avoid a 'single monolithic gateway'
Server-side aggregation pull
X-Domain SOEs
Bounded-Context (BC) Oriented Microservices
Microservice 01
Microservice 02
Microservice 03...n
Functionally cohesive µ-services with discrete data repositories aligned to specific business domain bounded context...
Modernize to "Cloud-Native"
Create Microservices around bounded context of the business domain.
Create clean data separation and instantiate separate DBs.
Create APIs targeting SOEs and other clients with autonomous api contract, json support, security, rate-limits, aggregation, caching etc.
rohit.sood@libertymutual.com

Pivotal

Pivotal

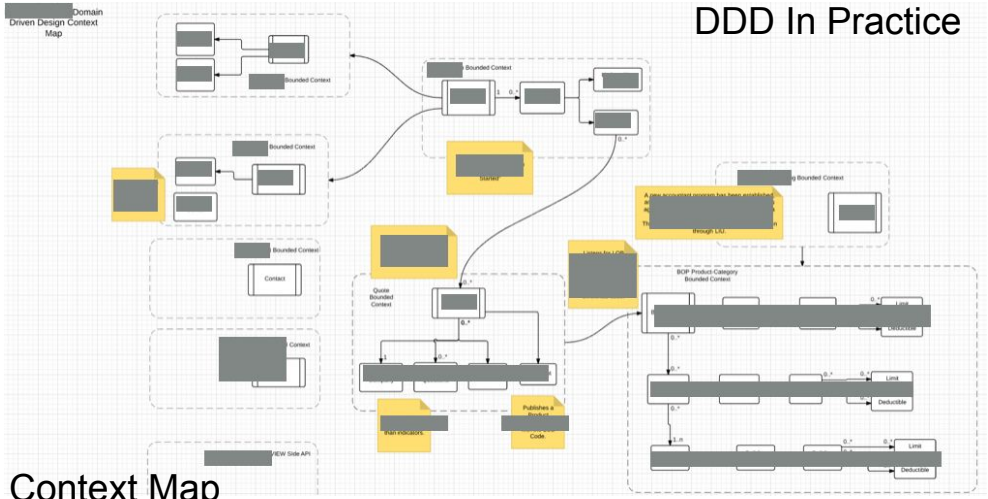
Weeks of coding can save you hours of designing.

Pivotal

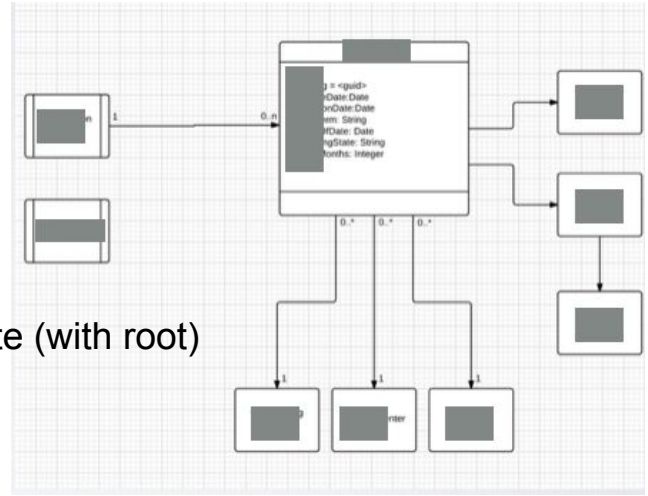

DDD to API-First
Domain Driven Design (Bounded Context def)
API Specification 80%
STEP 1
Select a Domain and create a model with entities, value objects and an Aggregate Root while collaborating with stakeholders. Alignment/ Collaboration is key.
STEP 2
Alongside domain modelling, spec out an API that matches. For example, create a Write-Side API for CQRS first and ensure the Commands are encapsulated behind HTTP POST invocations. Follow the "API First Recipe" - try to get to an 80% complete and correct API. Changes are this point are easy to make
STEP 3
Follow the "API First Recipe" - to accelerate development generate a Spring Boot Server scaffolding and begin business logic implementation. At the same time use the specification and begin contract testing.
API-First to Micro-services w/Contract Tests
Contract Testing
Micro-service Implementation
STEP 4
Ensure Contract tests are included in the build-pipeline for continuous validation.
In the future, API breaking change tests will be added to inform if new API specs are breaking e.g. endpoints removed or request/response changed.
CI/CD to Cloud
CI/CD & Iterate to "done"
BUILD PIPELINE
API BC1
API BC2
API BC3
API BC4
BI Policy API Product
Continuously across Epics
Add APIs to the API Product. API Product should be managed over time explicitly

Pivotal

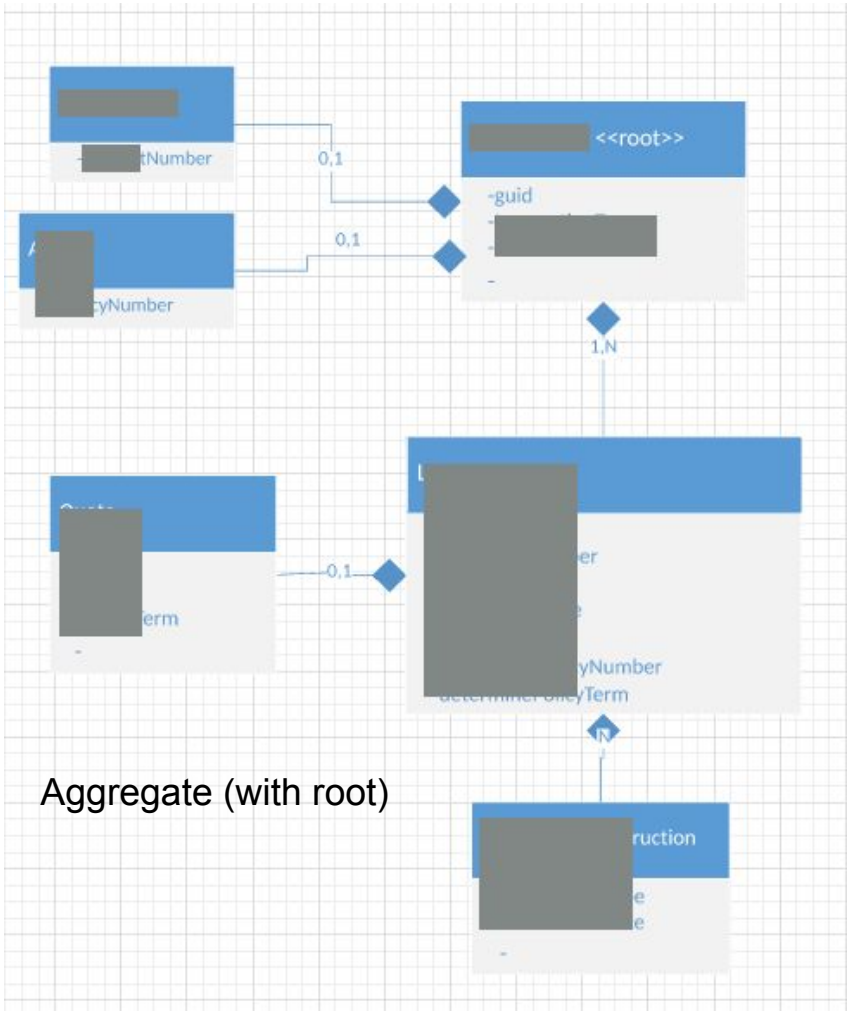
DDD In Practice



Context Map

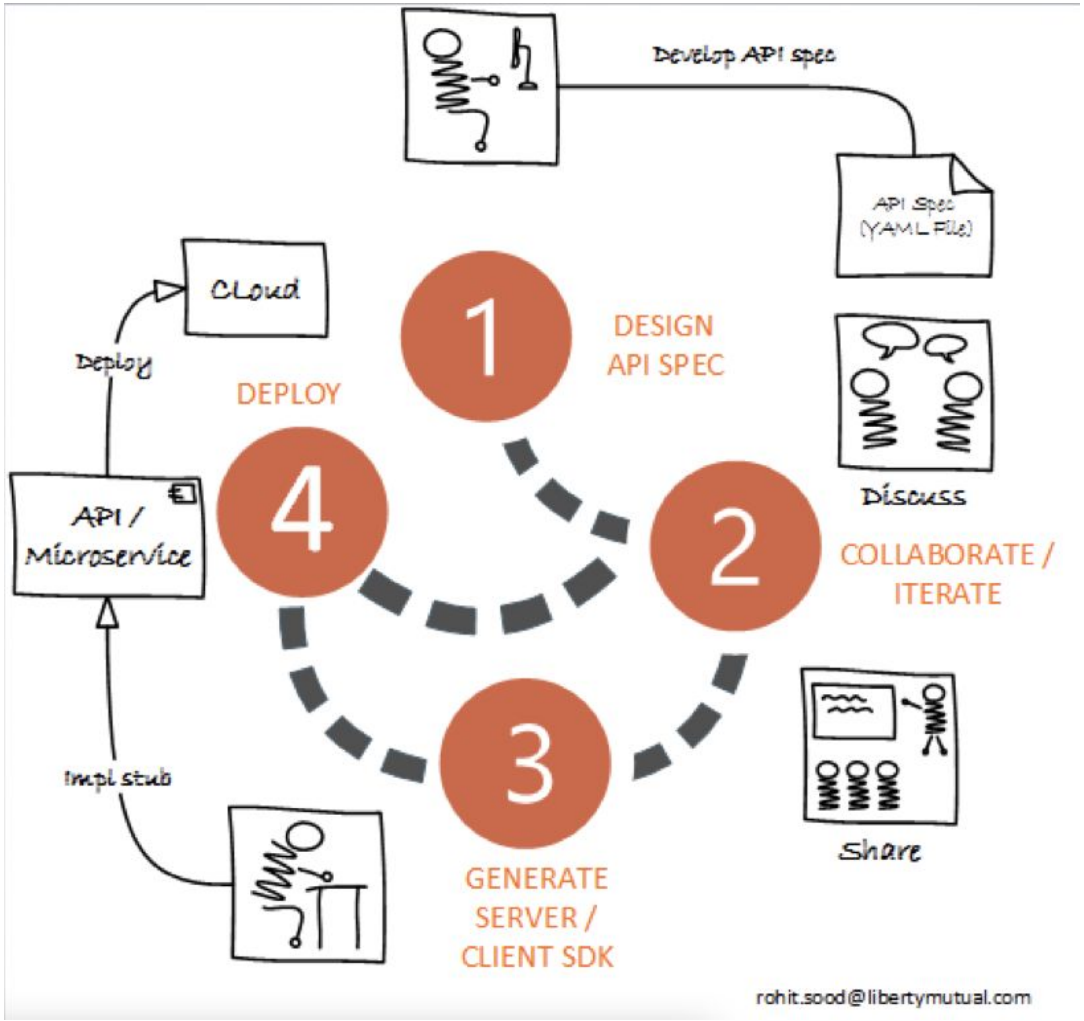


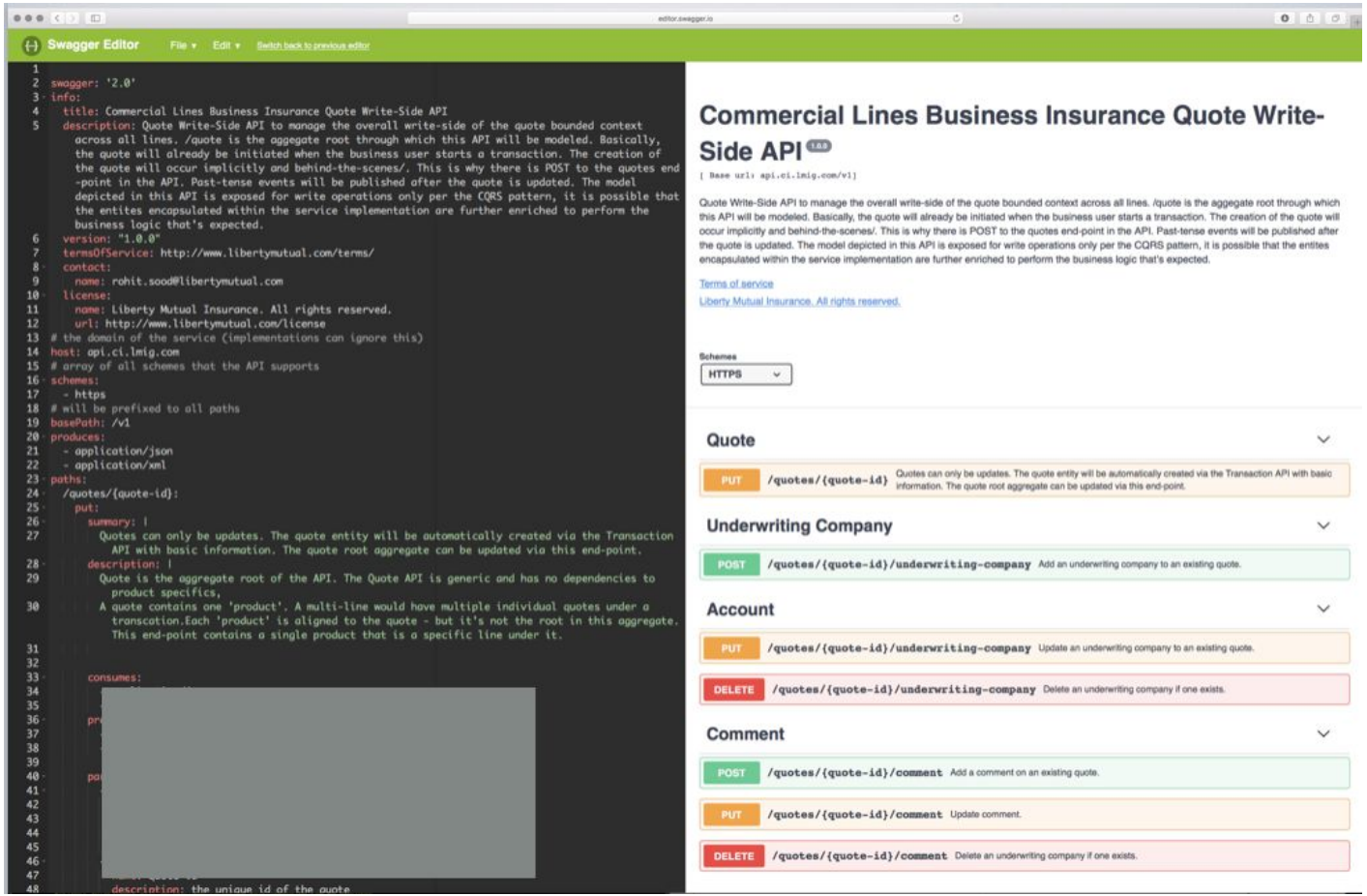
Aggregate (with root)



Aggregate (with root)

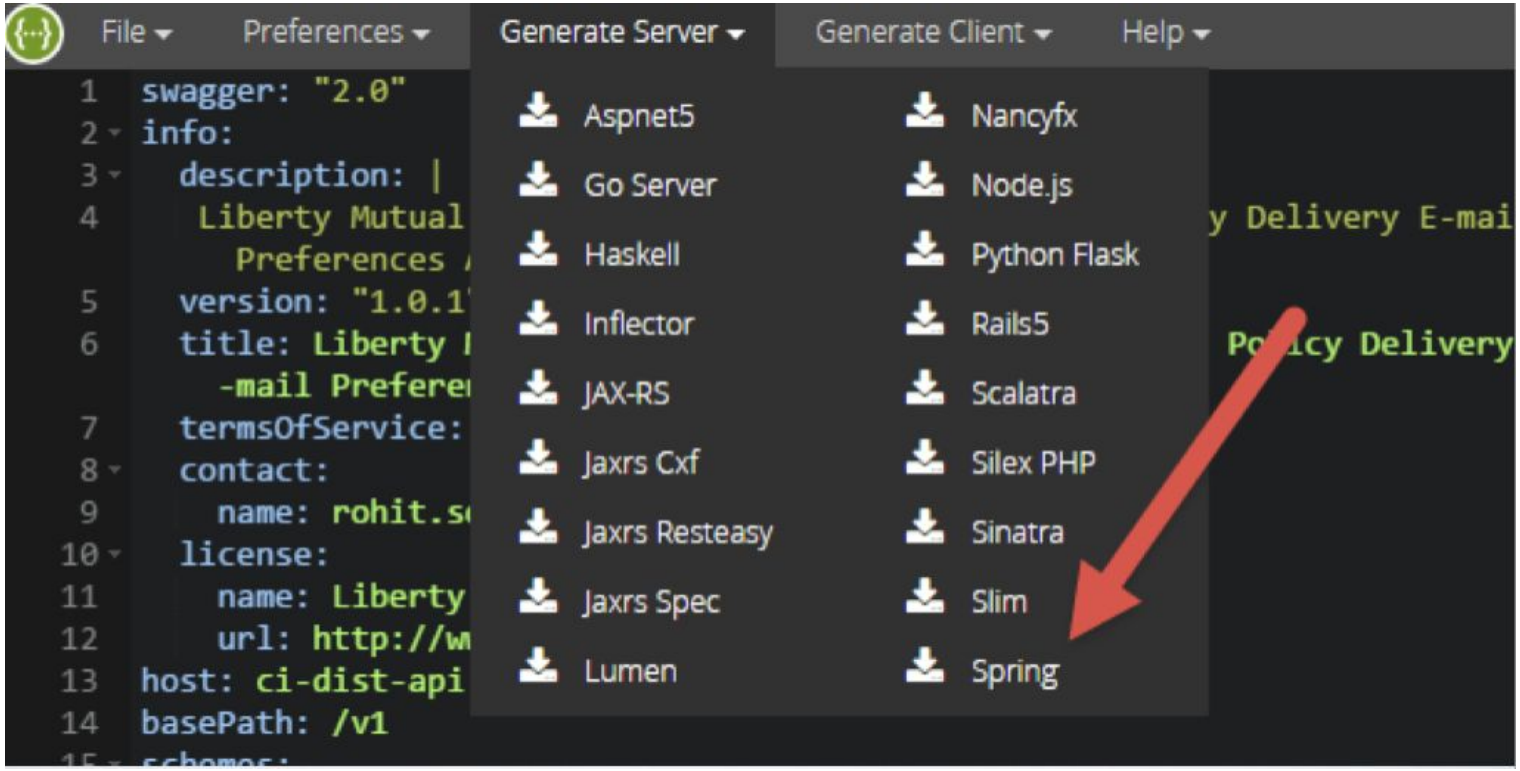
Pivotal


Develop API spec
API Spec
(YAML File)
Cloud
Deploy
1
DESIGN
API SPEC
DEPLOY
4
2
COLLABORATE /
ITERATE
Discuss
API /
Microservice
3
Impl stub
Share
GENERATE
SERVER /
CLIENT SDK
rohit.sood@libertymutual.com

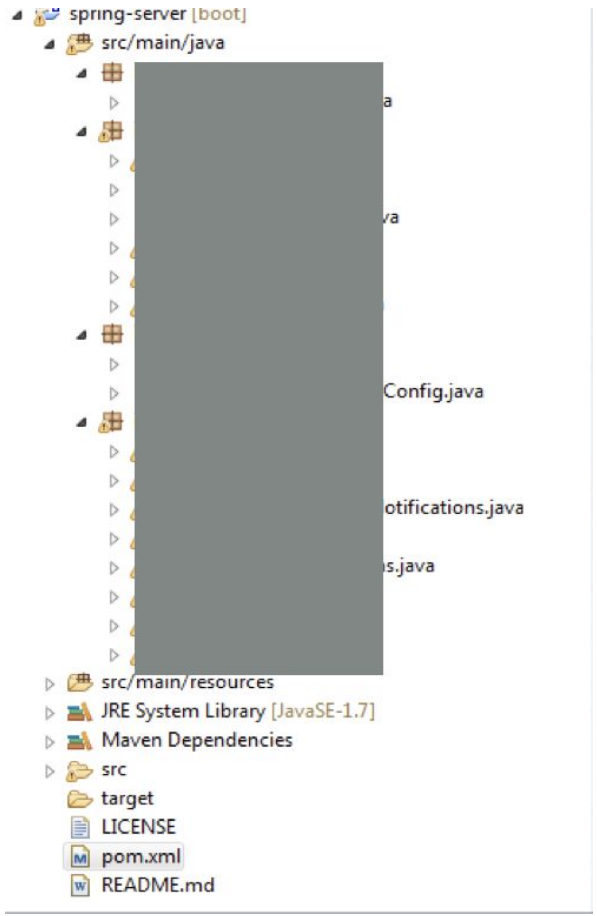


1. Design API Spec
2. Collaborate & Refactor

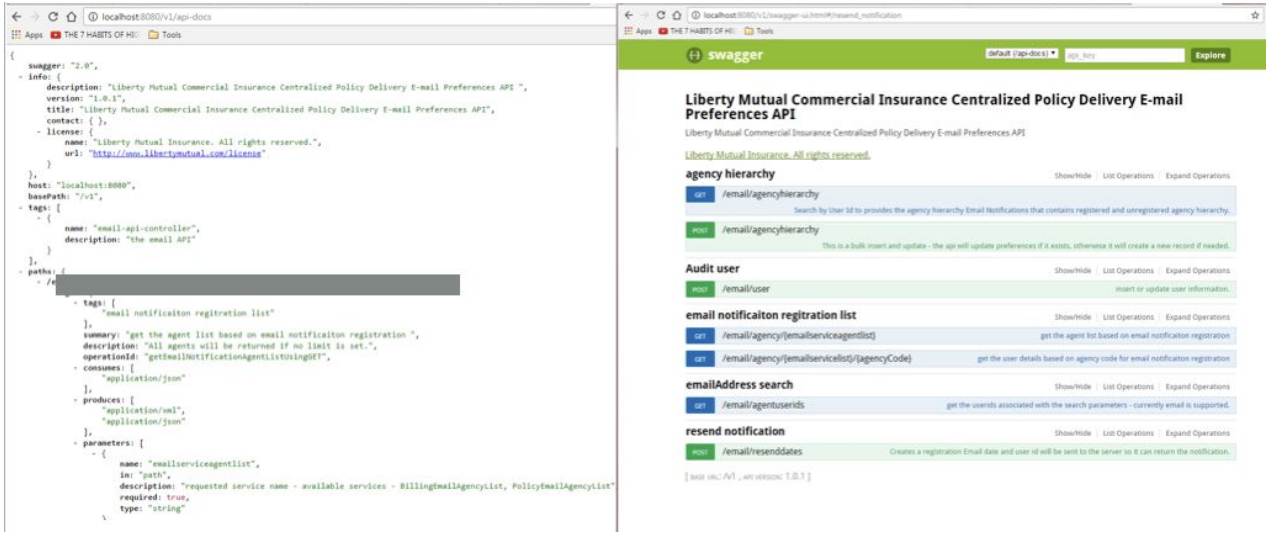
Pivotal



3. Generate Server (starter)



4. Refactor/Implement Logic/Test



5. Run

Pivotal

Experiences & Observations

Technical Debt is real but invisible.
Monoliths can lead to technical debt.
System Entropy always wins.
Conway's Law is true.
Microservices is not just about small services.
12 factors are important for CNA.
Go beyond the 12 factors - adopt API-First.
Breaking down monoliths with DDD.
Event Storming is a good Kickstarter to DDD.
Domain driven design should help lead to APIs first.
CQRS is a powerful pattern.
Event Sourcing is a difficult pattern - not for the faint of heart.
New technologies can bring new unknowns.
Pair programming worked very well.

Conclusions

- EDA is not the same as Event Sourcing
- Dealing with consistency is HARD.
- How do you recover ?
- Hours of design can save weeks of coding
- Pick your core architectural style
- Prefer the evolutionary approach to architecture
- Address the people challenges before the technical ones
- Event Storming can help you identify your core domain