Architecture principles, how to, patterns, examples

INTRODUCTION TO MICROSERVICES

October 2015, @SteveSfartz

Agenda

- Why Microservices ?
- Designing Microservices
- Microservices Patterns
- Netflix stack
- Comparing SOA styles
- Sump up

Why, Monolithic style, Scaling the Monolith

MICROSERVICES ARCHITECTURE

Microservices (2014)

In short, the microservice 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.

Martin Fowler, James Lewis

THE MONOLITHIC STYLE

- An application built as a single unit, 3 main parts
  - a client-side user interface ,
  - a database
  - and a server-side application that handles HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser.

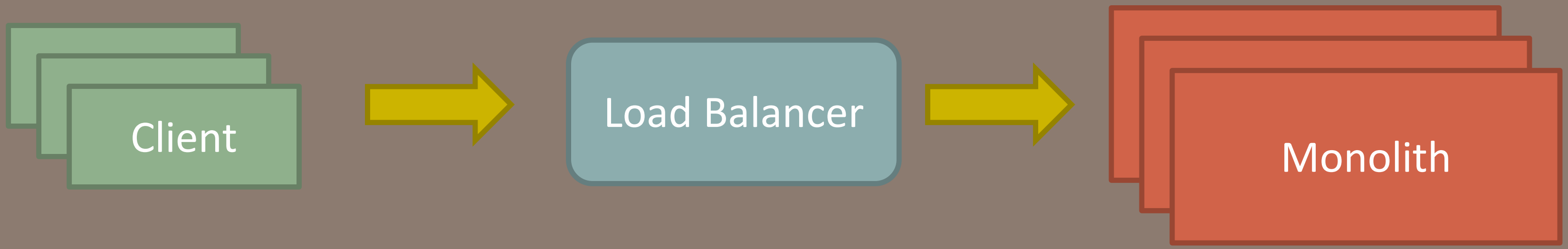
- This server-side application is a *monolith*
  - a single **logical** executable
  - any change requires building and deploying a new version

THE MONOLITHIC STYLE

- Natural way to build a system
  - code specific to each layer (UI, Logic, DB)
  - generally a minimum of 3 languages
  - divided into classes, functions and namespaces
- Locally built and tested on devs’ machines
- CI/CD pipeline to secure production

Scaling the Monolith

- Several Monolith instances behind a load balancer

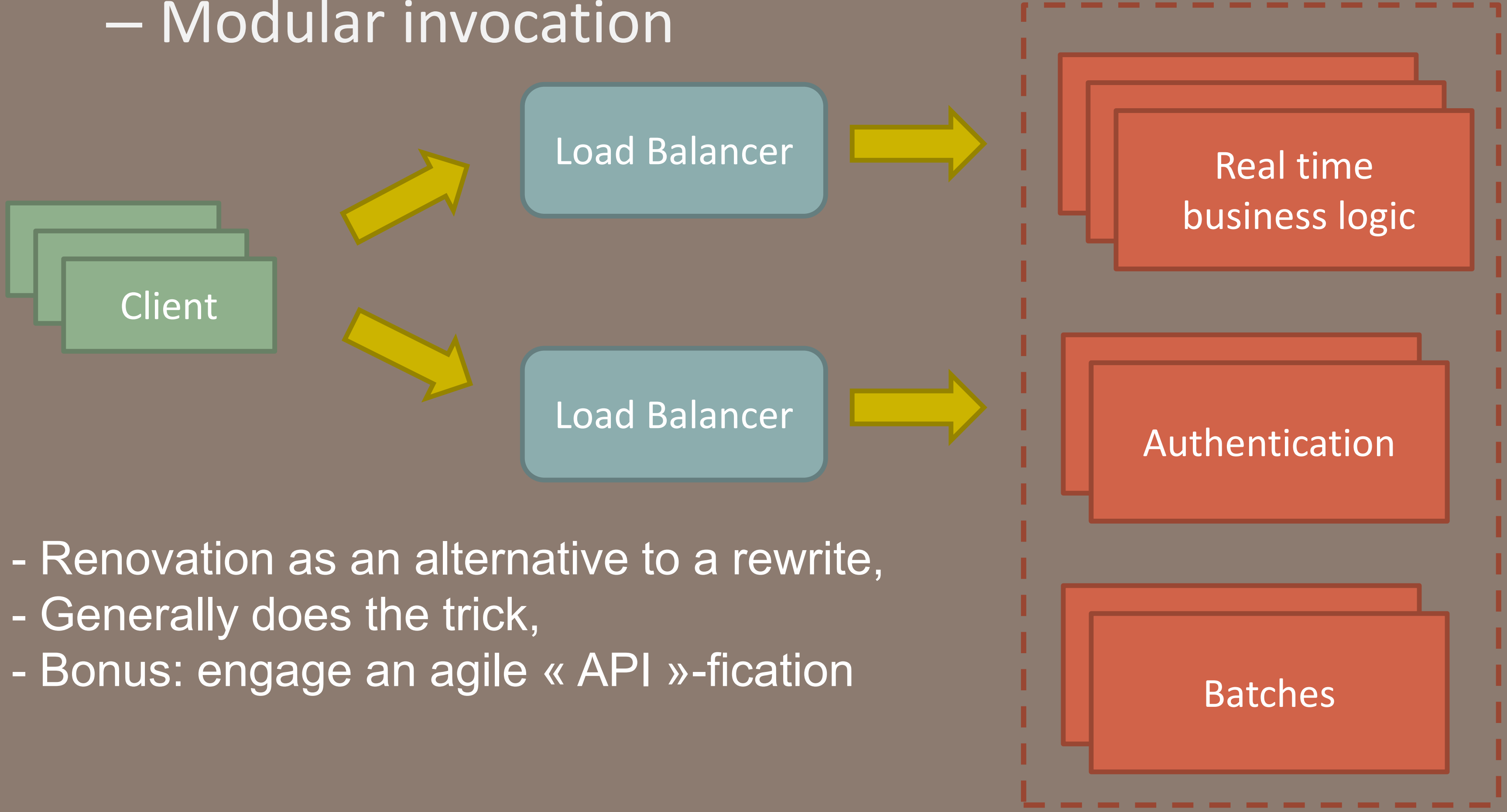


+ Quickest path to scale
+ High availability
- Routing traffic complexity
- Very large code base
- Change cycles tied together
- Limited scalability

⇒ lack of modularity

Making the Monolith more modular

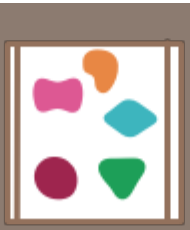
- Specialized instances of a single codebase
  - Modular invocation



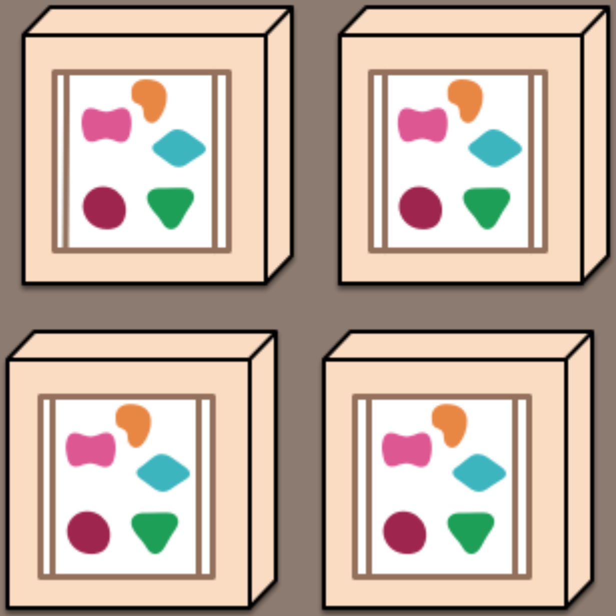
- Renovation as an alternative to a rewrite,
- Generally does the trick,
- Bonus: engage an agile « API »-fication

MONOLITHIC VS MICROSERVICES

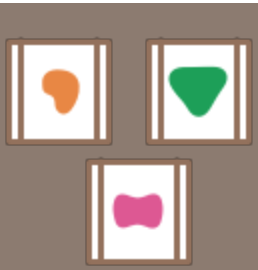
A monolithic application puts all its functionality into a single process...



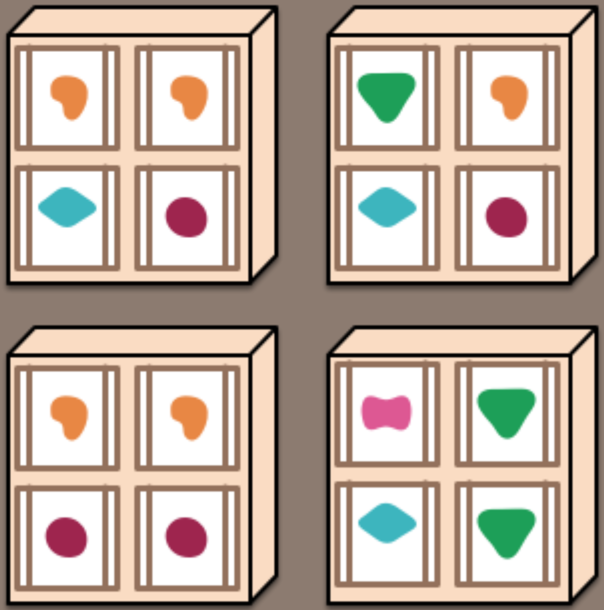
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...

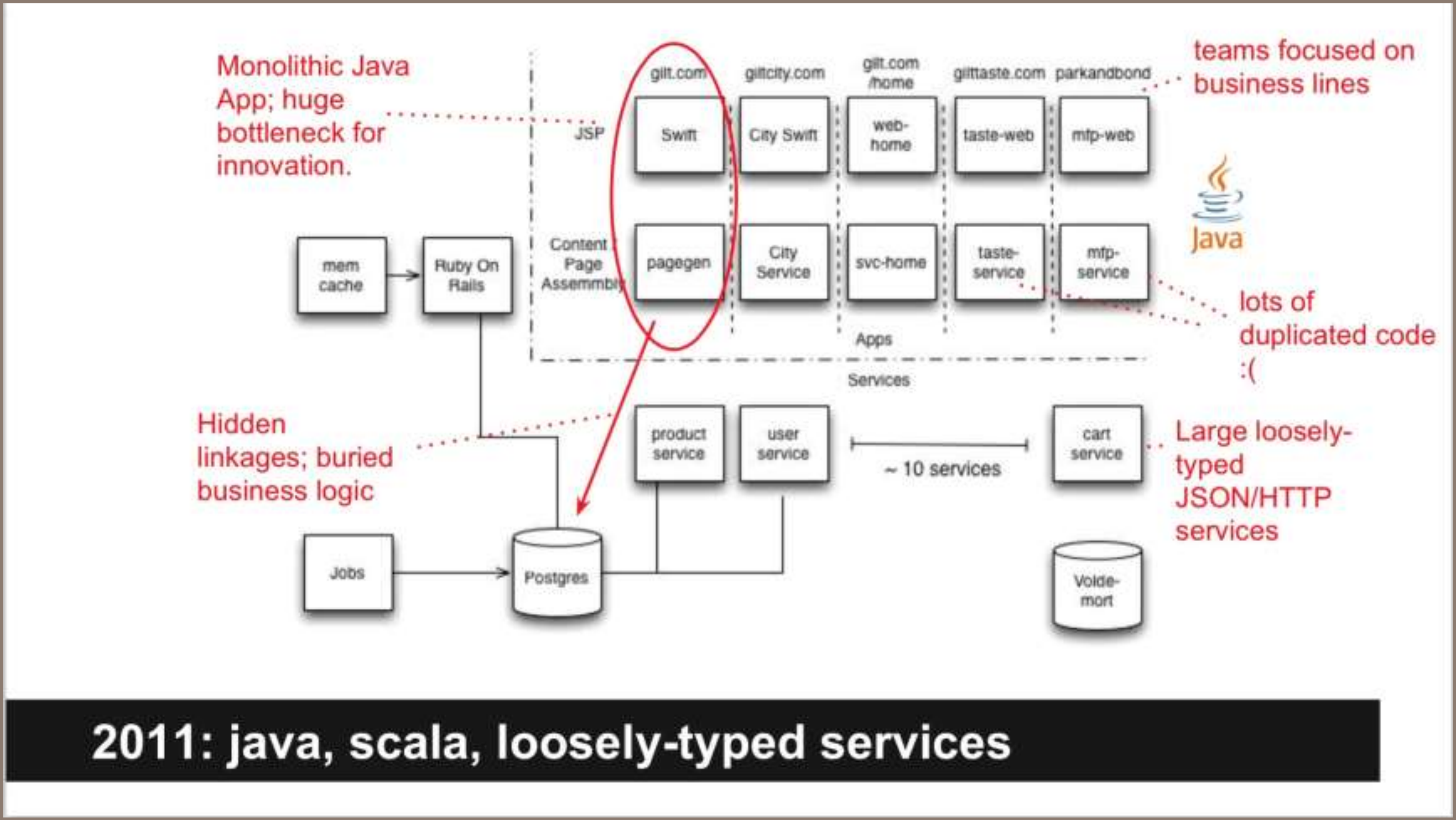


... and scales by distributing these services across servers, replicating as needed.



http://martinfowler.com/articles/microservices.html

GILT TESTIMONIAL



Monolithic Java App; huge bottleneck for innovation.

teams focused on business lines

lots of duplicated code :(

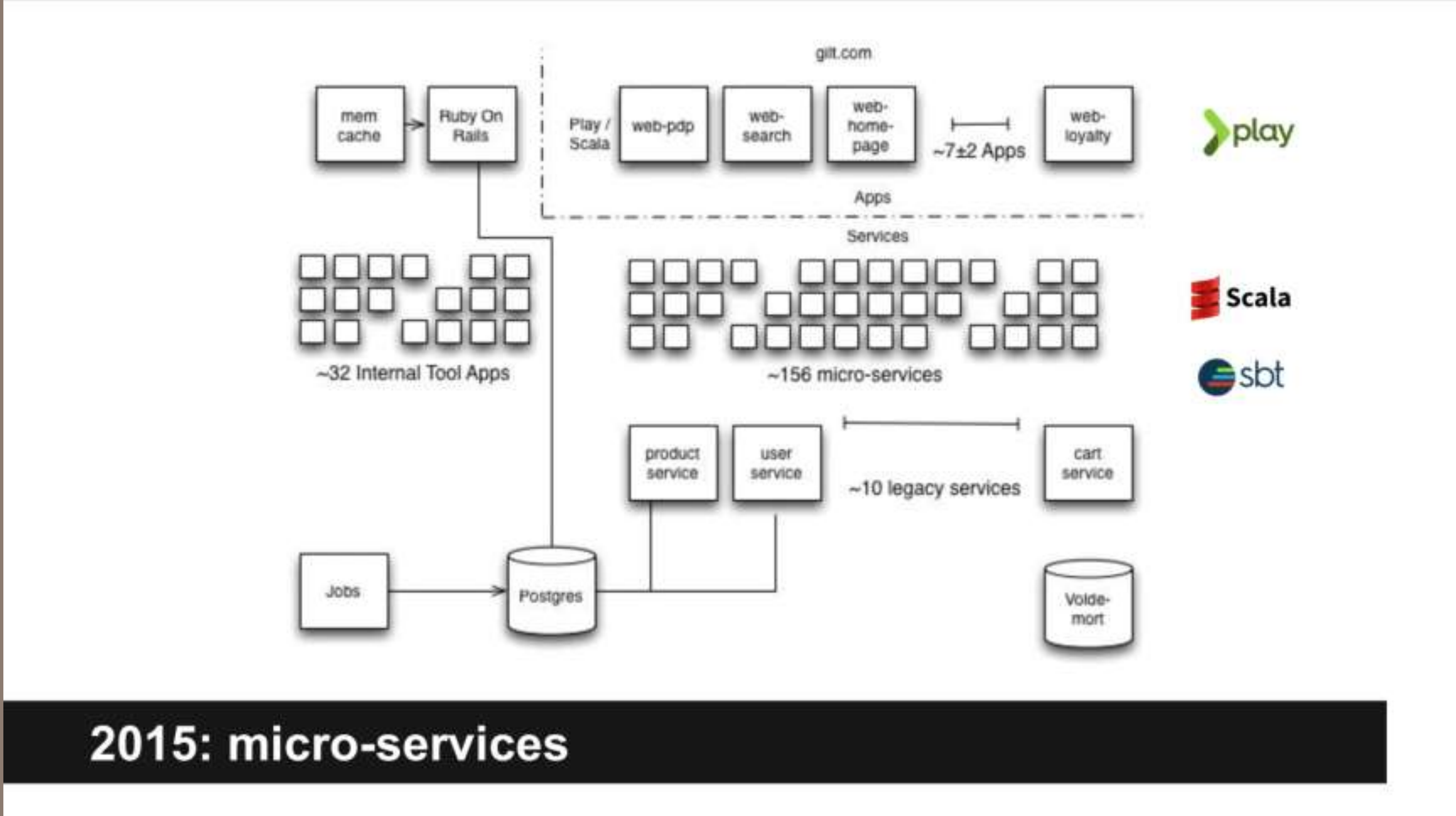
Hidden linkages; buried business logic

Large loosely-typed JSON/HTTP services

2011: java, scala, loosely-typed services

http://www.infoq.com/news/2015/04/scaling-microservices-gilt

GILT TESTIMONIAL



2015: micro-services

http://www.infoq.com/news/2015/04/scaling-microservices-gilt

Common Characteristics

DESIGNING FOR MICROSERVICES

Common Characteristics

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized governance
- Decentralized data management
- Infrastructure automation
- Design for failure
- Evolutionary design

http://martinfowler.com/articles/microservices.html

Componentization via Services

- Services as components rather than libraries
- Services avoid tight coupling by using **explicit remote call mechanisms**.
- Services are independently deployable and scalable

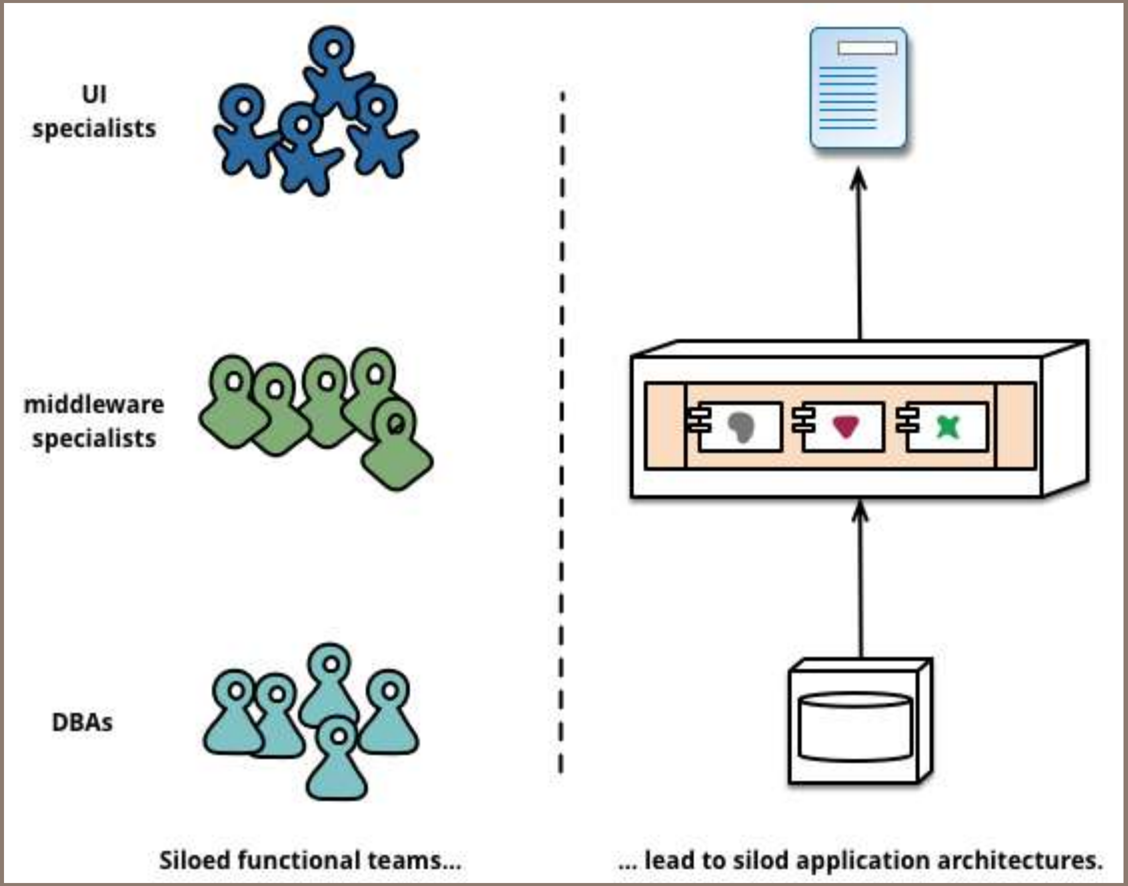
Componentization via Services

- Each service also provides a firm module boundary
  - business or technical,
  - even allowing for different services to be written in different programming languages,
  - they can also be managed by different teams .
- A Microservice may consist of multiple processes
  - that will always be developed and deployed together,
  - Ex : an application process and a database that's only used by that service.

ORGANIZED AROUND BUSINESS CAPABILITIES

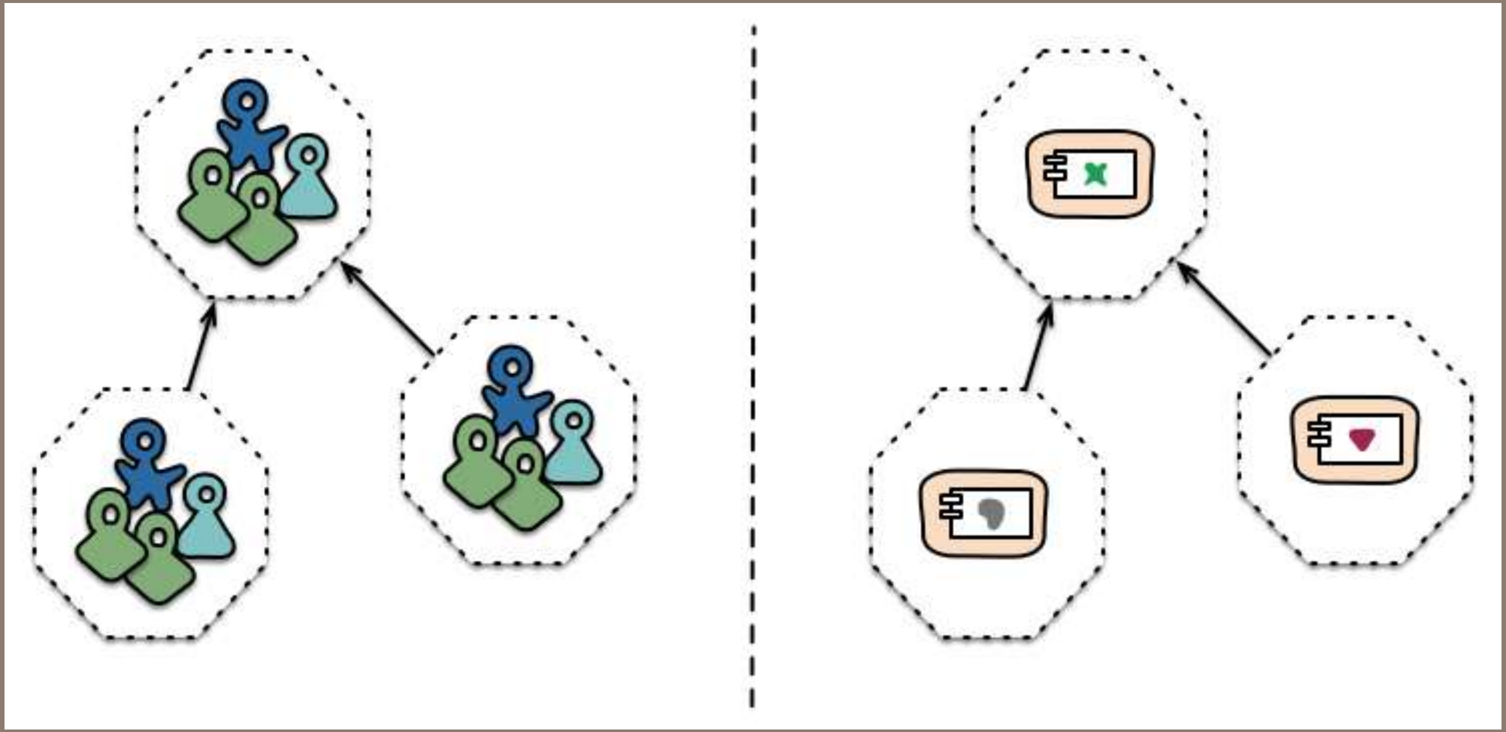
Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

Melvyn Conway, 1967.


UI specialists
middleware specialists
DBAs
Siloed functional teams...
... lead to silod application architectures.

ORGANIZED AROUND BUSINESS CAPABILITIES

- Microservices to solve Conway’s anti-pattern



Cross-functional teams….

… organized around capabilities

PRODUCTS NOT PROJECTS

- Standard project model:
  - deliver pieces of software which are then considered to be completed,
  - hand over to a maintenance organization and disband the project team
- The Microservices style
  - a team should own a product over its full lifetime
  - Amazon : You build => You run it

SMART ENDPOINTS AND DUMB PIPES

- Be as decoupled and as cohesive as possible
  - own domain logic,
  - act more as filters in the classical Unix sense
  - using simple RESTish protocols and lightweight messaging
- Smarts live in the services, not in-between the endpoints
  - No central tool / bus that includes sophisticated routing, transformations, process, business rules
- Pre-requisite : turn the chatty in-process communication of the monolith into coarser-grained network messaging

DECENTRALIZED GOVERNANCE

- The common microservices practice is to choose the best tool for each job
  - Tools are shared inside and outside the organization
  - Focus on common problems : data storage, IPC, infrastructure automation
  - Example : Netflix opensource libraries
- Favor independent evolution
  - Consumer-driven Service contracts,
  - Tolerant Reader Pattern

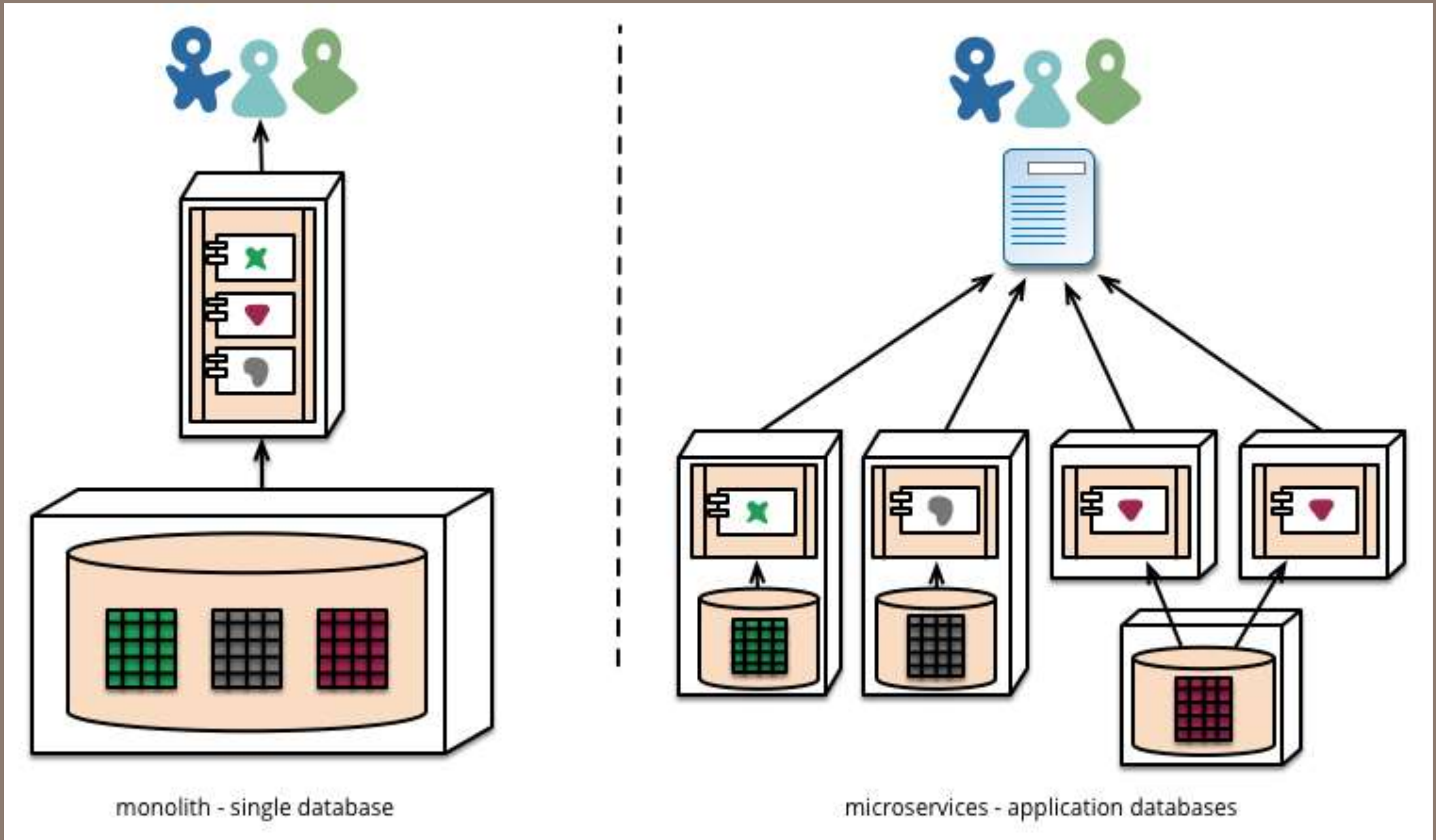
⇒Minimal over-heads create an opportunity to have teams responsible for all aspects

- Build their microservices and operate with 24/7 SLAs

Decentralized Data Management

- No unique data model approach
  - the conceptual model differs between microservices
  - Reinforce the separation of concerns
- Decentralized data storage
  - Polyglot persistence is frequent in microservices architectures
- Manage inconsistencies via the business practices in place throughout the organization
  - Common design : reversal processes versus 2PC distributed transactions

Decentralized Data Management


monolith - single database
microservices - application databases

INFRASTRUCTURE AUTOMATION

- CI/CD, Real-time monitoring, global system and fine grained resources dashboards,
- Has become a standard practice
  - – thanks to public cloud providers,
  - – but also opensource tools
- a QA stake for monoliths
- a pre-requisite for microservices

DESIGN FOR FAILURE

- Any service call can fail
  - Circuit Breaker pattern,
  - async consumption
    - Best practice : 1 to ZERO sync call
  - Ex : Netflix Stack
    - Hystrix / Eureka / Ribbon
  - /!\ provide interop & implementations for various langages
- Infrastructure pre-requisites
  - monitor and restore at scale
  - dedicated tooling for simulation
  - Ex : Netflix Simian Army (Chaos Monkey)

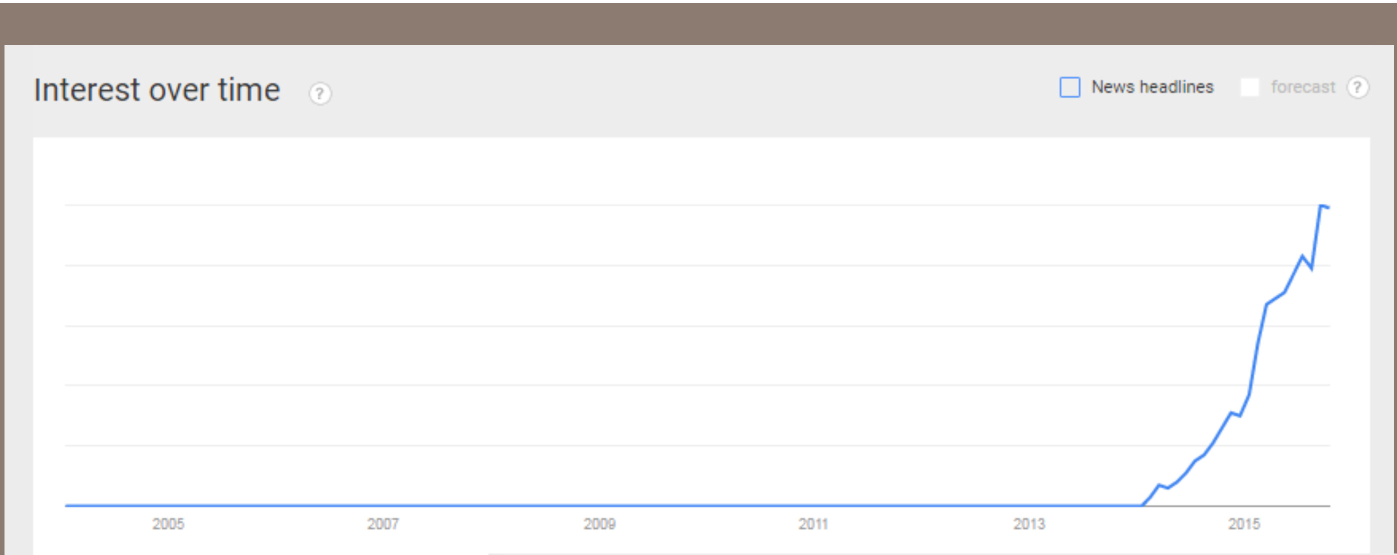
Evolutionary Design

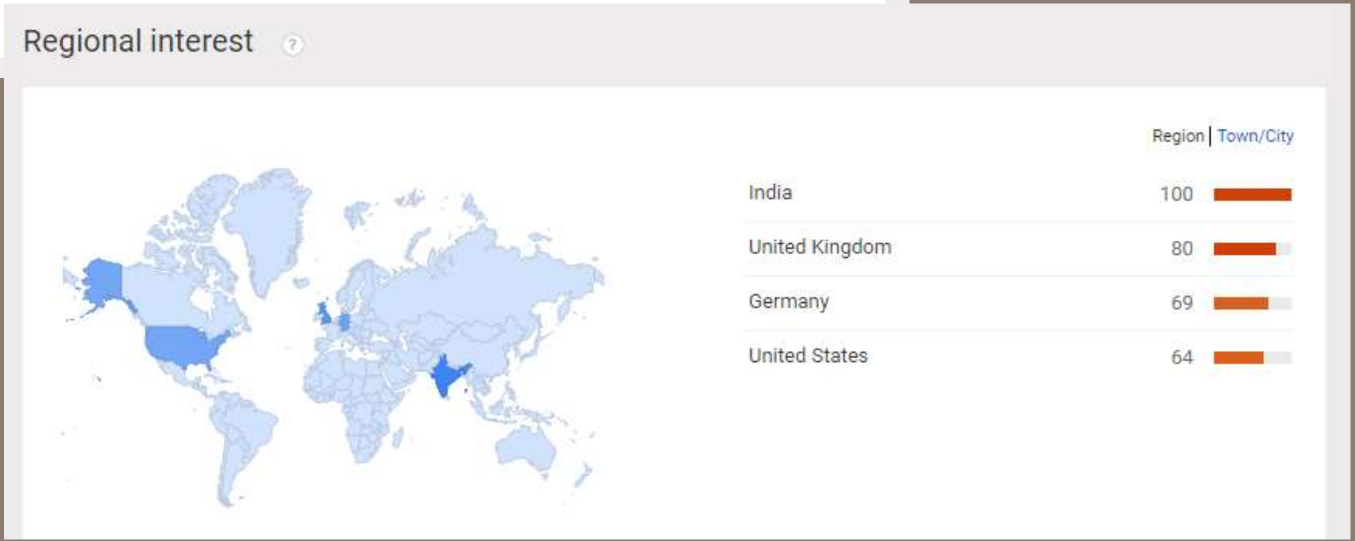
- More granular release planning
- Goal : change tolerance
  - Identify change impacts quickly (via automation, service contracts, automated dependency maps)
  - Fix rather than revert
  - In case of mandatory breaking change, use versioning
    - a last resort option in the microservices world

A bit of history, Best practices, Netflix Stack

REAL WORLD MICROSERVICES

Google Trends – October 2015





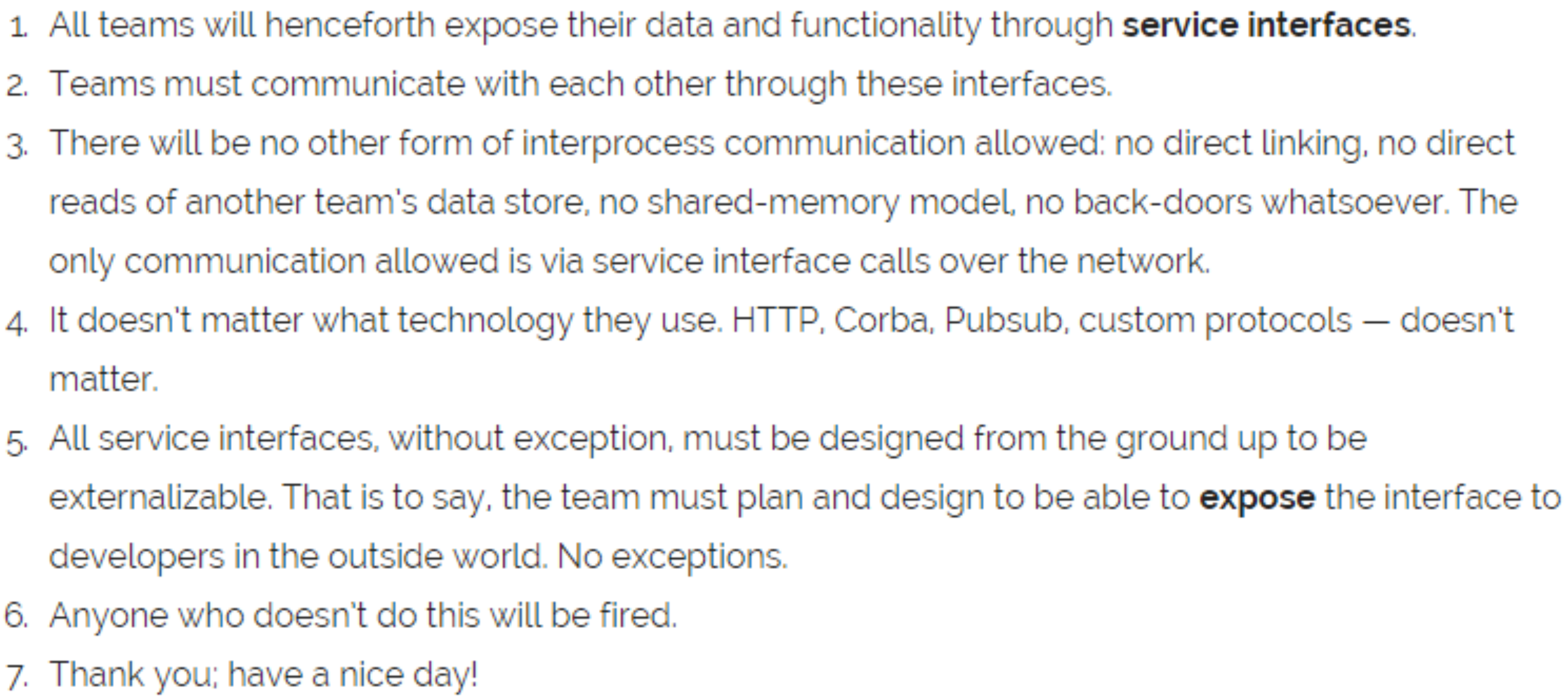
India	100	🟥
United Kingdom	80	🟥
Germany	69	🟥
United States	64	🟥

STATE OF THE ART

- Pioneers : Amazon, Netflix, Google
  - Started Opensourcing their building blocks in 2010
  - Democratized the Microservices architecture style
- Today
  - Large corporation for complex systems that need to evolve frequently (understand continuously)
  - Digital startups to ensure their core business will scale and embrace the long tail

Amazon DNA

- Microservices by design



1. All teams will henceforth expose their data and functionality through **service interfaces**.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols — doesn't matter.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to **expose** the interface to developers in the outside world. No exceptions.
6. Anyone who doesn't do this will be fired.
7. Thank you; have a nice day!

Jeff *Bezos*, 2002

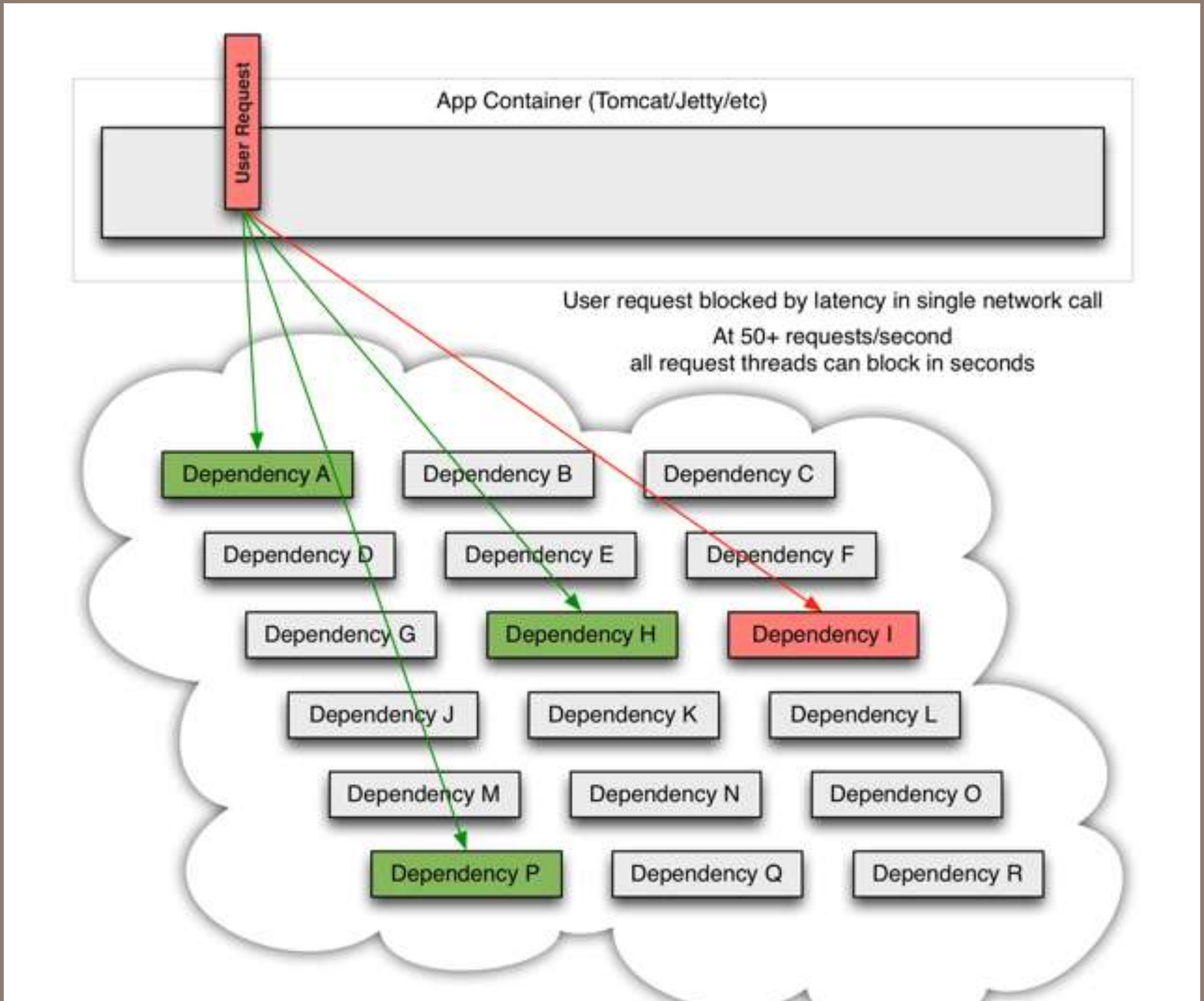
Netflix Ecosystem

- 100s of microservices
- 1,000s of daily production changes
- 10,000s of instances
- 100,000s of customer interactions per minute
- 1,000,000s of customers
- 1,000,000,000s of metrics
- 10,000,000,000 hours of streamed
- 10s of operations engineers

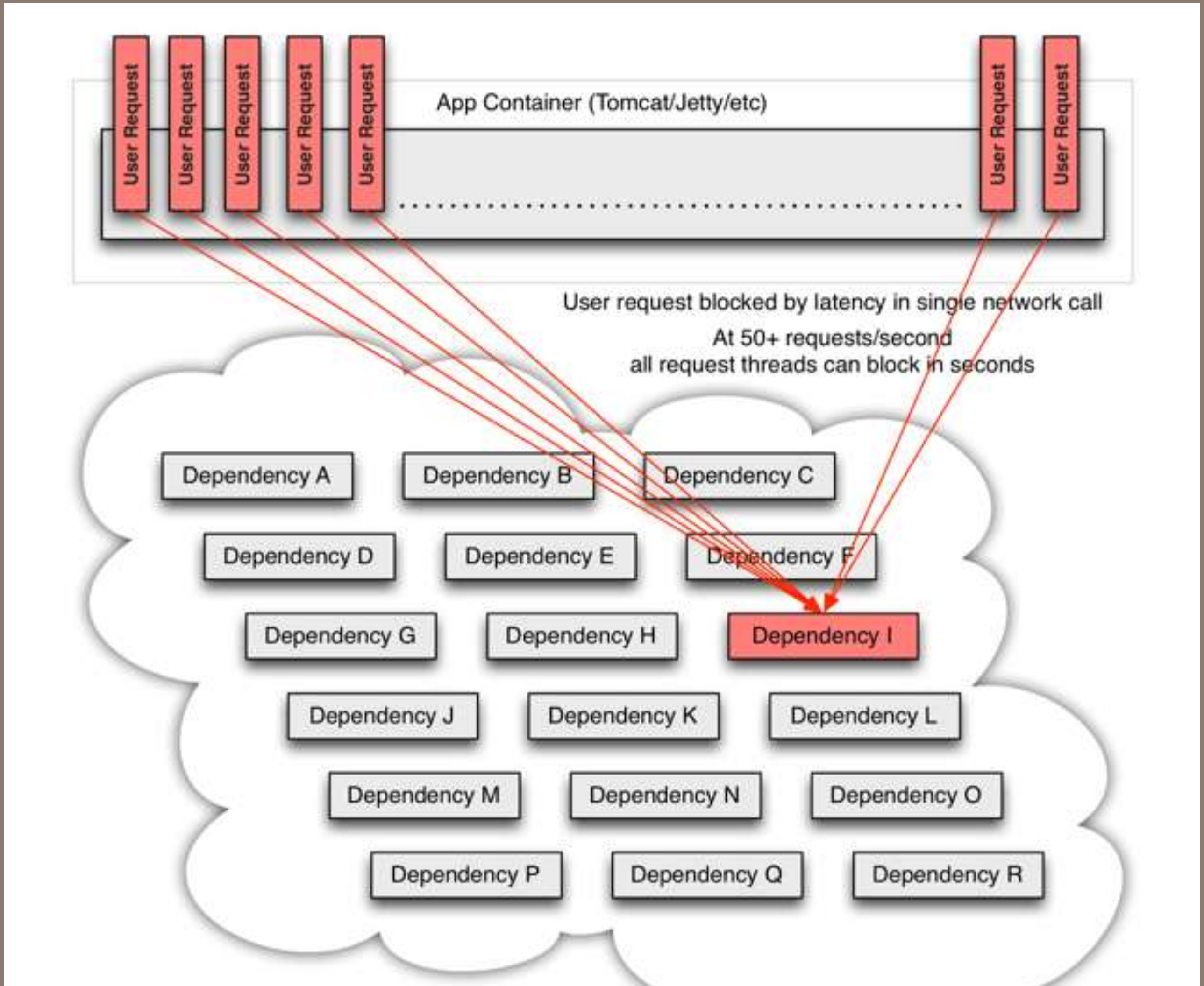
Resilience in Distributed Systems

- Failures are common
- An application that depends on 30 services
  - where each service has 99.99% uptime
  - 99.99^{30} = 99.7% uptime
  - 0.3% of 1 billion requests = 3,000,000 failures
  - 2+ hours downtime/month even if all dependencies have excellent uptime.
- Reality is generally worse.

When 1 backend becomes latent



AT 50+ REQ/S, ALL REQUEST THREADS CAN BLOCK IN SECONDS



Fail Fast, Fail Silent, Fallback

- Wrapping all calls to external systems (or “dependencies”) which typically executes within a separate thread
- Timing-out calls that take longer than thresholds you define.
- Maintaining a small thread-pool for each dependency;
  - if it becomes full, requests immediately rejected instead of queued up.
- Measuring successes, failures (exceptions thrown by client), timeouts, and thread rejections.
- Tripping a circuit-breaker to stop all requests to a particular service for a period of time,
  - either manually or automatically (error percentage threshold)
- Performing fallback logic when a request fails, is rejected, times-out, or short-circuits.
- Monitoring metrics and configuration changes in near real-time.

Services Discovery

- Typically, a Microservice gets created and destroyed often
- It is reconfigured on the fly
- In near real-time, others should find it
- Examples
  - Apache ZooKeeper
  - HashiCorp Consul
  - CoreOS Etcd
  - Netflix Eureka

Services Consumption

- Choose the best messaging infrastructure
  - RPC / REST style,
  - Request/Response, Streaming
  - HTTP, HTTP2, TCP, UDP
- Support async consumption / aggregation
  - Use parallel code structures of your clients
  - GO, ES7 (BabelJS, Google Traceur)
- Ease consumption via clients SDKs
  - Automated generated from API definitions
  - Service providers tend to opensource their SDKs
    - Facebook Parse, IBM Bluemix Mobile services

Services Consumption

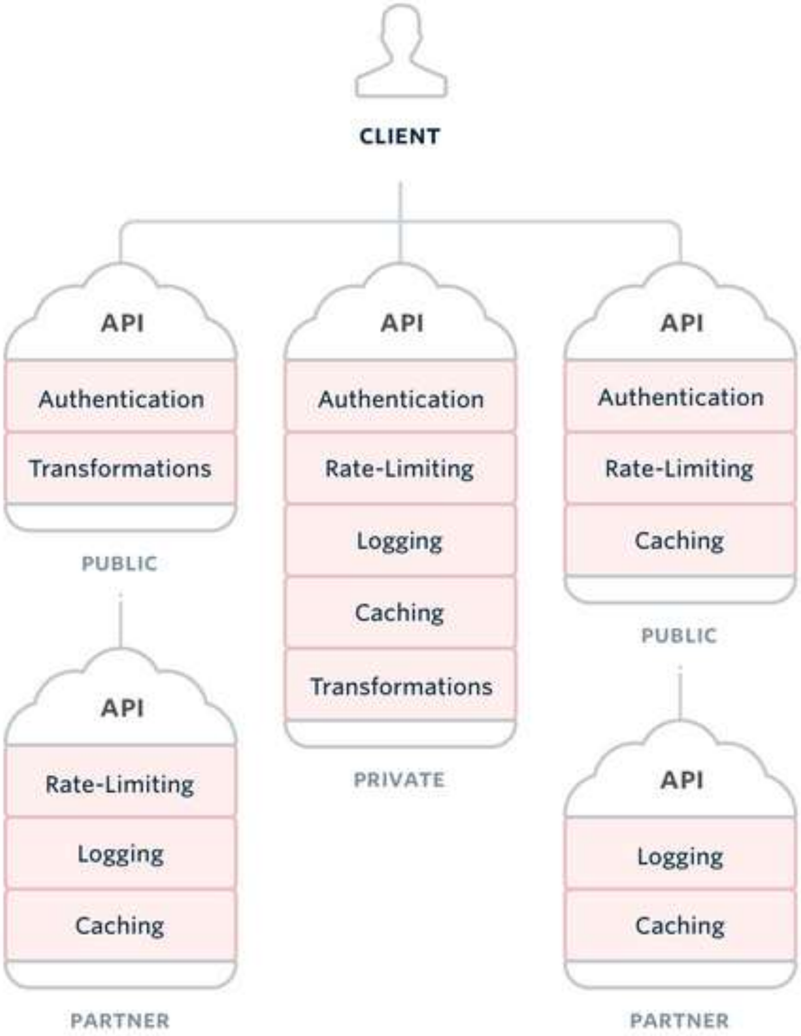
- Favor SDKs with large feature sets
  - Basic REST is not an option for microservices
  - integrate load balancing, fault tolerance, caching, service discovery, multiple transport protocols (HTTP, TCP, UDP)
- Netflix Ribbon
  - Inter Process Communication with built in software load balancers
  - Integrates Hystrix and Eureka
  - The primary usage model involves REST calls with various serialization scheme support.

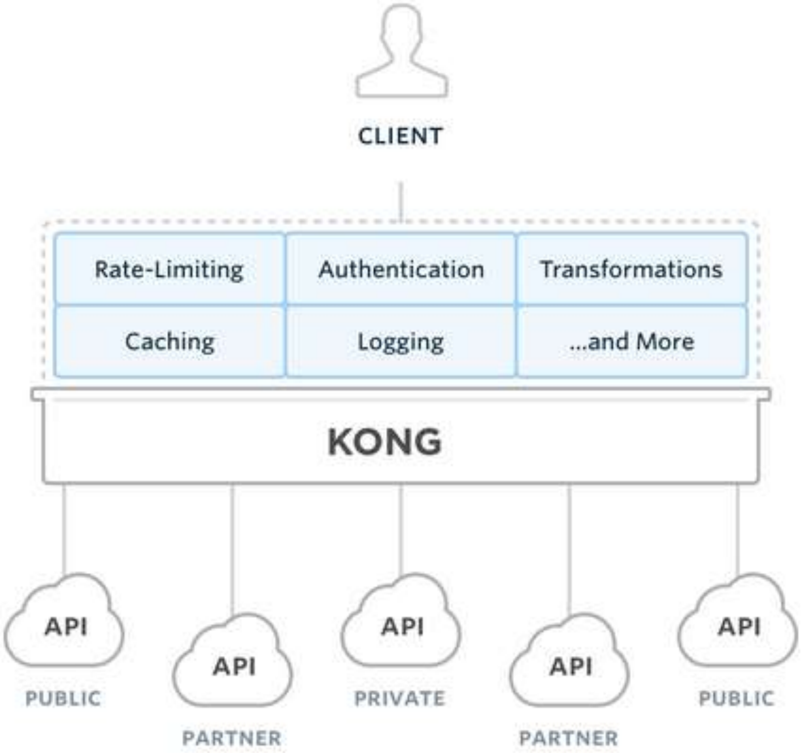
MICROSERVICES GATEWAY

- As an Edge service to access back-end microservices
  - Authentication and Security
  - Dynamic routing,
  - Insights and Monitoring
  - Stress testing : gradually increasing the traffic to a cluster in order to gauge performance.
  - Load Shedding : allocating capacity for each type of request and dropping requests that go over the limit.
  - Static Response handling : building some responses directly at the edge instead of forwarding them to an internal cluster
  - Multiregion Resiliency : routing requests across AWS regions in order to diversify the ELB usage and move the edge closer to users

MICROSERVICES GATEWAY

- Lots of building blocks
  - Nginx
  - Mashape Kong
- Needs integration with the other building blocks of the Microservices communication infrastructure
  - Example : Netflix Zuul





Mashape Kong

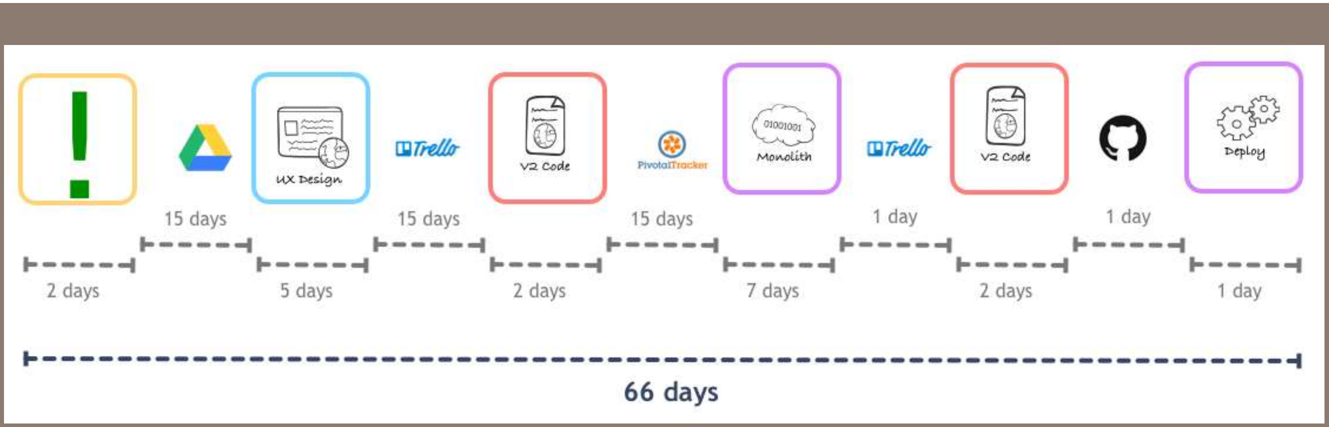
- ✗ Common functionality is duplicated across multiple services
- ✗ Systems tend to be monolithic and hard to maintain
- ✗ Difficult to expand without impacting other services
- ✗ Productivity is inefficient because of system constraints

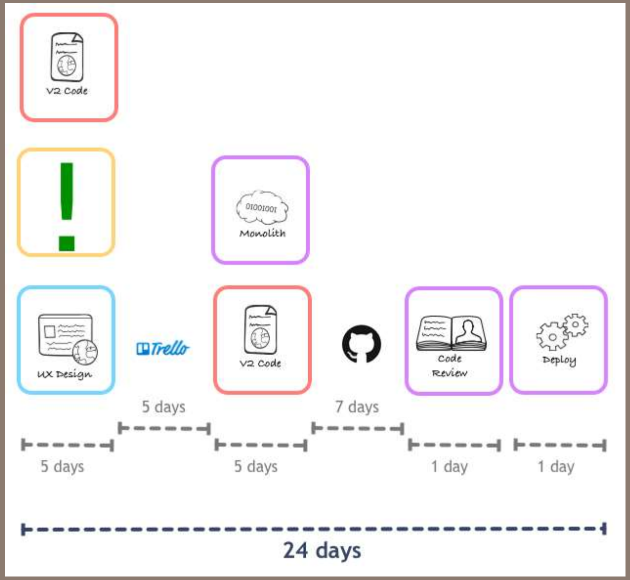
- ✓ Kong centralizes and unifies functionality into one place
- ✓ Build efficient distributed architectures ready to scale
- ✓ Expand functionality from one place with a simple command
- ✓ Your team is focused on the product, Kong does the REST

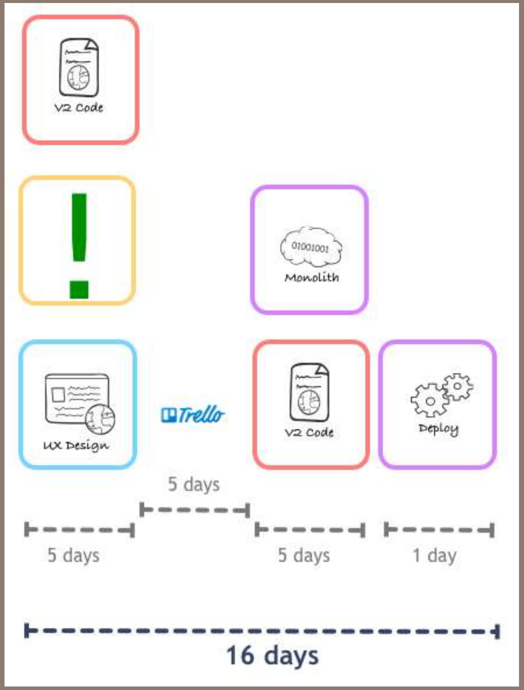
Committed Teams

- Ownership core to team organization
  - built into the management of the organization
  - make sure that teams have sufficient time to truly own the applications that they are in charge
    - “Products versus Projects” principle
    - “Functional versus Divisional” organizations
  - And give them 360 view on operations

SoundCloud Testimonial

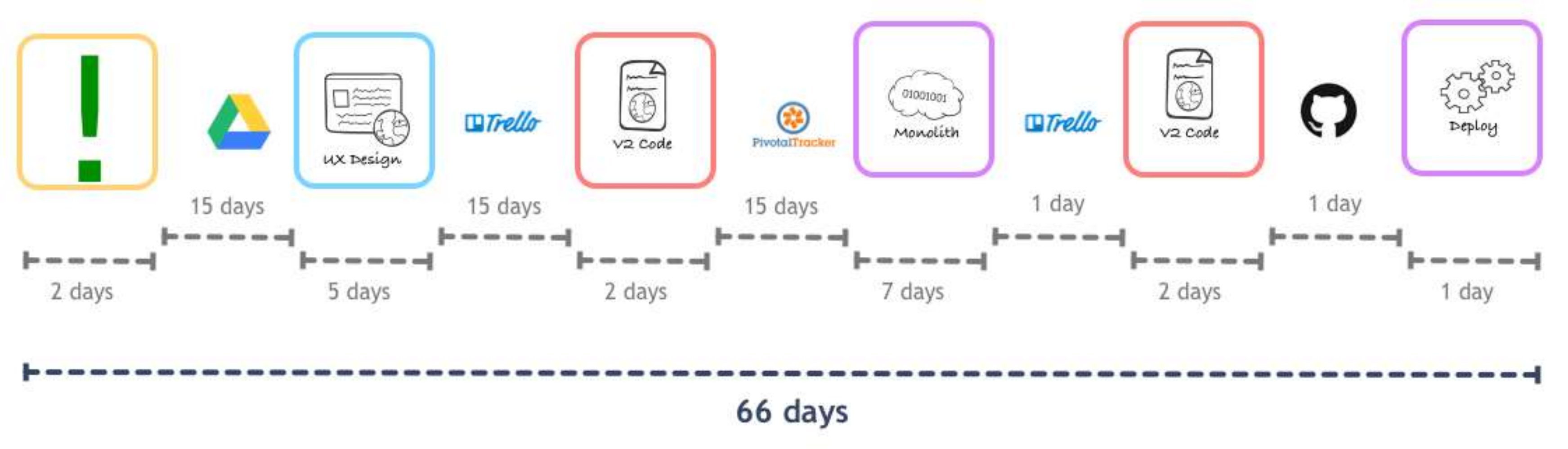






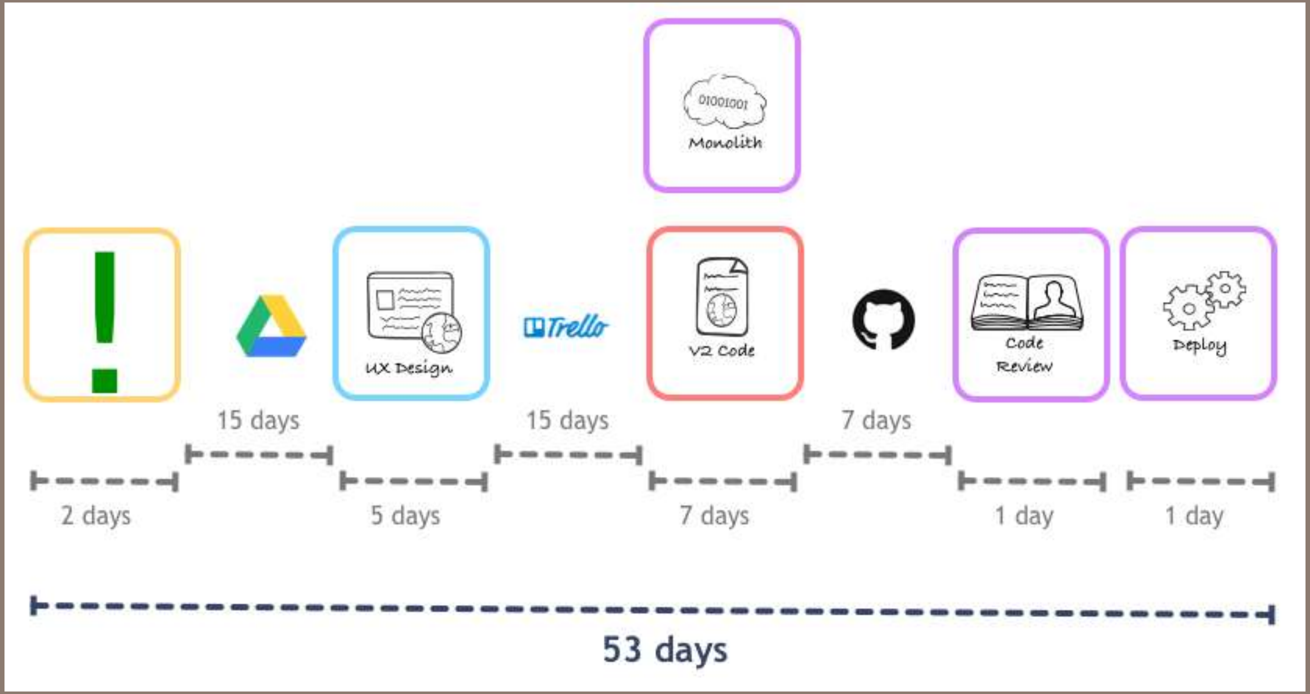
SoundCloud Testimonial

- Actual workflow to go live
  - 2 months, 11 steps
  - Main elephant is all the dance between front-end and back-end development
  - 11 days are doing actual development work



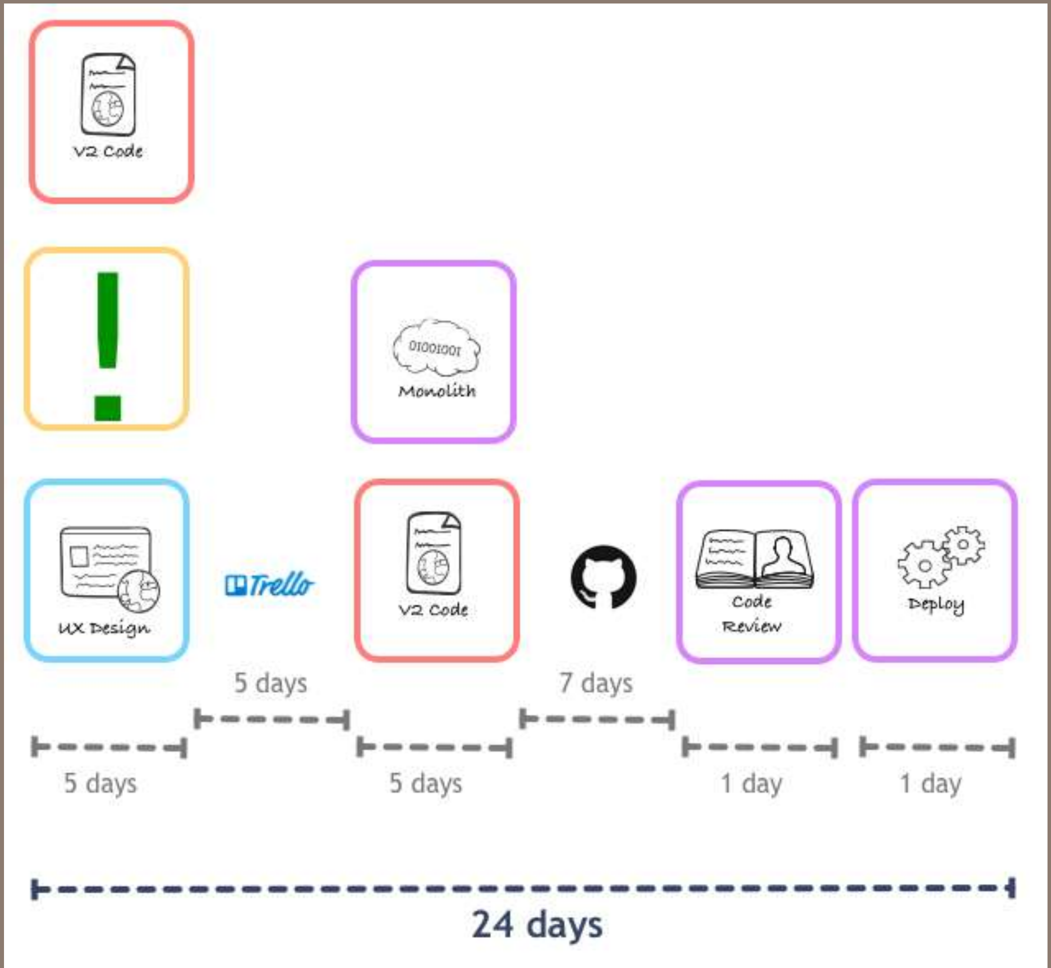
SoundCloud Testimonial

- Decision : pairing back-end and front-end devs
  - pair fully dedicated to a feature until its completion
  - Individually, each person ended up spending more time doing work per feature



SoundCloud Testimonial

- Designer, product manager, and front-end developer working close to each other

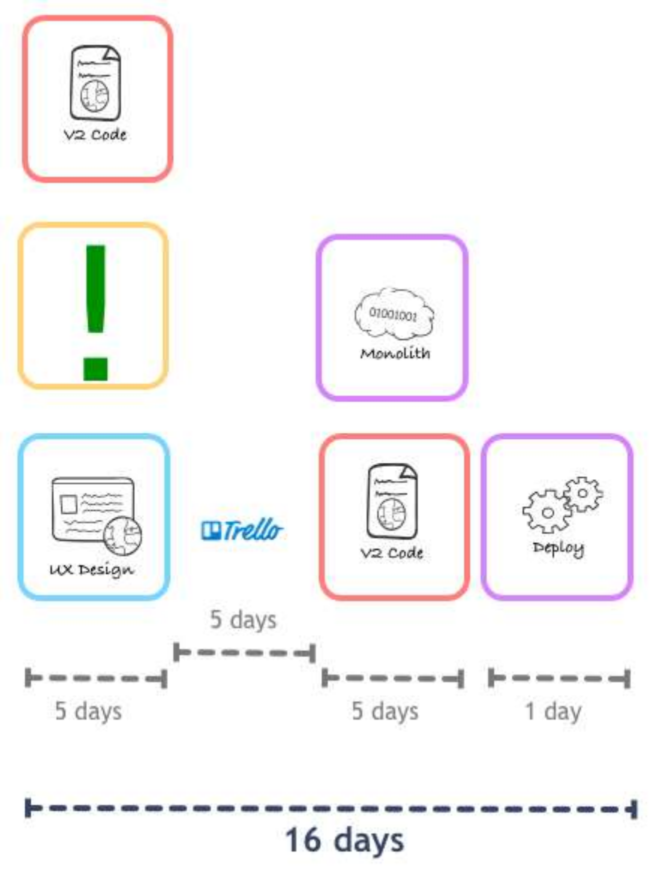


SoundCloud Testimonial

- The irreducible complexity of the monolith
  - *Why do we need Pull Requests?*
  - *Why do people make mistakes so often?*
  - *Why is the code base so complex?*
  - *Why do we need a single code base to implement the many components?*
  - *Why can't we have economies of scale for multiple, smaller, systems?*

SoundCloud Testimonial

- Isolated new features in dedicated microservices, isolated from the monolith
- New organization with team of 3 to 4 people
- Each team is responsible for decided whether parts of the Monolith are extracted and rewritten , or kept

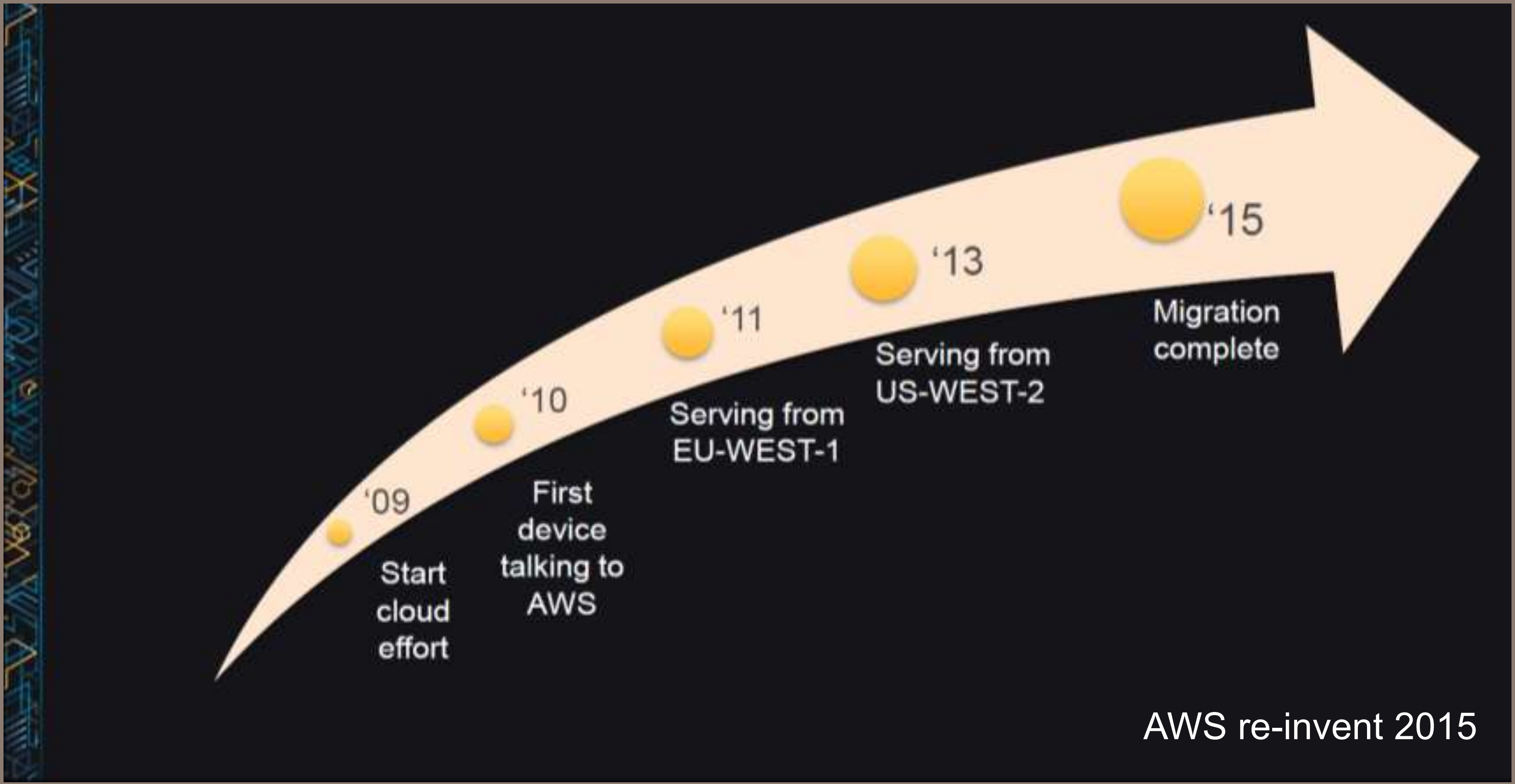


Committed Teams

- Ownership core to team organization
  - built into the management of the organization
  - make sure that teams have sufficient time to truly own the applications that they are in charge
    - “Products versus Projects” principle
    - “Functional versus Divisional” organizations
  - And give them 360 view on operations

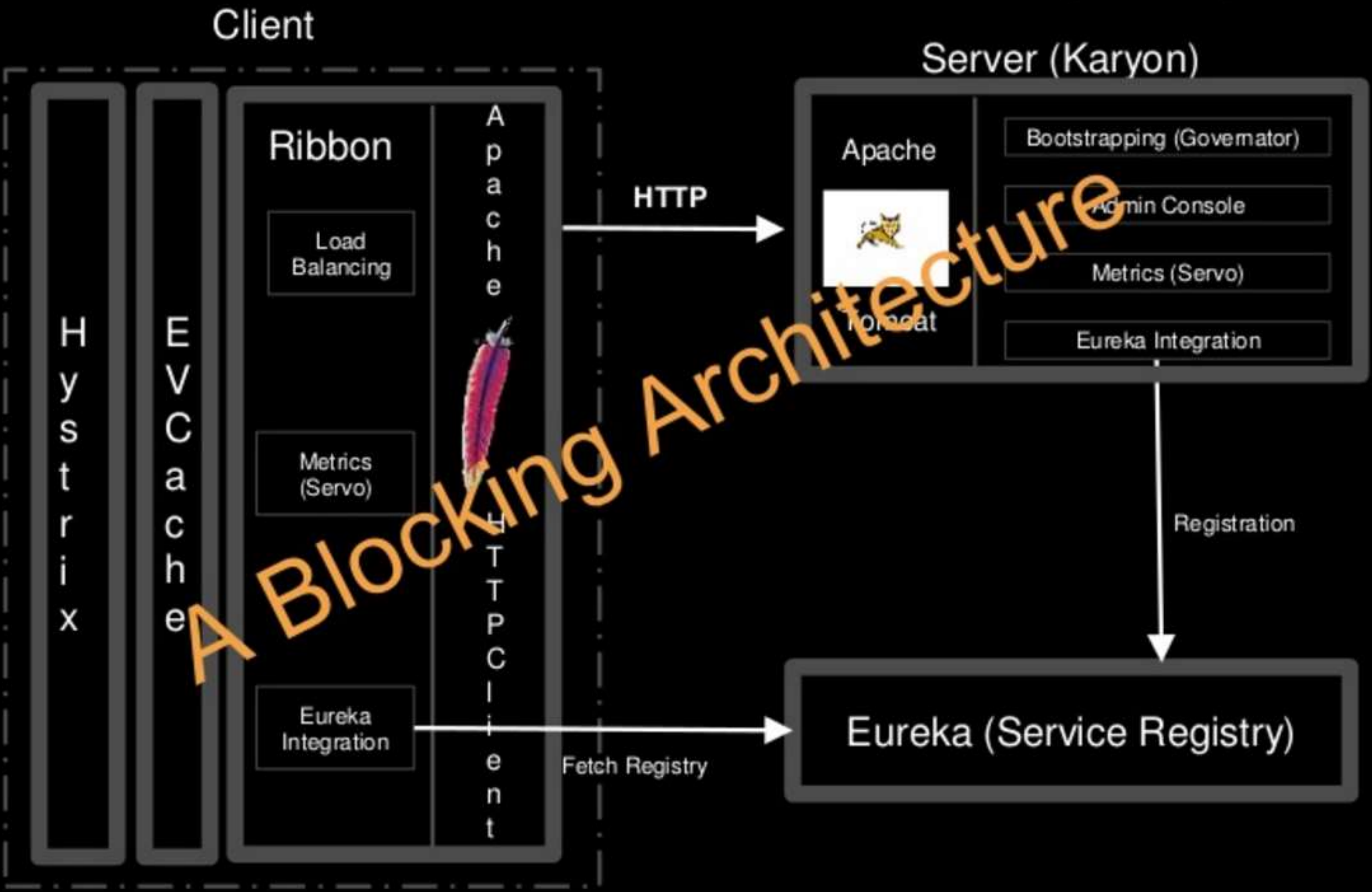
The Netflix Microservices journey

- Migration to AWS



AWS re-invent 2015

Netflix IPC Stack (1.0)



Client

Hystrix

EVCache

Ribbon

Load Balancing

Metrics (Servo)

Eureka Integration

Apache HTTPClient

HTTP

Server (Karyon)

Apache Tomcat

Bootstrapping (Governator)

Admin Console

Metrics (Servo)

Eureka Integration

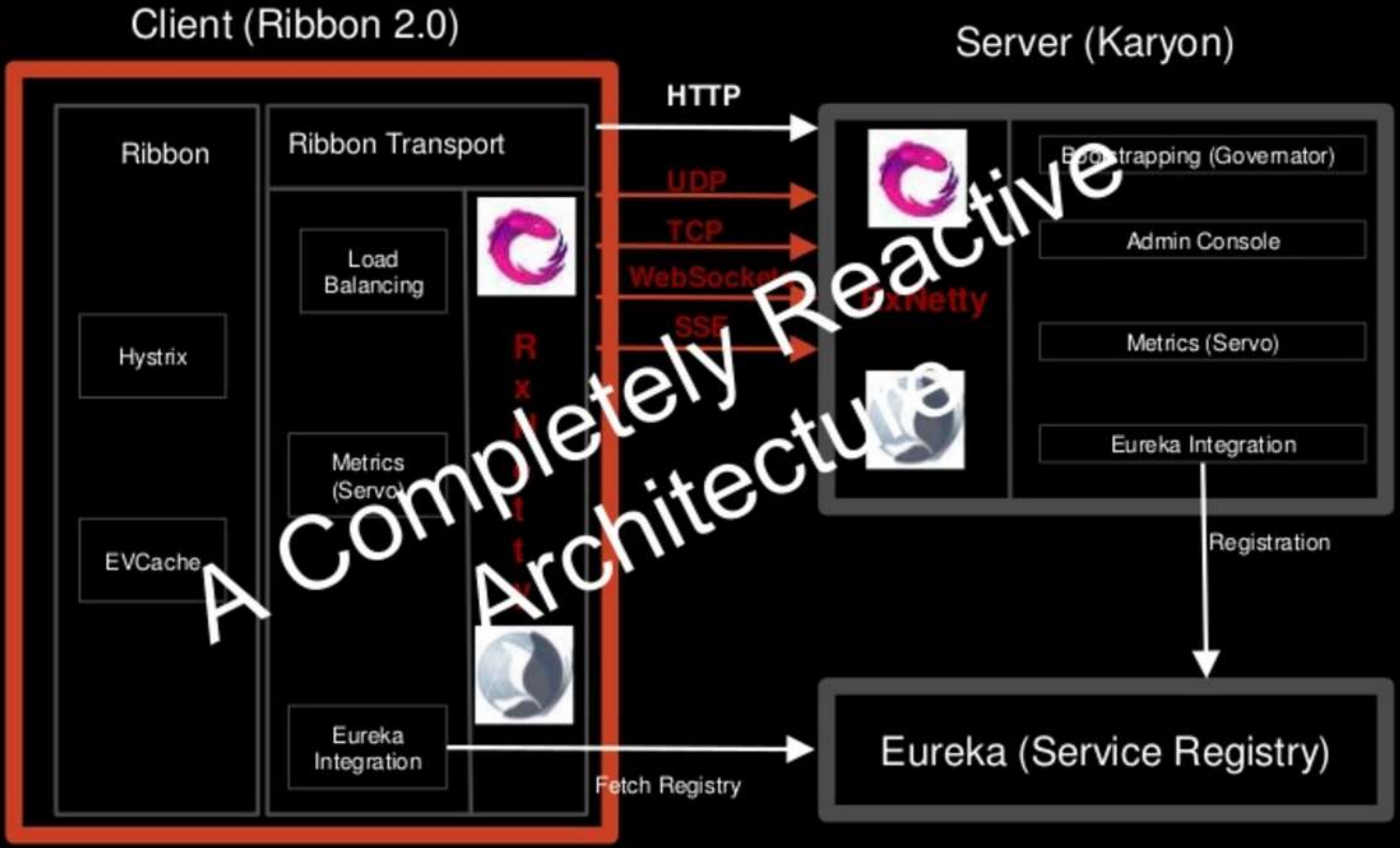
Registration

Fetch Registry

Eureka (Service Registry)

A Blocking Architecture

Netflix IPC Stack (2.0)



Client (Ribbon 2.0)

Ribbon

Hystrix

EVCache

Ribbon Transport

Load Balancing

Metrics (Servo)

Eureka Integration

RxNetty

HTTP

UDP

TCP

WebSocket

SSE

Fetch Registry

Server (Karyon)

RxNetty

Bootstrapping (Governator)

Admin Console

Metrics (Servo)

Eureka Integration

Registration

Eureka (Service Registry)

A Completely Reactive Architecture

Netflix Stack

- Solid communications
  - Hystrix : latency and fault tolerance library
  - Eureka : registry for resilient mid-tier load balancing and failover
  - Ribbon : client based smart load balancer
  - Servo : monitoring library
  - EVCache : distributed in-memory data store for AWS EC2
  - RxNetty : reactive extension adaptor for netty
  - Karyon : blueprint of a cloud ready microservice
  - Zuul : edge gateway
  - Falcor : js library for efficient data fetching

NETFLIX STACK

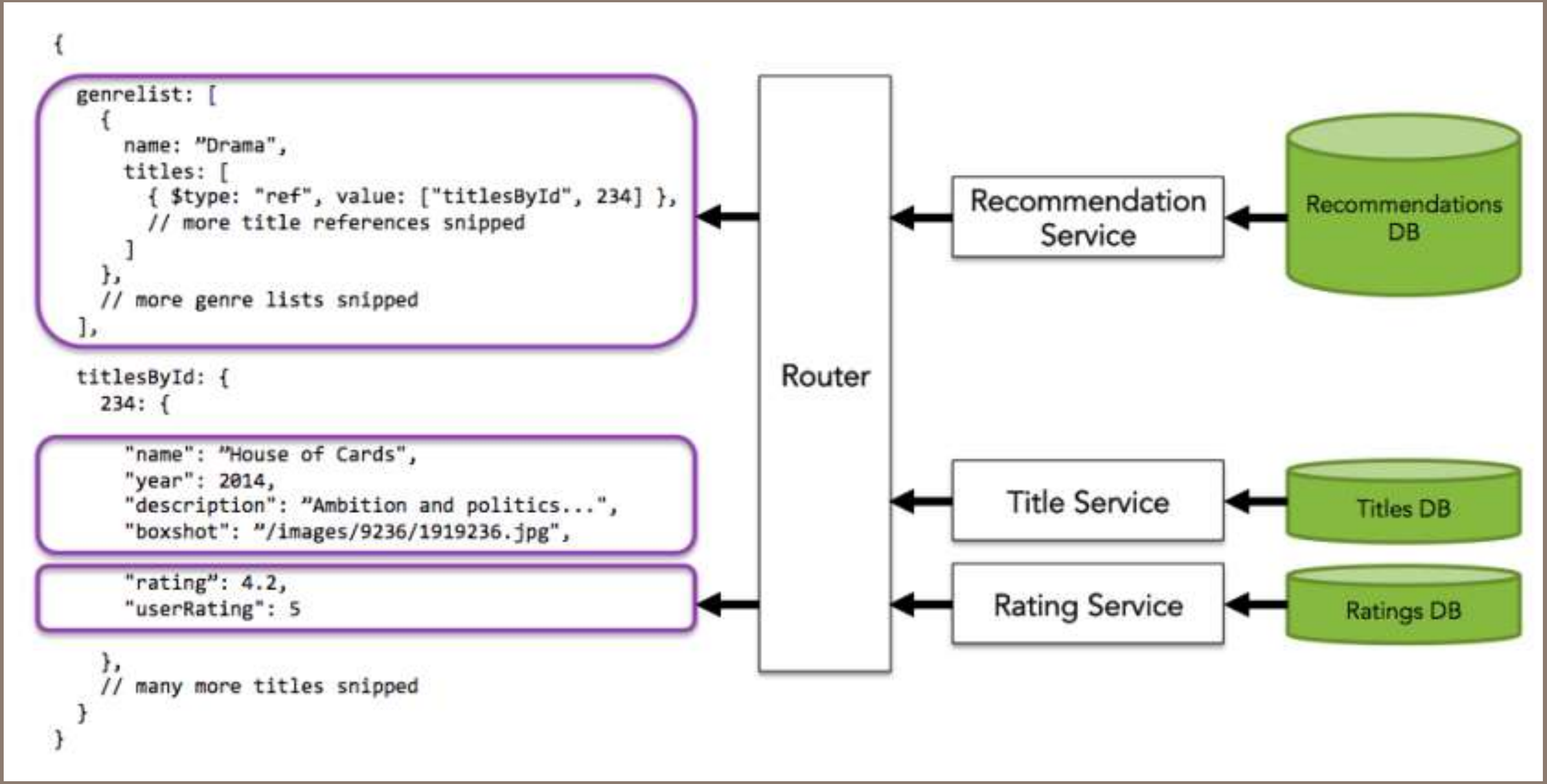
- Automation
  - Asgard : specialized AWS console (app deployments, management)
  - Spinnaker : microservices console (clusters, pipelines), not opensourced yet
  - Atlas : near real-time operational insights
  - Vector : exposes hand picked high resolution metrics from PCP – Performance Co-Pilot hosts
  - SimianArmy : services (Monkeys) in the cloud for generating various kinds of failures, detecting abnormal conditions, and testing our ability to survive them
  - Dependencies automatically documented from from real traffic analysis

Netflix / eureka

- Mid-tier load balancing
  - a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services
  - for the purpose of load balancing and failover of middle-tier servers,
  - the load balancing happens at the instance/server/host level
- Comes with a Java-based client
  - The client also has a built-in load balancer
  - The client instances know all the information about which servers they need to talk to.
  - Does basic round-robin load balancing.
    - At Netflix, a much more sophisticated load balancer wraps Eureka to provide weighted load balancing based on several factors like traffic, resource usage, error conditions to provide superior resiliency.

NETFLIX FALCOR

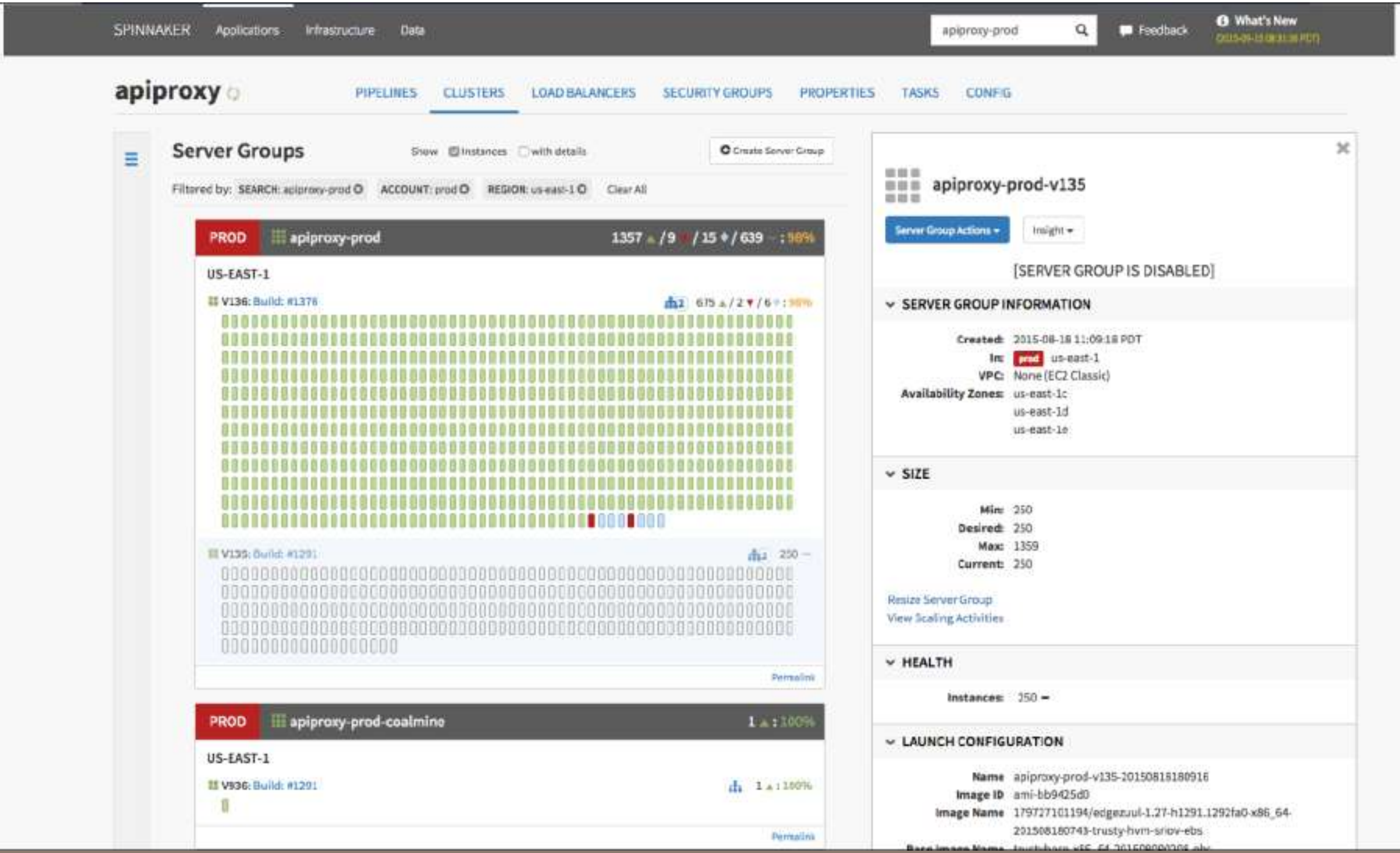
- Working with JSON Virtual Resources



```
{
  genrelist: [
    {
      name: ”Drama",
      titles: [
        { $type: "ref", value: ["titlesById", 234] },
        // more title references snipped
      ]
    },
    // more genre lists snipped
  ],
  titlesById: {
    234: {
      "name": ”House of Cards",
      "year": 2014,
      "description": ”Ambition and politics...",
      "boxshot": ”/images/9236/1919236.jpg",
      "rating”: 4.2,
      "userRating": 5
    },
    // many more titles snipped
  }
}
```

http://netflix.github.io/falcor/starter/what-is-falcor.html

SPINNAKER - MICROSERVICES CONSOLE



[SERVER GROUP IS DISABLED]

SERVER GROUP INFORMATION

Created: 2015-08-18 11:09:18 PDT
In: prod us-east-1
VPC: None (EC2 Classic)
Availability Zones: us-east-1c
us-east-1d
us-east-1e

SIZE

Min: 250
Desired: 250
Max: 1359
Current: 250

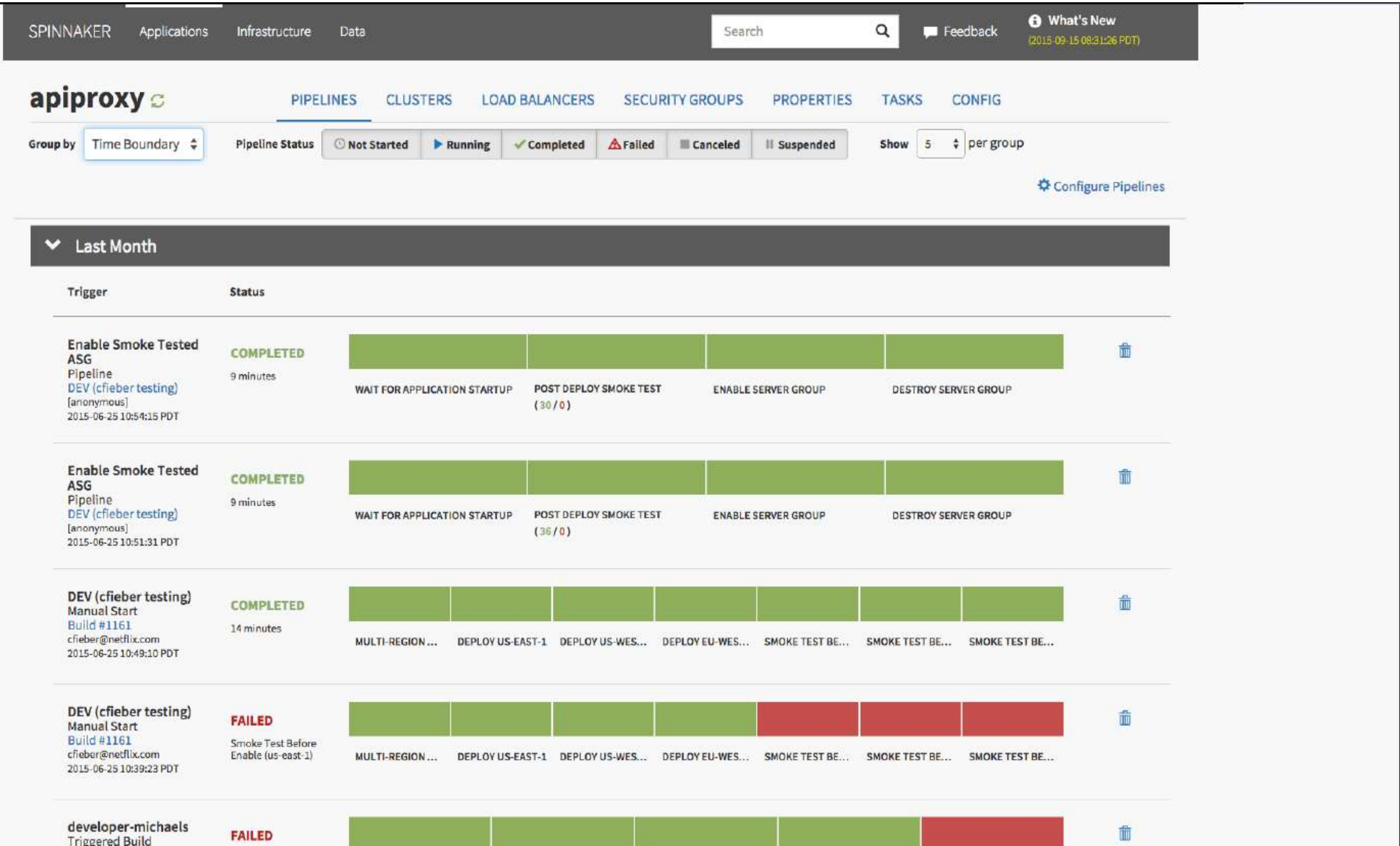
HEALTH

Instances: 250

LAUNCH CONFIGURATION

Name apiproxy-prod-v135-20150818180916
Image ID ami-bb9425d0
Image Name 179727101194/edgezuul-1.27-h1291.1292fa0-x86_64-201508180743-trusty-hvm-sriov-ebs

SPINNAKER - MICROSERVICES CONSOLE



apiproxy

Group by: Time Boundary

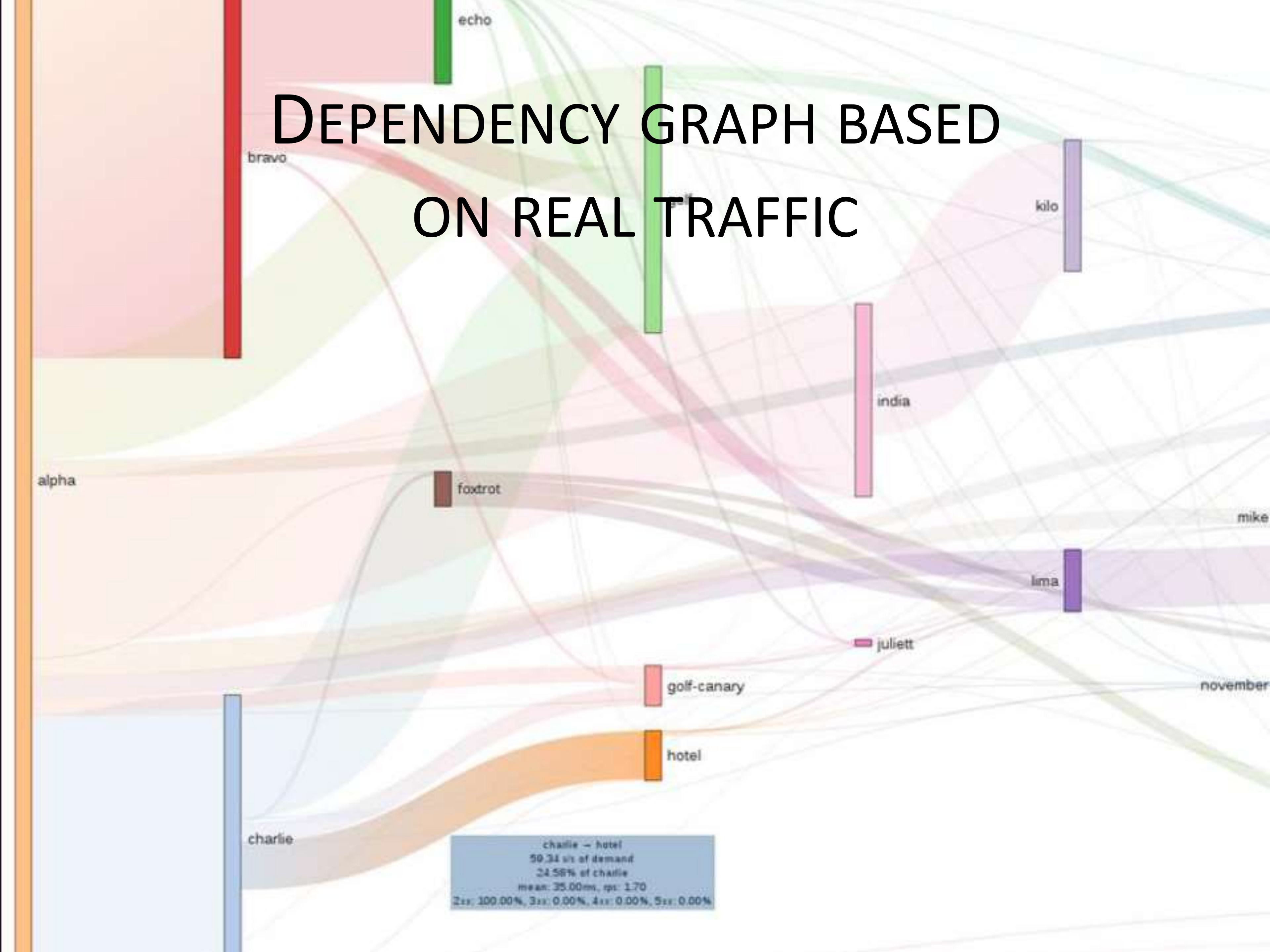
Pipeline Status: Not Started | ▶ Running | ✓ Completed | ⚠ Failed | Canceled | Suspended

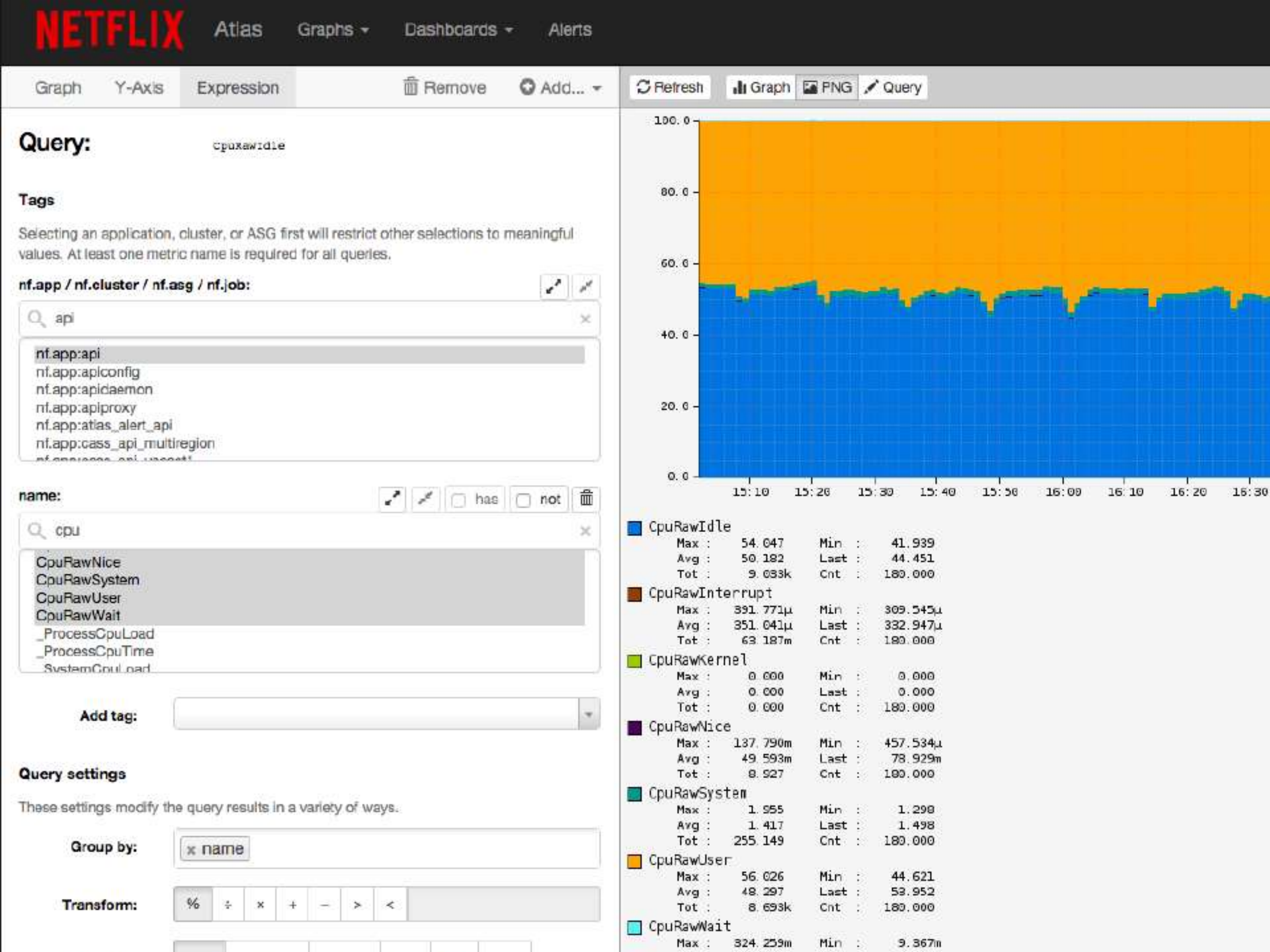
Show 5 per group

Last Month

Trigger	Status	
Enable Smoke Tested ASG Pipeline DEV (cfieber testing) [anonymous] 2015-06-25 10:54:15 PDT	COMPLETED 9 minutes	🟩 WAIT FOR APPLICATION STARTUP 🟩 POST DEPLOY SMOKE TEST (30/0) 🟩 ENABLE SERVER GROUP 🟩 DESTROY SERVER GROUP
Enable Smoke Tested ASG Pipeline DEV (cfieber testing) [anonymous] 2015-06-25 10:51:31 PDT	COMPLETED 9 minutes	🟩 WAIT FOR APPLICATION STARTUP 🟩 POST DEPLOY SMOKE TEST (36/0) 🟩 ENABLE SERVER GROUP 🟩 DESTROY SERVER GROUP
DEV (cfieber testing) Manual Start Build #1161 cfieber@netflix.com 2015-06-25 10:49:10 PDT	COMPLETED 14 minutes	🟩 MULTI-REGION ... 🟩 DEPLOY US-EAST-1 🟩 DEPLOY US-WES... 🟩 DEPLOY EU-WES... 🟩 SMOKE TEST BE... 🟩 SMOKE TEST BE... 🟩 SMOKE TEST BE...
DEV (cfieber testing) Manual Start Build #1161 cfieber@netflix.com 2015-06-25 10:39:23 PDT	FAILED Smoke Test Before Enable (us-east-1)	🟩 MULTI-REGION ... 🟩 DEPLOY US-EAST-1 🟩 DEPLOY US-WES... 🟩 DEPLOY EU-WES... 🟥 SMOKE TEST BE... 🟥 SMOKE TEST BE... 🟥 SMOKE TEST BE...
developer-michaels Triggered Build	FAILED	🟩 🟩 🟩 🟩 🟥

DEPENDENCY GRAPH BASED ON REAL TRAFFIC





Query: cpuRawIdle

Tags

Selecting an application, cluster, or ASG first will restrict other selections to meaningful values. At least one metric name is required for all queries.

nf.app / nf.cluster / nf.asg / nf.job:

api

- nf.app:api
- nf.app:apiconfig
- nf.app:apidaemon
- nf.app:apiproxy
- nf.app:atlas_alert_api
- nf.app:cass_api_multiregion

name: □ has □ not

cpu

- CpuRawNice
- CpuRawSystem
- CpuRawUser
- CpuRawWait
- _ProcessCpuLoad
- _ProcessCpuTime
- SystemCpuLoad

Add tag:

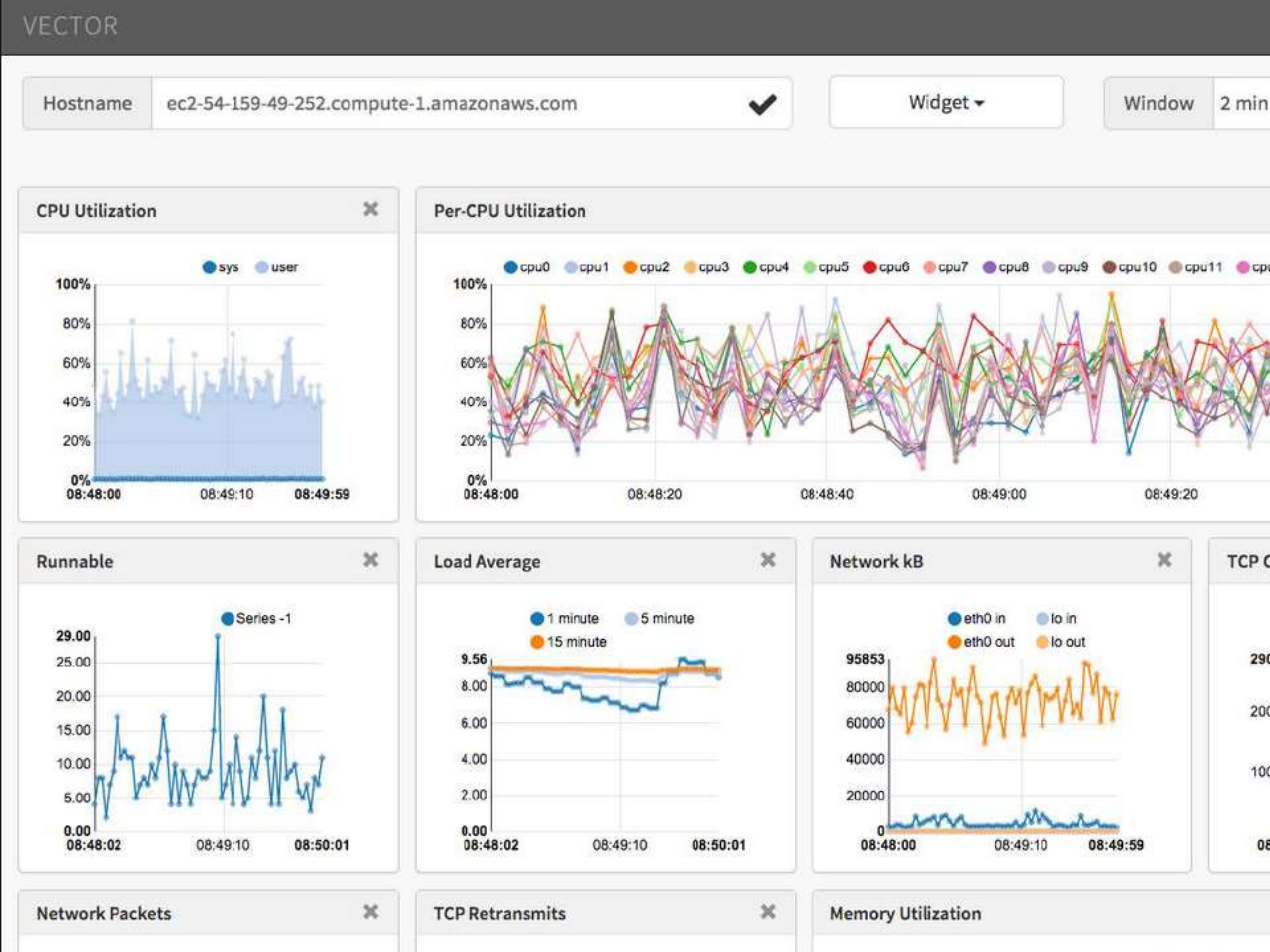
Query settings

These settings modify the query results in a variety of ways.

Group by: × name

Transform: % ÷ × + − > <

```
CpuRawIdle
    Max :    54.047    Min  :    41.939
    Avg :    50.182    Last :    44.451
    Tot :     9.033k   Cnt  :   180.000
CpuRawInterrupt
    Max :   391.771µ   Min  :   309.545µ
    Avg :   351.041µ   Last :   332.947µ
    Tot :    63.187m   Cnt  :   180.000
CpuRawKernel
    Max :     0.000    Min  :     0.000
    Avg :     0.000    Last :     0.000
    Tot :     0.000    Cnt  :   180.000
CpuRawNice
    Max :   137.790m   Min  :   457.534µ
    Avg :    49.593m   Last :    78.929m
    Tot :     8.927    Cnt  :   180.000
CpuRawSystem
    Max :     1.955    Min  :     1.298
    Avg :     1.417    Last :     1.498
    Tot :   255.149    Cnt  :   180.000
CpuRawUser
    Max :    56.026    Min  :    44.621
    Avg :    48.297    Last :    53.952
    Tot :     8.693k   Cnt  :   180.000
CpuRawWait
    Max :   324.259m   Min  :     9.367m
```



VECTOR

Hostname: ec2-54-159-49-252.compute-1.amazonaws.com

Window: 2 min

How big should be my microservices ?

RESTish microservices ?

Isn’t it SOA ?

The end of the monoliths ?

FAQ

How big ?

- Microservices ownership implies that each team is responsible for the entire lifecycle
  - functional vs divisional organizations
  - product management, development, QA, documentation, support
- Sizing depends on the system you’re building
  - Amazon 2PT principle - Two Pizza Teams
  - 6 to 10 people to build/deploy/maintain a microservice
  - an average microservice at Gilt consists of 2000 lines of code, 5 source files, and is run on 3 instances in production

RESTish Microservices ?

- REST
  - Web lingua franca, 100% interoperable
  - development cost is generally higher
  - best practices : provide client sdks, (ex : generated from Swagger/RAML or other API description languages)
  - performance issues if not well-designed (chattiness)
  - best practices : experience based and coarser grained APIs
- RPC
  - optimized communication via binary formats
  - automated generation from IDL, polyglot by default
  - integrated support multiples scenarios : request/response, streaming, bi-directional streaming

How RESTish Microservices ?

- RPC vs REST style depends on the system you're building and teams existing skills set
- Whatever the style, your microservices architecture MUST provide
  - Services Discovery,
  - Reliable Communications,
  - Operational insights (logs, monitoring, alerts, real time analysis)

GOT IT, BUT ISN’T IT SOA ?

- SOA so what ?
  - Enterprise SOA
  - Event-driven architecture (Pub/Sub)
  - Streaming Services (real-time time series, bidirectional)
  - Container-Services (ala Docker)
  - Nanoservices (ala AWS Lambda)

- Simply stated : Microservices are a SOA style for systems whose first goal is to scale

  ⇒ in details, let’s see how microservices differ from…

Microservices vs Enterprise SOA

- Enterprise SOA is often seen as
  - multi-year initiatives, costs millions
  - complex protocols with productivity and interoperability challenges
  - central governance model that inhibits change
- Enterprise SOA is more about integrating siloed monoliths
  - generally via a smart and centralized service bus
- Microservices is scalable SOA
  - an architectural style to design, develop and deploy a large and complex system, so that it is easier to scale and evolve

vs Enterprise SOA

√ Componentization via Services

√ Organized around Business Capabilities

Products not Projects

Smart endpoints and dumb pipes

Decentralized governance

Decentralized data management

Infrastructure automation

v Design for failure

√ Evolutionary design

VS EVENT DRIVEN ARCHITECTURE

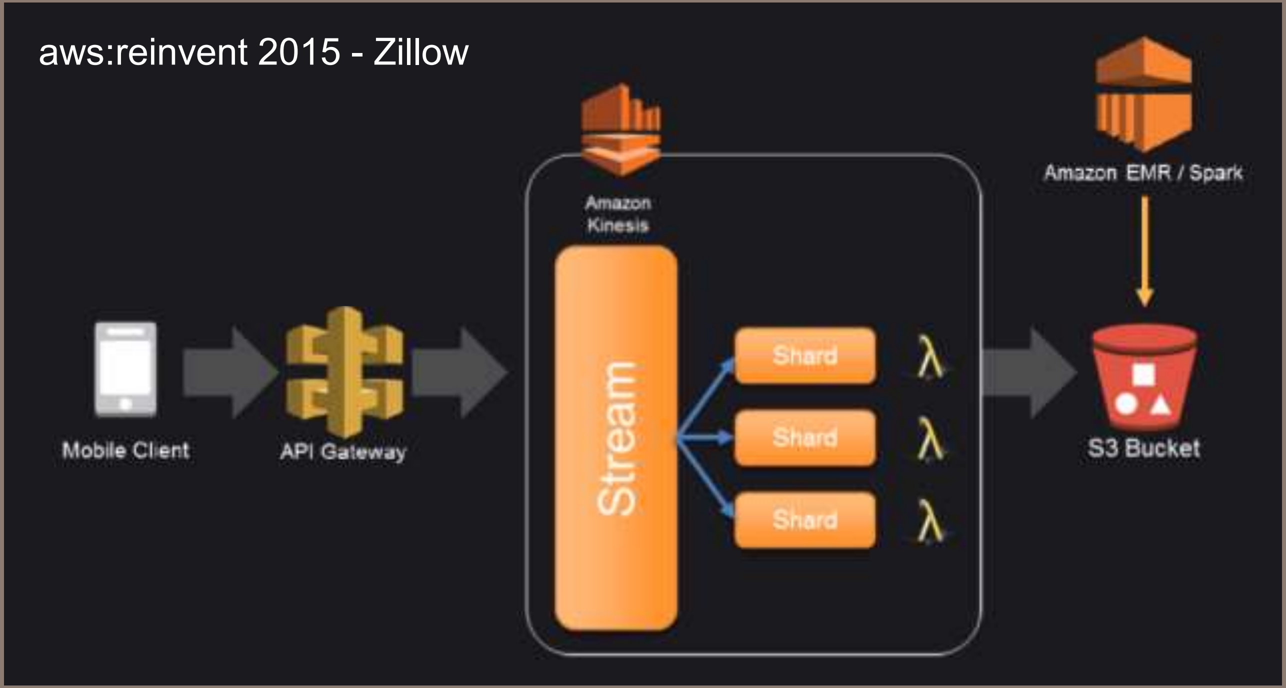
- EDA fits well with Document oriented systems

and information flows

- Communication between microservices can be a mix of RPC (ie, P2P calls) and EDA calls
- See EDA as a communication pattern for your microservices
- Can address choreography, orchestration, pipeline requirements

VS STREAMING SERVICES

- Streaming services fit well
  - if you have large volumes (log entries, IoT),
  - and/or if you aim at real time analysis
- Data ingestion endpoint of a microService
  - Real time analysis of a mobile app usage


aws:reinvent 2015 - Zillow

VS CONTAINER SERVICES

- Containers provide the infrastructure to deploy your microservices independently one from another
- See Container Services as a building block of your global microservices architecture

VS NANOSERVICES

- Nanoservices are small pieces of code (functions)
  - Example : AWS Lambda, Auth0 Webtasks
- A microservice may leverage 1+ nanoservices

SUMP UP

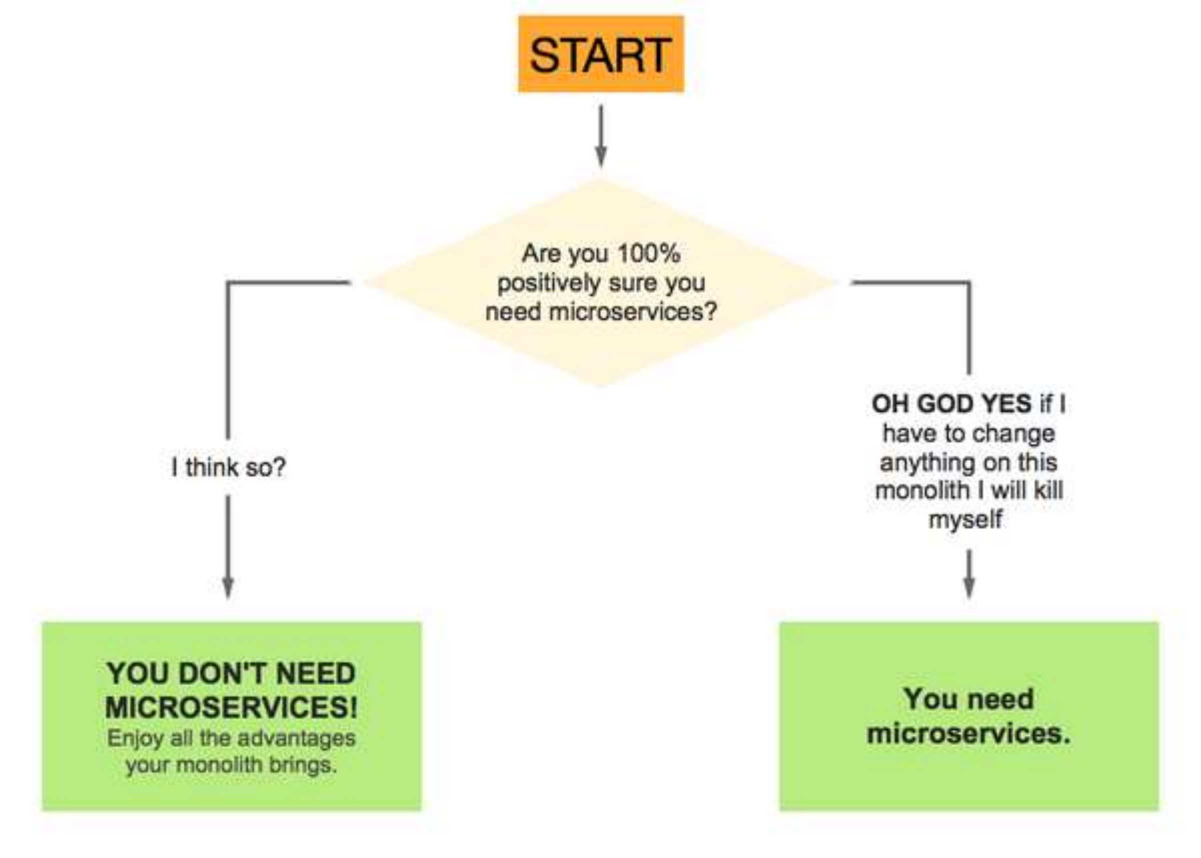
Microservices Patterns

- Solid communications
  - Fault tolerant librairies
  - Service discovey
- Committed teams
  - Devops culture
  - Code/Test/Deploy/Support, 24/7
  - Automation
- Ownership
  - Organisation aligned with the strategy
  - Insights via real time monitoring

Microservices is a long journey

- Several years to implement
  - Communications, infrastructure, automation, monitoring, teams organization
  - The price to pay for extreme agility in a complex system at scale
- Prepare for iterative reworks
  - Multiples languages => maintainability
  - Numerous building blocks => updates & security
  - EOL of microservices stacks : manage your technical debt
  - Dependency Hell : keep control of your microservices segmentation
  - Ownership : is your organization ready ?

The End of Monoliths ?



START
Are you 100% positively sure you need microservices?
I think so?
OH GOD YES if I have to change anything on this monolith I will kill myself
YOU DON'T NEED MICROSERVICES! Enjoy all the advantages your monolith brings.
You need microservices.

http://www.stavros.io/posts/microservices-cargo-cult/

From Monoliths to Microservices

- Velocity of innovation for complex systems
  - Keep your monolith as is if you don't need to speed up features delivery
- To prepare for the journey
  - switch from layered architecture to internal APIs,
  - automate integration and deployment,
  - reorganize from divisional to functional teams committed to business and owning their code

REFERENCES

Microservices, Martin Fowler, James Lewis

- http://martinfowler.com/articles/microservices.html

A day in the life of a Netflix Engineer using 37% of the internet

- http://fr.slideshare.net/AmazonWebServices/dvo203-the-life-of-a-netflix-engineer-using-37-of-the-internet

Dzone – Geting started with Microservices

- https://dzone.com/refcardz/getting-started-with-microservices

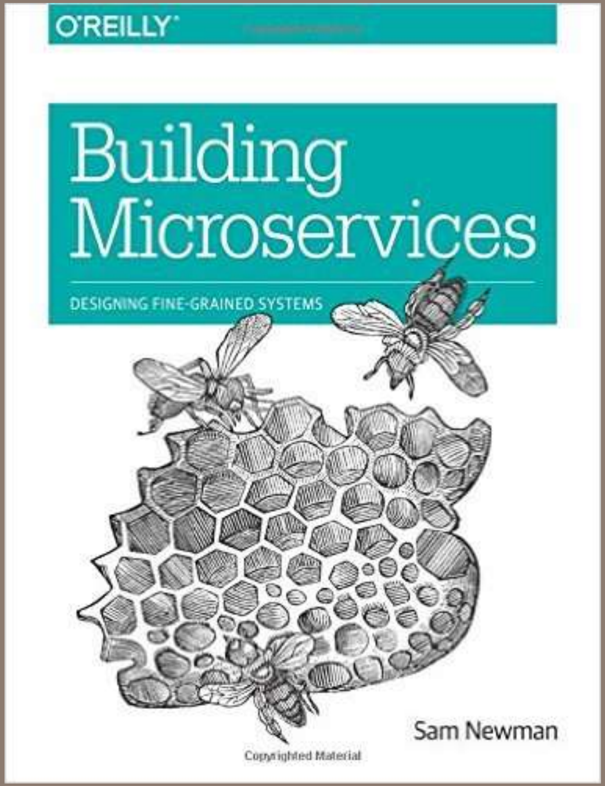
SoundCloud – How we ended up with microservices

- http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html

gRPC - boilerplate to high-performance scalable APIs

- http://fr.slideshare.net/AboutYouGmbH/robert-kubis-grpc-boilerplate-to-highperformance-scalable-apis-codetalks-2015

To Go Further


O'REILLY
Building Microservices
DESIGNING FINE-GRAINED SYSTEMS
Sam Newman
Copyrighted Material