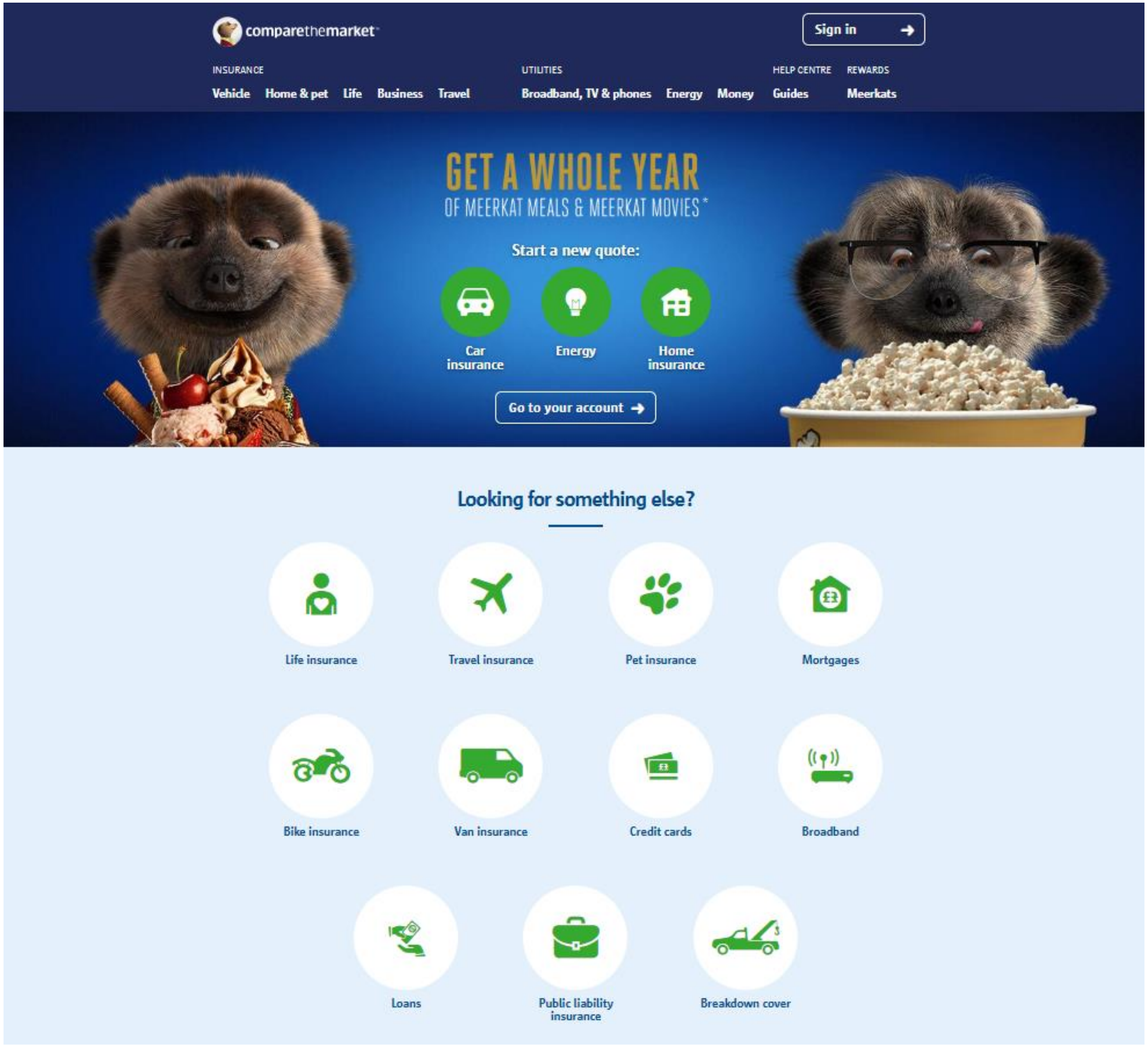
- Why microservices?
- Essence of microservices
- Refactoring microservice-based architectures
- Concluding remarks
- Case studies
    - comparethemarket.com



UK’s most popular (insurance) price comparison website

comparethemarket.com

comparethemarket.com

- *Price comparison web site, various product verticals*
- *Initially one team, one system*
- *Rapid growth of customer numbers and staff*
- *Change*
  - *from monolith architecture to microservices with single responsibility*
  - *from relational DB (point of contention, changes breaking other teams'apps) to Docker-based approach with mongoDB EE*
- *Achievement*
  - *ability to change quickly to catch new consumer trends and tech adoption trends*

comparethemarket.com

- Issues of monolith
  - coordinate releases amongst teams
  - long feature cycles
  - changes negatively impacting other product teams
  - not being able to expose their functionality for partners through APIs
  - inability to scale a single database with 200+ tables and a single point of failure any other way but vertically
- Solution

  Change the layout of the internal teams from groups of specialists (UI, DBA, MW) to **autonomous product teams** that could change what was theirs through **microservices**, a set of smaller applications chained together instead of a single, master application

The pros and cons of microservices: Lessons learned as Comparethemarket.com moves away from a single, monolithic application. ComputerWorldUK. June 2016.

Benefits

1 **More manageable**

“By taking a single complicated thing and breaking it into smaller things, they become easier to reason about.” (Unix design philosophy)

2. **Flexible tech stack**

“We haven’t had to commit to a tech stack, we use the right tools for the problem e.g. if we have a machine learning problem we use Python as there is a wealth of open source libraries we can make use of. If we just want to do a restful API we use Node.js.”

3. **Reduced cost of failure**

“Big services tend fail in a big way. If your monolithic application isn’t working you aren’t making money. In a microservice world if a service stops working it doesn’t take down the entire businesses ability to make money. It drives innovation and shortens the build, measure, learn cycle so people feel more empowered to experiment and it is freeing to build things that don’t have to last ten years.”

4. **More specialized**

“You can use microservices to specialise what you do. So if you have customer data you can put that into a single microservice, wrap that in layers of additional security and not burden the rest of your services with that same constraint. In a monolithic world there needs to be that highest common denominator.”

5. **Independence**

“Teams can have their own backlog of change and scale and release independently which allows the organisation as a whole to move faster.”

Drawbacks

1. Complexity

“By splitting one large thing into small bits you end up having more bits.”

2. Disparate data sources

“You end up with many data sources and so how do you deal with that data?”

3. Creating a distributed monolith

“You have to put some effort into design upfront. Just taking a monolith and breaking it into microservices won’t cut it if those bits were tightly coupled to start with, you will just end up with tightly coupled microservices, which is a distributed monolith, which is the worst of all worlds.”

To minimise these drawbacks:

- **investment in automation**

  ”Trying to realise **continuous delivery** by making releases reliable and repeatable and the whole process a non-event, rather than a 4am in the morning cold sweat process.”

- **ensure that all monitoring and alerting frameworks looked the same**

  “all of our microservices emit a standard set of metrics for latency, throughput etc. All of these metrics have a standardised threshold.”

- **realistic approach to faults and failures**

  “by accepting the fact that computers break and networks fail we have focused on becoming fault tolerant.”

  if a non-critical dependency goes down a service will still **respond in a degraded form**

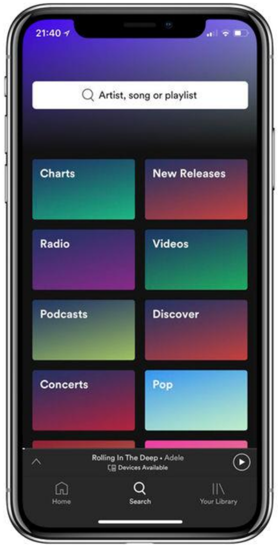
  if the failure is a critical dependency like the database, it **fails fast** “so that the client doesn’t expend lots of resources waiting for a response that doesn’t come back”

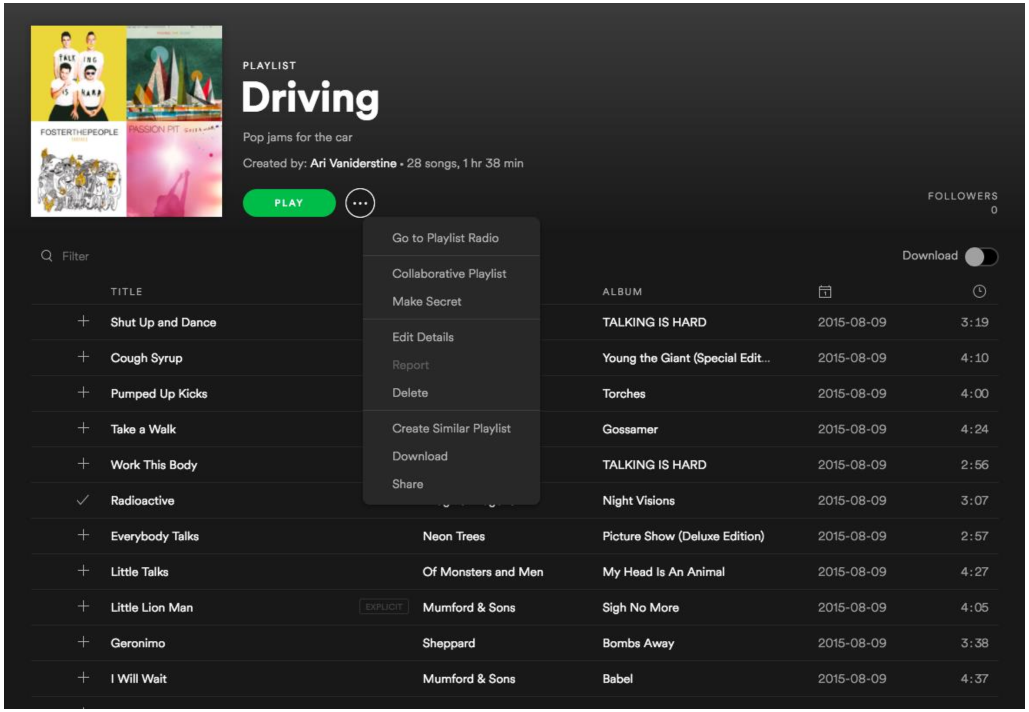
- **data management**

  “Each microservice emits data in the form of a JSON object. This gives us a real-time view of what is happening across our estate and is scalable.”

  “Each microservice has a private data store so no one else relies on the structure of the data which aligns with the Mongo no-schema metadata approach. The **team can change the structure of the data in that store without having to coordinate**.”

- Why microservices?
- Essence of microservices
- Refactoring microservice-based architectures
- Concluding remarks
- Case studies
  - comparethemarket.com
  - Spotify





PLAYLIST

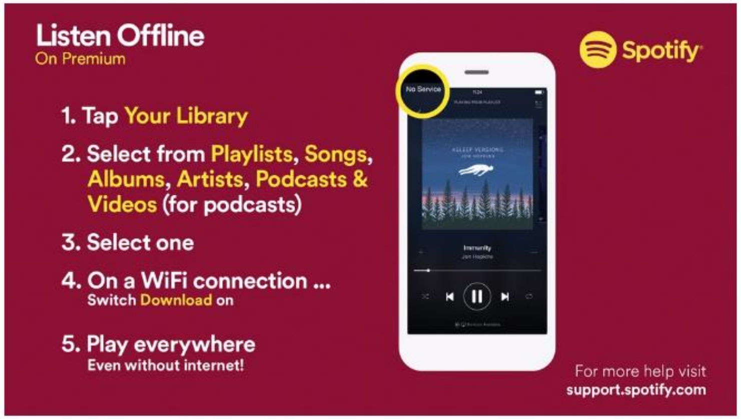
Driving

Pop jams for the car

Created by: Ari Vaniderstine • 28 songs, 1 hr 38 min

FOLLOWERS
0

	TITLE			ALBUM		
+	Shut Up and Dance			TALKING IS HARD	2015-08-09	3:19
+	Cough Syrup			Young the Giant (Special Edit...	2015-08-09	4:10
+	Pumped Up Kicks			Torches	2015-08-09	4:00
+	Take a Walk			Gossamer	2015-08-09	4:24
+	Work This Body			TALKING IS HARD	2015-08-09	2:56
✓	Radioactive			Night Visions	2015-08-09	3:07
+	Everybody Talks		Neon Trees	Picture Show (Deluxe Edition)	2015-08-09	2:57
+	Little Talks		Of Monsters and Men	My Head Is An Animal	2015-08-09	4:27
+	Little Lion Man	EXPLICIT	Mumford & Sons	Sigh No More	2015-08-09	4:05
+	Geronimo		Sheppard	Bombs Away	2015-08-09	3:38
+	I Will Wait		Mumford & Sons	Babel	2015-08-09	4:37



Listen Offline
On Premium

1. Tap Your Library
2. Select from Playlists, Songs, Albums, Artists, Podcasts & Videos (for podcasts)
3. Select one
4. On a WiFi connection ...
   Switch Download on
5. Play everywhere
   Even without internet!

For more help visit
support.spotify.com

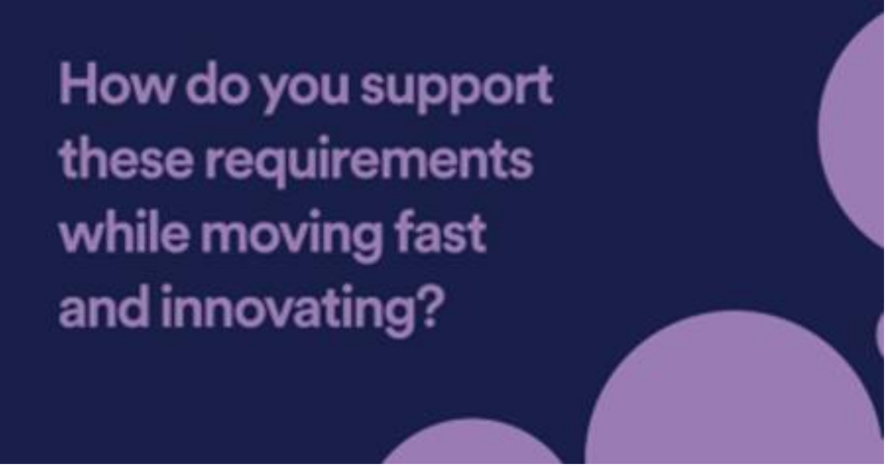
Some requirements

- Scale to **millions of users**
- Support **multiple platforms**
  - Web, mobiles, laptops, embedded devices, game platforms, …
- Handle **complex business rules**
- **Be competitive** in a fast moving market
  - React quickly
  - Innovate

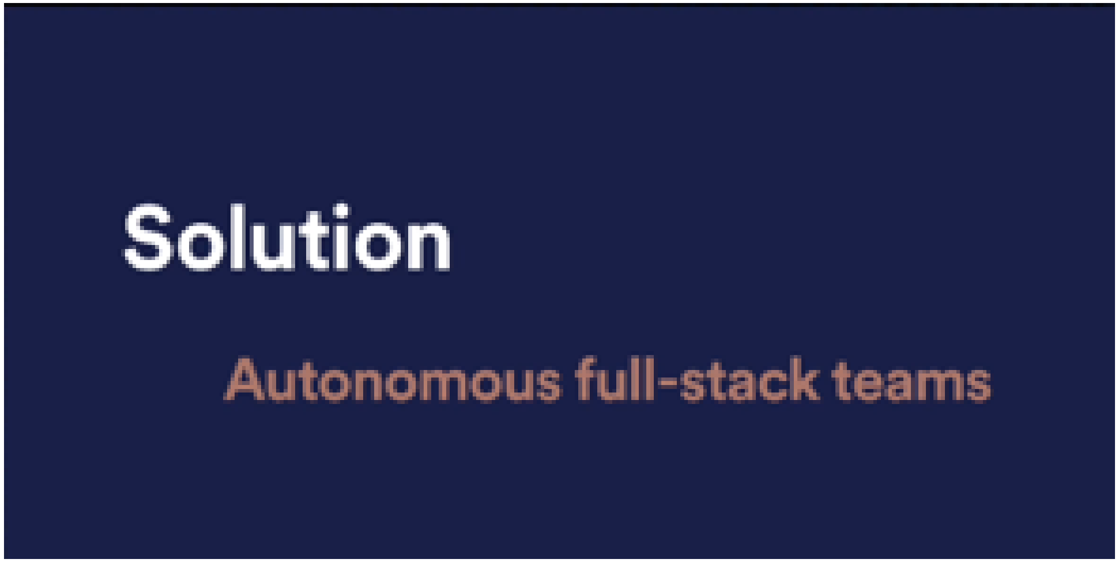
K. Goldsmith. Microservices at Spotify. GOTO 2015.

Some numbers

- 75M+ Monthly Active Users
  - Long user sessions (average 23 mins), white noise all night
- 58 countries
- >20K songs added per day
- >2B user-generated playlists (+ 75M playlists created by Spotify)
- Incredibly complex business rules
  - choice of which version of track is served depends on business rules
  - different licensing agreements & contracts across different countries and labels
- Lots of competition



How do you support these requirements while moving fast and innovating?



Solution

Autonomous full-stack teams

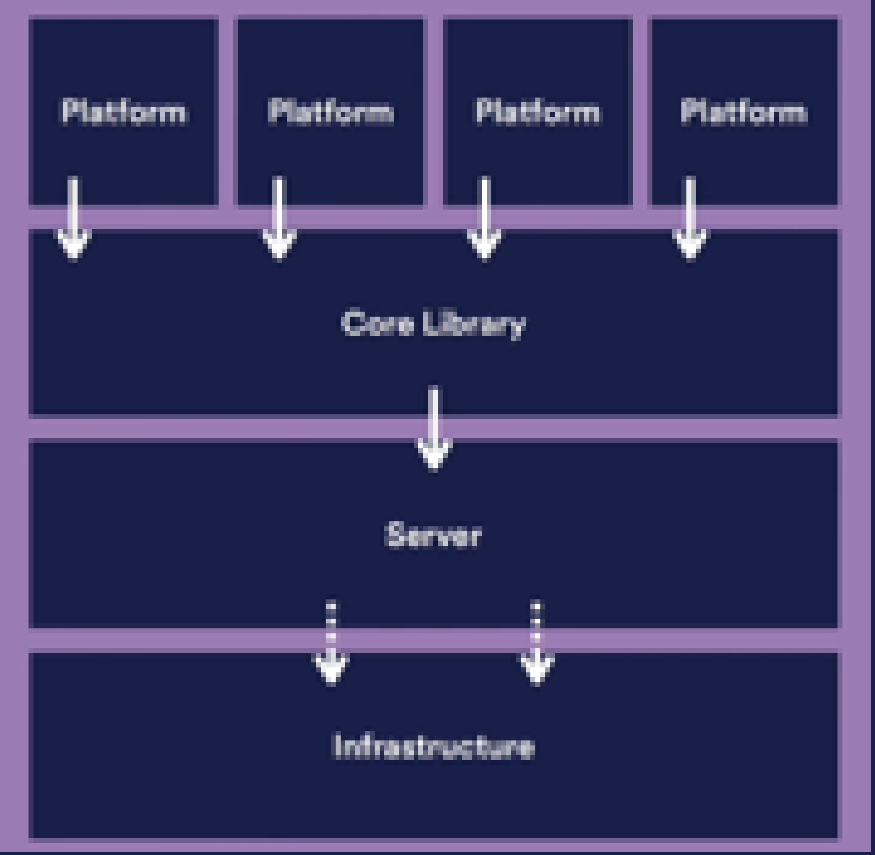
Autonomous

adjective
au·ton·o·mous - \ȯ-ˈtä-nə-məs\

(of a country or region) having the freedom to govern itself or control its own affairs.
"the federation included sixteen autonomous republics"

having the freedom to act independently.

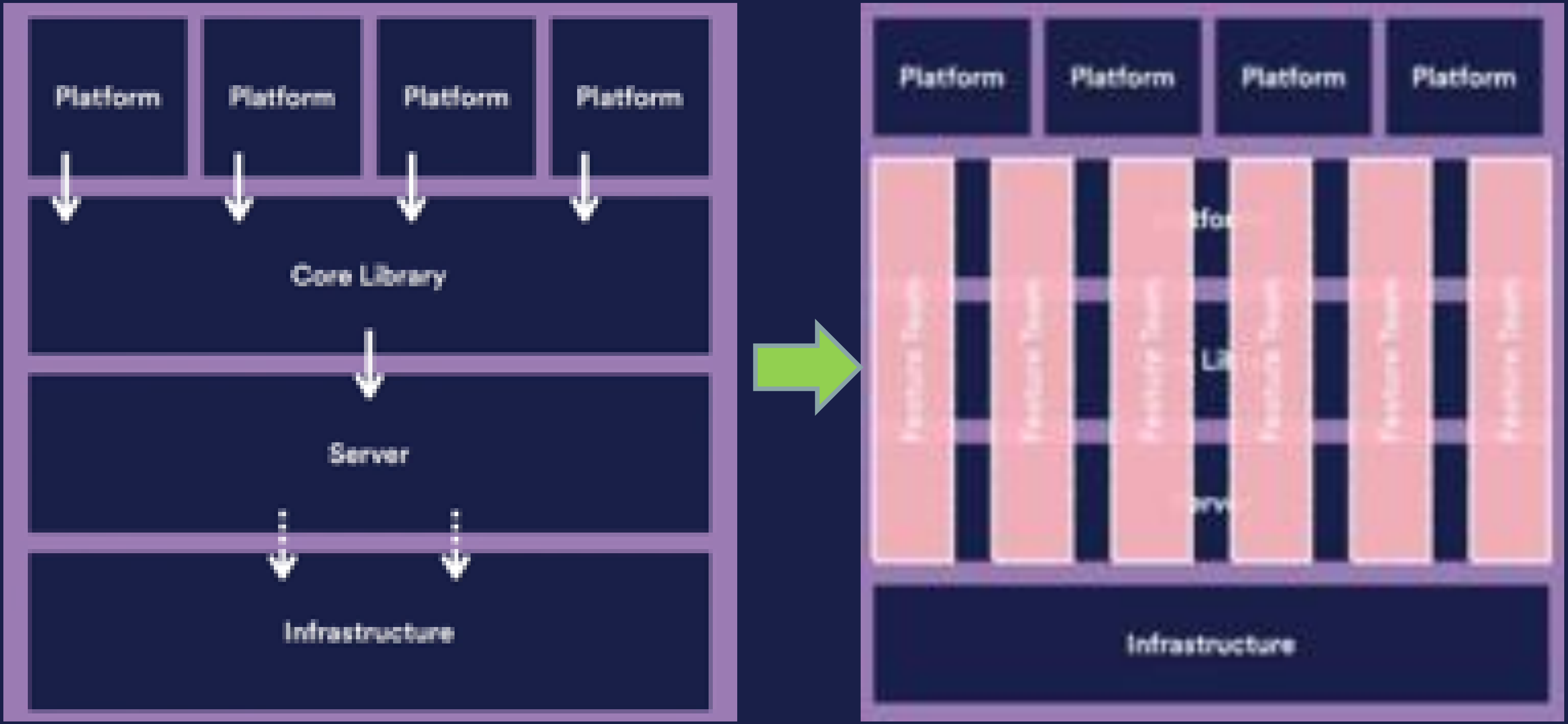
"school governors are legally autonomous"
synonyms: self-governing, independent, sovereign, free, self-ruling, self-determining, autarchic; self-sufficient
"an autonomous republic"

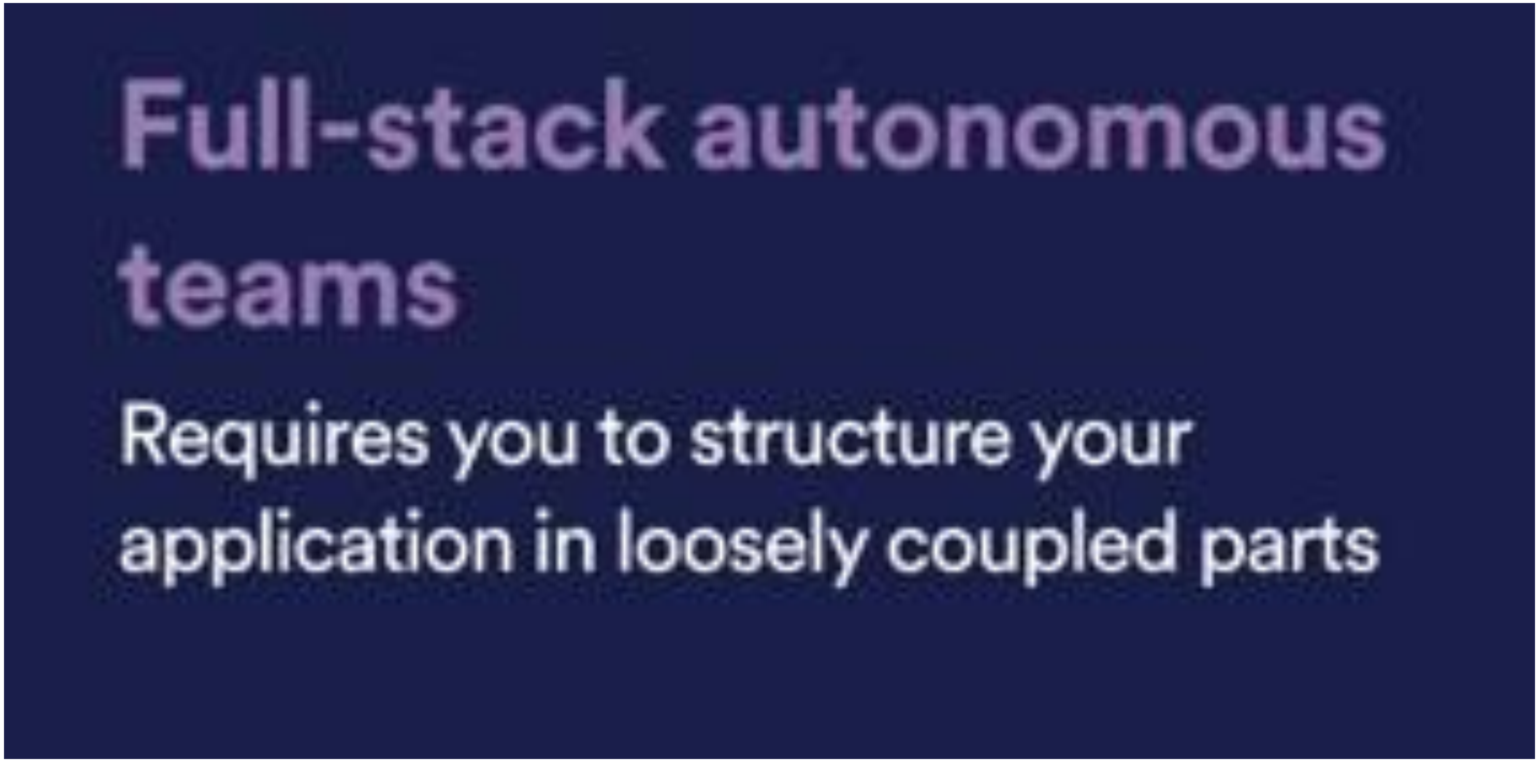


- Standard way of building connected apps
- «Horizontal» teams
- Hard to build new feature: lots of asking to other teams

Client UX implementation *depends on*
Core Library implementation *depends on*
Server implementation *depends on*
Infrastructure implementation

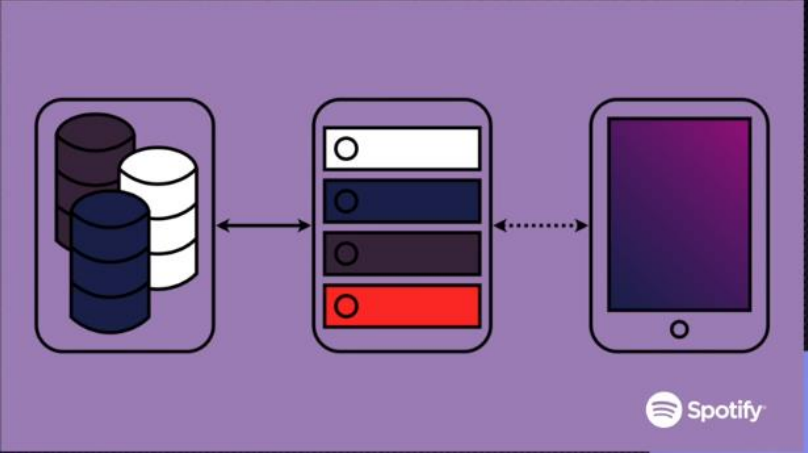
Full–stack autonomous teams

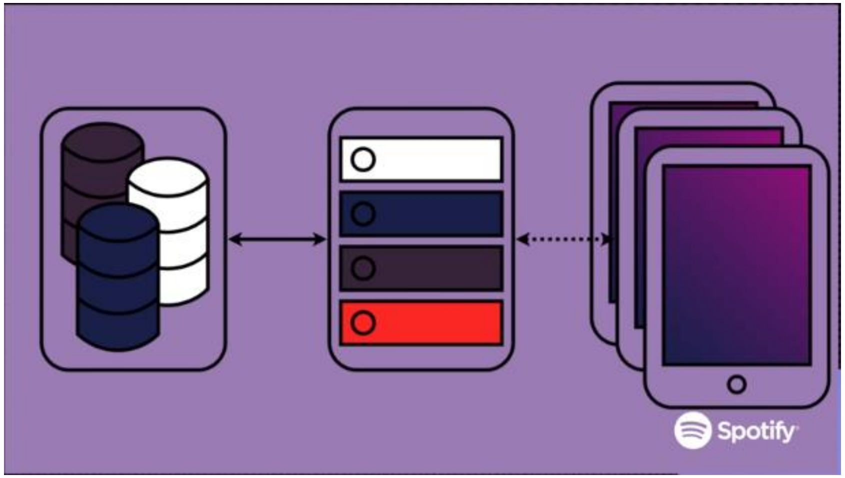




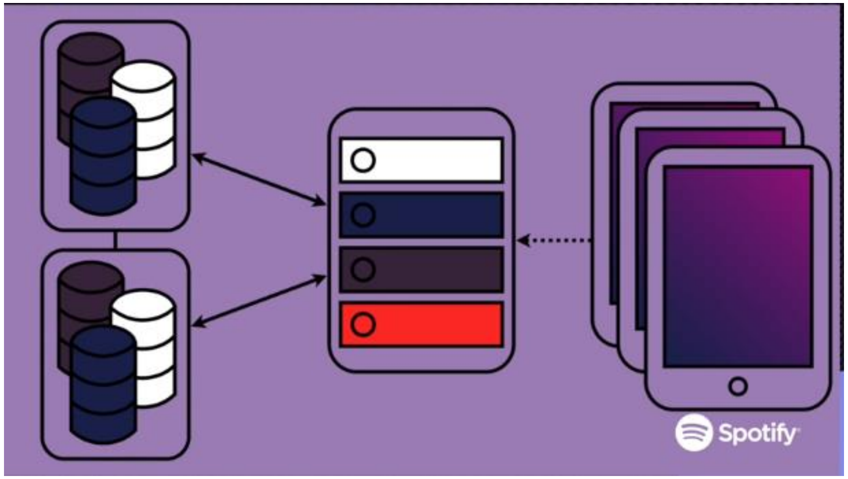
Full-stack autonomous teams

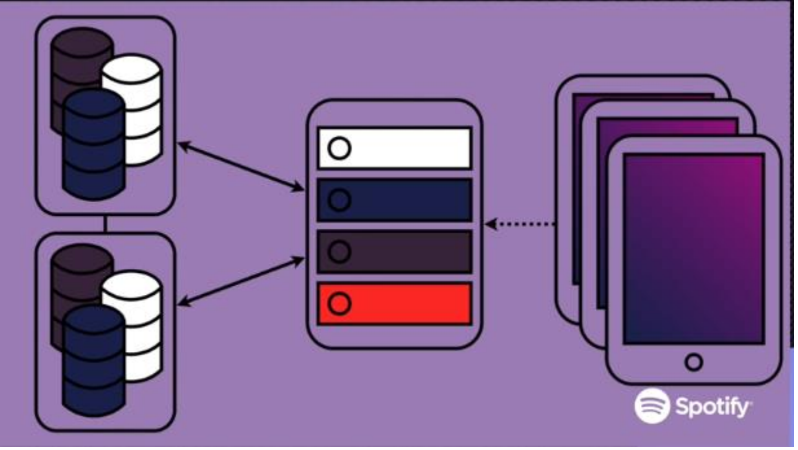
Requires you to structure your application in loosely coupled parts

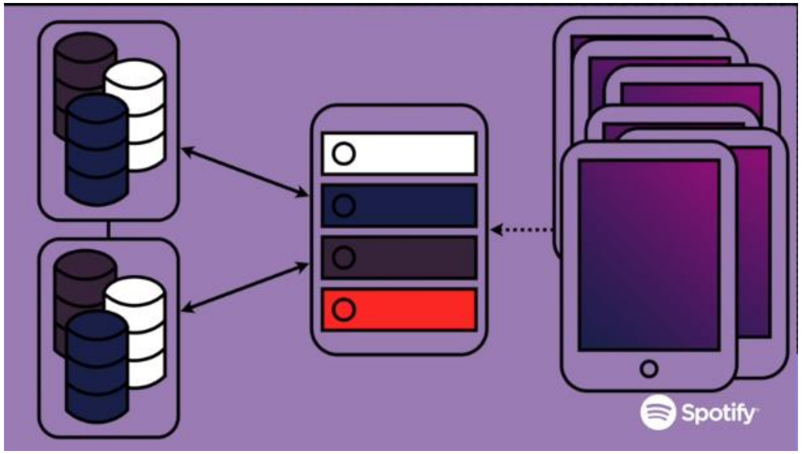




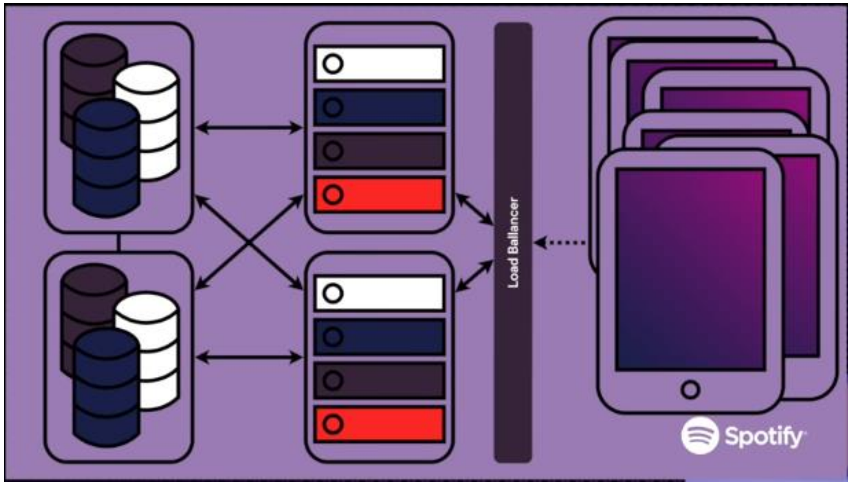
Datastore slow …





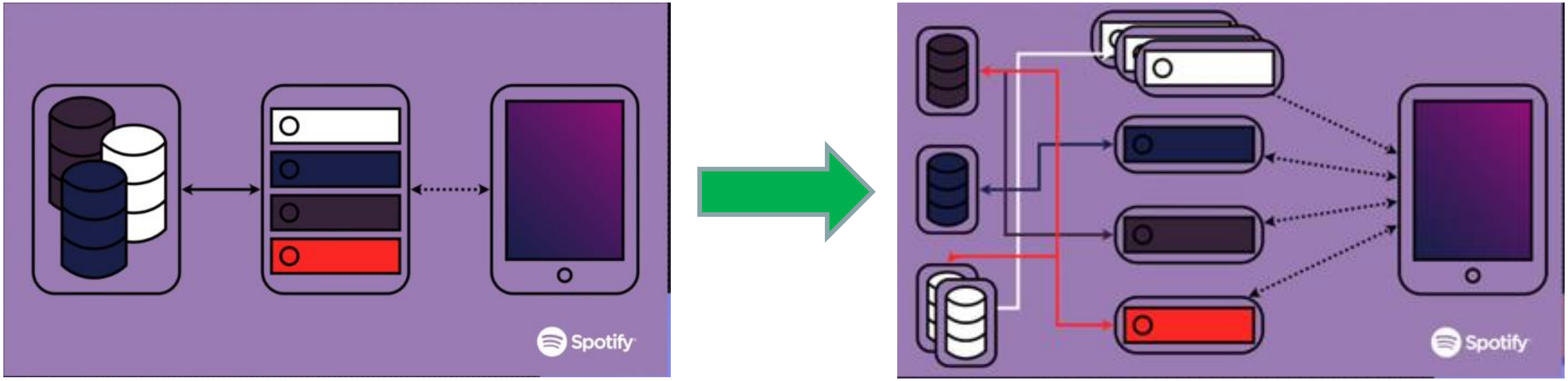


Server is slow …



Scaling?

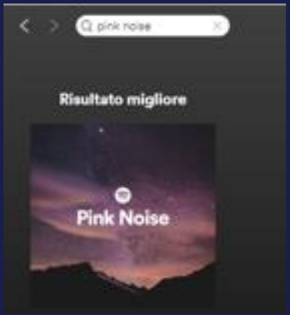
Changing?


Spotify
Spotify

Microservices yay!

- **Easier to scale** based on real-world bottlenecks
- **Easier to test**
- **Easier to deploy**
- **Easier to monitor**
- Can be **versioned independently**
- Are **less susceptible to large failures**

Crucial for embedded devices. *Manufacturers will not update device ever, lamp will always use (old) API version*



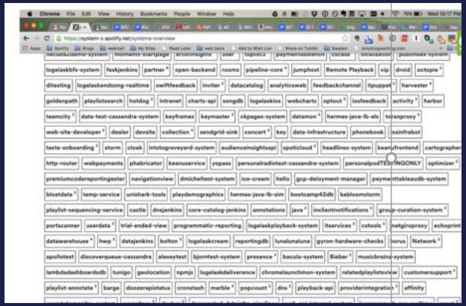
Users do no realize if «suggest service» fails

Microservices boo!

- Are harder to monitor

Thousands of instances running together

- Need good documentation / discovery tools

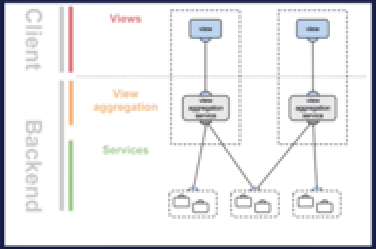

Chrome File Edit View History Bookmarks People Window Help
https://system-z.spotify.net/systems-overview

logelaskbfs-system feskjenkins partner open-backend rooms pipeline-core jumphost Remote Playback vip droid octopie
ditesting logelaskendsong-realtime swiftfeedback inviter datacatalog analyticsweb feedbackchannel itpuppet harvester
goldenpath playlistsearch hotdog intranet charts-api songdb logelaskios webcharts optout iosfeedback activity harbor
teamcity data-test-cassandra-system keyframes keymaster ckpages-system datamon hermes-java-lb-els toranproxy
web-site-developer dealer devsite collection sendgrid-sink concert key data-infrastructure phonebook iosinfrabot
taste-onboarding storm cloak intstograveyard-system audienceinsightsapi spoticloud headlines-system keanufrontend cartographer
http-router webpayments phabricator keanuservice yopass personalradiotest-cassandra-system personalpodTESTINGONLY optimizer
premiumcodereportingester navigationview dmicheltest-system ice-cream hello gcp-deloyment-manager paymenttableaudb-system
bloatdata temp-service unishark-tools playdemographics hermes-java-lb-sim bootcamp42db kabloomstorm
playlist-sequencing-service castle dnsjenkins core-catalog-jenkins annotations java inclientnotifications group-curation-system
portscanner userdata trial-ended-view programmatic-reporting logelaskplayback-system itservices cstools netgiroproxy echoprint
datawarehouse hwp datajenkins bolton logelaskcream reportingdb lunalunaluna gyron-hardware-checks horus Network
apollotest discoverqueue-cassandra alexeytest bjorntest-system presence bacula-system Bieber musicbrainz-system
lambdadashboardsdb tunigo geolocation npmjs logelaskdeliverance chromelaunchmon-system relatedplaylistsview customersupport
playlist-annotate barge doozerapistatus cronstash marble popcount dns playback-api providerintegration affinity

Many services named by developers out of context

- Create increased latency

Lots of services calling other services



Netflix view aggregation model

- Microservices for years now
- Production environment running in Spotify DCs (test environments run on AWS)
- Super happy of architecture
  - great for **scaling**
  - great for **rewriting** when needed
  - developers **deploying** by themselves whenever they want to

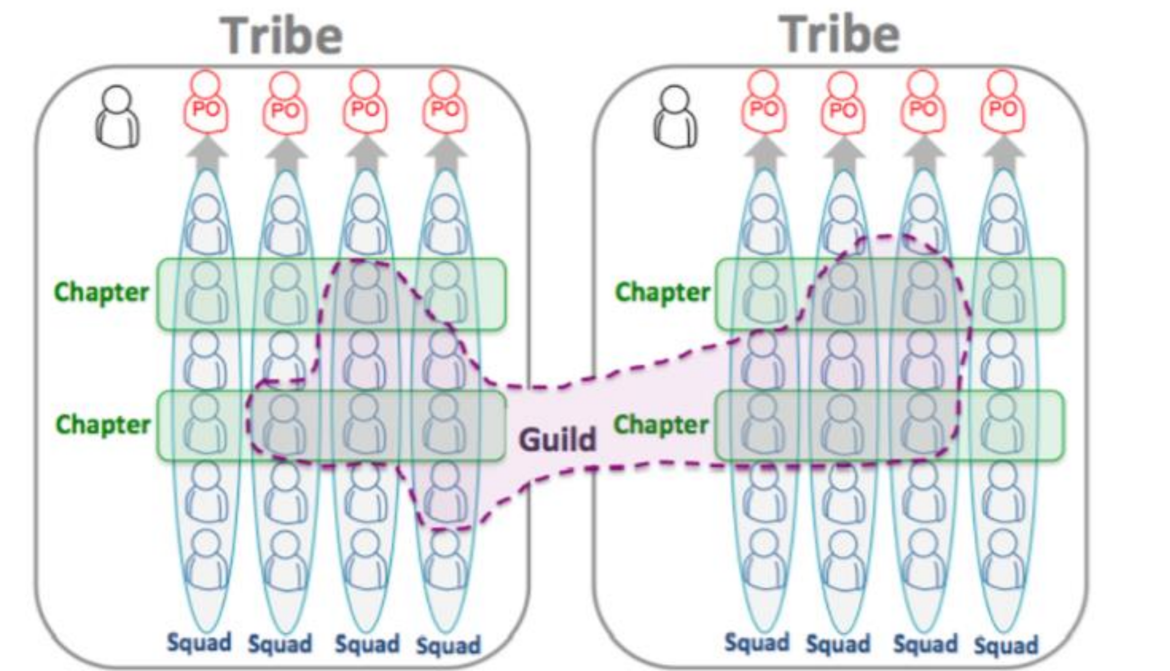
Some numbers

- 90+ teams (squads)
- 600+ developers
- 5 development offices in 2 continents
- 1 product!

- 810 active services
- ~10 services per squad

Q: «squads»?

A: Agile team organization …

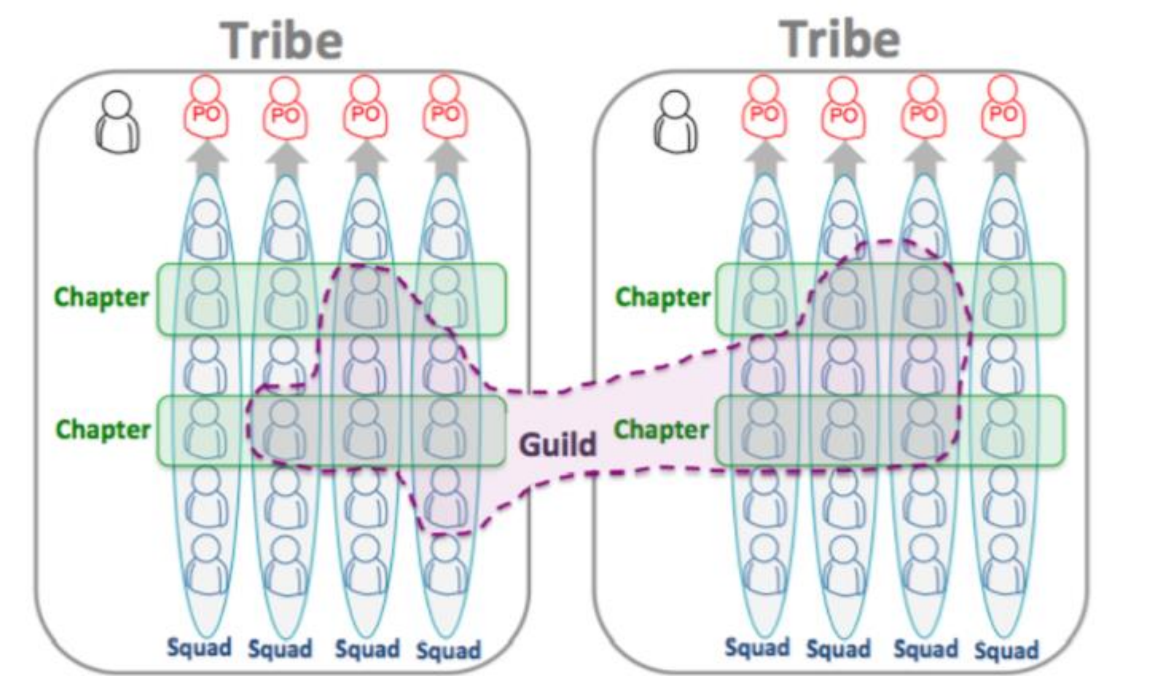


A **squad** has a dedicated Product Owner who feeds user stories to build. Squads have all the skills and tools needed to design, develop, test and release to production, being an autonomous, self-organising team, experts in their product area.

A **tribe** is a collection of squads within the same business area (e.g., mobile). The squads within a tribe sit in the same area. Spotify implements shared lounges to create inter-squad interaction, and regular informal team events and get together where squads share what they are working on. Tribe Leader is responsible for providing the right environment for all the squads.

Chapters are team members working within a special area (e.g., front office developers, back office developers, database admins, testers). Chapter members exchange ideas to promote innovation and ‘cross pollination’ across teams.

A **guild** is a community of members across the organization with shared interests (e.g., web technology, test automation), who want to share knowledge, tools code and practices.



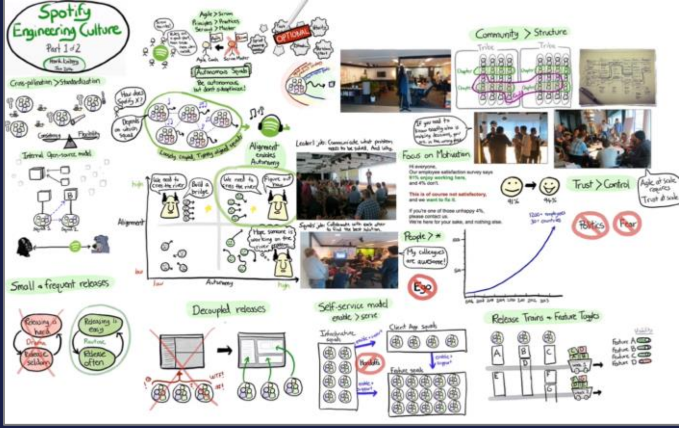
Q: How is code reviewing performed at Spotify?

A: Chapters (sometimes guilds) do code reviews for squads.
Two «+1» required to merge.

Spotify engineering culture

[Part I]

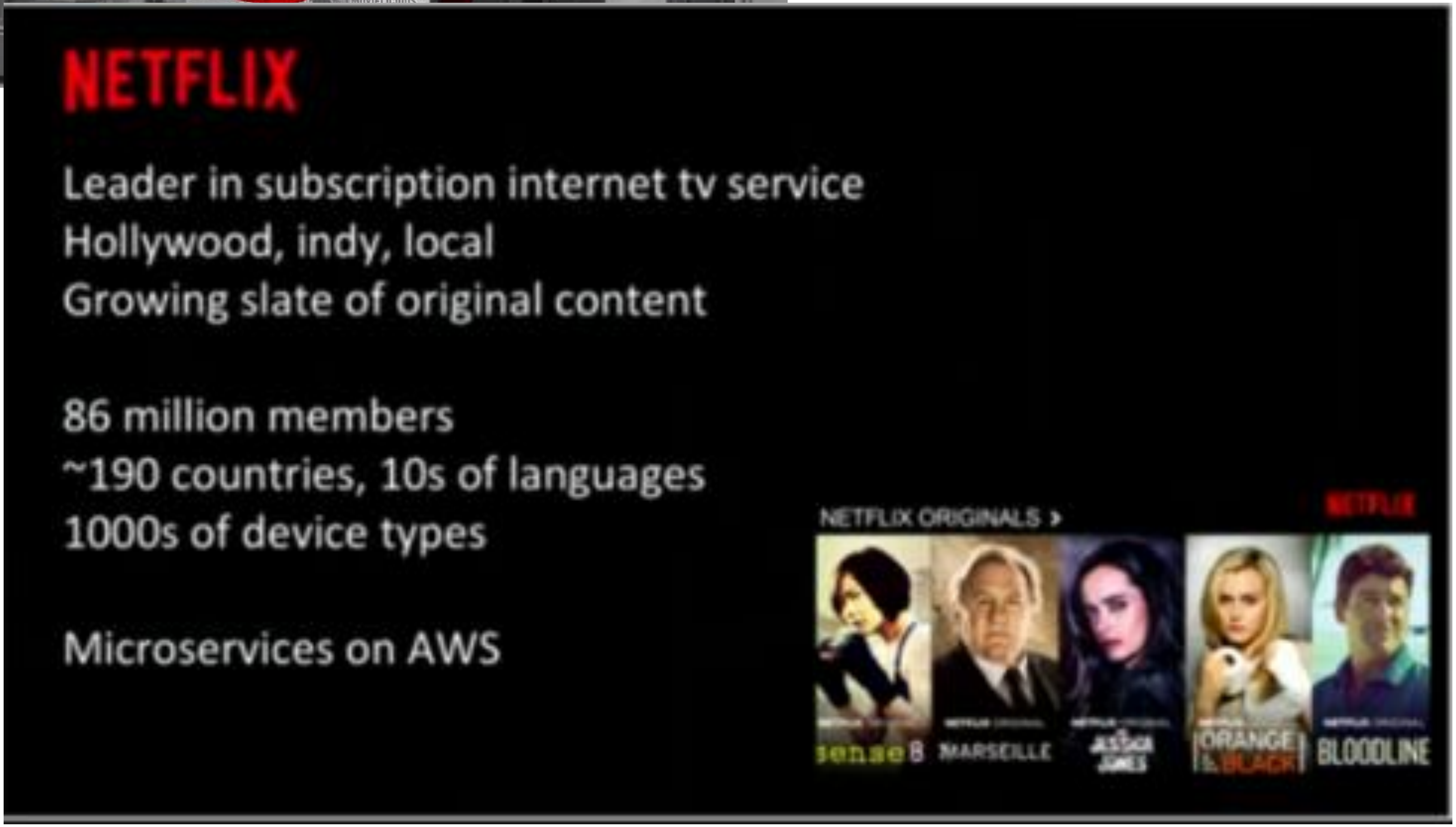
- Agile principles
  - Loosely coupled, tightly aligned squads
  - Cros- pollination
  - Internal open-source model for code
  - Frequent releases, enabled by decoupling
- Focus on people: motivation, community, trust


Spotify Engineering Culture
Part 1/2
Cross-pollination > Standardization
Internal Open-source model
Small + frequent releases
Decoupled releases
Self-service model
Community > Structure
Focus on Motivation
Trust > Control
Ego
Politics
Fear
Release Trains + Feature Toggles

- Fail fast → learn fast → improve fast
  - Spotify is a fail-friendly environment
    - ❑ Fail walls, post-mortems, retrospectives
  - Failure recovery (rather than failure avoidance)
  - Limited blast radius via decoupled architecture and gradual rollout
- Product development approach
  - Lean startup principles (think|build|ship|tweak)
  - Impact > Velocity
  - Innovation > Predictability
  - Value delivery > Plan fullfillment
- 10% hack time
- Experiment-friendly culture

- Waste-repellent culture
  - «if it wroks, keep it, otherwise dump it»
  - Minimize need of big projects
  - Minimal bureaucracy to avoid chaos
  - Improvement boards, Toyota improvement «kata»
- «Health culture heals broken process»
  - Culture-focused roles (agile coaches)
  - Boot camps
  - Story telling

- Why microservices?
- Essence of microservices
- Refactoring microservice-based architectures
- Concluding remarks
- Case studies
  - comparethemarket.com
  - Spotify
  - Netflix



NETFLIX

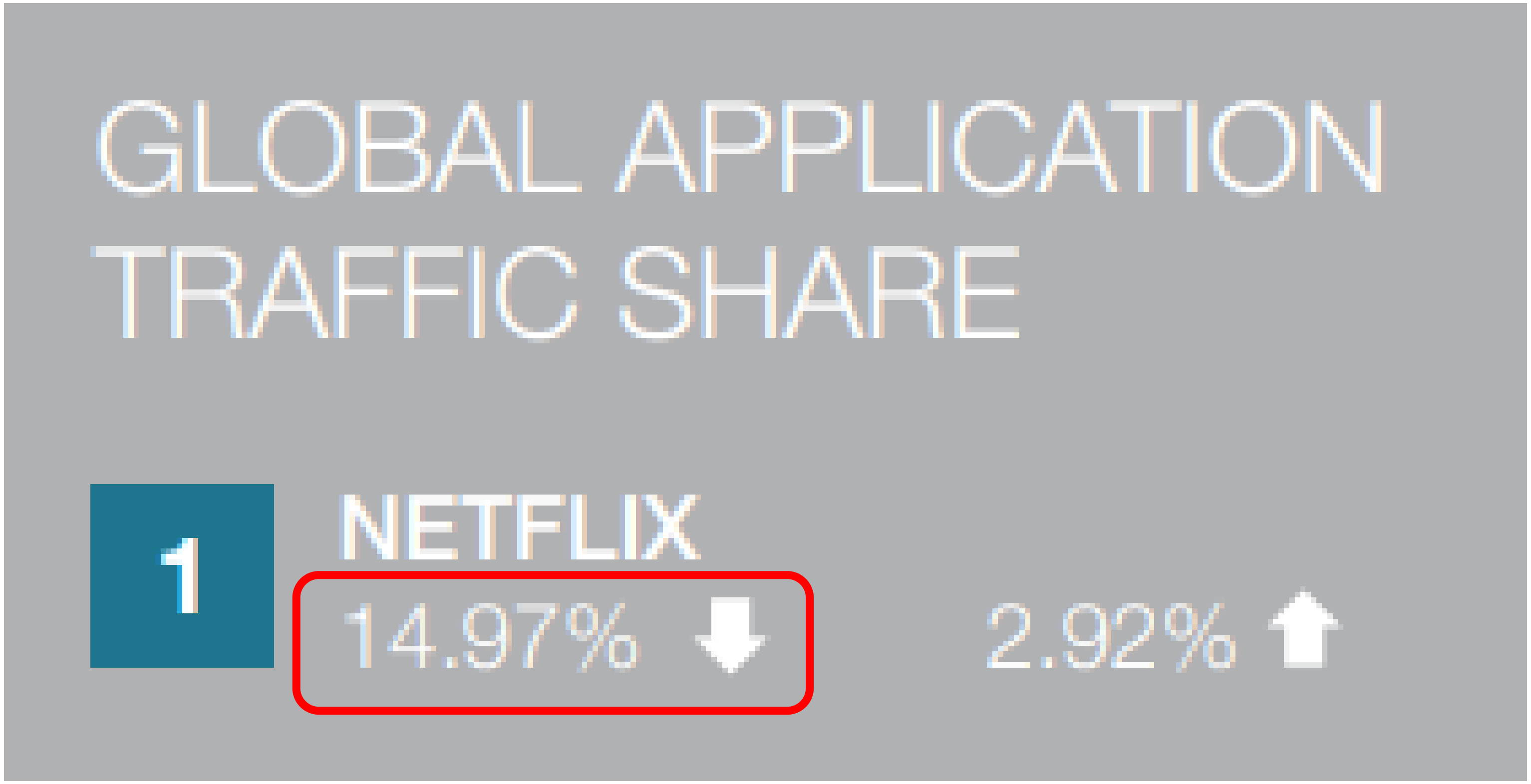
Leader in subscription internet tv service
Hollywood, indy, local
Growing slate of original content

86 million members
~190 countries, 10s of languages
1000s of device types

Microservices on AWS

SANDVINE

The Global Internet
Phenomena Report
October 2018



GLOBAL APPLICATION TRAFFIC SHARE

1 NETFLIX

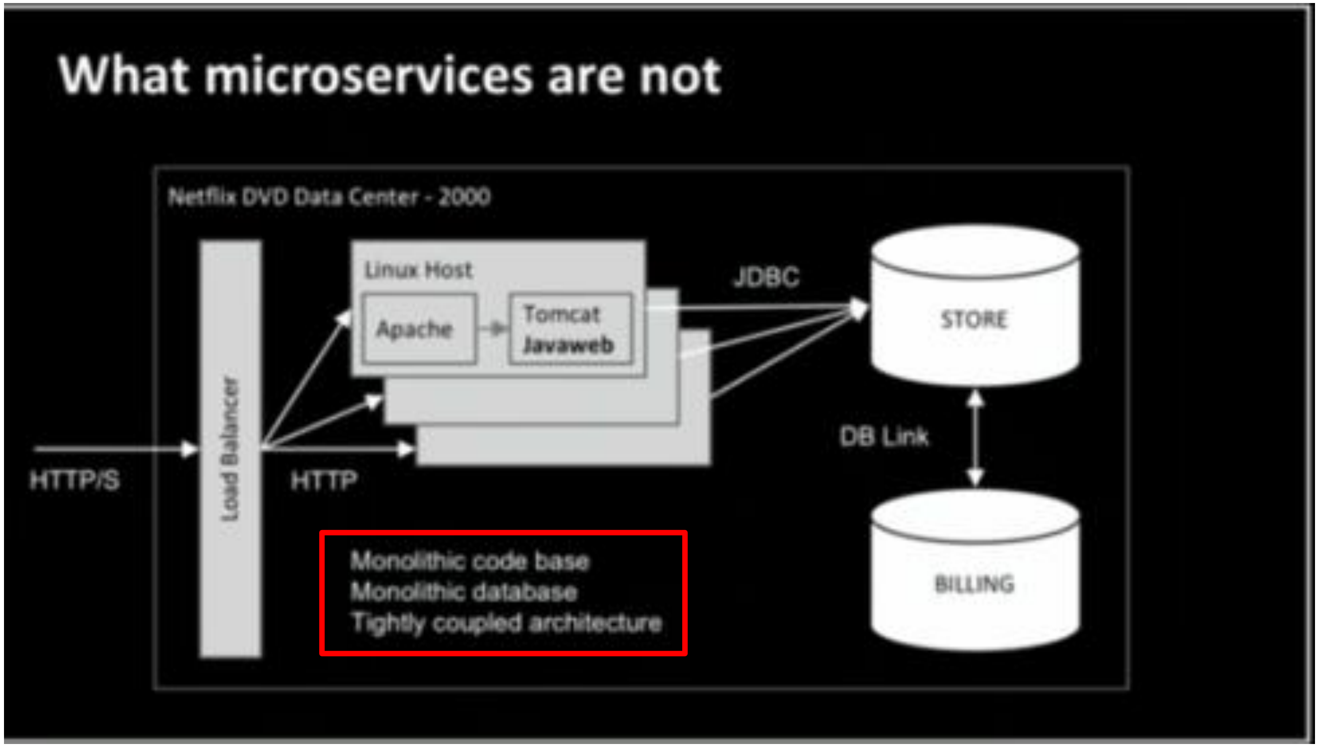
14.97% ↓

2.92% ↑

aste
Neil Hunt
CHIEF PRODUCT OFFICER, NETFLIX
AWS re:Invent

- Netflix figures
  - 86M members
  - 150M hours of streaming per day
  - 100,000+ AWS instances
- Migration to AWS
  - [2008, 2016]
  - improved productivity and scalability
- 500+ microservices
- Many tools
  - Spinnaker (CICD)
  - Atlas (monitoring)
  - Simian Army (testing reliability)
  - …
- Shift to containers
  - Titus (container management environment)
  - then AWS Blox …

- **Microservices**

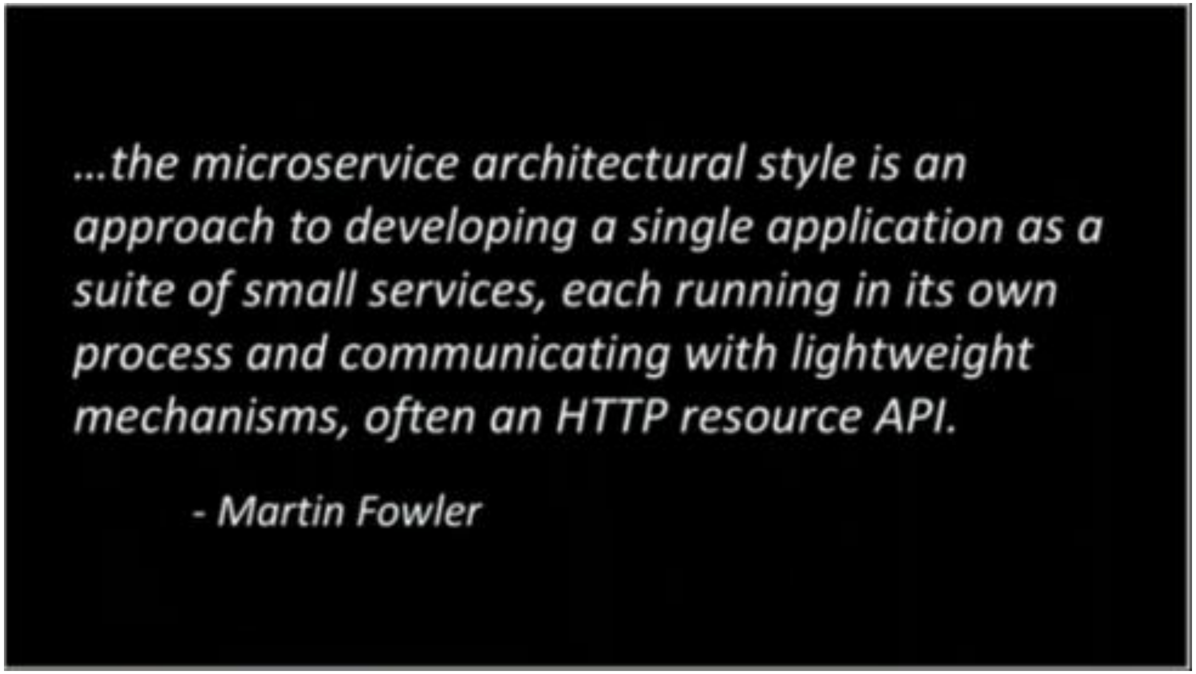


What microservices are not

Monolithic code base
Monolithic database
Tightly coupled architecture

Hard to scale
Hard to modify

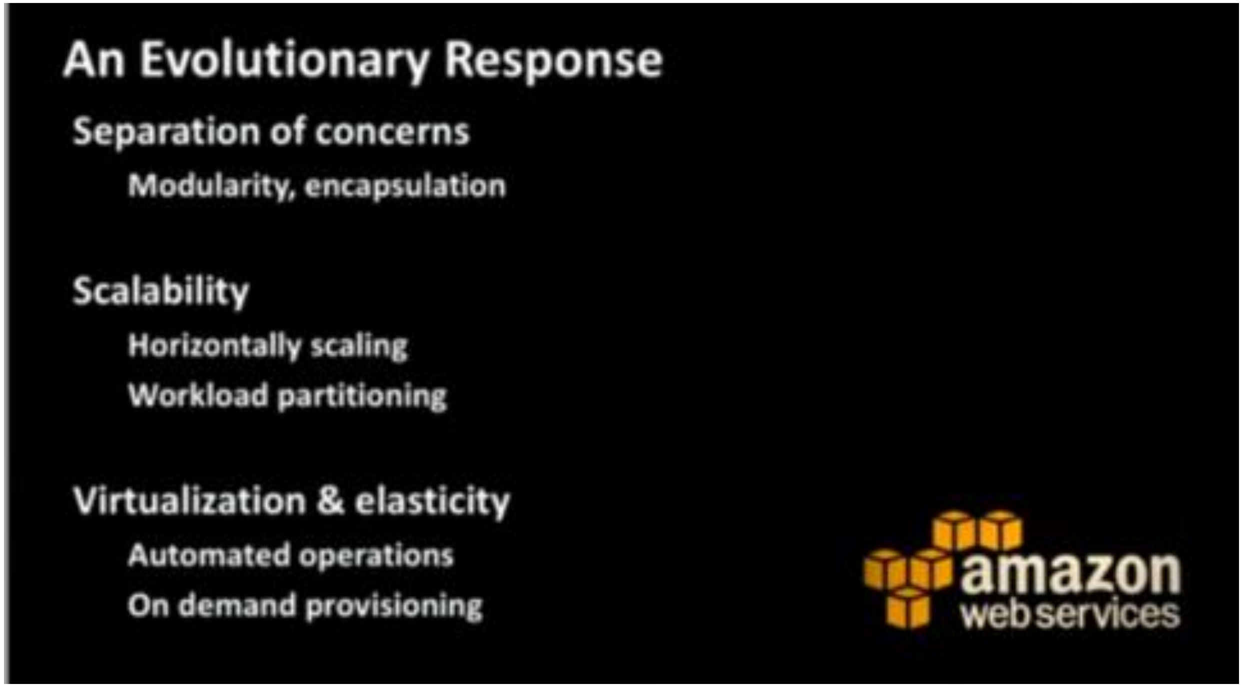
What is a microservice?



…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.

- Martin Fowler

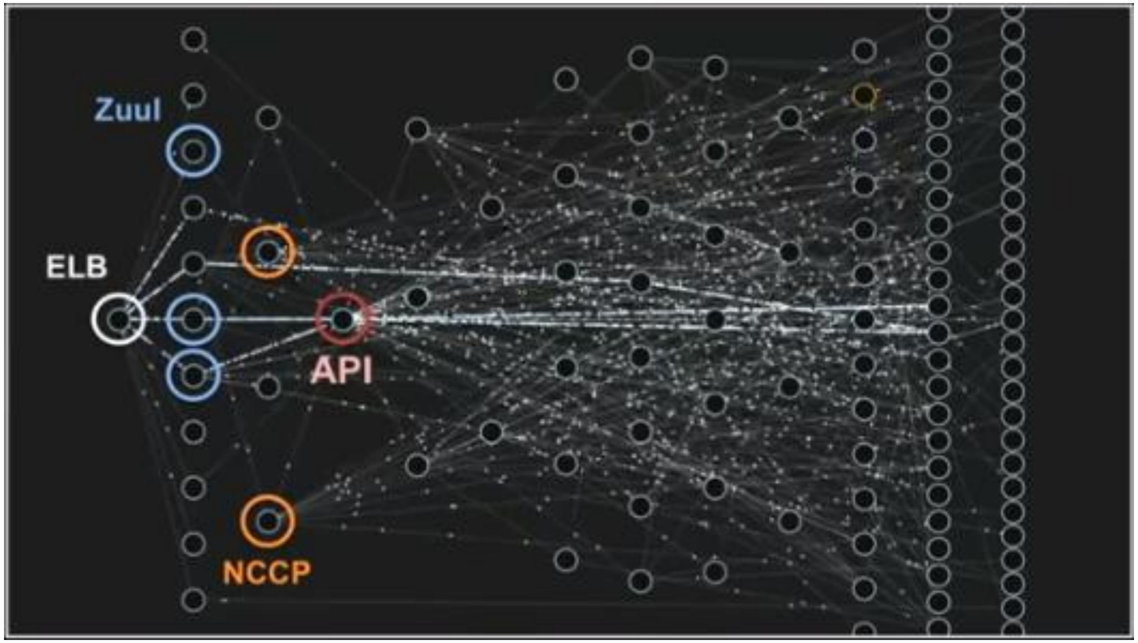
What is a microservice?

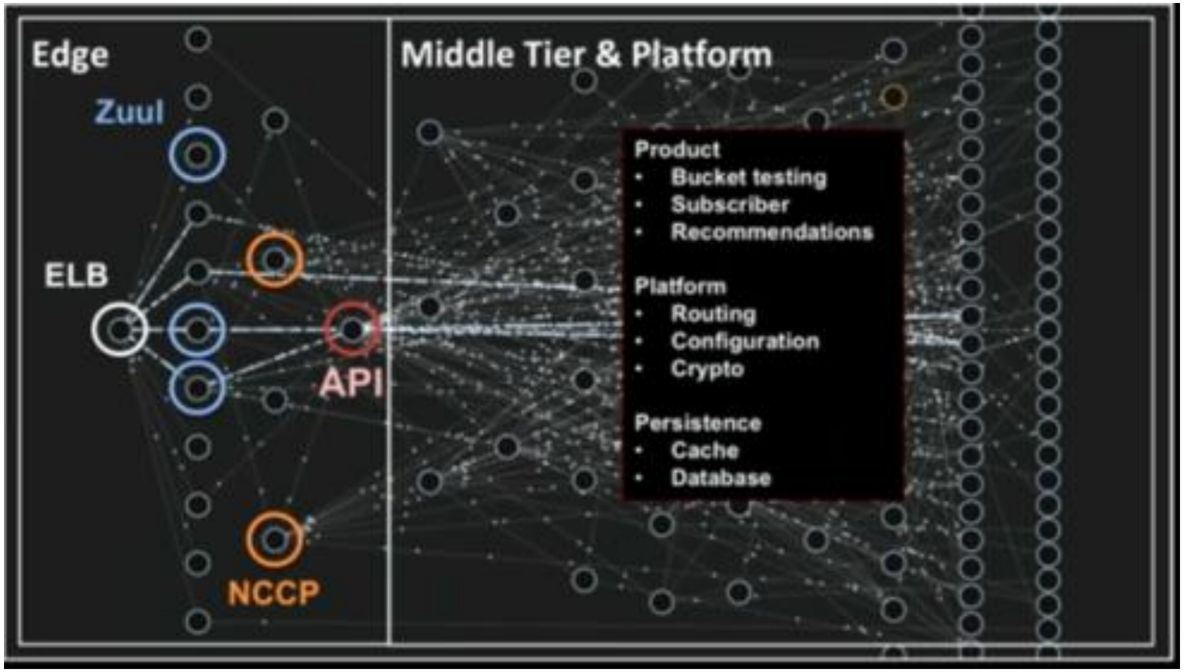


An Evolutionary Response

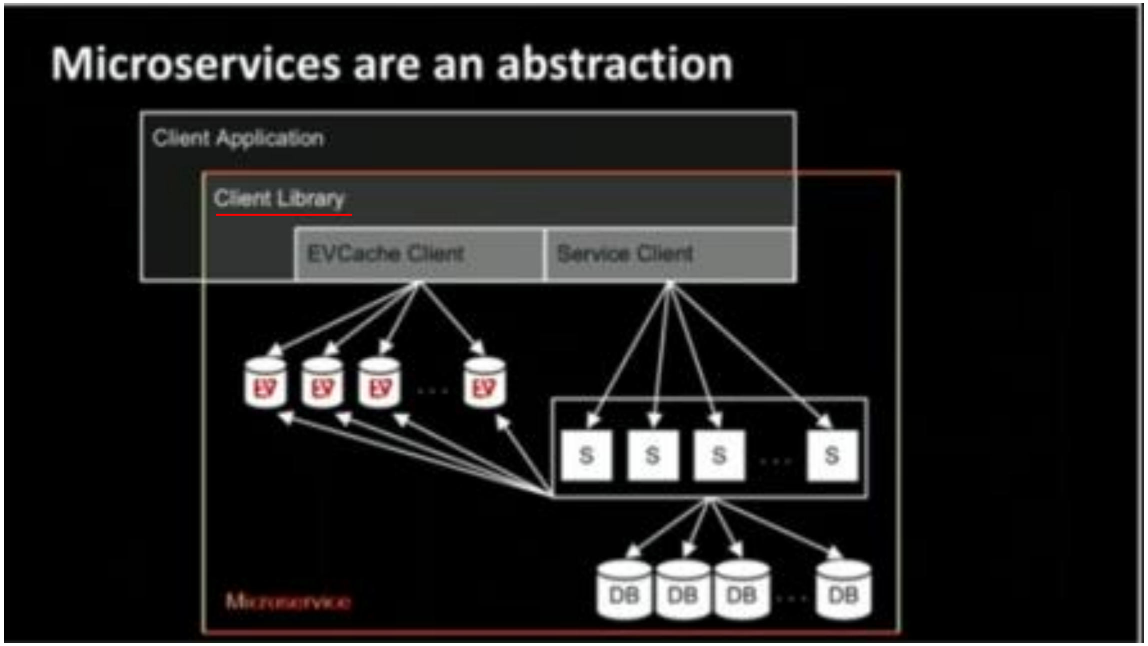
- Separation of concerns
  - Modularity, encapsulation
- Scalability
  - Horizontally scaling
  - Workload partitioning
- Virtualization & elasticity
  - Automated operations
  - On demand provisioning

amazon
web services


Zuul
ELB
API
NCCP



- **ELB**: AWS Elastic Load Balancing
- **Zuul**: proxy layer, performs dynamic routing
- **NCCP**: legacy tier, supporting earlier devices
- Netflix's **API** gateway, calling all other services

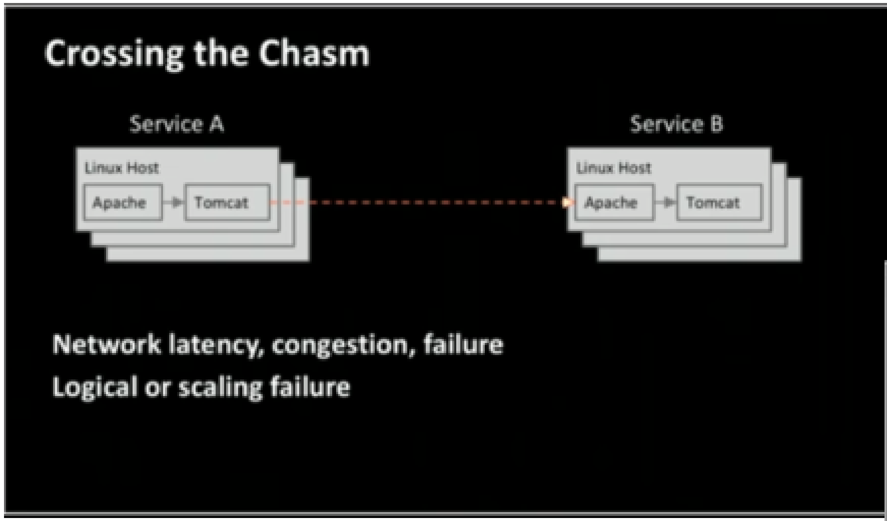


Microservices are an abstraction

… microservices are not always small

- Microservices
- Challenges
  - **Dependecy**
  - Scale
  - Variance
  - Change

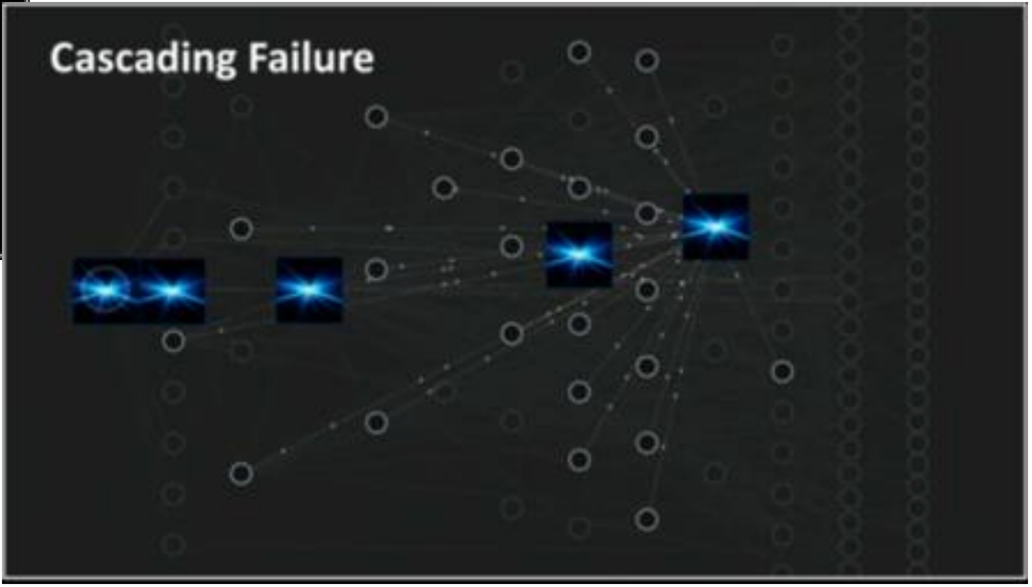
Dependency: Inter-service requests



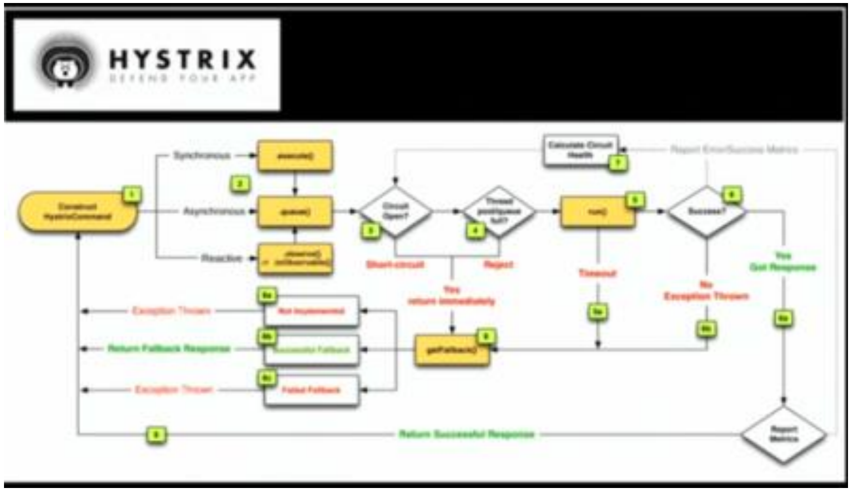
Crossing the Chasm

Network latency, congestion, failure

Logical or scaling failure



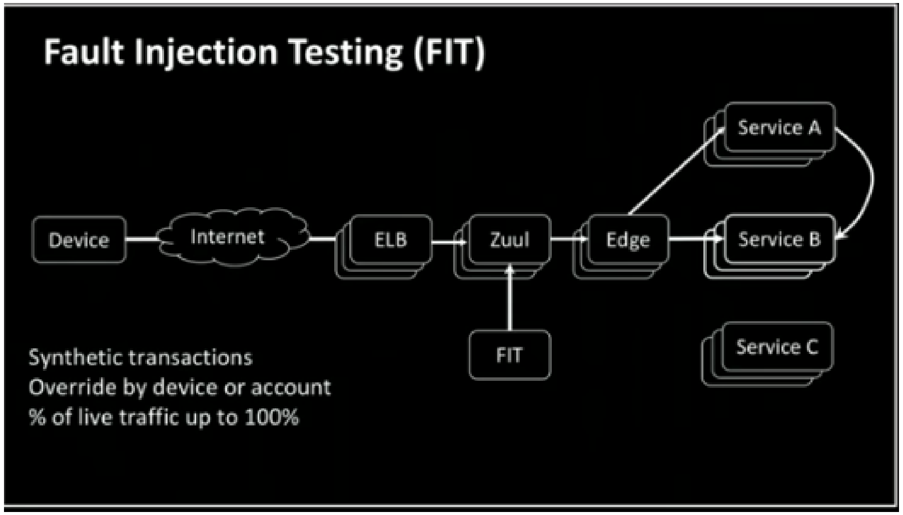
Cascading Failure



Timeouts and retries
Fallback (to allow customers to continue)
Isolated thread pools and circuits

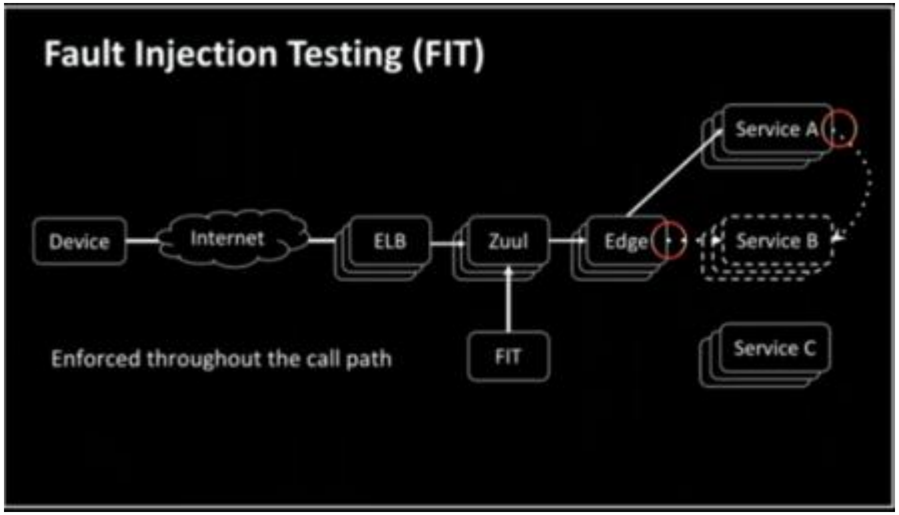
Dependency: Inter-service requests (cont.)

How do you know if it works?



Fault Injection Testing (FIT)

Synthetic transactions
Override by device or account
% of live traffic up to 100%

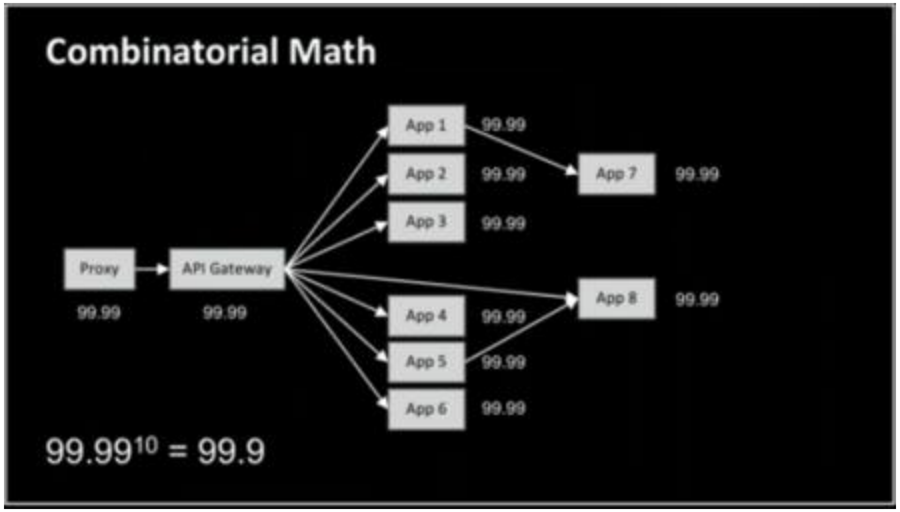


Fault Injection Testing (FIT)

Enforced throughout the call path

Dependency: Inter-service requests (cont.)

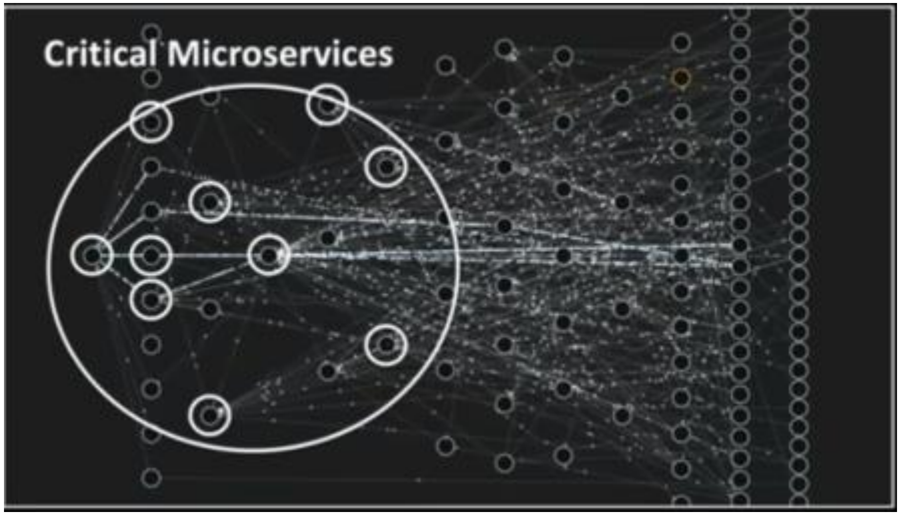
How to constrain testing scope?



Combinatorial Math

$99.99^{10} = 99.9$

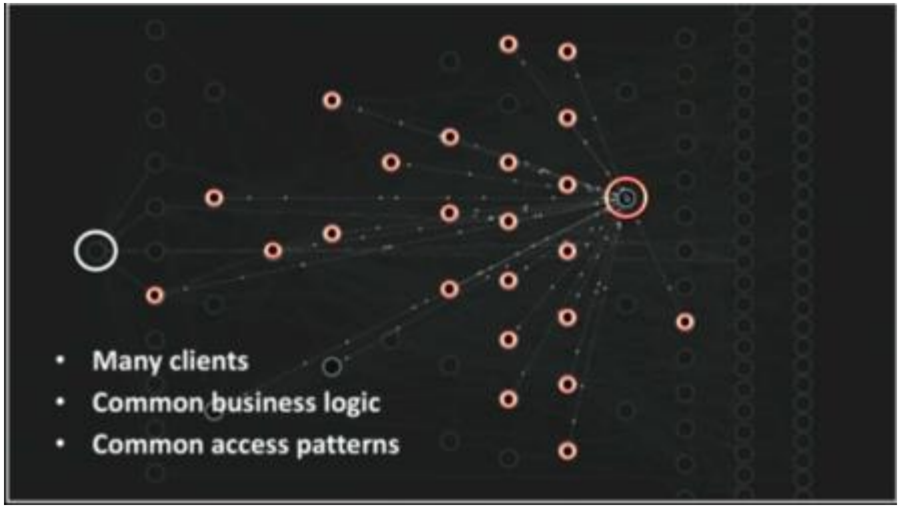
8-9 hours a year



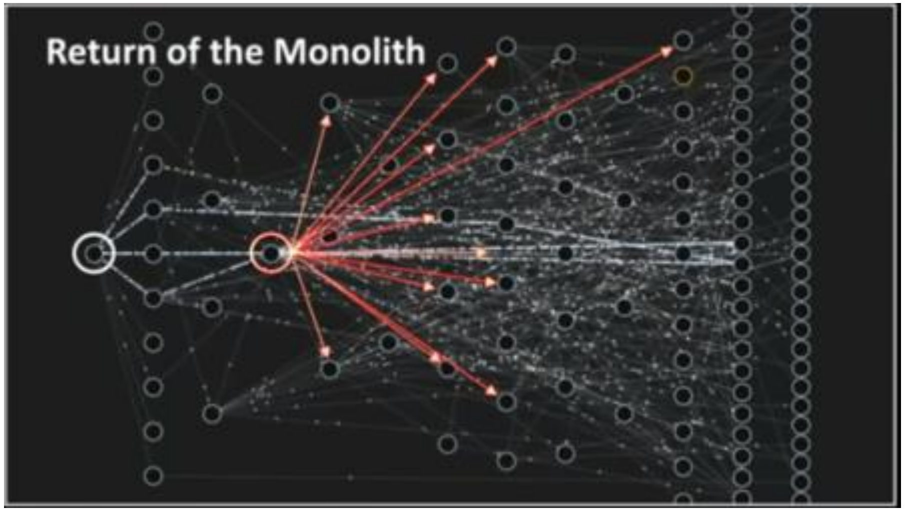
Critical Microservices

Dependency: Client libraries

Client libraries are useful



- Many clients
- Common business logic
- Common access patterns



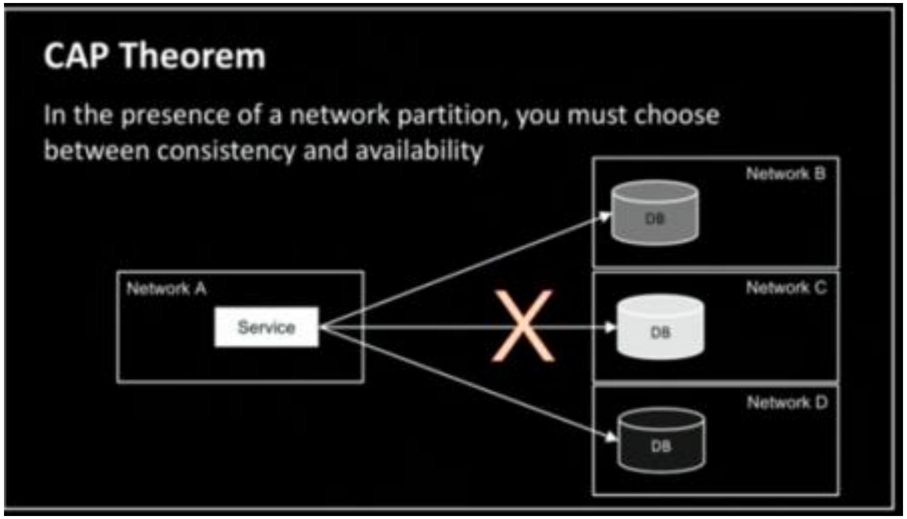
Return of the Monolith

Parasitic Infestation

Heap consumption
Logical defects
Transitive dependencies

→ **simplify client libraries**

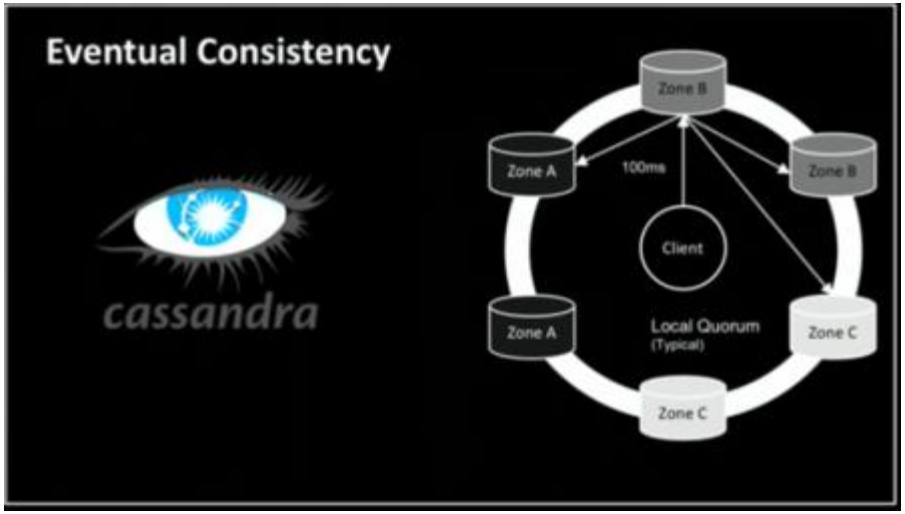
Dependency: Persistence



CAP Theorem

In the presence of a network partition, you must choose between consistency and availability

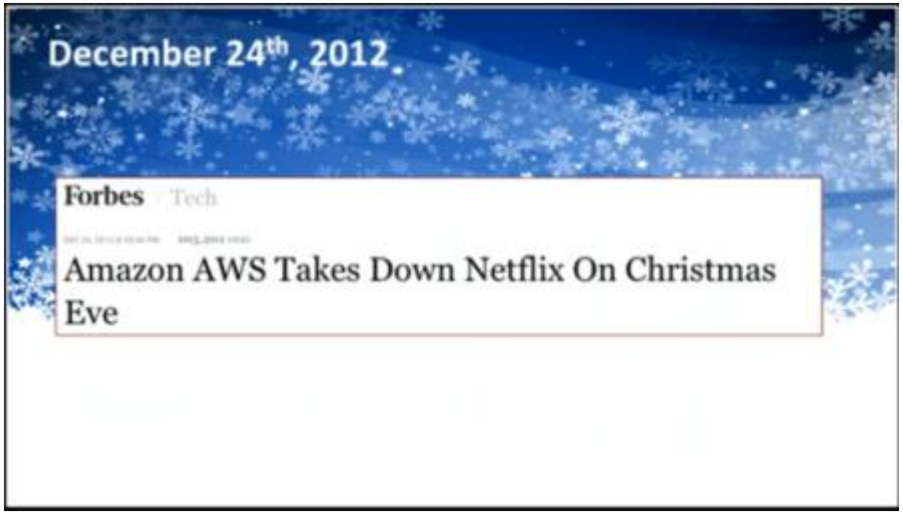
Service wants to write copy of same data in 3 DBs.
What if it cannot write in one of them? Fail?



Eventual Consistency

«write to the ones you can ge to, and then fix it up afterwards»
Local quorum

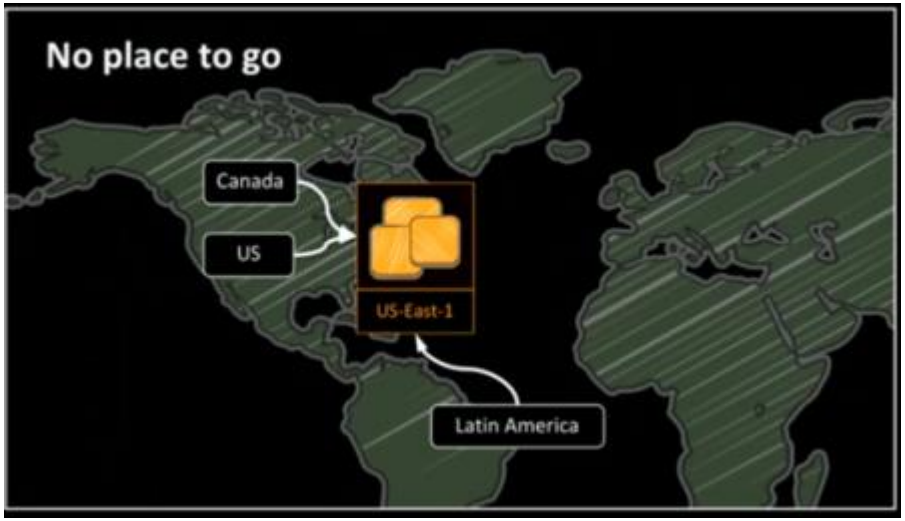
Dependency: Infrastructure



December 24th, 2012

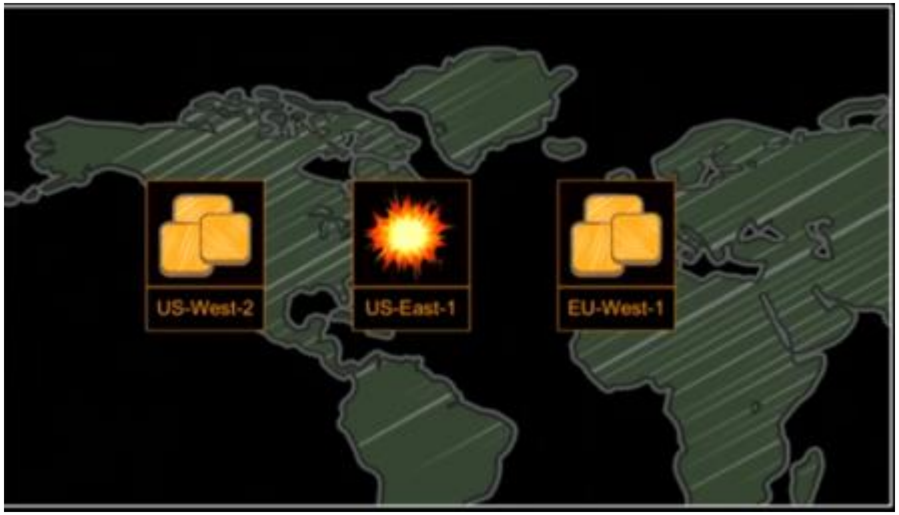
Forbes Tech

Amazon AWS Takes Down Netflix On Christmas Eve



No place to go

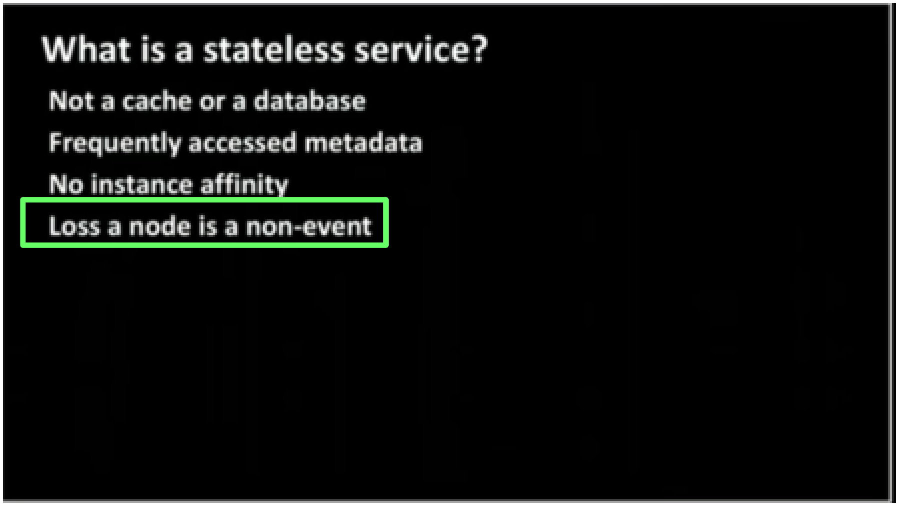
All in Us East 1



Multi region strategy

- Microservices
- Challenges
  - Dependecy
  - **Scale**
  - Variance
  - Change

Scale: Stateless services



What is a stateless service?

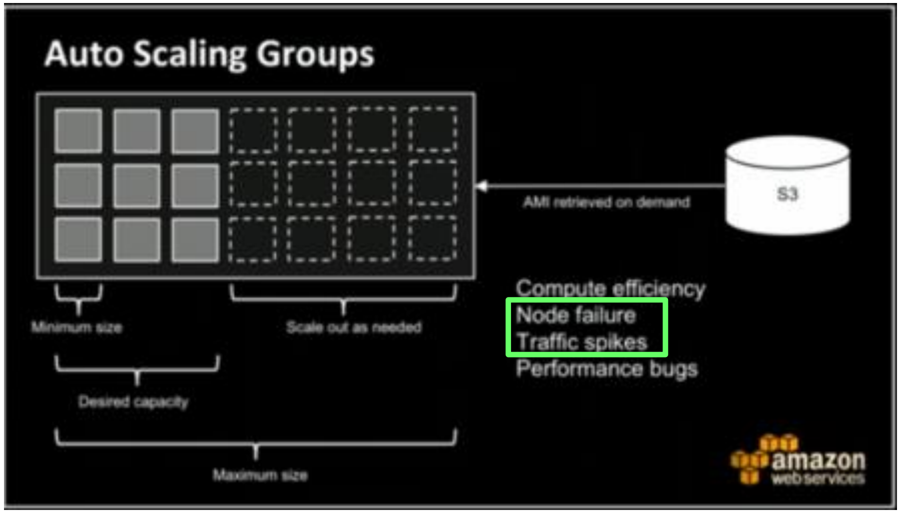
Not a cache or a database

Frequently accessed metadata

No instance affinity

Loss a node is a non-event

replicate



Auto Scaling Groups

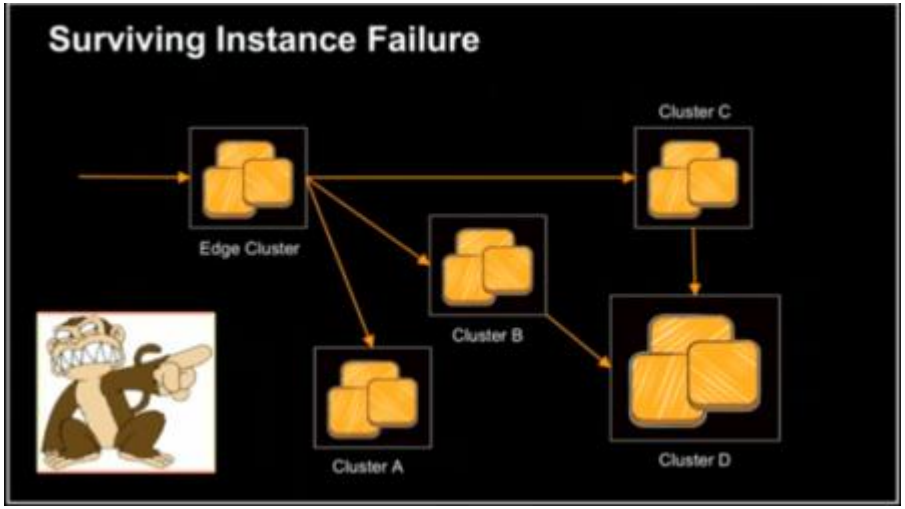
Compute efficiency

Node failure

Traffic spikes

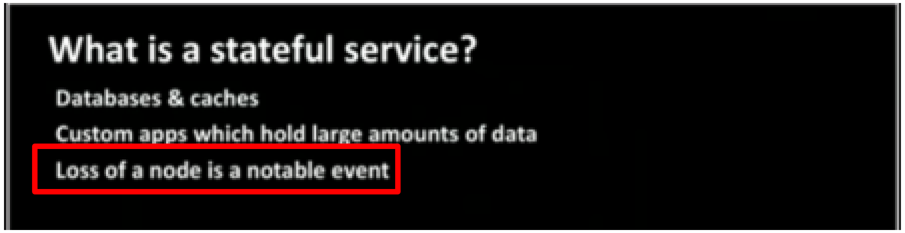
Performance bugs

and test with Chaos Monkey



Surviving Instance Failure

Scale: Stateful services

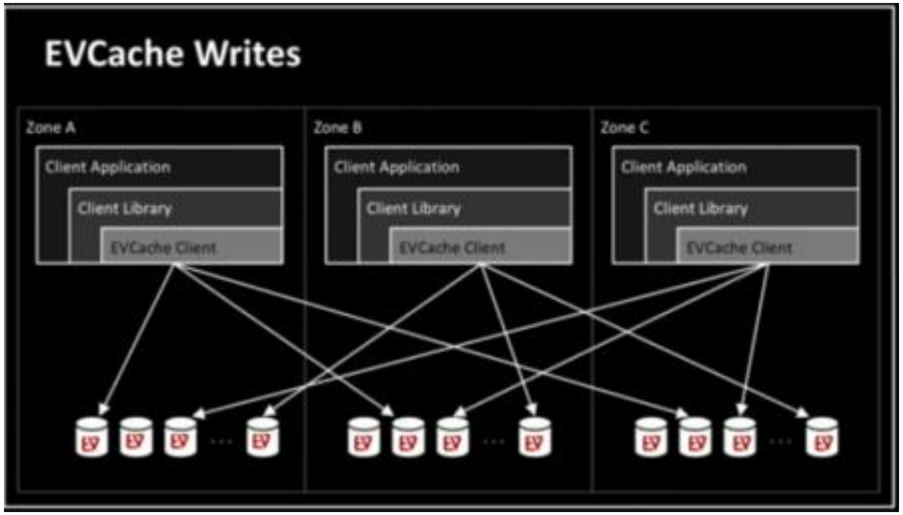


What is a stateful service?

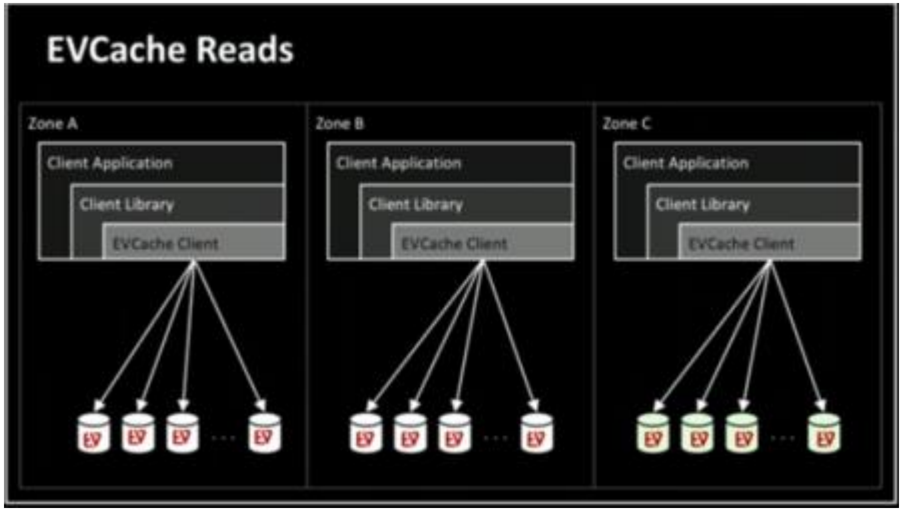
Databases & caches

Custom apps which hold large amounts of data

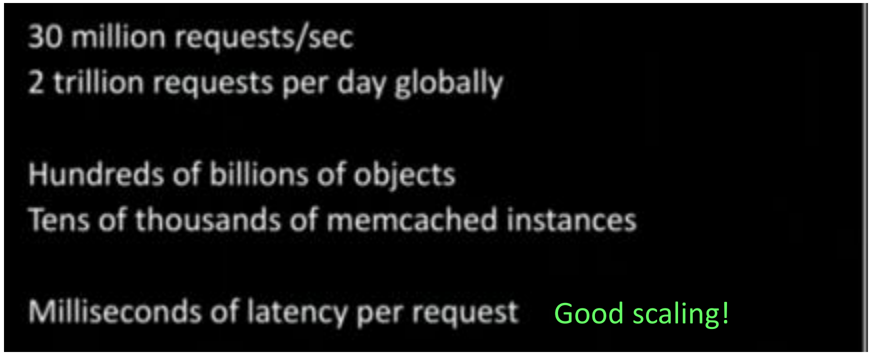
Loss of a node is a notable event



Multiple copies written out to multiple nodes,
in different availability zones



Local reads, but app can fall back
to reading across availability zones



30 million requests/sec

2 trillion requests per day globally

Hundreds of billions of objects

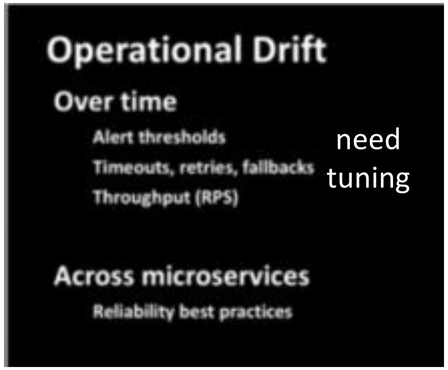
Tens of thousands of memcached instances

Milliseconds of latency per request

Good scaling!

- Microservices
- Challenges
  - Dependecy
  - Scale
  - **Variance**
  - Change

(Unintentional) Variance: Operational drift



Operational Drift

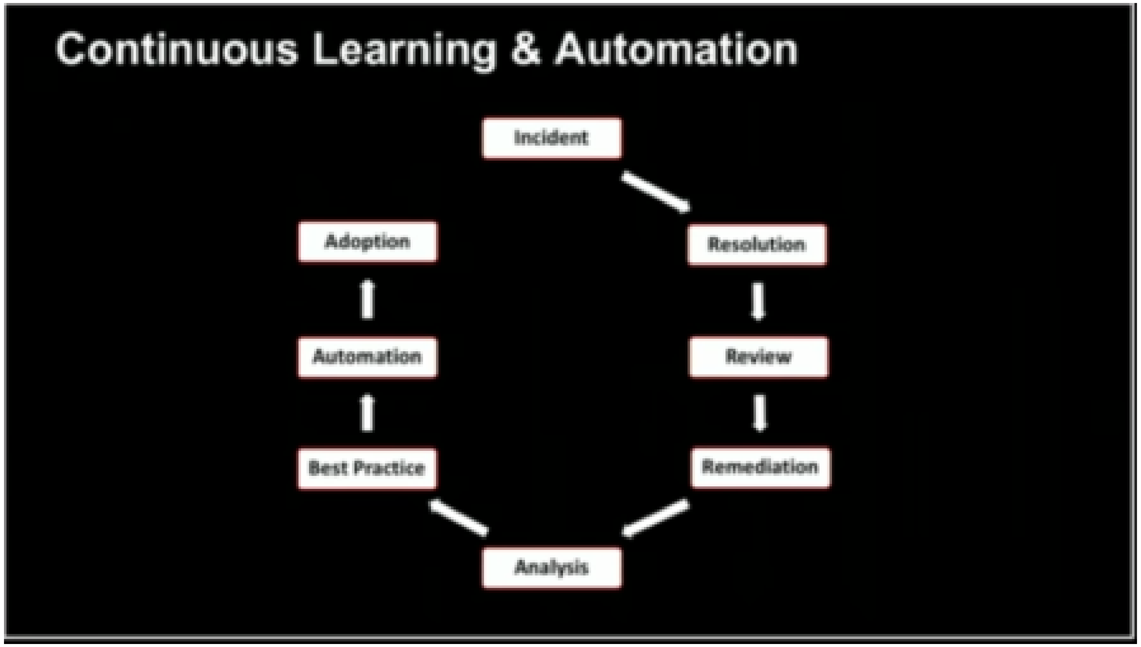
Over time

- Alert thresholds
- Timeouts, retries, fallbacks
- Throughput (RPS)

need tuning

Across microservices

- Reliability best practices



Continuous Learning & Automation

Incident

Resolution

Review

Remediation

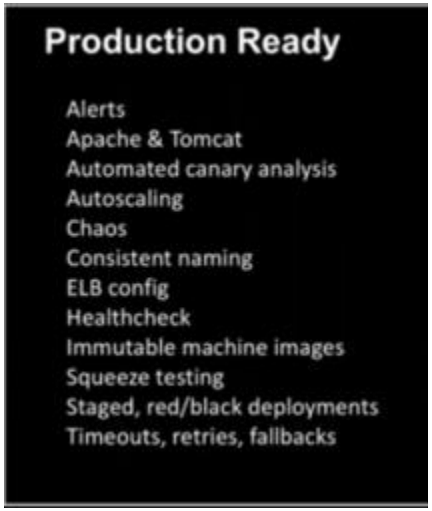
Analysis

Best Practice

Automation

Adoption

Best practices (checklist)



Production Ready

- Alerts
- Apache & Tomcat
- Automated canary analysis
- Autoscaling
- Chaos
- Consistent naming
- ELB config
- Healthcheck
- Immutable machine images
- Squeeze testing
- Staged, red/black deployments
- Timeouts, retries, fallbacks

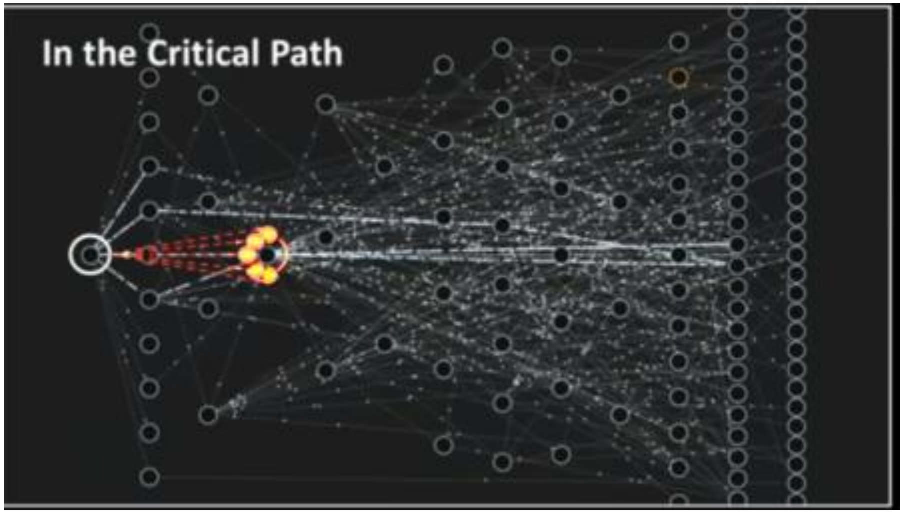
(Intentional) Variance: Polyglot & Containers

The Paved Road

- Stash
- Nebula/Gradle
- BaseAMI/Ubuntu
- Jenkins
- Spinnaker
- Runtime Platform

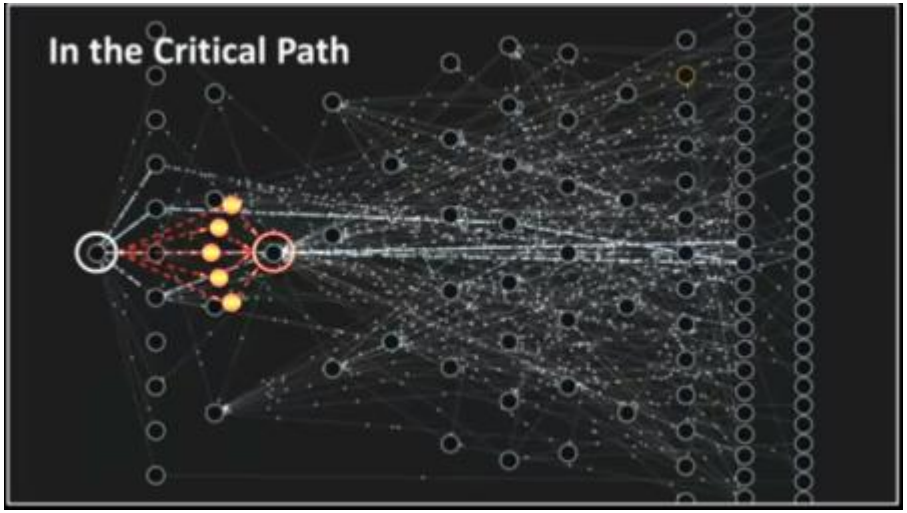
Engineering clients were going off road

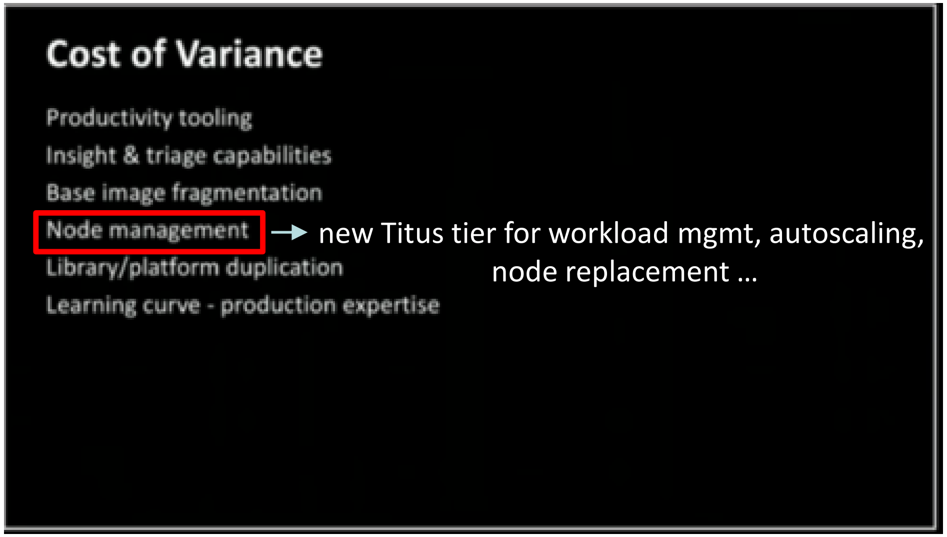
(Intentional) Variance: Polyglot & Containers



Problems in putting these technologies in the critical path
API gateway integrating scripts that can act as endpoints → monolith pattern

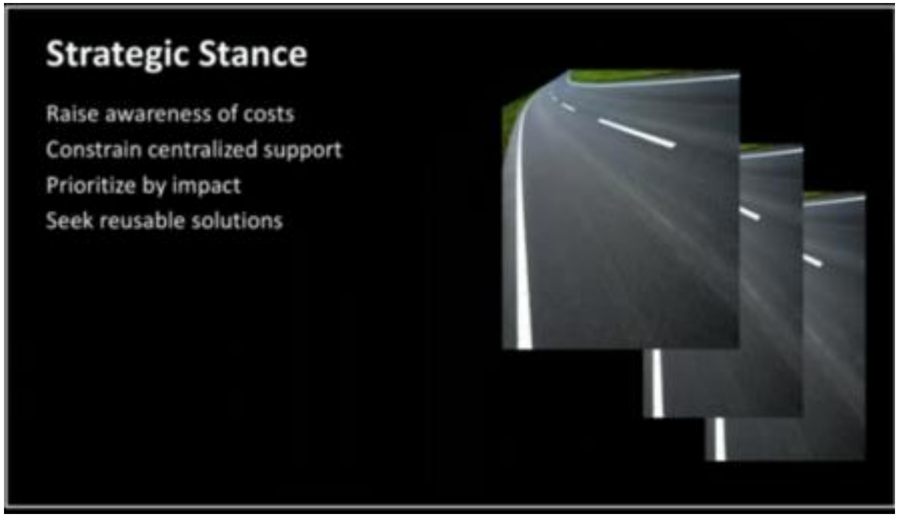
Solution: Push endpoints out of API service
(into nodejs apps in Docker containers)





Cost of Variance

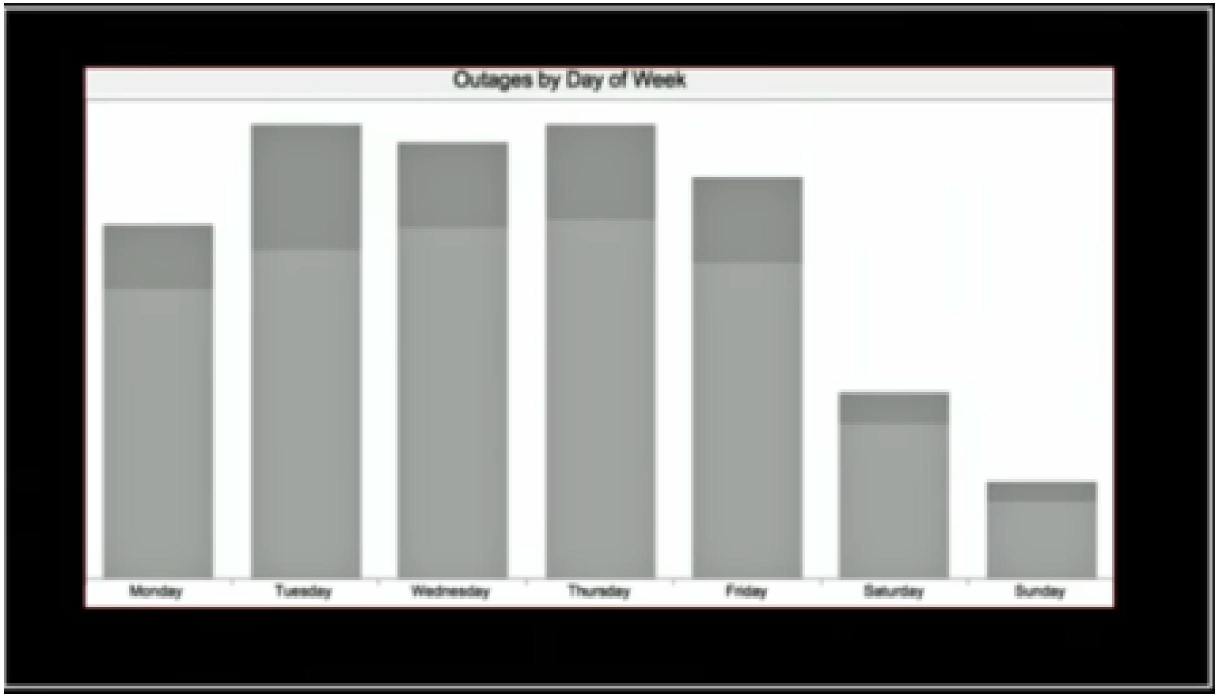
Productivity tooling
Insight & triage capabilities
Base image fragmentation
Node management → new Titus tier for workload mgmt, autoscaling, node replacement …
Library/platform duplication
Learning curve - production expertise

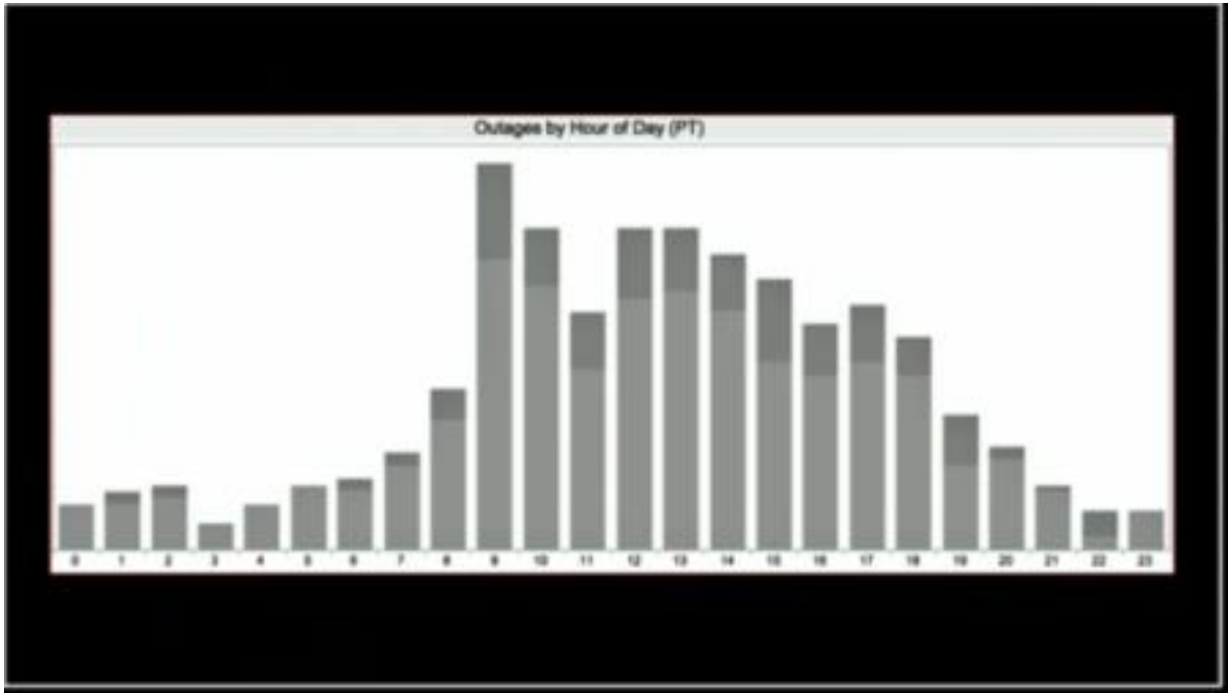


Strategic Stance

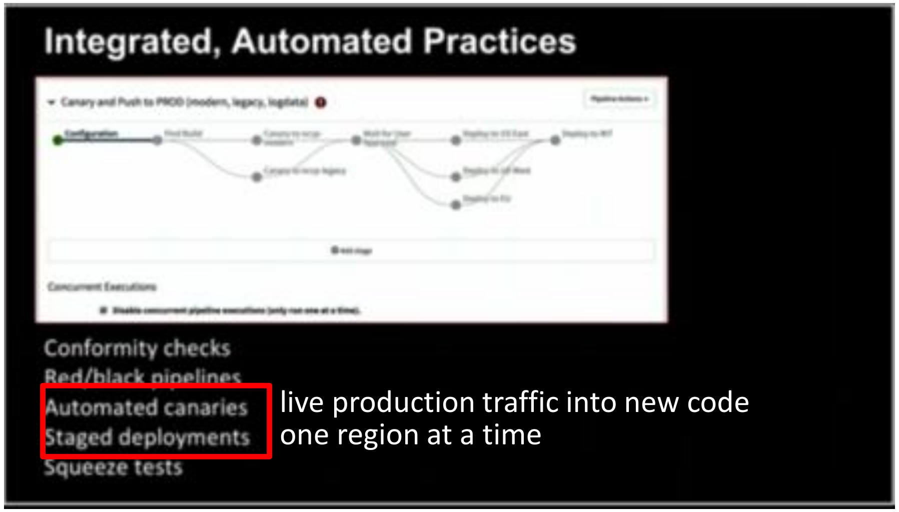
Raise awareness of costs
Constrain centralized support
Prioritize by impact
Seek reusable solutions

- Microservices
- Challenges
  - Dependecy
  - Scale
  - Variance
  - **Change**


Outages by Day of Week
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday


Outages by Hour of Day (PT)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

How do we achieve velocity with confidence?



Integrated, Automated Practices

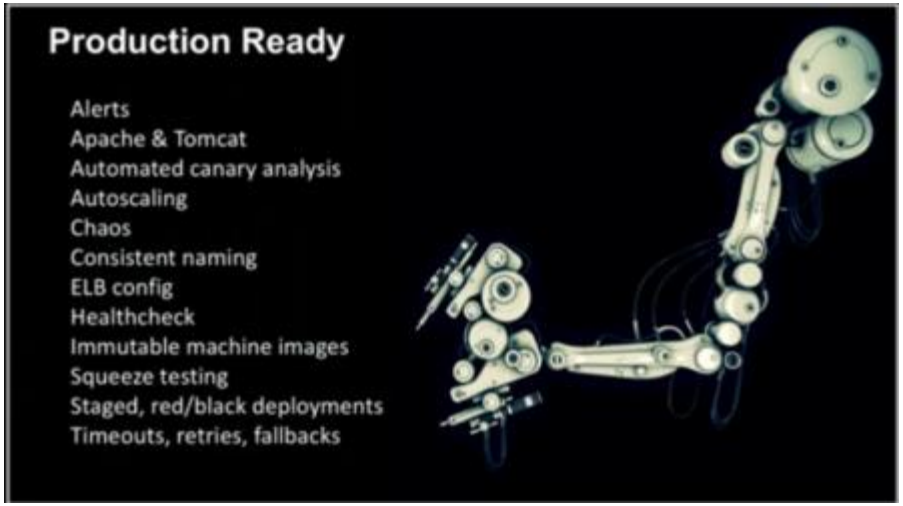
Canary and Push to PROD (modern, legacy, logdata)

Concurrent Executions

☑ Disable concurrent pipeline executions (only run one at a time).

Conformity checks
Red/black pipelines
Automated canaries
Staged deployments
Squeeze tests

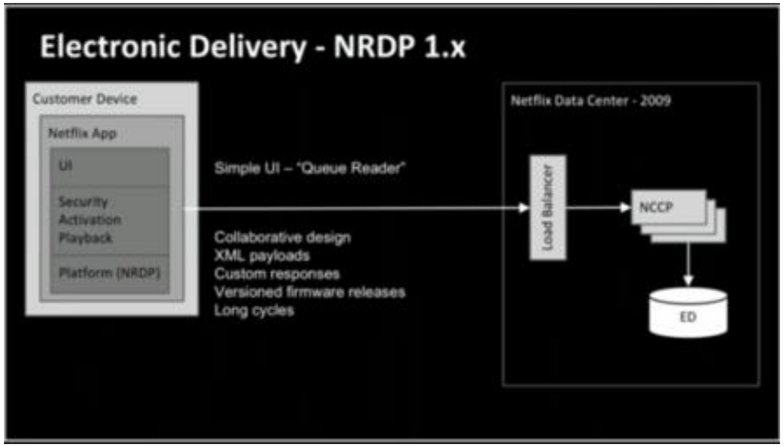
live production traffic into new code
one region at a time



Production Ready

- Alerts
- Apache & Tomcat
- Automated canary analysis
- Autoscaling
- Chaos
- Consistent naming
- ELB config
- Healthcheck
- Immutable machine images
- Squeeze testing
- Staged, red/black deployments
- Timeouts, retries, fallbacks

- Microservices
- Challenges
  - Dependecy
  - Scale
  - Variance
  - Change
- **Organization & architecture**

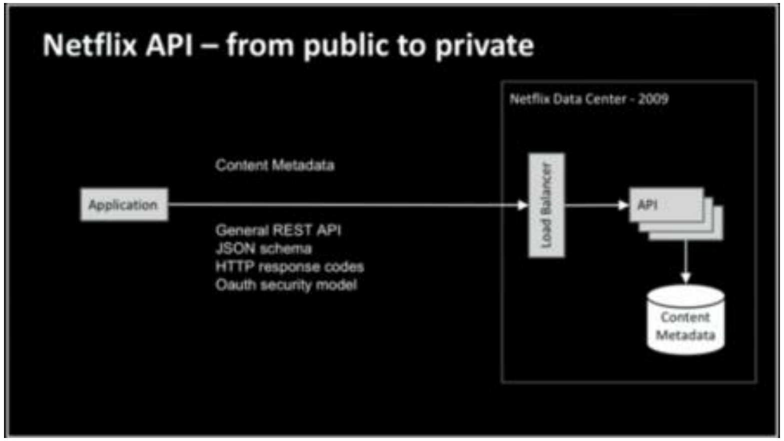


Electronic Delivery - NRDP 1.x

Simple UI – "Queue Reader"

Collaborative design
XML payloads
Custom responses
Versioned firmware releases
Long cycles

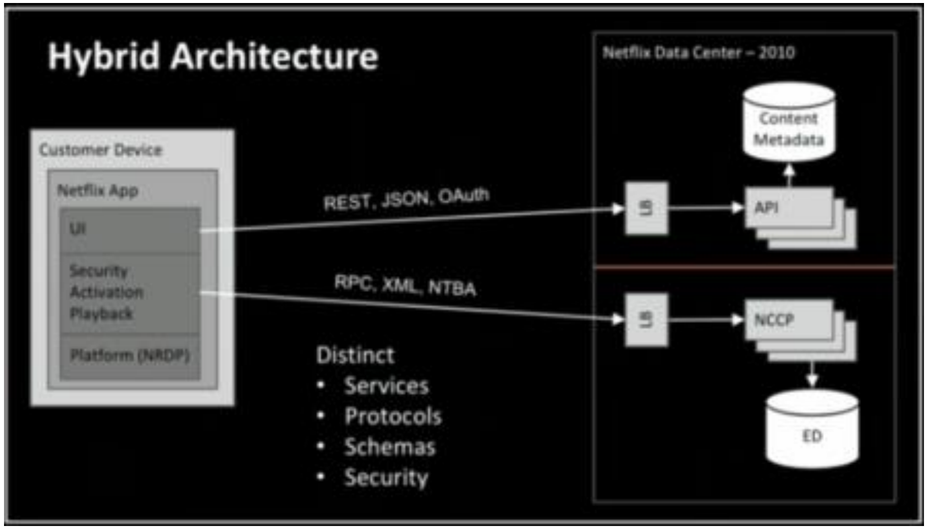
Netflix API - let a 1000 flowers bloom!



Netflix API – from public to private

Content Metadata

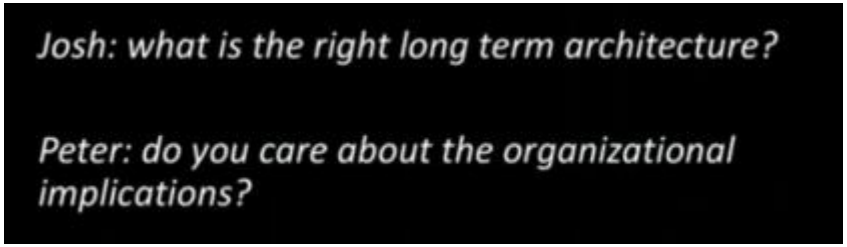
General REST API
JSON schema
HTTP response codes
Oauth security model



Hybrid Architecture

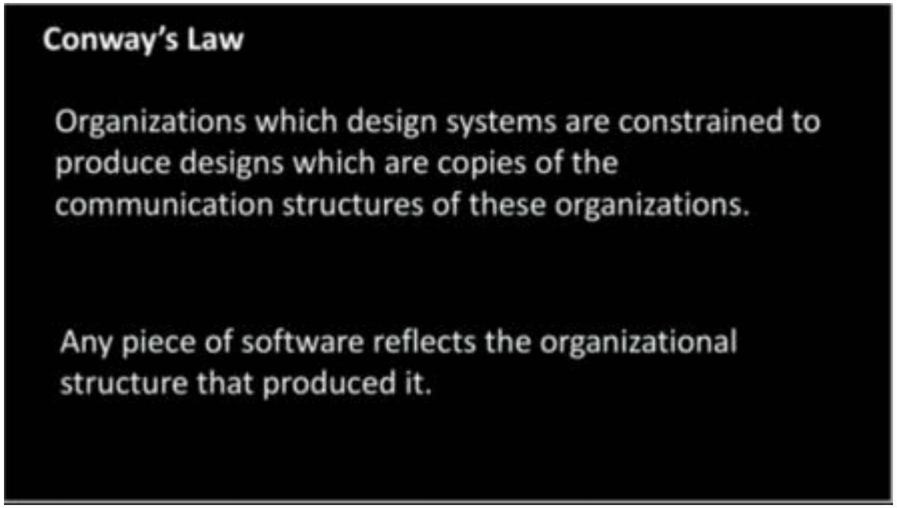
Distinct

- Services
- Protocols
- Schemas
- Security



Josh: what is the right long term architecture?

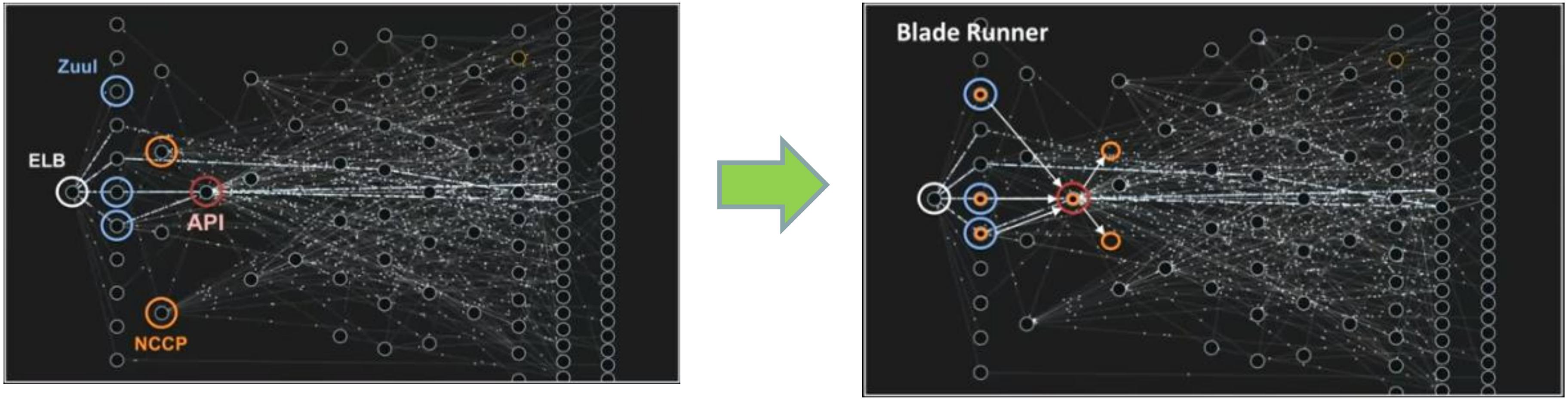
Peter: do you care about the organizational implications?



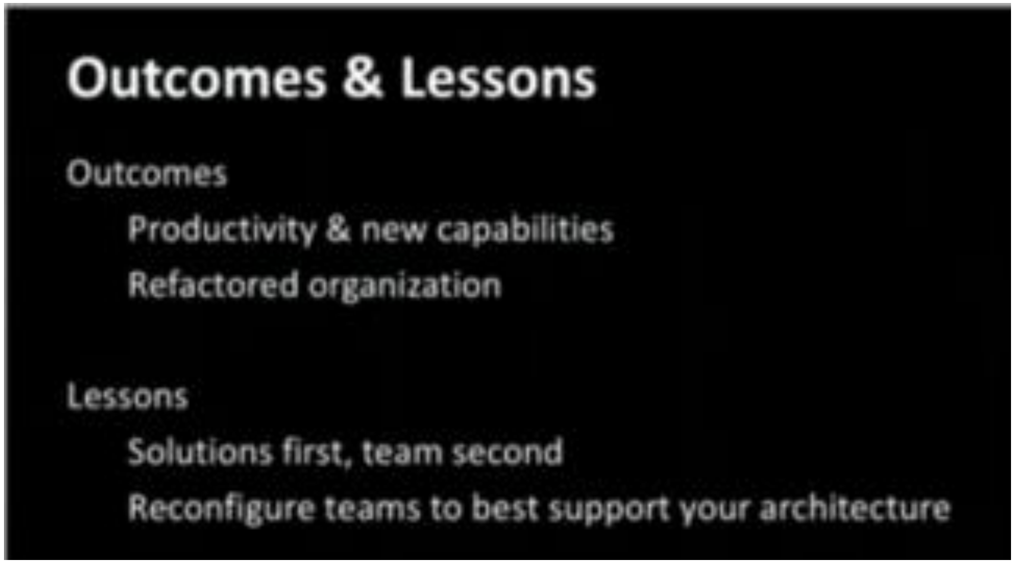
Conway's Law

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

Any piece of software reflects the organizational structure that produced it.



NCCP capabilities decomposed into smaller microservices and integrated into Zuul proxy layer and API gateway



Outcomes & Lessons

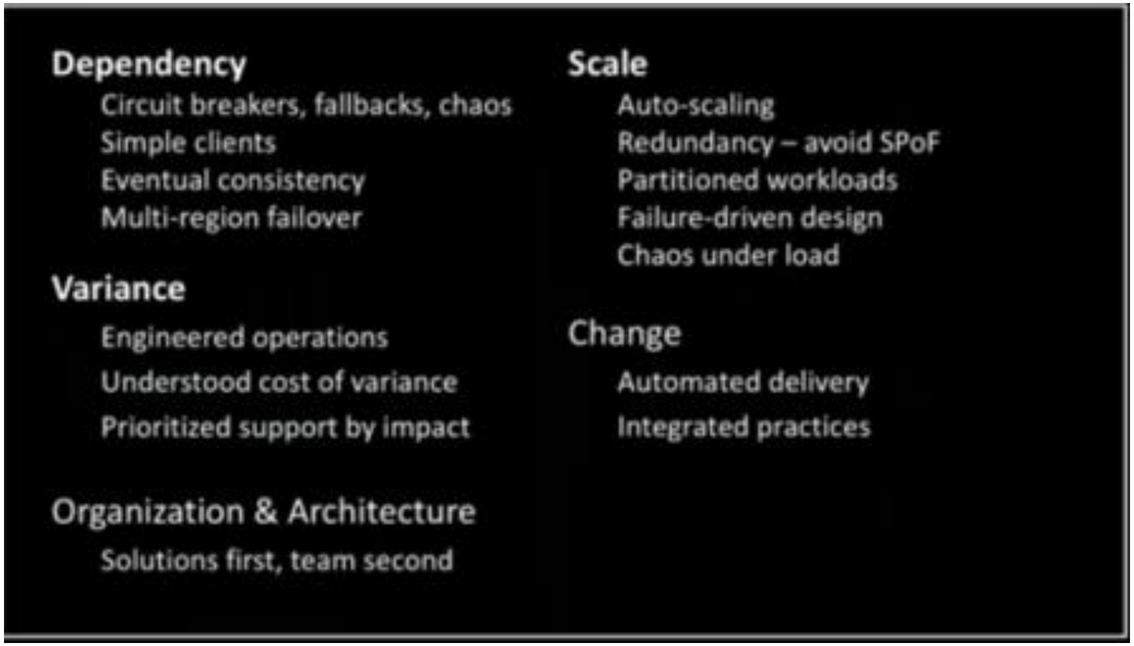
Outcomes

- Productivity & new capabilities
- Refactored organization

Lessons

- Solutions first, team second
- Reconfigure teams to best support your architecture

Summary



Dependency
- Circuit breakers, fallbacks, chaos
- Simple clients
- Eventual consistency
- Multi-region failover

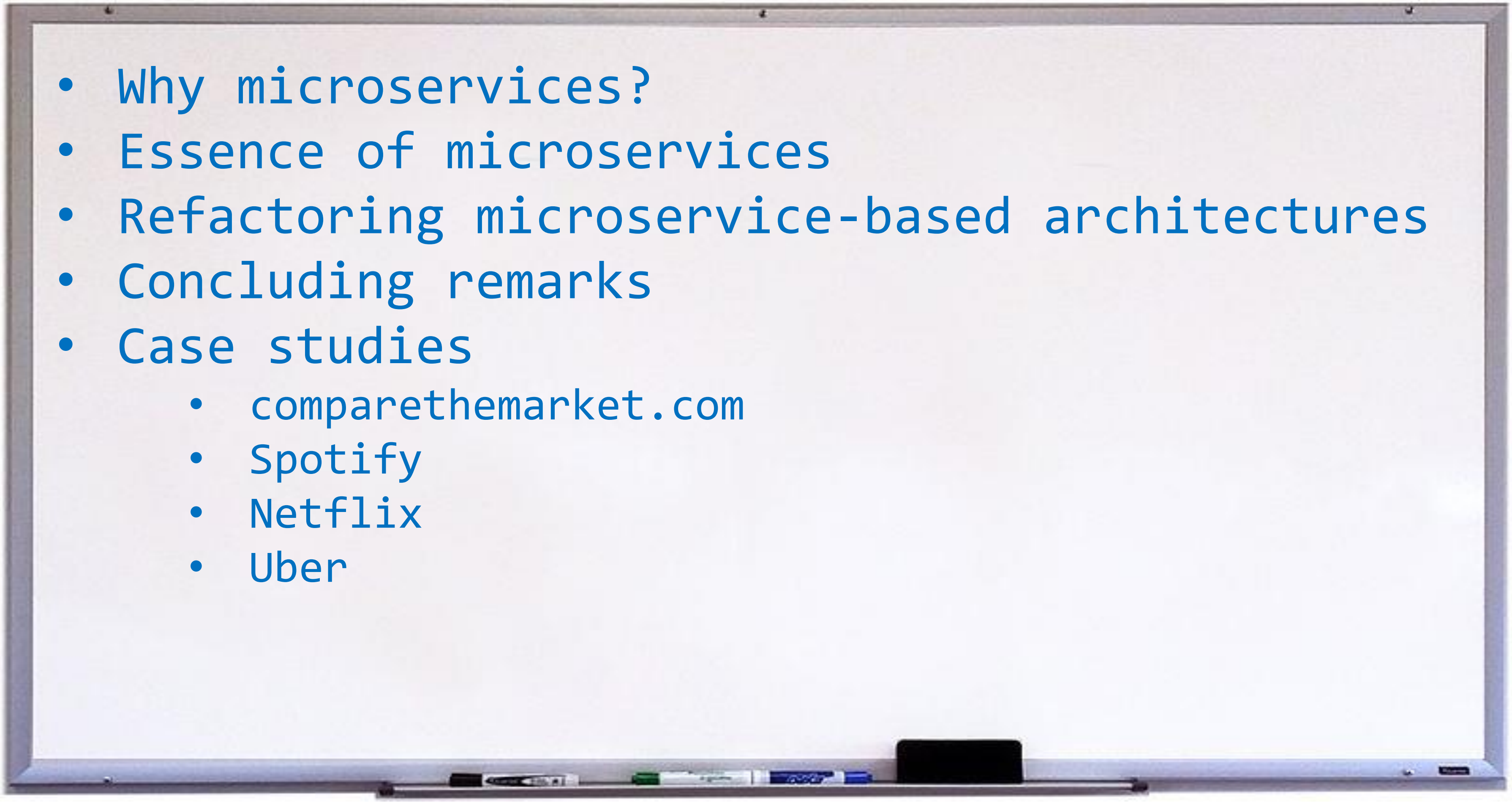
Variance
- Engineered operations
- Understood cost of variance
- Prioritized support by impact

Organization & Architecture
- Solutions first, team second

Scale
- Auto-scaling
- Redundancy – avoid SPoF
- Partitioned workloads
- Failure-driven design
- Chaos under load

Change
- Automated delivery
- Integrated practices

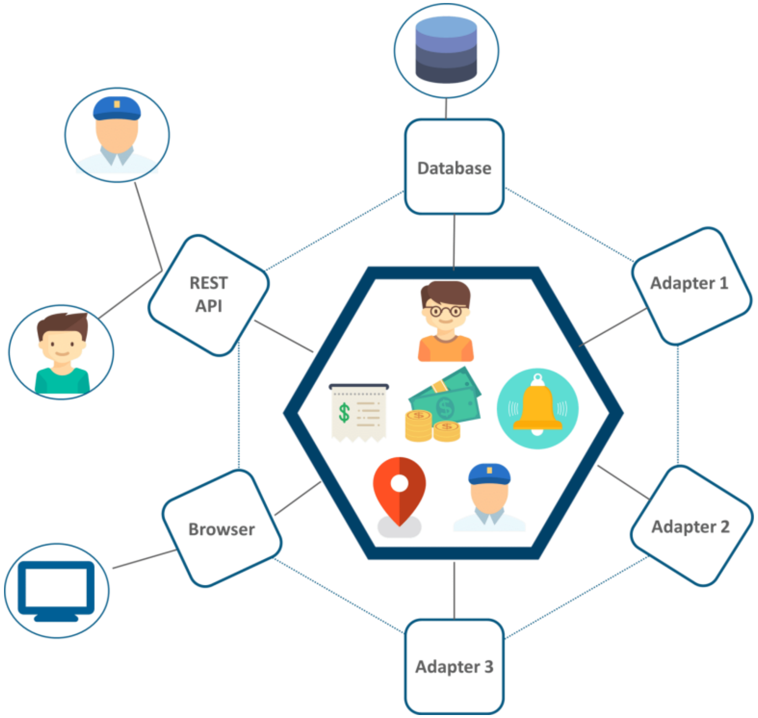
NETFLIX
DEVICE PARTNER LIFECYCLE TEAM



- Why microservices?
- Essence of microservices
- Refactoring microservice-based architectures
- Concluding remarks
- Case studies
    - comparethemarket.com
    - Spotify
    - Netflix
    - Uber

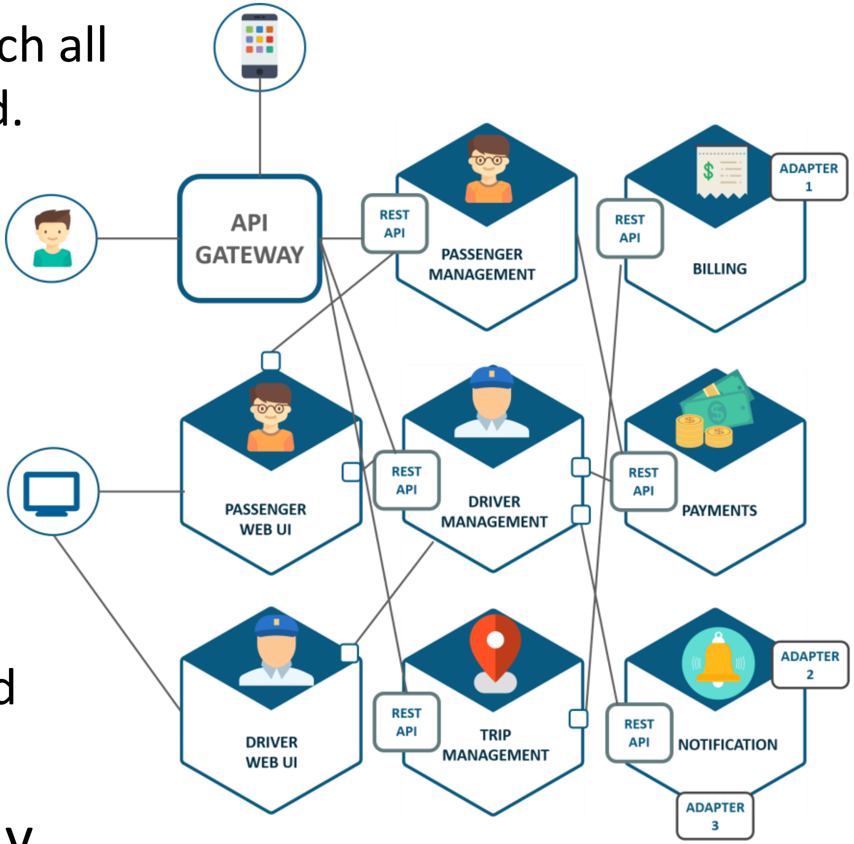
Uber’s initial architecture

- Monolithic architecture built for a single offering in a single city
  - a REST API with which passenger and driver connect
  - three different adapters used to perform billing, payments, sending emails/messages
  - a MySQL database to store all data
- Problems when expanding worldwide
  - all the features had to be re-built, deployed and tested again and again to update a single feature
  - fixing bugs became extremely difficult in a single repository as developers had to change the code again and again
  - scaling the features simultaneously with the introduction of new features worldwide was quite tough to be handled together



Uber’s solution

- Break monolith into microservices
  - introduction of API Gateway through which all the drivers and passengers are connected. From the API Gateway,
  - all the internal points (passenger management, driver management, trip management, etc.) connected from API Gateway
- Individual separate deployable units performing separate functionalities
  - E.g., billing microservices can be deployed independently from the rest
- All features can now scale individually
  - e.g., people searching for cabs > people actually booking & paying cabs


API GATEWAY
REST API
PASSENGER MANAGEMENT
REST API
BILLING
ADAPTER 1
PASSENGER WEB UI
REST API
DRIVER MANAGEMENT
REST API
PAYMENTS
DRIVER WEB UI
REST API
TRIP MANAGEMENT
REST API
NOTIFICATION
ADAPTER 2
ADAPTER 3