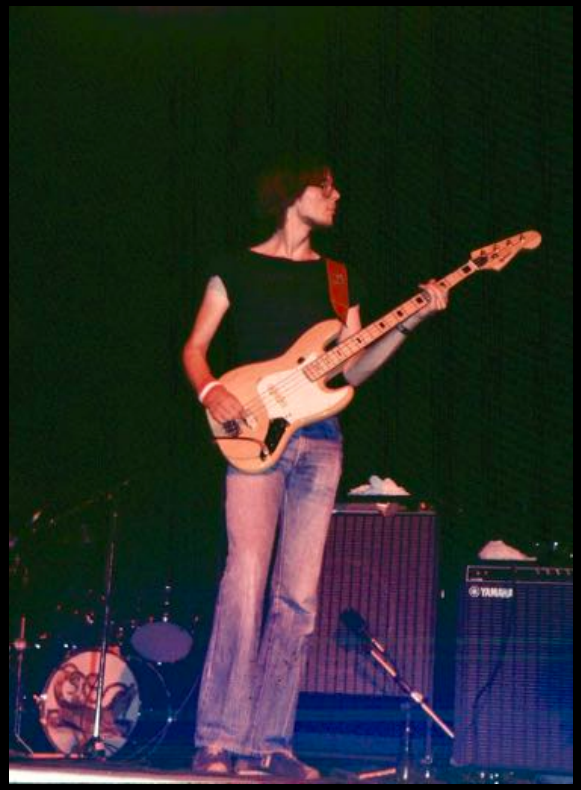

Adrian's Greatest Hits, B-Sides and Re-issues
Adrian Cockcroft | YOW | December 2022
@adrianco@mastodon.social



Adrian in 1982
https://soundcloud.com/adrian-cockcroft/black-tiger-dont-look-back

Like a classic rock band on their annual "farewell and we aren't dead yet tour"

I know you just want to hear the hits, but I want to sneak in some stuff I like too…

(Headbanging optional)

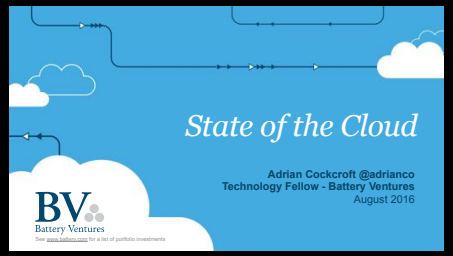
Setlist

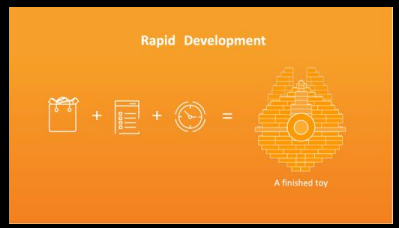
Black Tiger - Don't Look Back - 1982
Netflix in the Cloud - QconSF 2010, Cassandra Summit 2011
Microservices - Various MicroXchg Berlin talks
Cloud Trends - GigaOM Structure - 2016 reissue
Communicating Sequential Goroutines - Gophercon - 2016
Lego spaceships and the kitchen sink - AWS - 2017-2019
DevSusOps - a track from the new album you don't want to hear

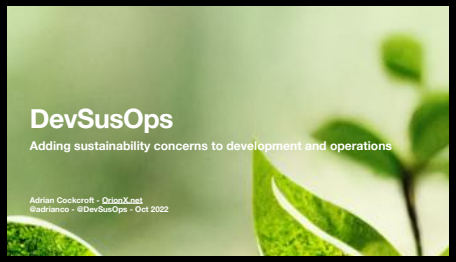
Encore

Bottleneck Analysis - GOTO Aarhus - 2013

This talk features "singles" from these "albums"







Netflix in the Cloud

Nov 6, 2010

Adrian Cockcroft
@adrianco #netflixcloud
http://www.linkedin.com/in/adriancockcroft

We stopped building our own datacenters

Capacity growth rate is accelerating, unpredictable

Product launch spikes - iPhone, Wii, PS3, XBox

Datacenter is large inflexible capital commitment

"The cloud lets its users focus on delivering differentiating business value instead of wasting valuable resources on the ***undifferentiated heavy lifting*** *that makes up most of IT infrastructure."*

Werner Vogels

Amazon CTO

amazon web services

NETFLIX

What, Why and How?

The details…

Goals

- Faster
  - **Lower latency** than the equivalent datacenter web pages and API calls
  - Measured as mean and 99th percentile
  - For both first hit (e.g. home page) and in-session hits for the same user
- Scalable
  - **Avoid needing any more datacenter capacity** as subscriber count increases
  - No central vertically scaled databases
  - Leverage AWS elastic capacity effectively
- Available
  - Substantially **higher robustness and availability** than datacenter services
  - Leverage multiple AWS availability zones
  - No scheduled down time, no central database schema to change
- Productive
  - Optimize **agility** of a large development team with automation and tools
  - Leave behind complex tangled datacenter code base (~8 year old architecture)
  - Enforce clean layered interfaces and re-usable components

Old Datacenter vs. New Cloud Arch

Central SQL Database	Distributed Key/Value NoSQL
Sticky In-Memory Session	Shared Memcached Session
Chatty Protocols	Latency Tolerant Protocols
Tangled Service Interfaces	Layered Service Interfaces
Instrumented Code	Instrumented Service Patterns
Fat Complex Objects	Lightweight Serializable Objects
Components as Jar Files	Components as Services

Tangled Service Interfaces

- Datacenter implementation is exposed
  - Oracle SQL queries mixed into business logic
- Tangled code
  - Deep dependencies, false sharing
- Data providers with sideways dependencies
  - Everything depends on everything else

Anti-pattern affects productivity, availability

Untangled Service Interfaces

- New Cloud Code With Strict Layering
  - Compile against interface jar
  - Can use spring runtime binding to enforce
- Service interface **is** the service
  - Implementation is completely hidden
  - Can be implemented locally or remotely
  - Implementation can evolve independently

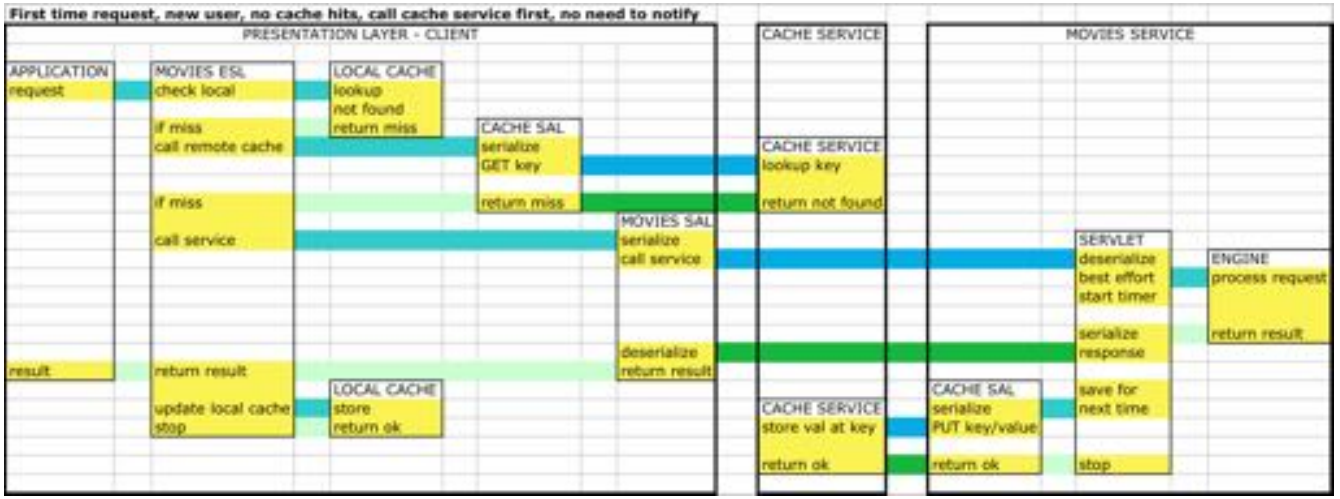
Untangled Service Interfaces

Two layers:

- SAL - Service Access Library
  - Basic serialization and error handling
  - REST or POJO's defined by data provider
- ESL - Extended Service Library
  - Caching, conveniences
  - Can combine several SALs
  - Exposes faceted type system (described later)
  - Interface defined by data consumer in many cases

Service Interaction Pattern

Sample Swimlane Diagram



NETFLIX

Boundary Interfaces

- Isolate teams from external dependencies
  - Fake SAL built by cloud team
  - Real SAL provided by data provider team later
  - ESL built by cloud team using faceted objects
- Fake data sources allow development to start
  - e.g. Fake Identity SAL for a test set of customers
  - Development solidifies dependencies early
  - Helps external team provide the right interface

One Object That Does Everything

- Datacenter uses a few big complex objects
  - Movie and Customer objects are the foundation
  - Good choice for a small team and one instance
  - Problematic for large teams and many instances
- False sharing causes tangled dependencies
  - Unproductive re-integration work

Anti-pattern impacting productivity and availability

An Interface For Each Component

- Cloud uses faceted Video and Visitor
  - Basic types hold only the identifier
  - Facets scope the interface you actually need
  - Each component can define its own facets
- No false-sharing and dependency chains
  - Type manager converts between facets as needed
  - video.asA(PresentationVideo) for www
  - video.asA(MerchableVideo) for middle tier

Response to 2010 talk was a mixture of incomprehension and confusion. Most people thought we were crazy and would be back in our datacenters when it failed…

Replacing Datacenter Oracle with Global Apache Cassandra on AWS

July 11, 2011

Adrian Cockcroft
@adrianco #netflixcloud
http://www.linkedin.com/in/adriancockcroft

Get stuck with wrong config

Wait Wait File tickets

Ask permission Wait Wait

Wait Things We Don't Do Wait

Run out of space/power

Plan capacity in advance

Have meetings with IT Wait

NETFLIX

Netflix could not build new datacenters fast enough

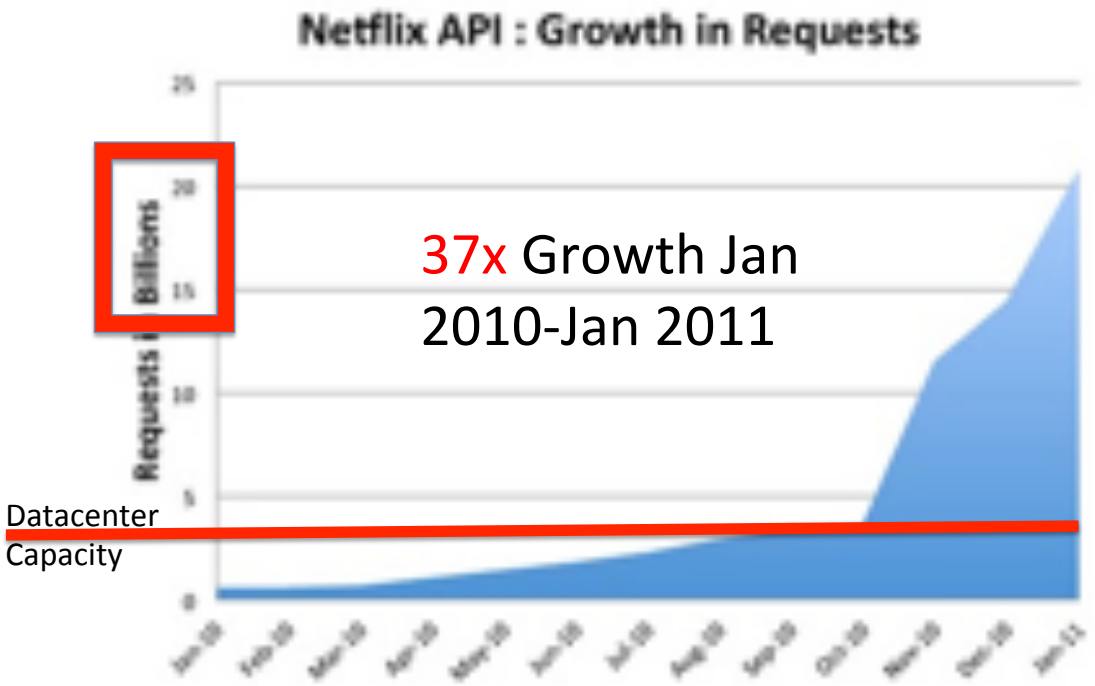
Capacity growth is accelerating, unpredictable

Product launch spikes - iPhone, Wii, PS3, XBox

NETFLIX

Out-Growing Data Center

http://techblog.netflix.com/2011/02/redesigning-netflix-api.html



NETFLIX

High Availability

- Cassandra stores 3 local copies, 1 per zone
  - Synchronous access, durable, highly available
  - Read/Write One fastest, least consistent - ~1ms
  - Read/Write Quorum 2 of 3, consistent - ~3ms
- AWS Availability Zones
  - Separate buildings
  - Separate power etc.
  - Close together

Remote Copies

- Cassandra duplicates across AWS regions
  - Asynchronous write, replicates at destination
  - Doesn't directly affect local read/write latency
- Global Coverage
  - Business agility
  - Follow AWS…
- Local Access
  - Better latency
  - Fault Isolation

NETFLIX

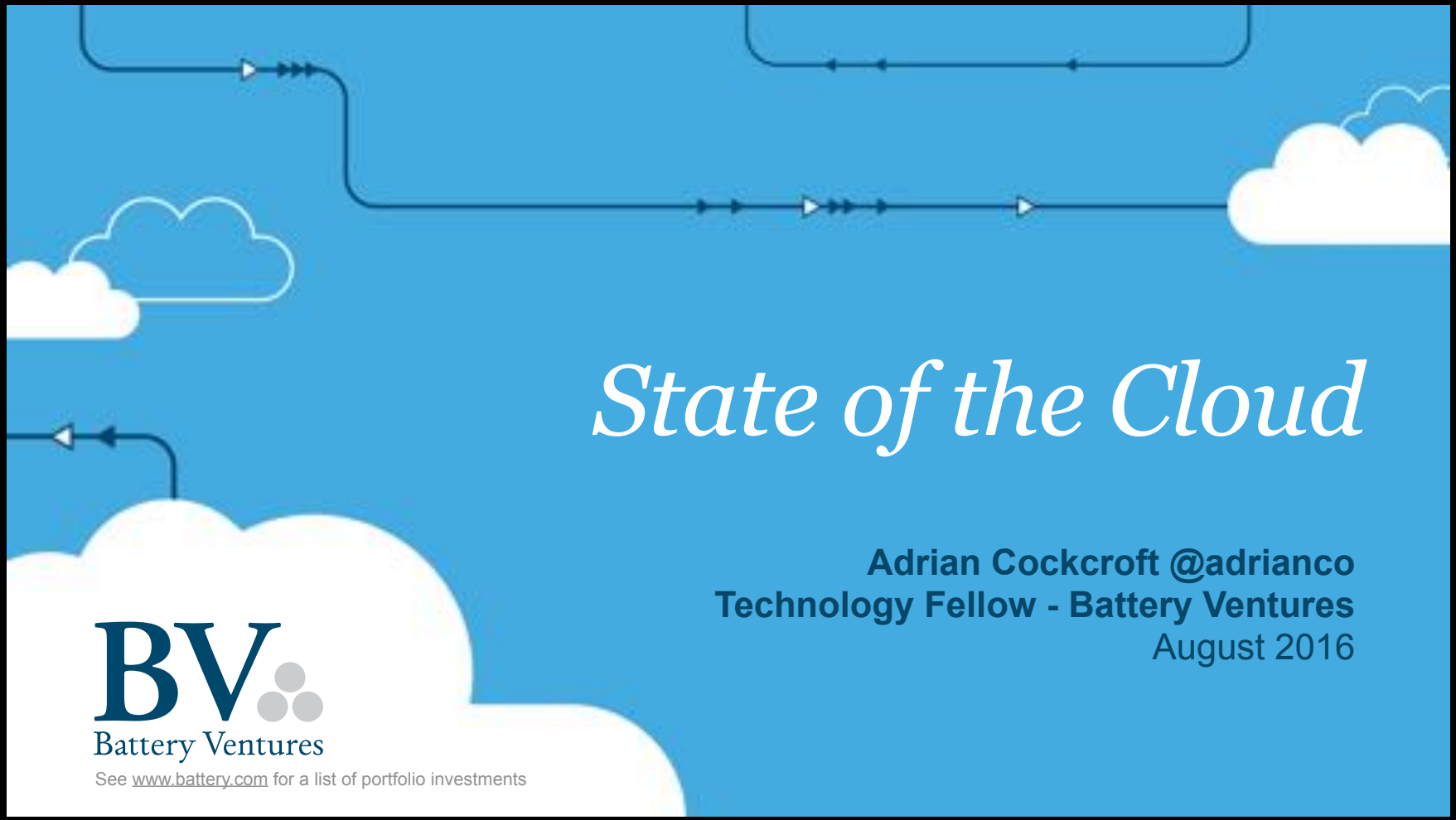
Chaos Monkey

- Make sure systems are resilient
  - Allow any instance to fail without customer impact
- Chaos Monkey hours
  - Monday-Thursday 9am-3pm random instance kill
- Application configuration option
  - Apps now have to opt-out from Chaos Monkey
- Computers (Datacenter or AWS) randomly die
  - Fact of life, but too infrequent to test resiliency

Architecture design control
Be sure you can auto-scale down!

NETFLIX

Response to 2011 progress was that Netflix was a Unicorn, and while it might work for us, it wasn’t relevant to others

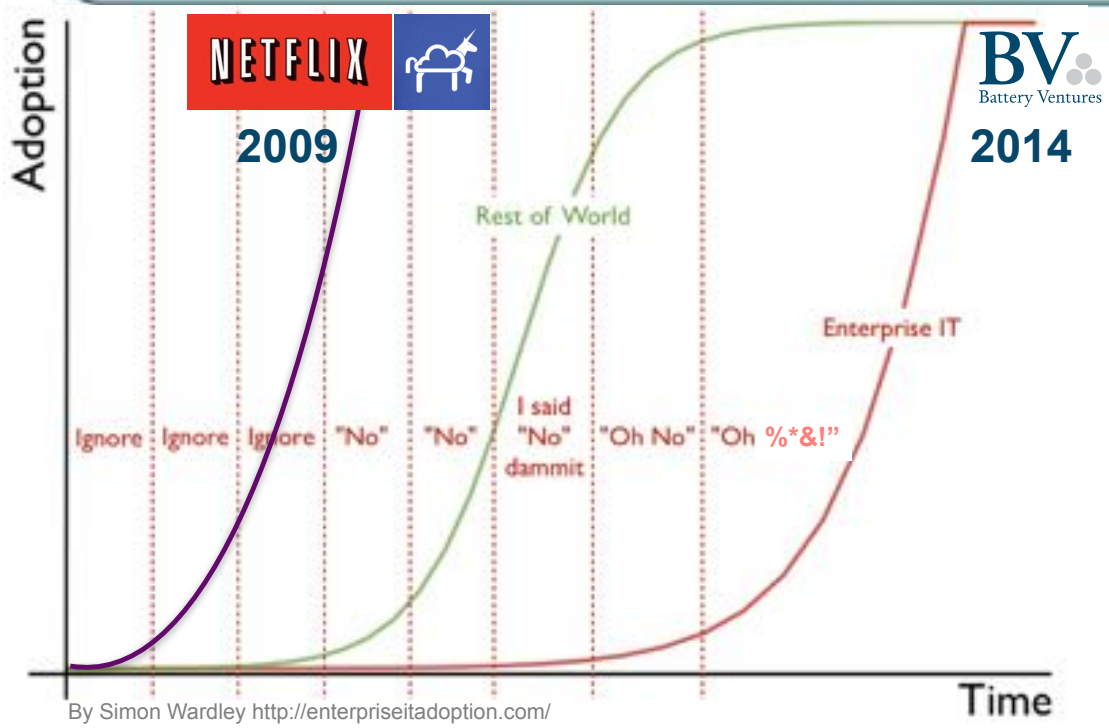

State of the Cloud
Adrian Cockcroft @adrianco
Technology Fellow - Battery Ventures
August 2016
BV
Battery Ventures
See www.battery.com for a list of portfolio investments

Previous Cloud Trend Updates

GigaOM Structure May 2014
D&B Cloud Innovation July 2015
GigaOM Structure November 2015

Trends from 2014: Noted as appropriate

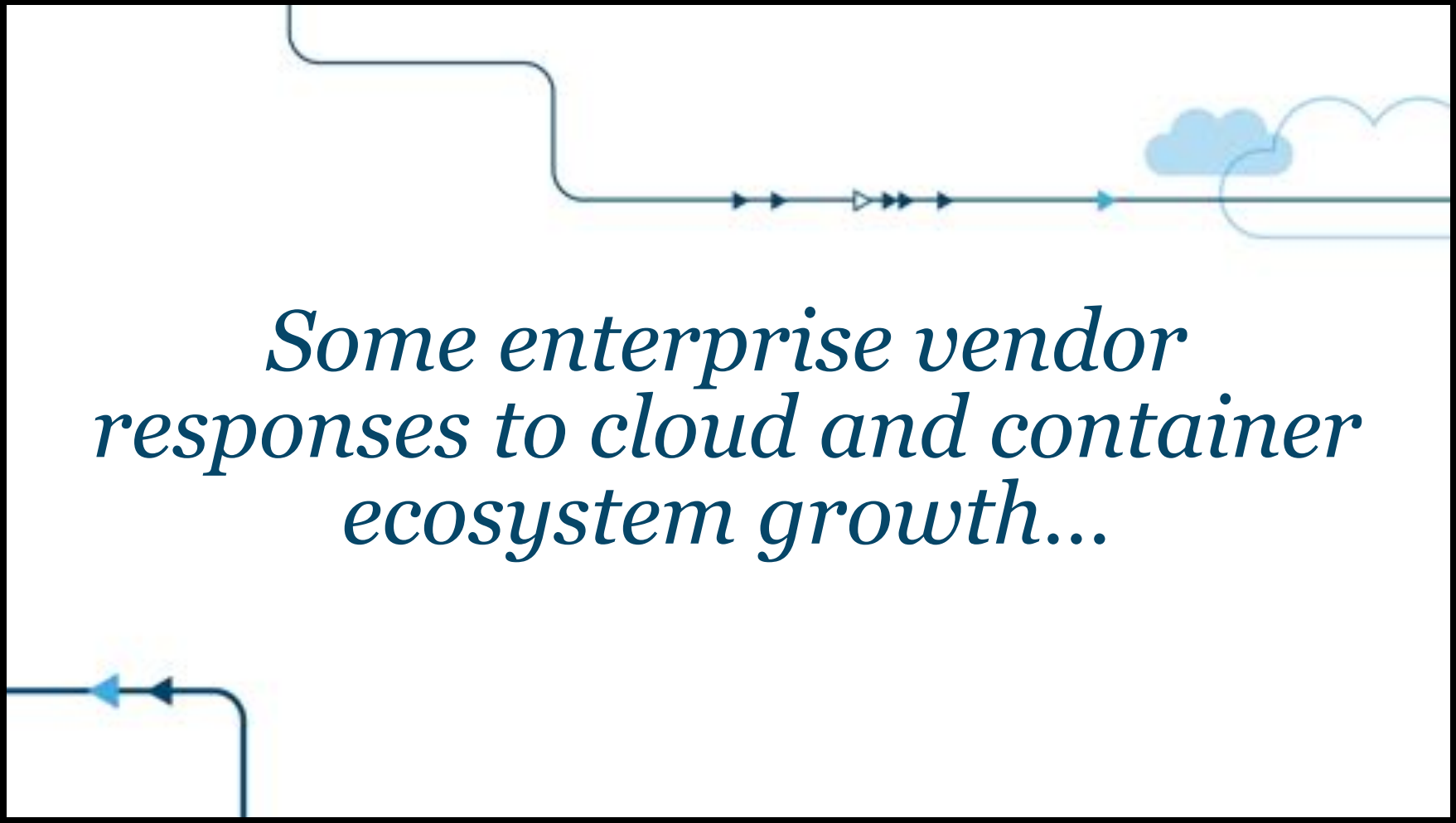
Why am I here?



@adrianco's job at the intersection of cloud and Enterprise IT, looking for disruption and opportunities.

Disruptions in 2016 coming from server-less computing and teraservices.

In 2014 Enterprises finally embraced public cloud and in 2015 serious deployments are under way.


Some enterprise vendor responses to cloud and container ecosystem growth...

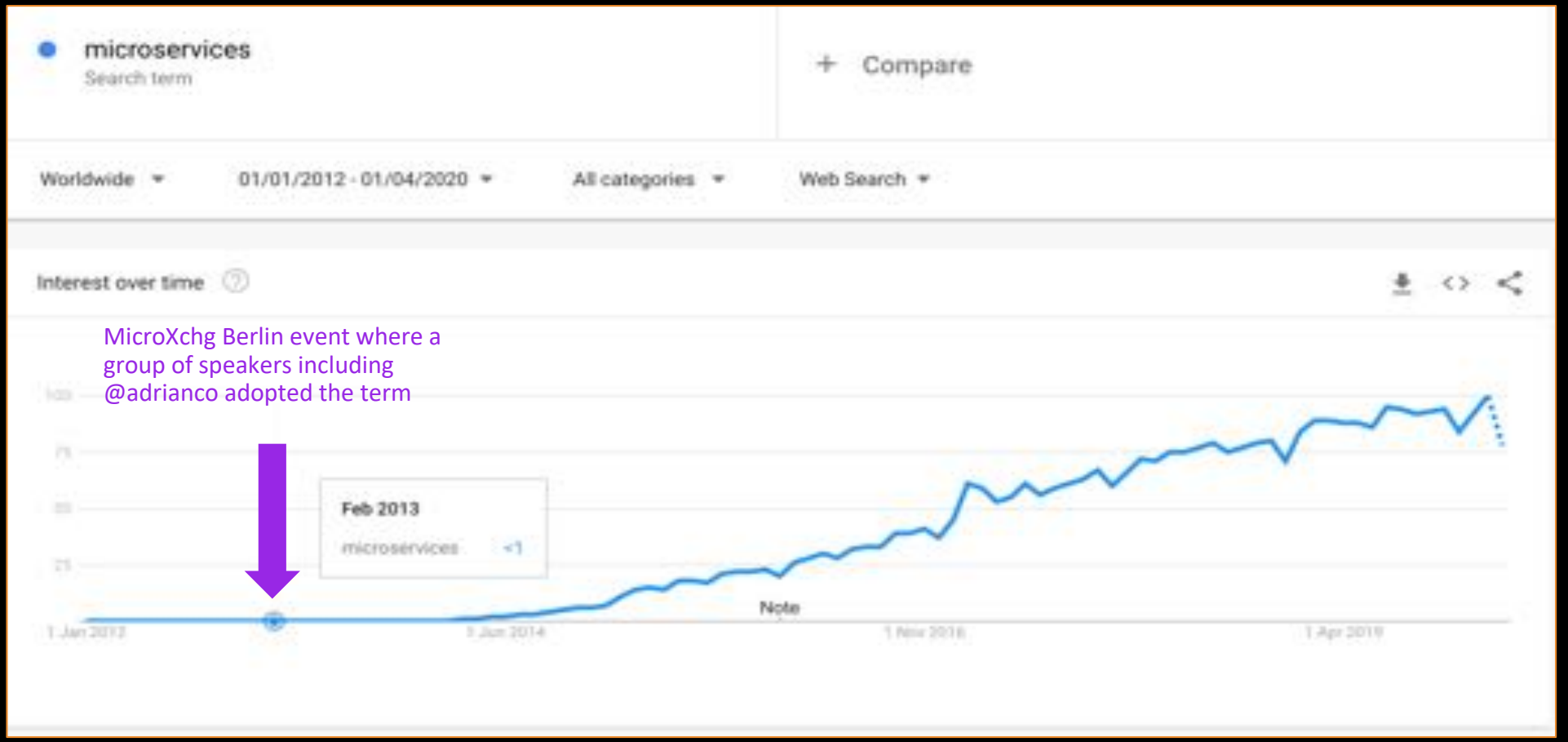
The ship is sinking, let's re-brand as a submarine!

EMC² *The ship is sinking, let's merge with a submarine!*

Look! we cut our ship in two really quickly!

Hewlett Packard
Enterprise

Trends: Microservices

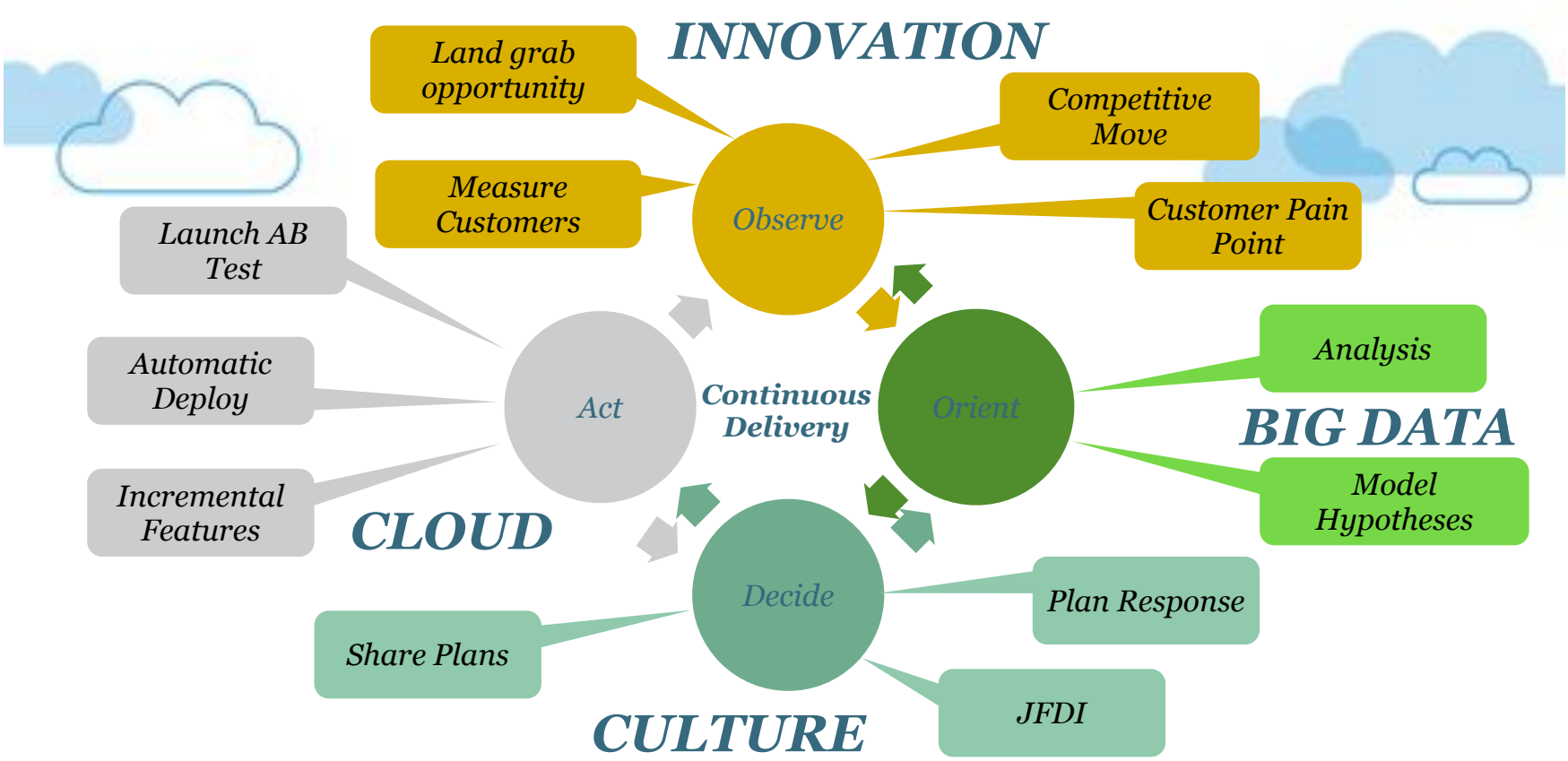



Microservices: Why, What and How
Adrian Cockcroft @adrianco
Technology Fellow - Battery Ventures
October 2016
BV
Battery Ventures

Typical reactions to my Netflix talks…

What I learned from my time at Netflix

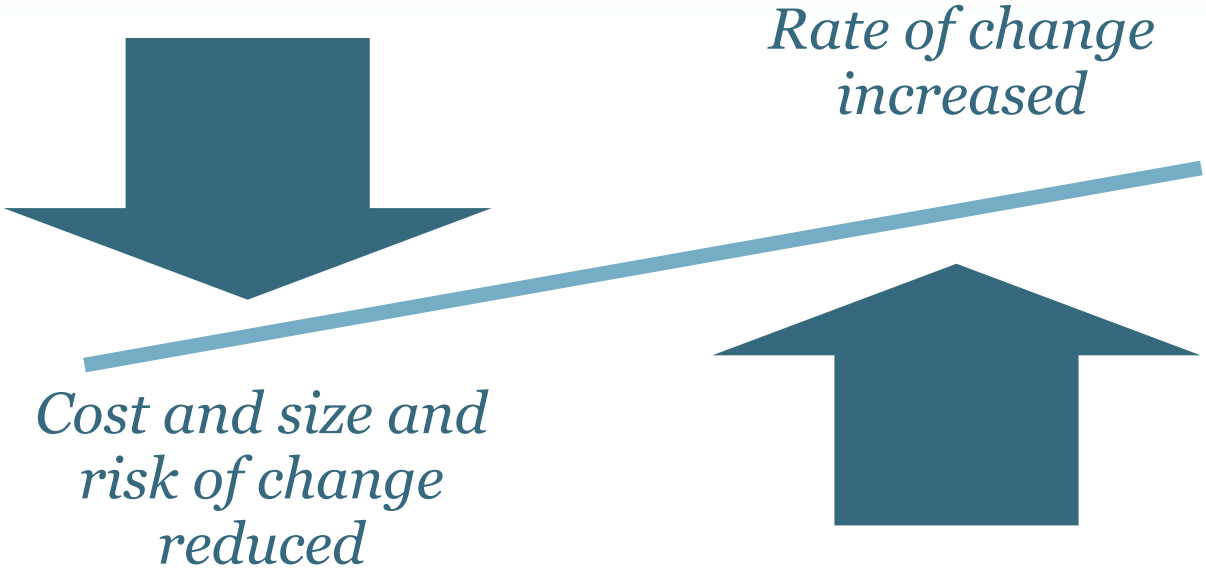
- *Speed wins in the marketplace*
- *Remove friction from product development*
- *High trust, low process, no hand-offs between teams*
- *Freedom and responsibility culture*
- *Don't do your own undifferentiated heavy lifting*
- *Use simple patterns automated by tooling*
- *Self service cloud makes impossible things instant*


INNOVATION
Land grab opportunity
Competitive Move
Measure Customers
Customer Pain Point
Observe
Launch AB Test
Automatic Deploy
Incremental Features
Act
Continuous Delivery
Orient
Analysis
BIG DATA
Model Hypotheses
CLOUD
Decide
Plan Response
Share Plans
JFDI
CULTURE

Non-Destructive Production Updates

- *"Immutable Code" Service Pattern*
  - *Existing services are unchanged, old code remains in service*
  - *New code deploys as a new service group*
  - *No impact to production until traffic routing changes*
- *A|B Tests, Feature Flags and Version Routing control traffic*
  - *First users in the test cell are the developer and test engineers*
  - *A cohort of users is added looking for measurable improvement*

What Happened?



It’s what you know that isn’t so

- *Make your assumptions explicit*
- *Extrapolate trends to the limit*
- *Listen to non-customers*
- *Follow developer adoption, not IT spend*
- *Map evolution of products to services to utilities*
- *Re-organize your teams for speed of execution*

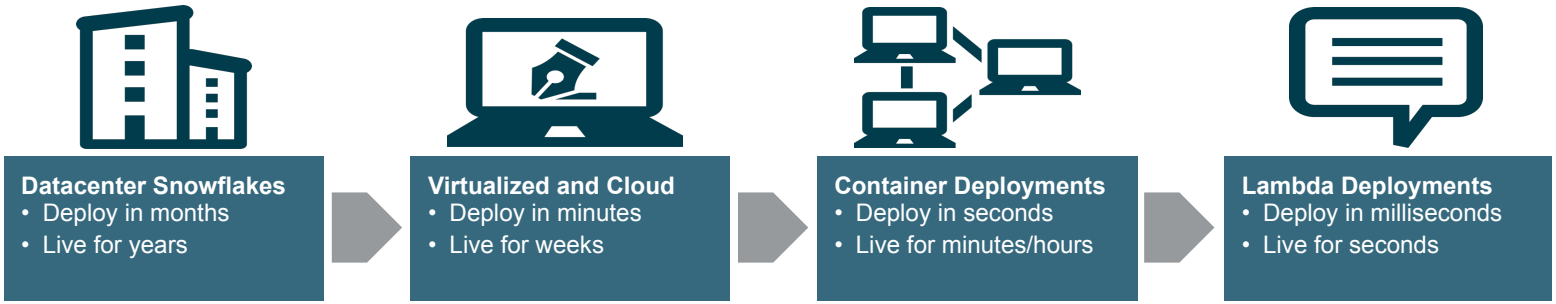
If every service has to be updated at the same time it's not loosely coupled

A Microservice Definition

Loosely coupled service oriented architecture with bounded contexts

If you have to know too much about surrounding services you don't have a bounded context. See the Domain Driven Design book by Eric Evans.

Speeding Up The Platform



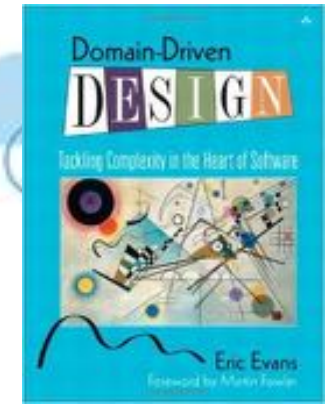
AWS Lambda is leading exploration of serverless architectures in 2016

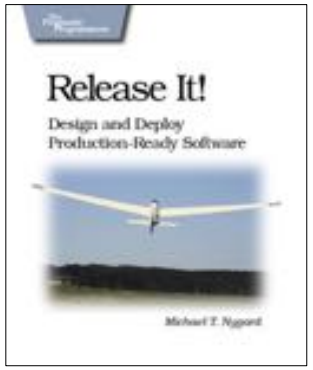
Separate Concerns with Microservices

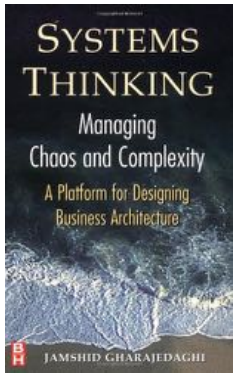
- *Invert Conway's Law – teams own service groups and backend stores*
- *One "verb" per single function micro-service, size doesn't matter*
- *One developer independently produces a micro-service*
- *Each micro-service is it's own build, avoids trunk conflicts*
- *Deploy in a container: Tomcat, AMI or Docker, whatever…*
- *Stateless business logic. Cattle, not pets.*
- *Stateful cached data access layer using replicated ephemeral instances*

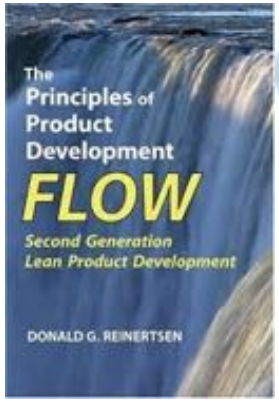
http://en.wikipedia.org/wiki/Conway's_law

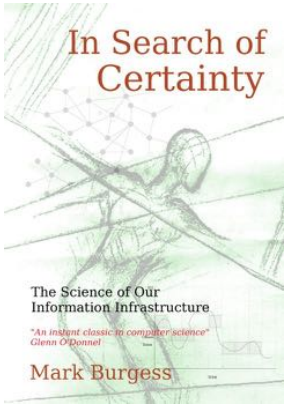
Inspiration

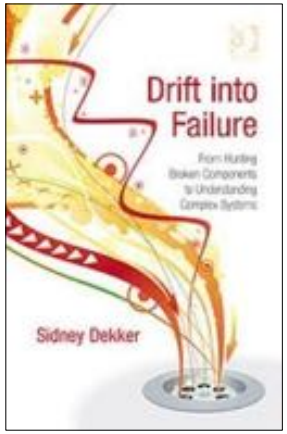


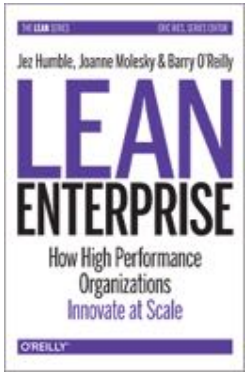


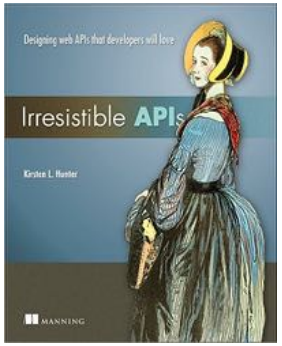


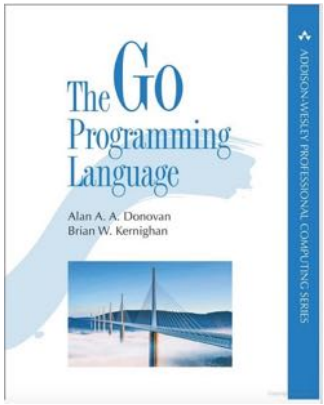












What's Missing?

Advanced Microservices Topics

Failure injection testing
Versioning, routing
Binary protocols and interfaces
Timeouts and retries
Denormalized data models
Monitoring, tracing
Simplicity through symmetry

@adrianco
BV
Battery Ventures

Benefits of version aware routing

Immediately and safely introduce a new version
Canary test in production
Use DIY feature flags, 

, A|B tests with Wasabi

Route clients to a version so they can't get disrupted
Change client or dependencies but not both at once

Eventually remove old versions
Incremental or infrequent "break the build" garbage collection

@adrianco
BV
Battery Ventures

Timeouts and Retries

Connection timeout vs. request timeout confusion

Usually setup incorrectly, global defaults

Systems collapse with "retry storms"

Timeouts too long, too many retries

Services doing work that can never be used

@adrianco
BV
Battery Ventures

Timeouts and Retries

Bad config: Every service defaults to 2 second timeout, two retries

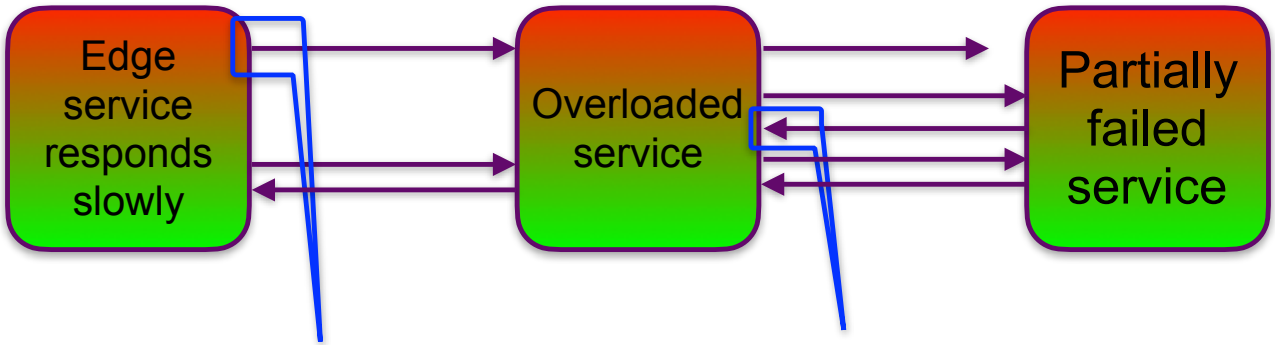
If anything breaks, everything upstream stops responding

Retries add unproductive work

@adrianco
BV Battery Ventures

Timeouts and Retries

Bad config: Every service defaults to 2 second timeout, two retries



First request from Edge timed out so it ignores the successful response and keeps retrying. Middle service load increases as it's doing work that isn't being consumed

Timeout and Retry Fixes

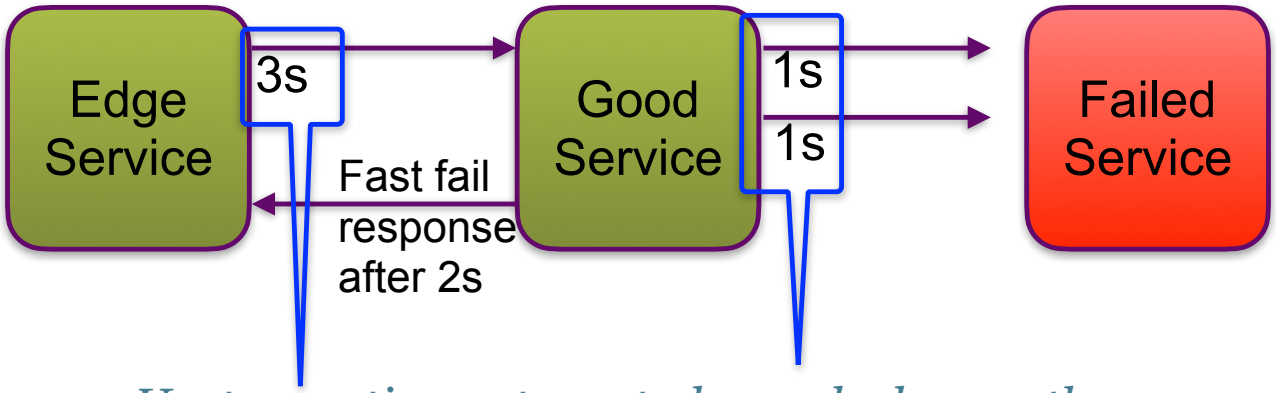
Cascading timeout budget
Static settings that decrease from the edge
or dynamic budget passed with request

How often do retries actually succeed?
Don't ask the same instance the same thing
Only retry on a different connection

@adrianco
BV
Battery Ventures

Timeouts and Retries

Budgeted timeout, one retry



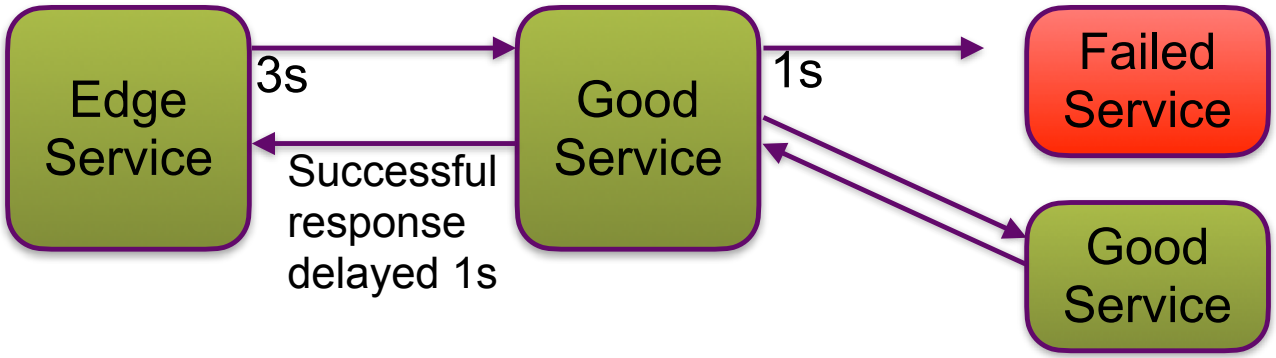
*Upstream timeout must always be longer than total downstream timeout * retries delay*

No unproductive work while fast failing

@adrianco
BV
Battery Ventures

Timeouts and Retries

Budgeted timeout, failover retry



For replicated services with multiple instances never retry against a failed instance

No extra retries or unproductive work

@adrianco
BV
Battery Ventures

"We see the world as increasingly more complex and chaotic because we use inadequate concepts to explain it. When we understand something, we no longer see it as chaotic or complex."

Jamshid Gharajedaghi - 2011
Systems Thinking: Managing Chaos and Complexity: A Platform for Designing Business Architecture

@adrianco
BV
Battery Ventures

I wanted to learn Go, and build something I could talk about at events. I ported an actor-based simulator from Occam to Go and generated large complex simulated microservice graphs with it using Go channels as networks.

Then I gave a talk at Gophercon about the history of channels…

Communicating Sequential Goroutines

Adrian Cockcroft @adrianco
Technology Fellow - Battery Ventures
July 2016

Agenda

1978	*Communicating Sequential Processes*
1983	*Occam*
	How Channels Work
1992	*Pi-Calculus*
2013	*The Life of Occam-Pi*
2006	*Occam-Pi based simulation*
	Pi-Calculus ideas in Go
	Go Applications

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays
CR Categories: 4.20, 4.22, 4.32

"...the concepts and notations introduced in this paper (although described in the next section in the form of a programming language fragment) ***should not be regarded as suitable for use as a programming language,*** *either for abstract or for concrete programming. They are at best only a partial solution to the problems tackled."*

CSP Issues:

Not a full language
Hard to read
Process addressing

David May's Occam Language

Extremely simple and elegant implementation of CSP as a language

Adds named channels
Designed as the assembly language for Transputer hardware

```
Occam is intended to be the smallest language which  is  adequate
for  its purpose; however, suggestions for further simplification
would be welcome.
```

Comparing Occam and Go

Parallel Channel Assignment

```
PROC main(CHAN out)
  VAR x,y:
  SEQ
    x := 1
    CHAN c:
    PAR
      c!x
      c?y
    out!y
:
```

```
func main() {
          var x, y int
          x = 1
          c := make(chan int)
          var wg sync.WaitGroup
          wg.Add(2)
          go func() { defer wg.Done(); c <- x }()
          go func() { defer wg.Done(); y = <-c }()
          wg.Wait()
          fmt.Println(y)
}
```

Pi-Calculus
Robin Milner 1992

A Calculus of Mobile Processes, I

ROBIN MILNER

University of Edinburgh, Scotland

JOACHIM PARROW

Swedish Institute of Computer Science, Kista, Sweden

AND

DAVID WALKER

University of Warwick, England

We present the π-calculus, a calculus of communicating systems in which one can naturally express processes which have changing structure. Not only may the component agents of a system be arbitrarily linked, but a communication between neighbours may carry information which changes that linkage. The calculus is an extension of the process algebra CCS, following work by Engberg and Nielsen, who added mobility to CCS while preserving its algebraic properties. The π-calculus gains simplicity by removing all distinction between variables and constants; communication links are identified by *names*, and computation is represented purely as the communication of names across links. After an illustrated description of how the π-calculus generalises conventional process algebras in treating mobility, several examples exploiting mobility are given in some detail. The important examples are the encoding into the π-calculus of higher-order functions (the λ-calculus and combinatory algebra), the transmission of processes as values, and the representation of data structures as processes. The paper continues by presenting the algebraic theory of *strong bisimilarity* and *strong equivalence*, including a new notion of equivalence indexed by *distinctions*—i.e., assumptions of inequality among names. These theories are based upon a semantics in terms of a labeled transition system and a notion of *strong bisimulation*, both of which are expounded in detail in a companion paper. We also report briefly on work-in-progress based upon the corresponding notion of *weak* bisimulation, in which internal actions cannot be observed.

We present the π-calculus, a calculus of communicating systems in which one can naturally express processes which have changing structure. Not only may the component agents of a system be arbitrarily linked, but a communication between neighbours may carry information which changes that linkage.

In this paper we do not present the basic semantics of the calculus; this is done in our companion paper (Milner, Parrow, and Walker, 1989), in the same style as in CCS, namely as a labeled transition system defined by structural inference rules. In that paper the notions of strong bisimulation and strong equivalence are also defined; the latter is a congruence relation, so it may be understood as (strong) semantic equality. Here, we shall rely somewhat upon analogy with the transitions of CCS agents. In particular, we assume simple transitions such as

$$(\cdots + \bar{y}x.P + \cdots) \mid (\cdots + y(z).Q + \cdots) \xrightarrow{\tau} P \mid Q\{x/z\}$$

and simple equations such as

$$(y)(\bar{y}x.P \mid y(z).Q) = \tau.(y)(P \mid Q\{x/z\})$$

It's easy to show that...

This paper is incomprehensible!

A triumph of notation over comprehension.

Simple equations such as...

It is illuminating to see how the encoding of a particular example behaves. Consider $(\lambda xx)N$; first, we have

$$[\![\lambda xx]\!]v \equiv v(x)(w).\bar{x}w.$$

So, assuming x not free in N,

$$\begin{aligned}
[\![(\lambda xx)N]\!]u &\equiv (v)([\![\lambda xx]\!]v \mid (x)\bar{v}xu.x(w).[\![N]\!]w) \\
&\equiv (v)(v(x)(w).\bar{x}w \mid (x)\bar{v}xu.x(w).[\![N]\!]w) \\
&= \tau.(v)(x)(v(w).\bar{x}w \mid \bar{v}u.x(w).[\![N]\!]w) \\
&= \tau.\tau.(v)(x)(\bar{x}u \mid x(w).[\![N]\!]w) \\
&= \tau.\tau.\tau.(v)(x)(\mathbf{0} \mid [\![N]\!]u) = \tau.\tau.\tau.[\![N]\!]u.
\end{aligned}$$

More generally, it is easy to show that

$$[\![(\lambda xM)N]\!]u \approx [\![M\{N/x\}]\!]u, \qquad (36)$$

Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Life of occam-Pi

Peter H. WELCH

School of Computing, University of Kent, UK

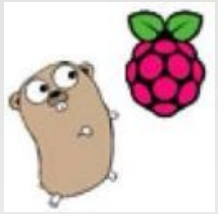
p.h.welch@kent.ac.uk

Abstract. This paper considers some questions prompted by a brief review of the history of computing. Why is programming so hard? Why is concurrency considered an *"advanced"* subject? What's the matter with *Objects*? Where did all the Maths go? In searching for answers, the paper looks at some concerns over fundamental ideas within *object orientation* (as represented by modern programming languages), before focussing on the concurrency model of communicating processes and its particular expression in the occam family of languages. In that focus, it looks at the history of occam, its underlying philosophy (*Ockham's Razor*), its semantic foundation on Hoare's CSP, its principles of *process oriented design* and its development over almost three decades into occam-π (which blends in the concurrency dynamics of Milner's π-calculus). Also presented will be an urgent need for rationalisation – occam-π is an experiment that has demonstrated significant results, but now needs time to be spent on careful review and implementing the conclusions of that review. Finally, the future is considered. In particular, is there a future?

Keywords. process, object, local reasoning, global reasoning, occam-pi, concurrency, compositionality, verification, multicore, efficiency, scalability, safety, simplicity

...looks at the history of occam, its underlying philosophy (Ockham's Razor), its semantic foundation on Hoare's CSP, its principles of process oriented design and its development over almost three decades into occam-π (which blends in the concurrency dynamics of Milner's π-calculus).

Go-π



Dynamic Channel Protocol
Actor Pattern
Partitioned Service Registry
Logging and Tracing

Dynamic Channel Protocol

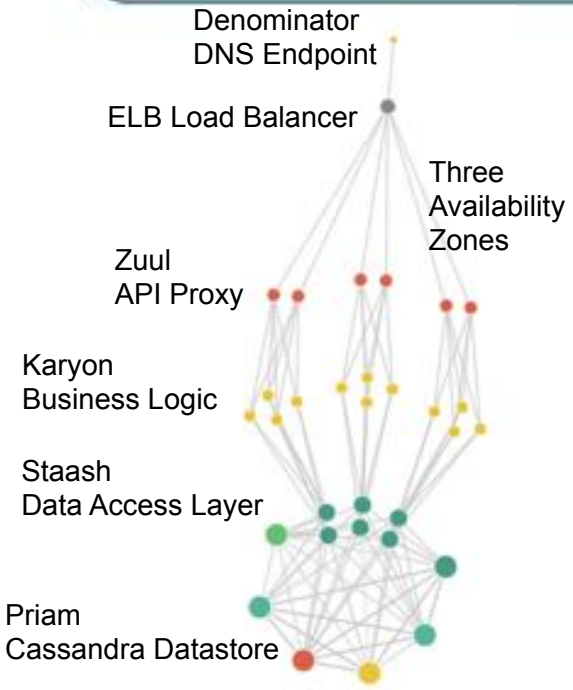
https://github.com/adrianco/spigo/tree/master/tooling/gotocol

Imposition/Intention? https://en.wikipedia.org/wiki/Promise_theory

```
ch <-gotocol.Message{gotocol.GetRequest, listener, now, ctx, "why?"}

// Message structure used for all messages, includes a channel of itself
type Message struct {
        Imposition   Impositions  // request type
        ResponseChan chan Message // place to send response messages
        Sent         time.Time    // time at which message was sent
        Ctx          Context      // message context
        Intention    string       // payload
}
```

Simulated Microservices



Model and visualize microservices
Simulate interesting architectures
Generate large scale configurations
Eventually stress test real tools

Code: github.com/adrianco/spigo
Simulate Protocol Interactions in Go
Visualize with D3
See for yourself: http://simianviz.surge.sh
Follow @simianviz for updates

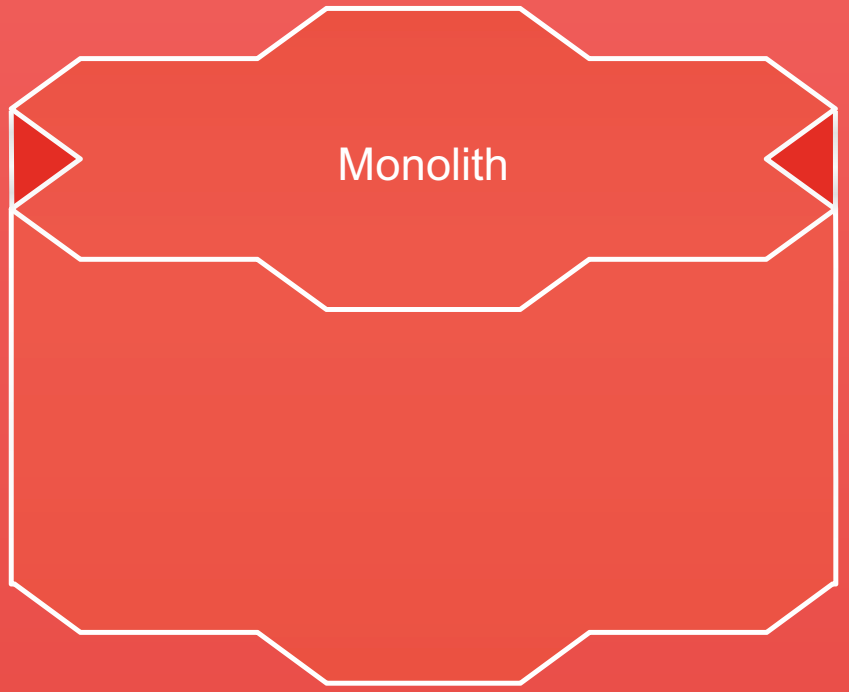
Conclusions

CSP is too limited
π-Calculus syntax is incomprehensible
Occam-Pi makes CSP and π-Calculus readable
Go concurrency syntax is clumsy in places but works
Showed some useful channel based Go-π idioms
Pass channels over channels for dynamic routing
Go works well for actor like simulation

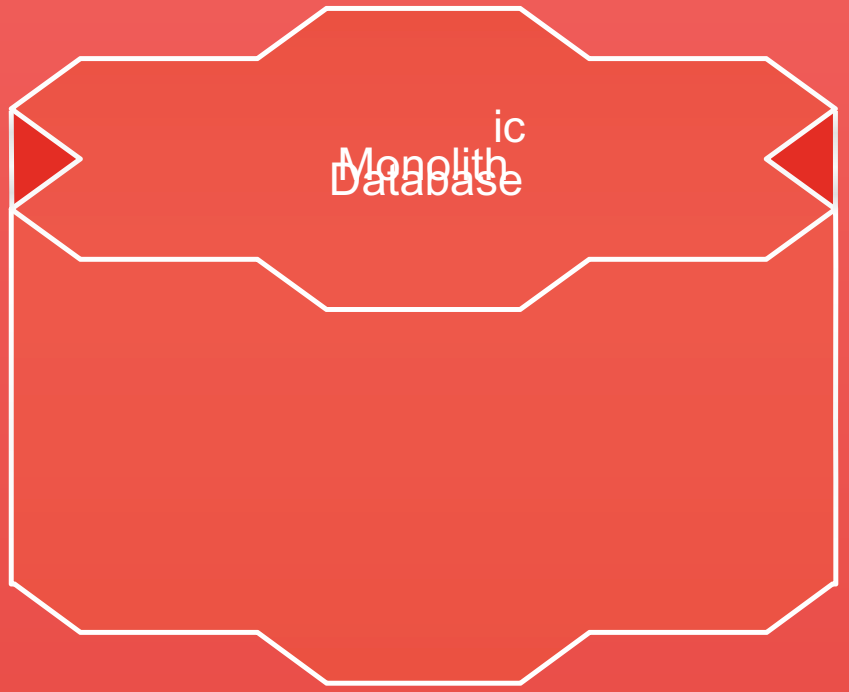
Adrian signed to a new label (AWS) at the end of 2016 and now had a much bigger production budget for making slides look cool, and a PR department to keep him from being too controversial!

The New De-Normal

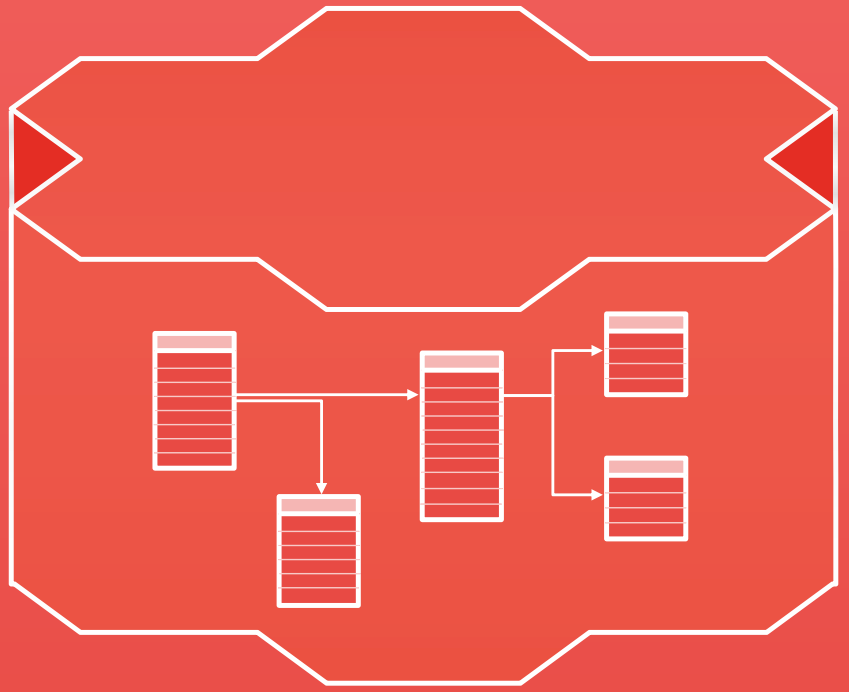
Expensive, Hard to Create and Run



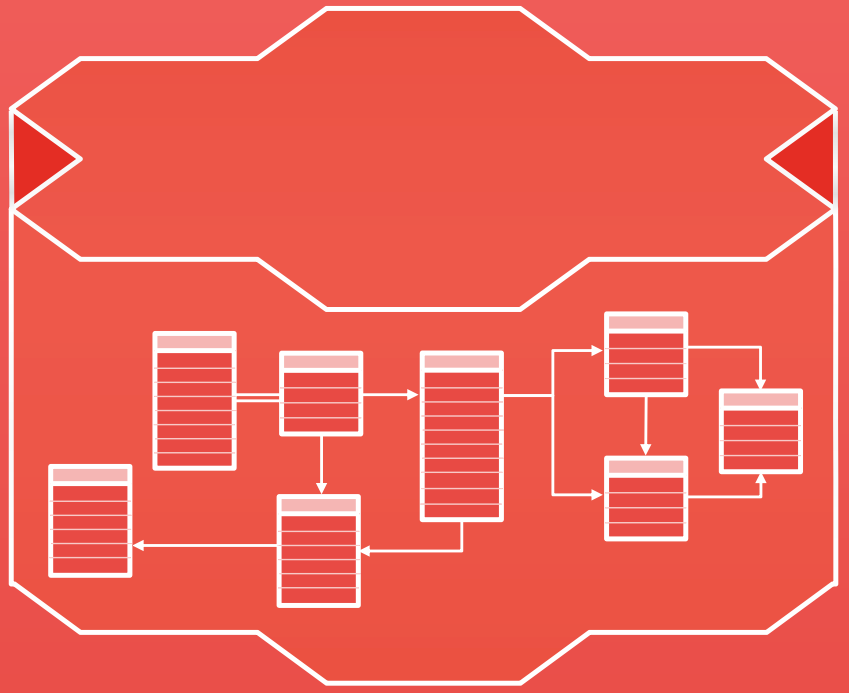
Expensive,
Hard to Create
and Run



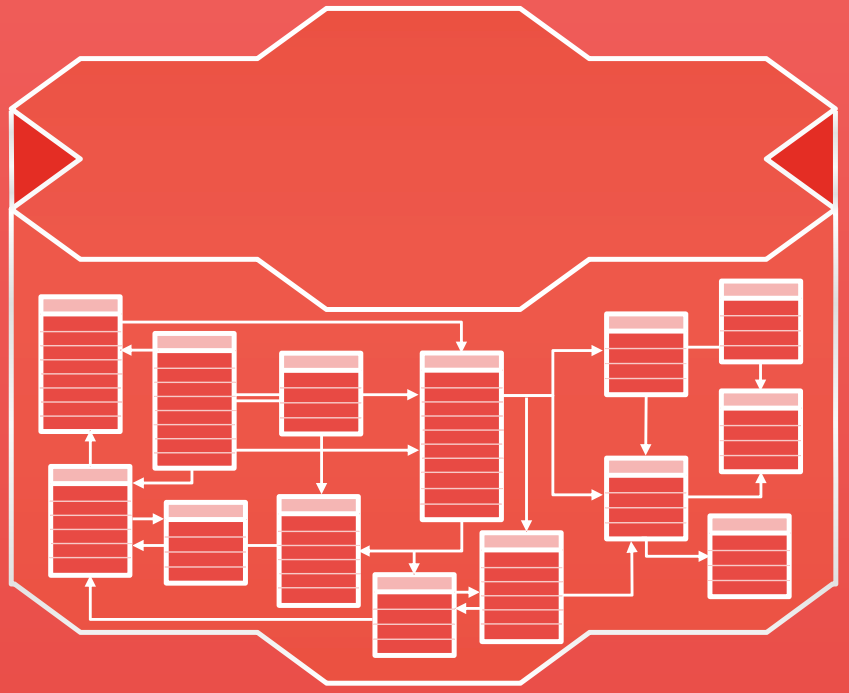
Database Schema
Entity Relationship



Database Schema
Entity Relationship



Database Schema
Entity Relationship

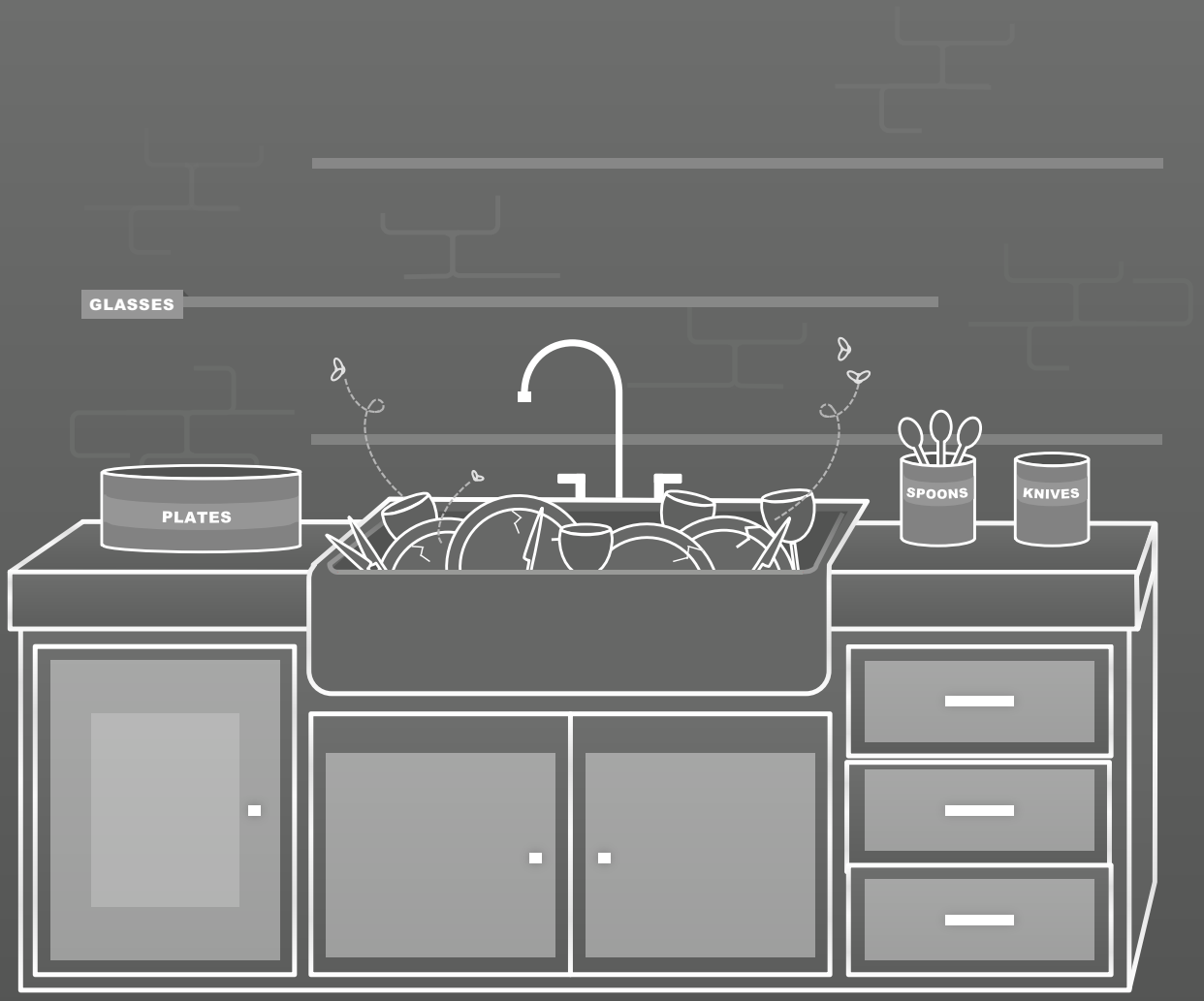


Kitchen Sink Analogy

Kitchen Sink Cleanup

Kitchen Sink Cleanup

Kitchen Sink Cleanup



Kitchen Sink Cleanup

Kitchen Sink Cleanup

Kitchen Sink Cleanup

Kitchen Sink Cleanup

Consistency Problem

How Many Complete Sets Are There?

Consistency Problem

How Many Complete Sets Are There?

Consistency Problem

How Many Complete Sets Are There?

Adding a New Use Case

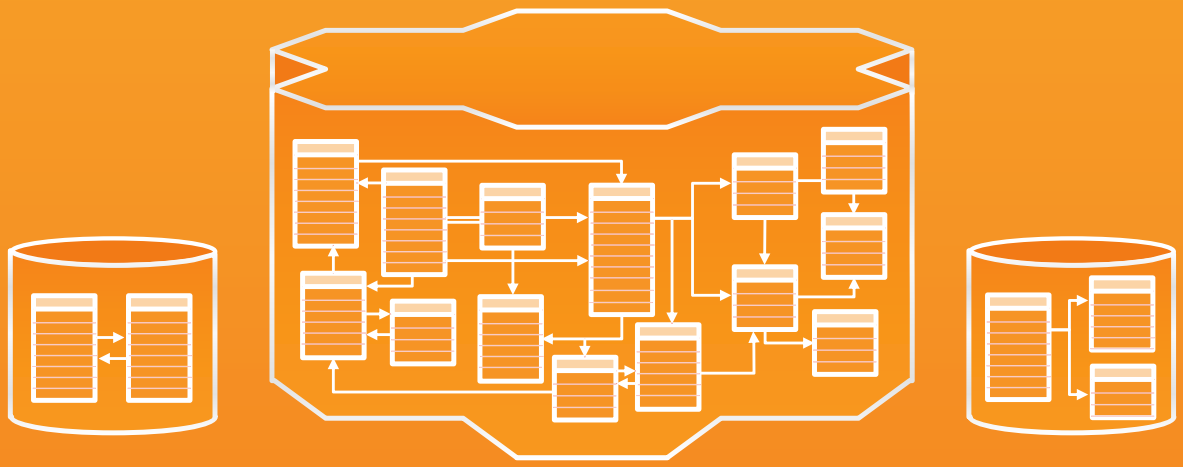
Adding a New Use Case

Cloud Makes it Easy to Add New Databases

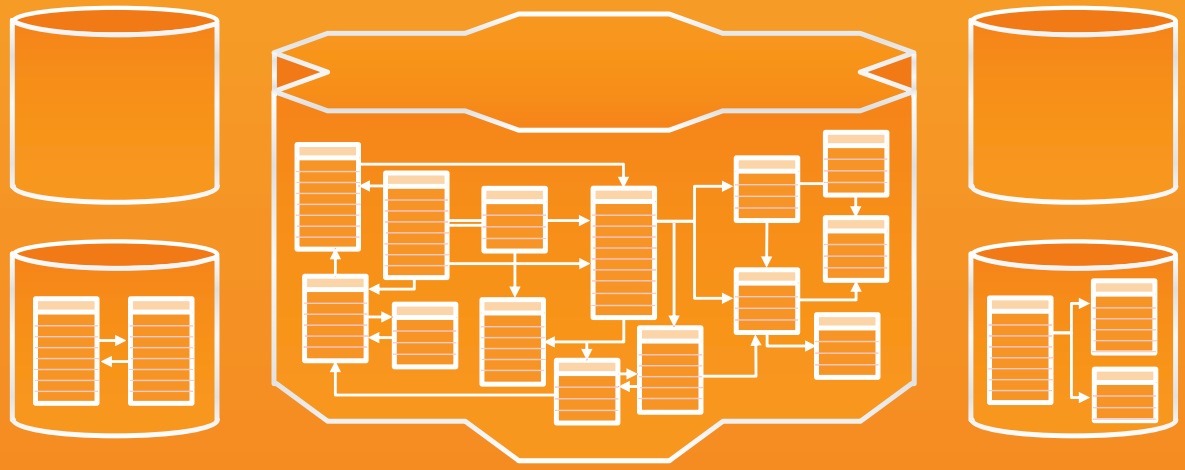
Amazon DMS

Amazon DynamoDB

Amazon RDS



Untangle and Migrate Existing “Kitchen Sink” Schemas

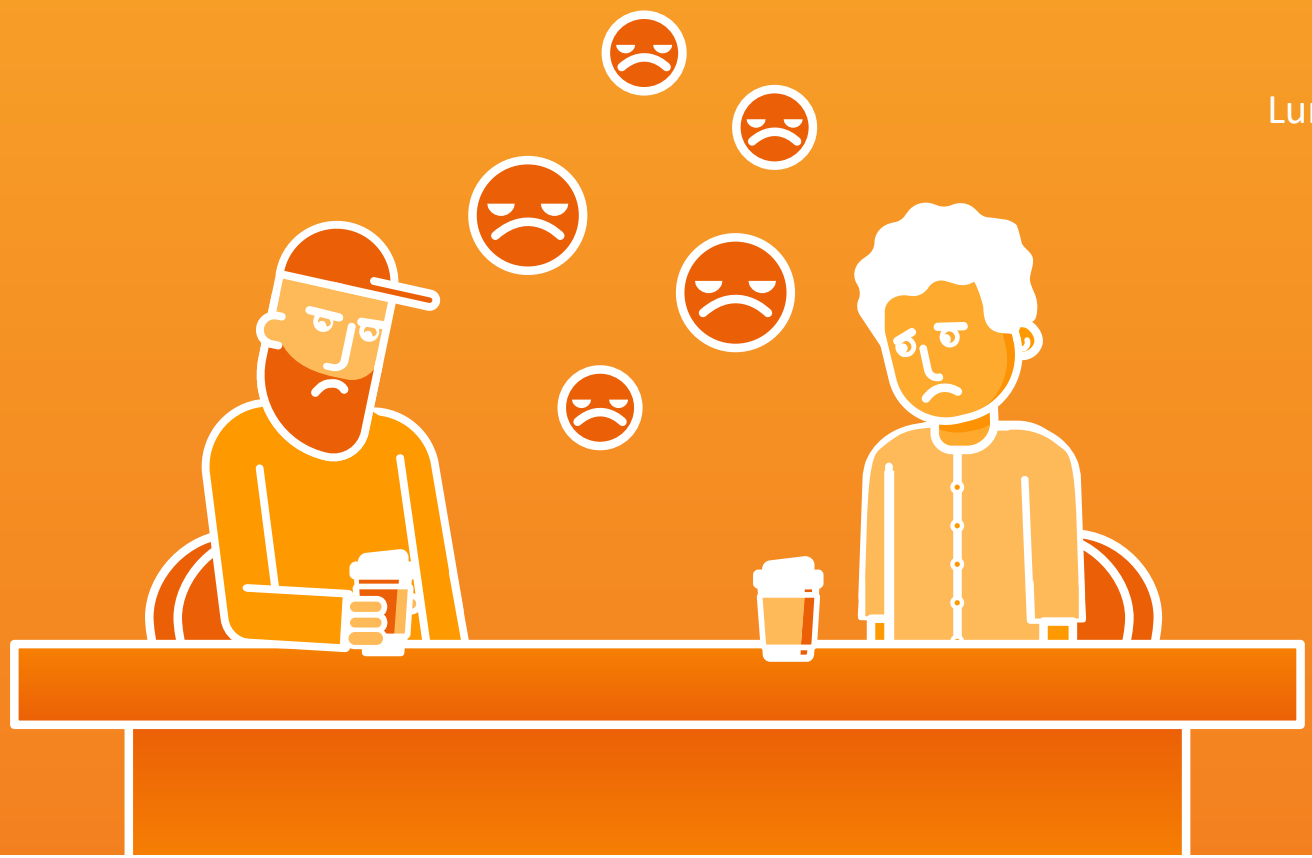


Untangle and Migrate Existing "Kitchen Sink" Schemas

I started to collect stories, scribbled them as rough ideas on my iPad with Apple pencil, sent them to the graphic designer, and got some cool decks back…

Conventional Development

Conventional Development



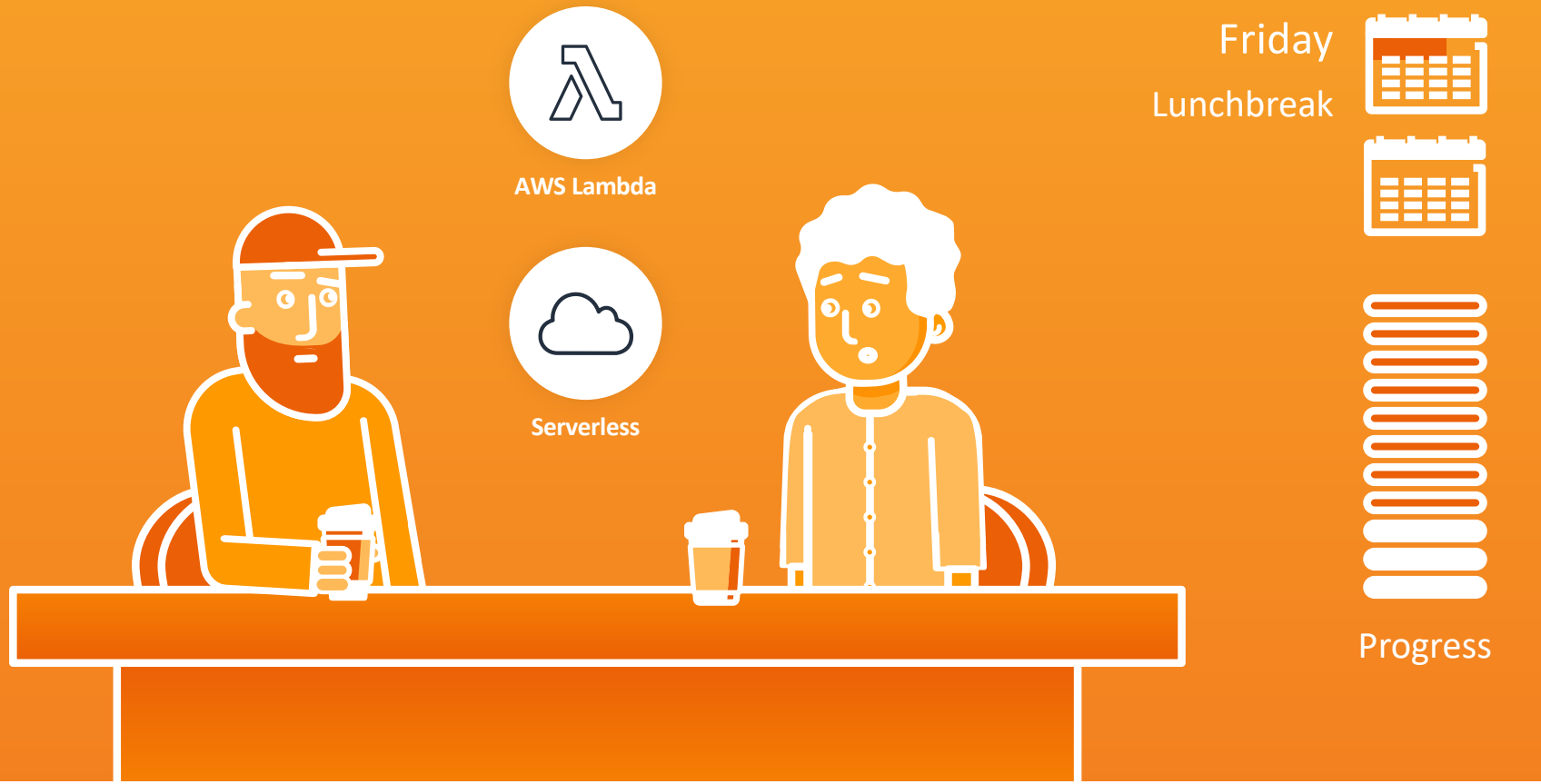
2 months left

Friday

Lunchbreak

Progress

Conventional Development



Serverless Development

2 months left

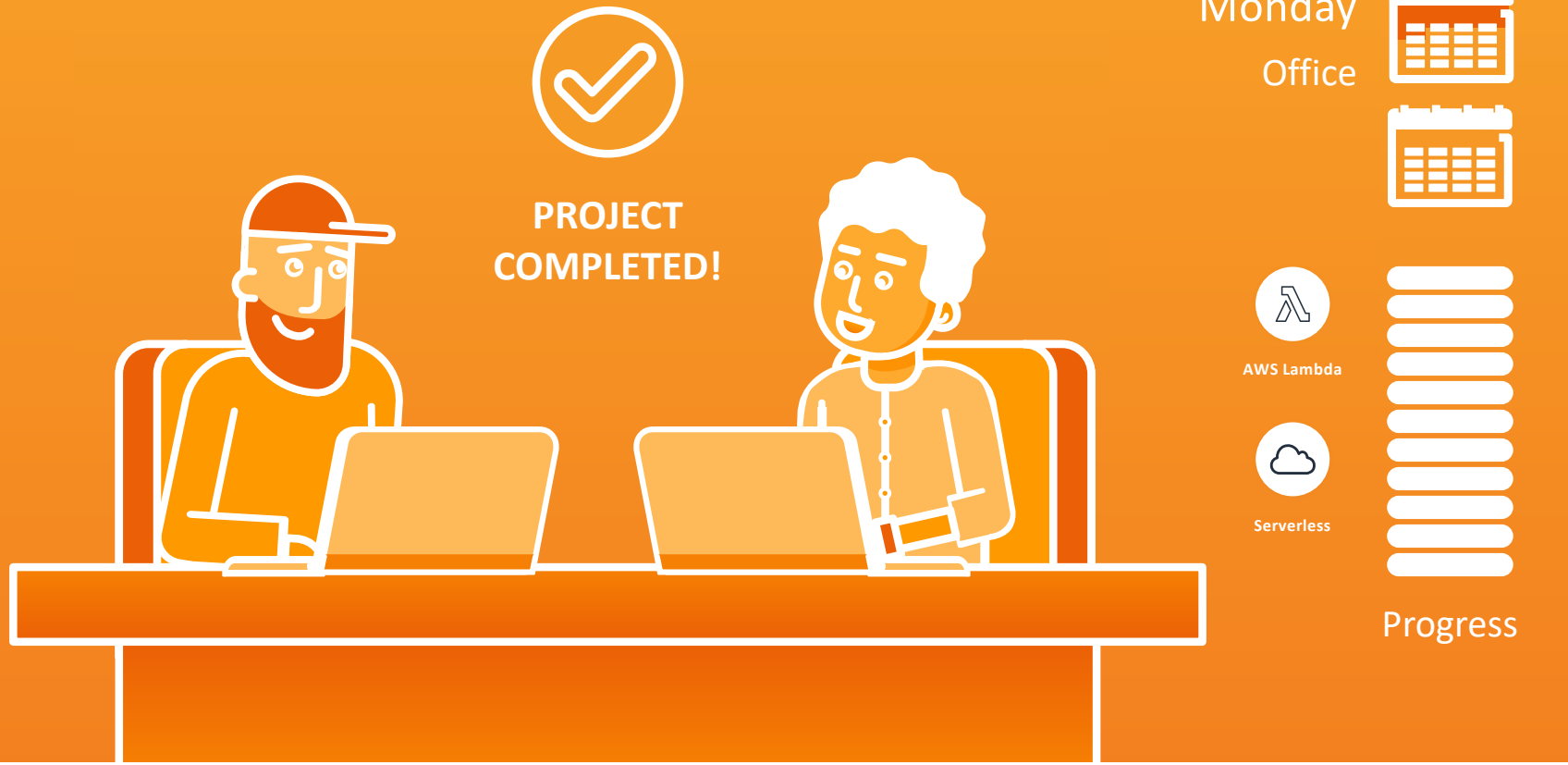
Home

AWS Lambda

Serverless

Progress

Serverless Development



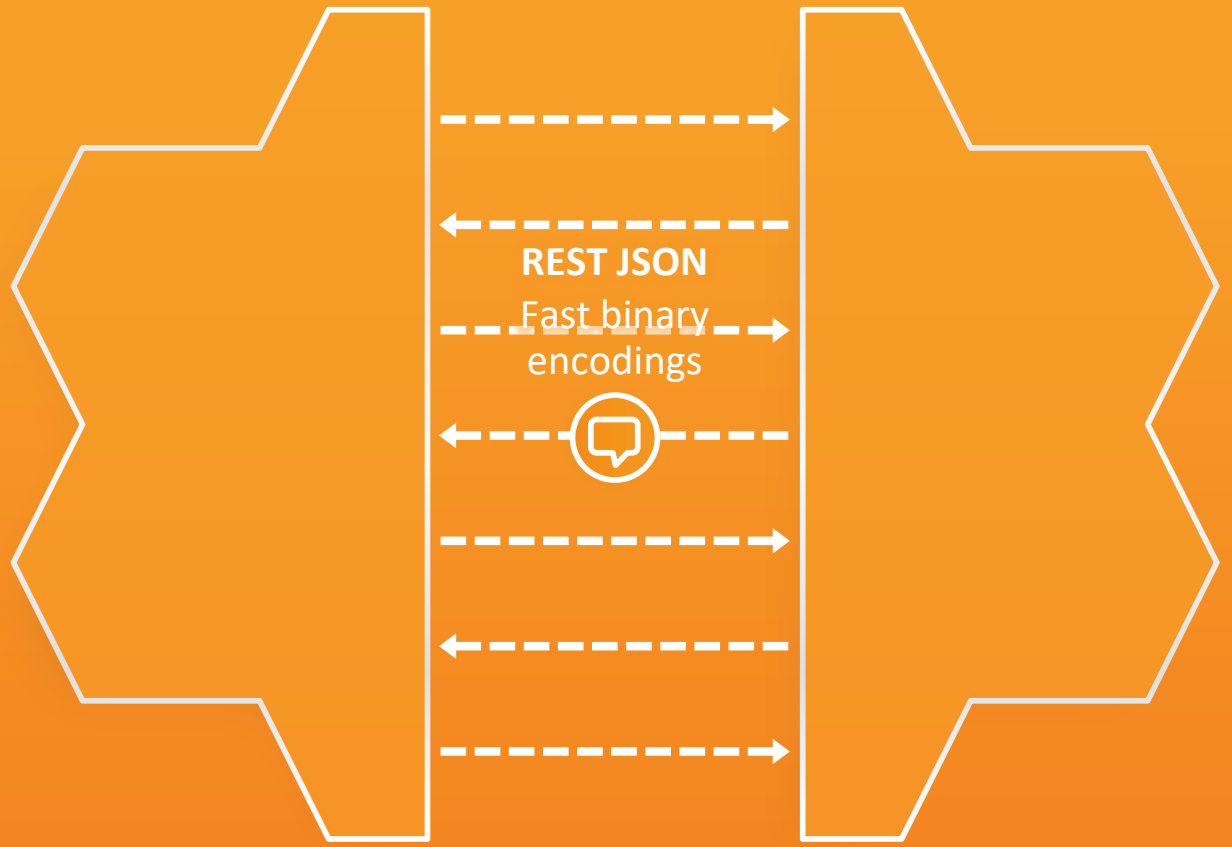
Serverless Development

Serverless Development

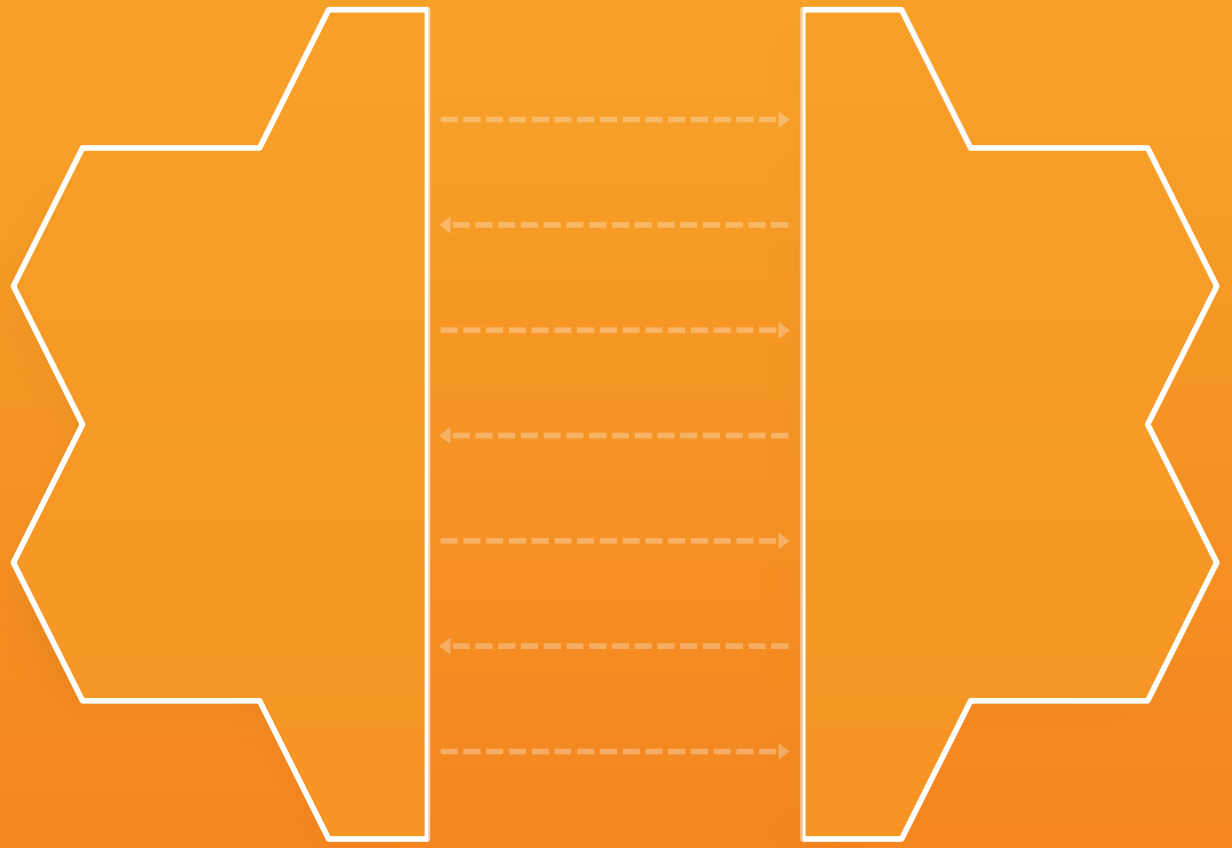
Shipped application 1 month early

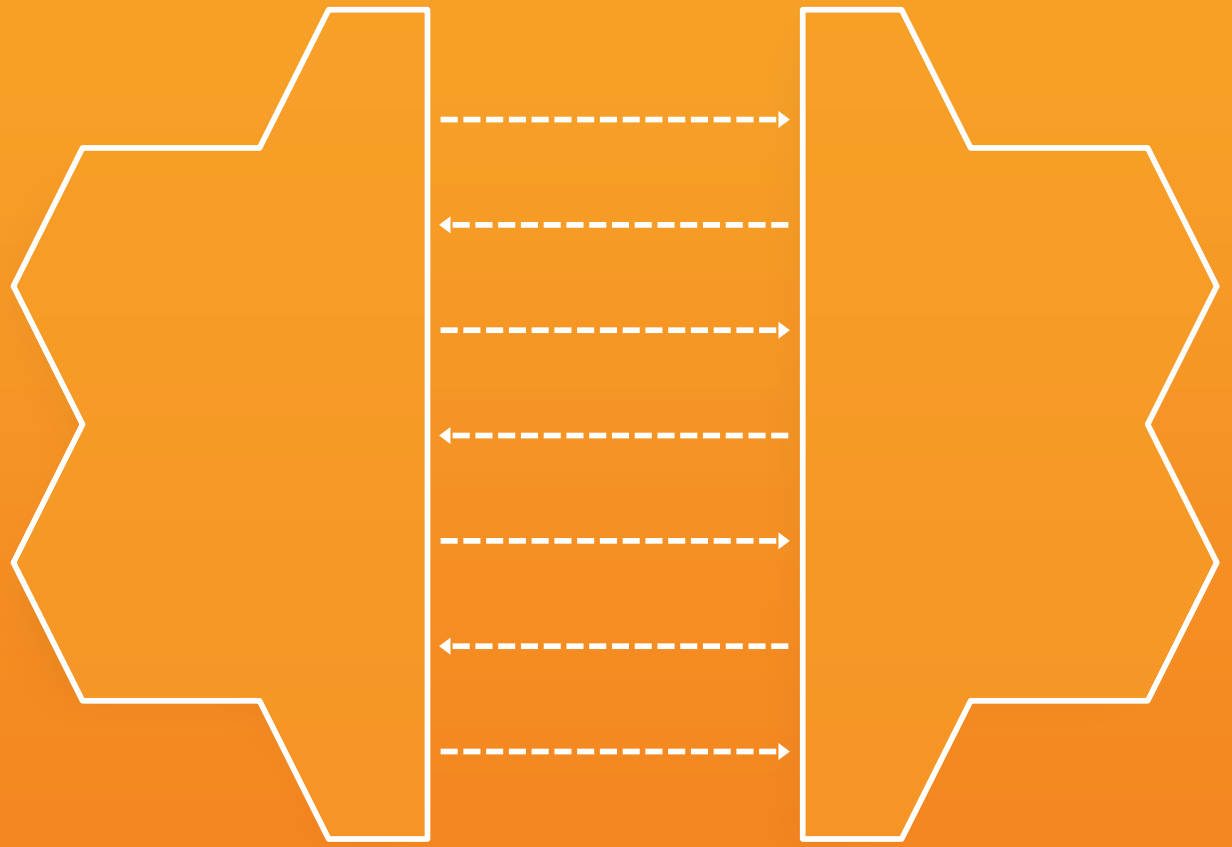
What is different about serverless?

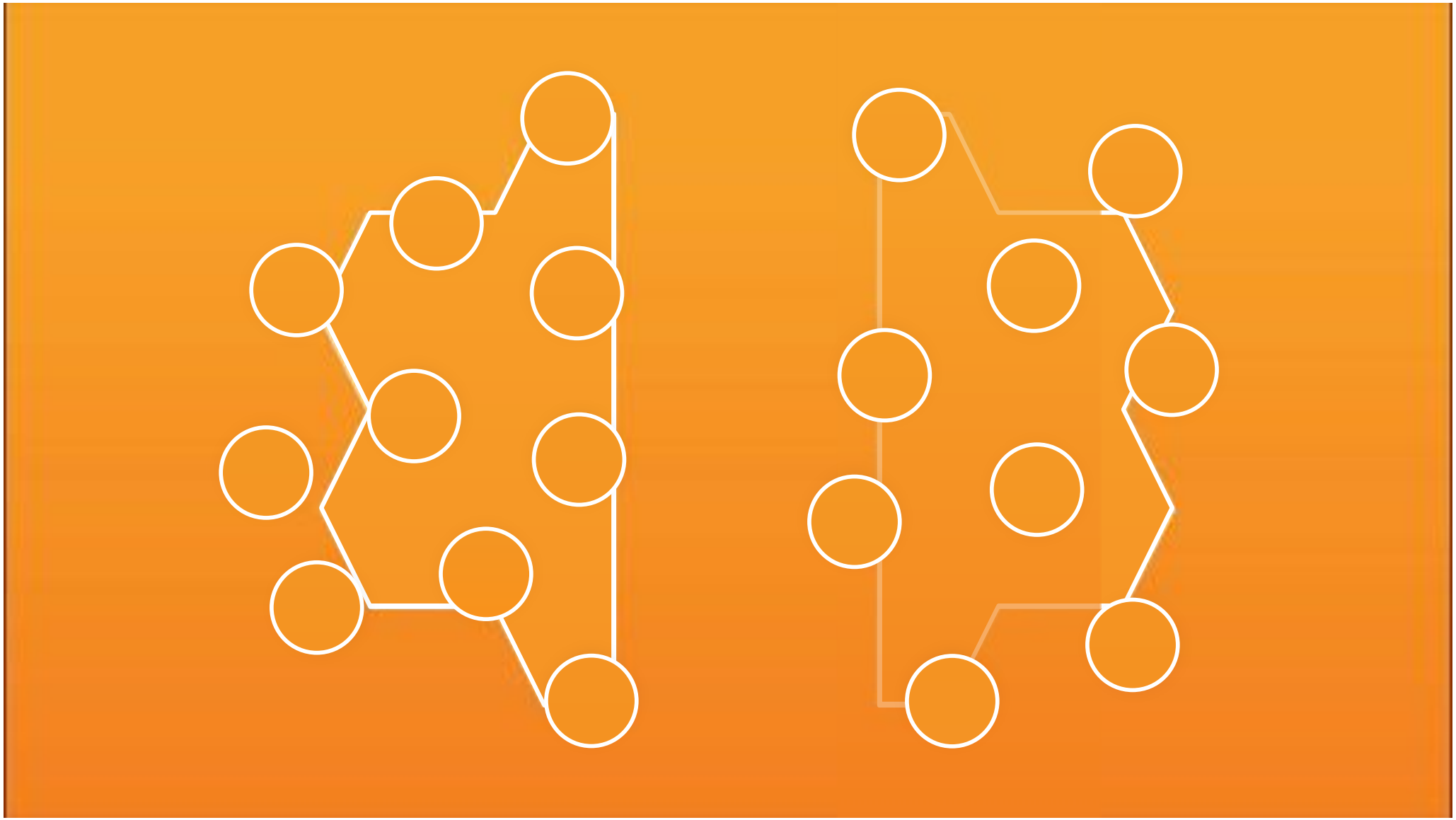
Splitting Monoliths

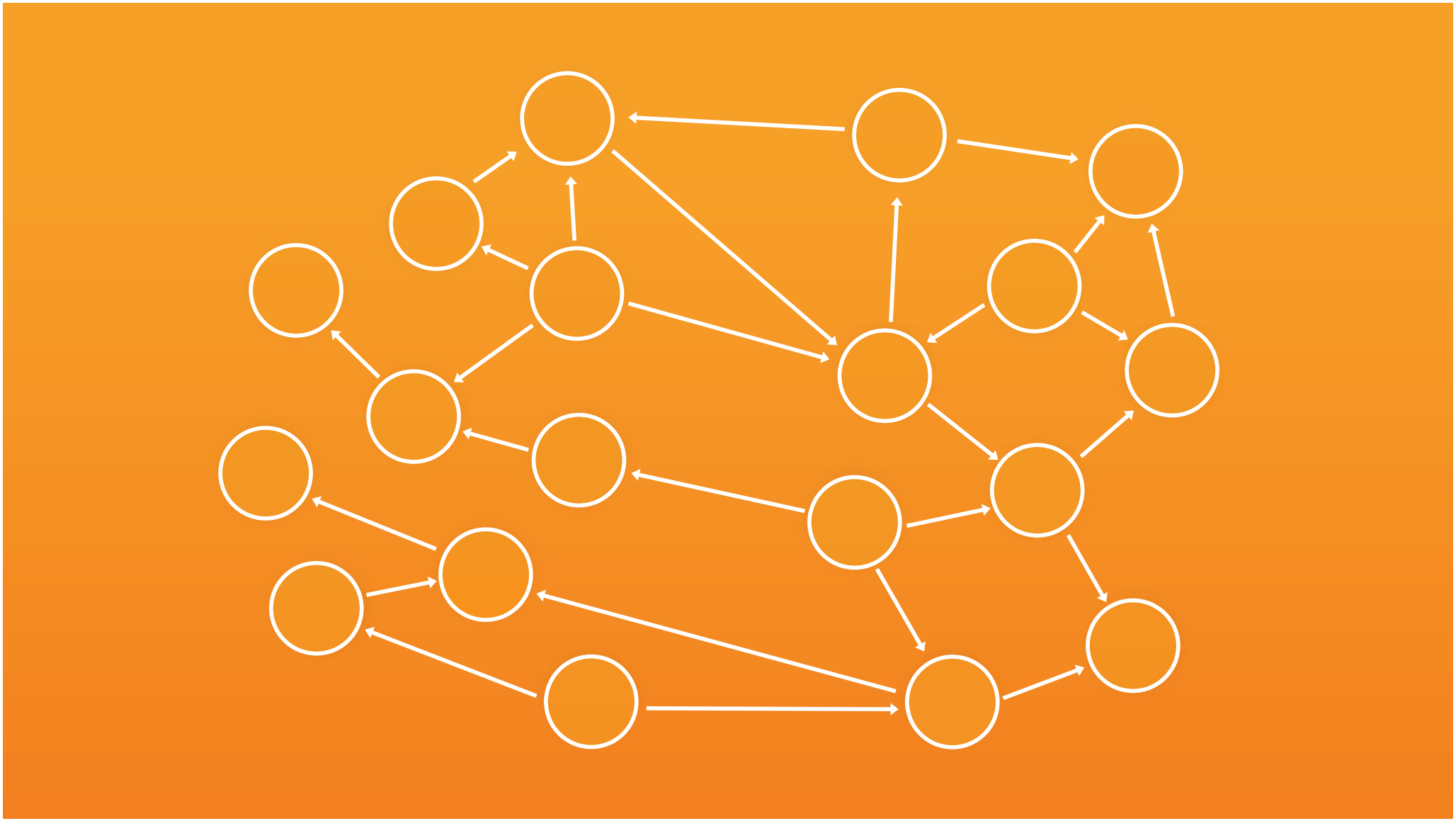


Splitting Monoliths









Microservices

Microservices to Functions

Standard building brick services provide standardized platform capabilities

Amazon API Gateway

Amazon SQS

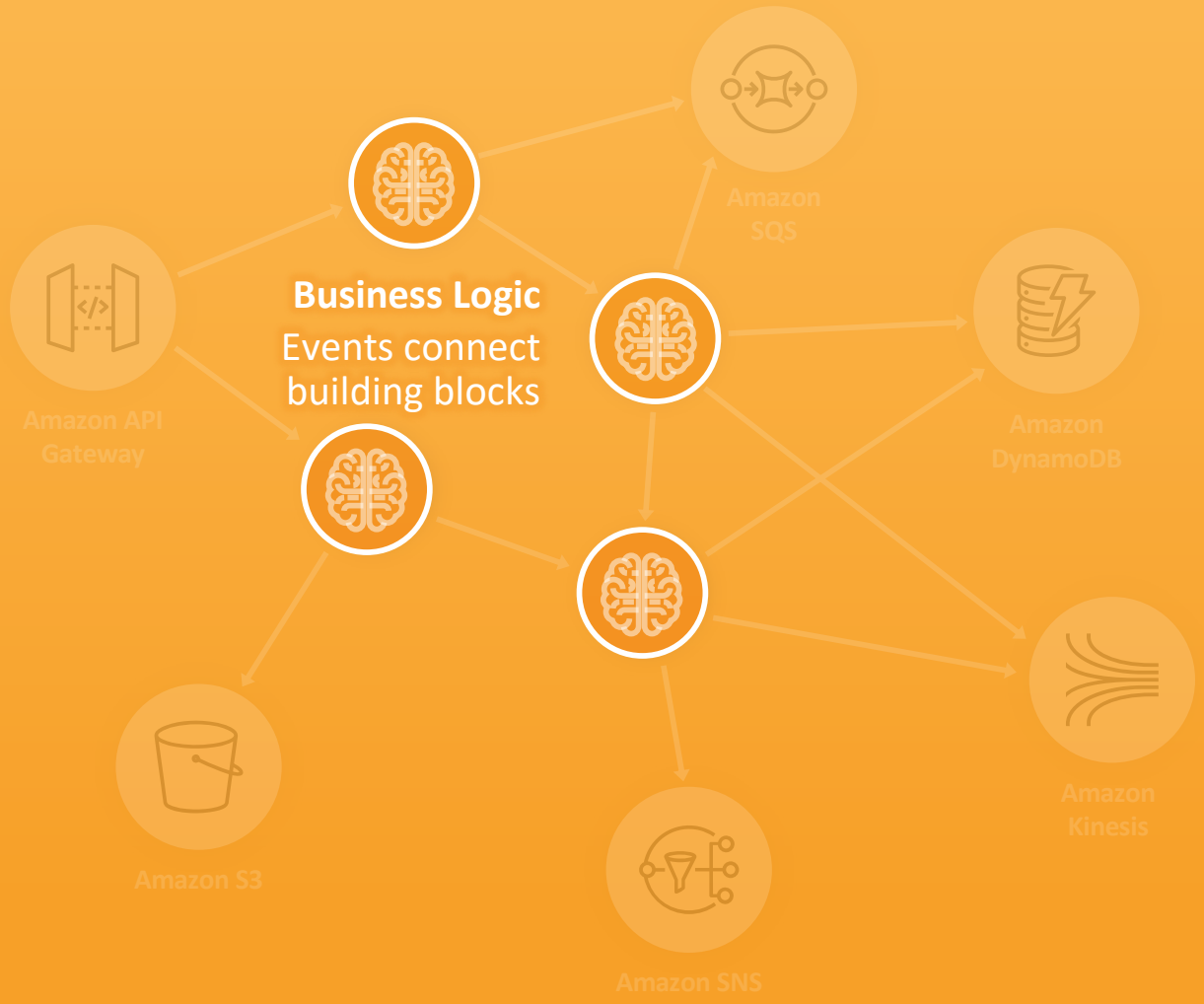
Amazon DynamoDB

Amazon S3

Amazon SNS

Amazon Kinesis

Microservices to Functions



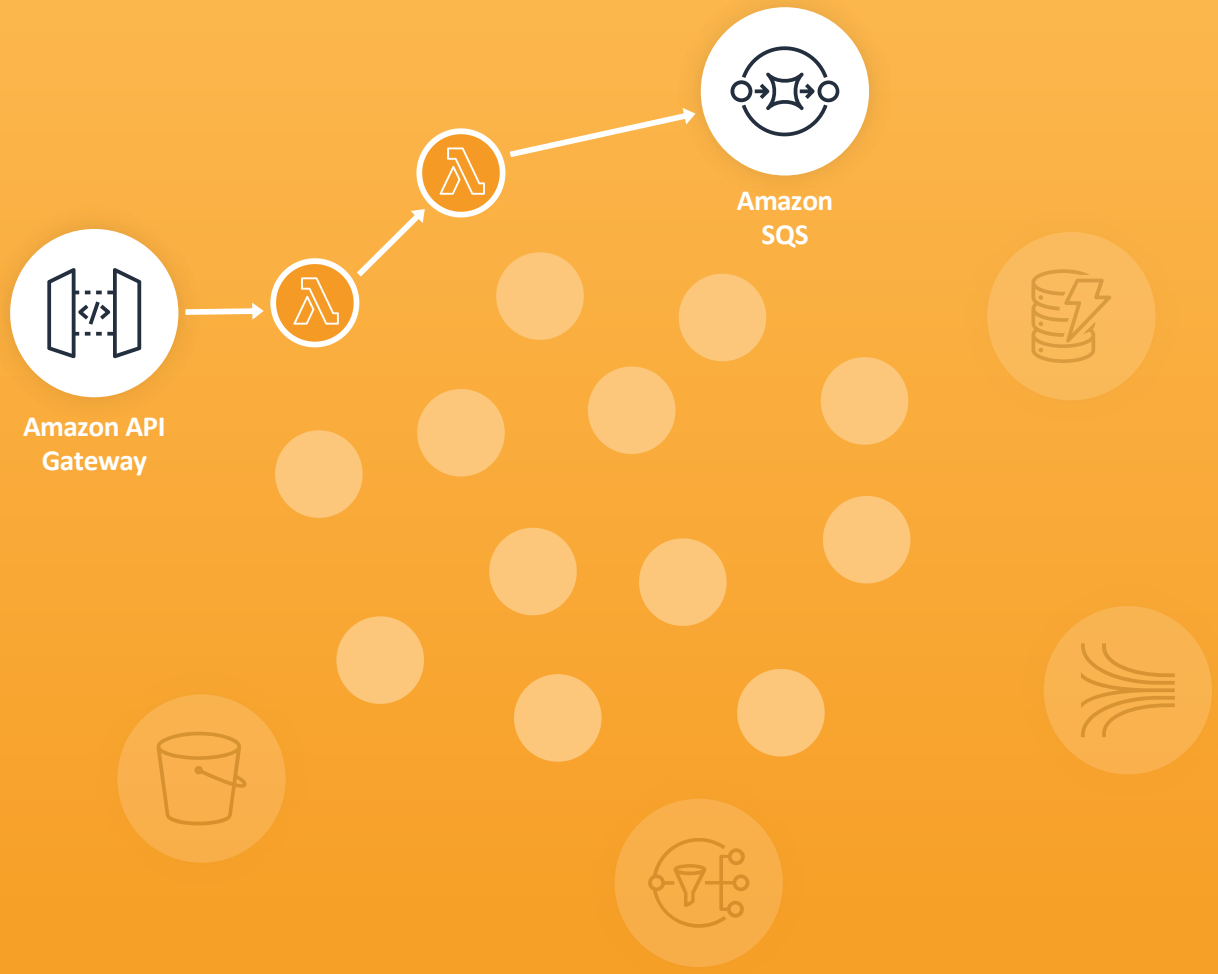
Microservices to Functions

Microservices to Functions

Microservices to Ephemeral

Microservices to Ephemeral Functions

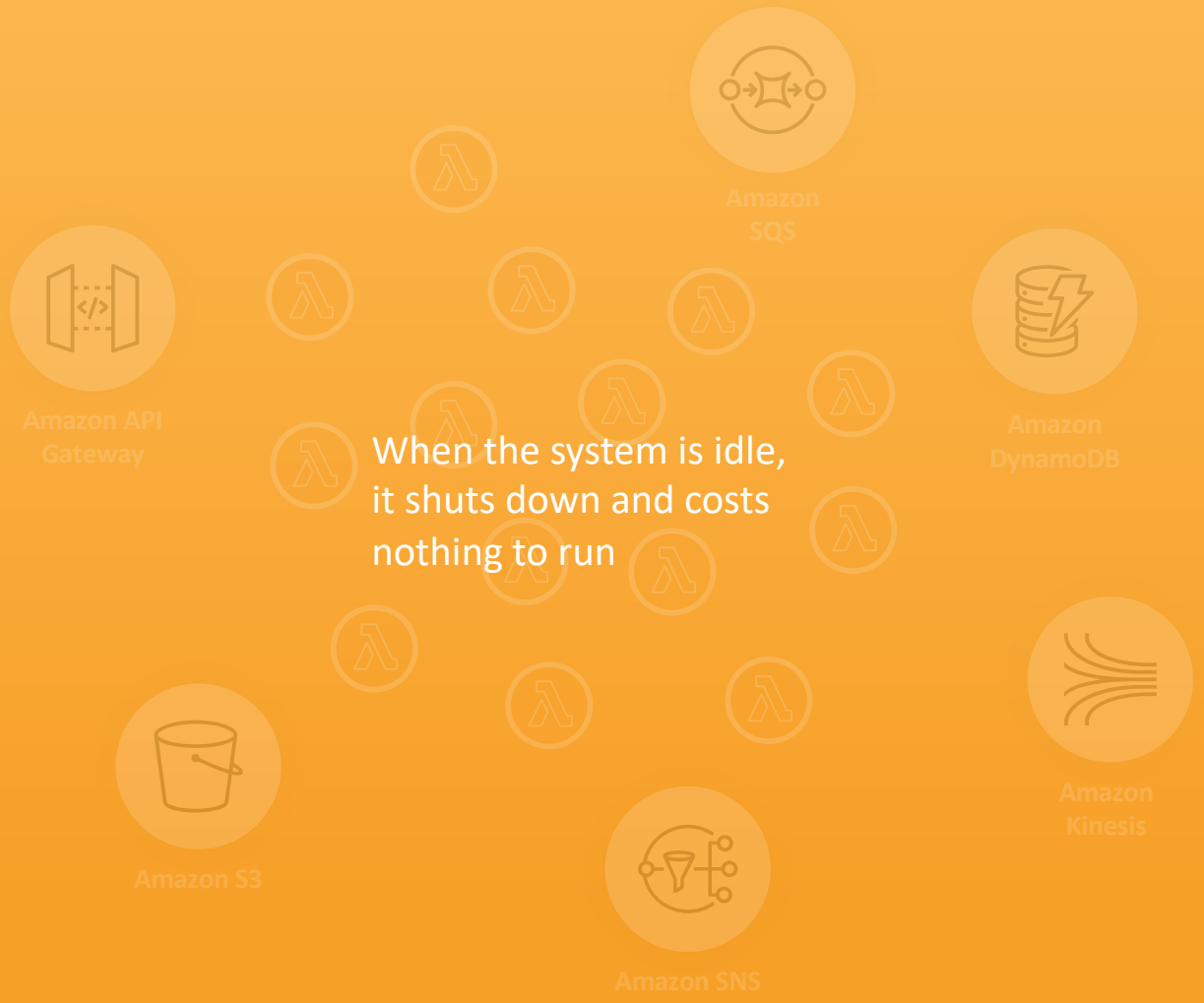
Microservices to Ephemeral Functions



Microservices to Ephemeral Functions

Microservices to Ephemeral Functions

Microservices to Ephemeral Functions

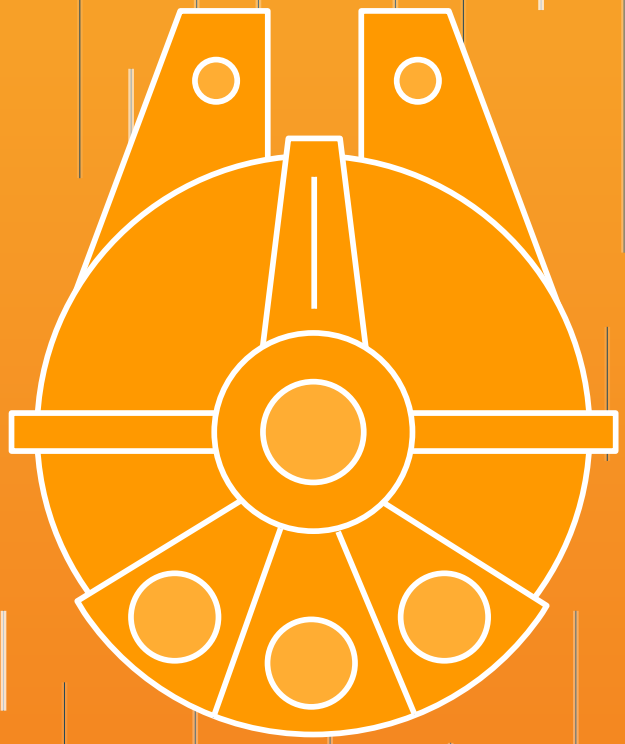


When the system is idle, it shuts down and costs nothing to run

So WHY is it so fast to write a serverless app?

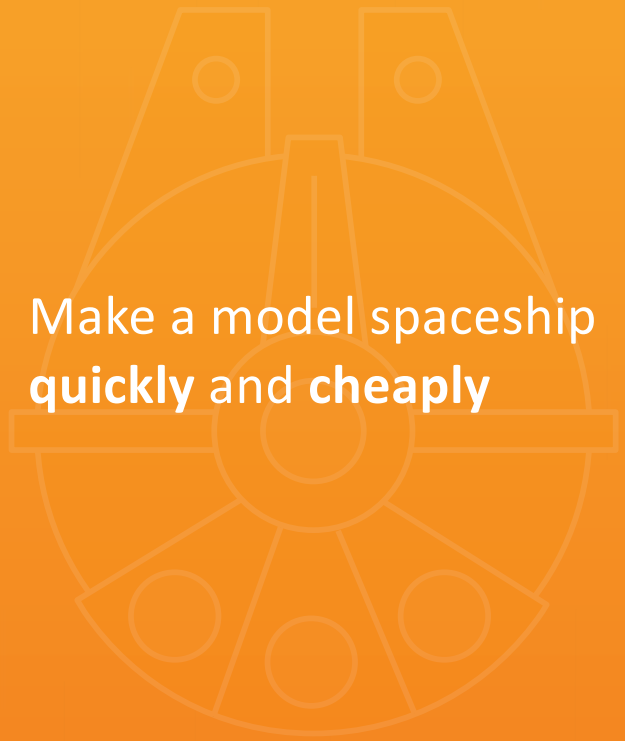
An analogy…

What is the user need?

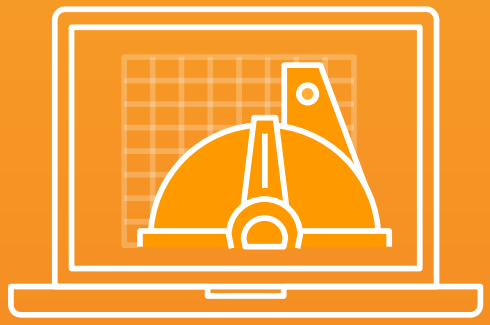


What is the problem you are trying to solve?

→ Make a model spaceship **quickly** and **cheaply**



Traditional Development

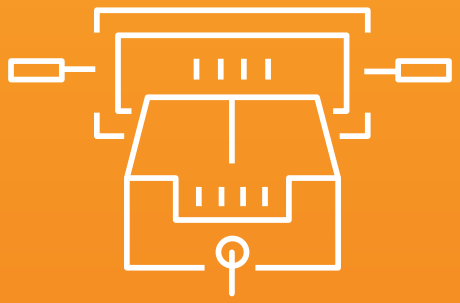


Design a prototype

Traditional Development

Carve from
modelling clay

Traditional Development



Make molds

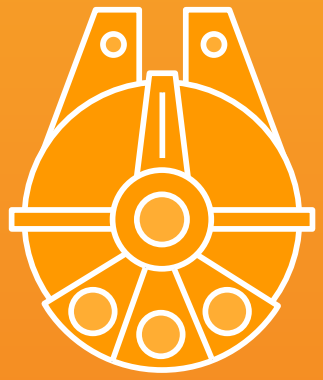
Traditional Development

Produce injection molded parts

Traditional Development

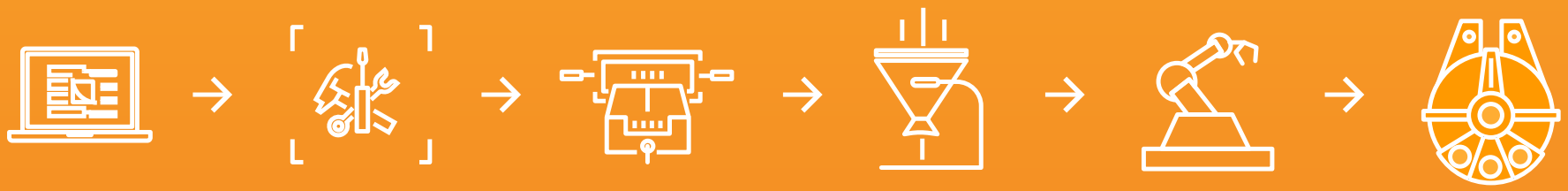
Assemble parts

Traditional Development



Sell finished toy

Traditional Development



Design a prototype

Carve from modelling clay

Make molds

Produce injection molded parts

Assemble parts

Sell finished toy

Rapid Development

Big bag of blocks

+

Instructions

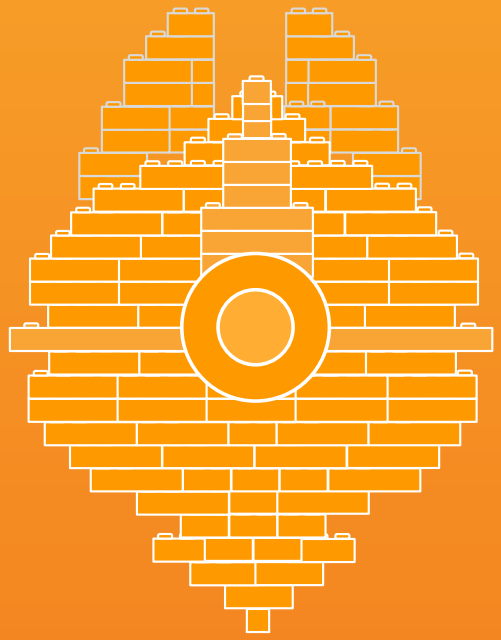
+

A few hours

Rapid Development

A finished toy

Rapid Development

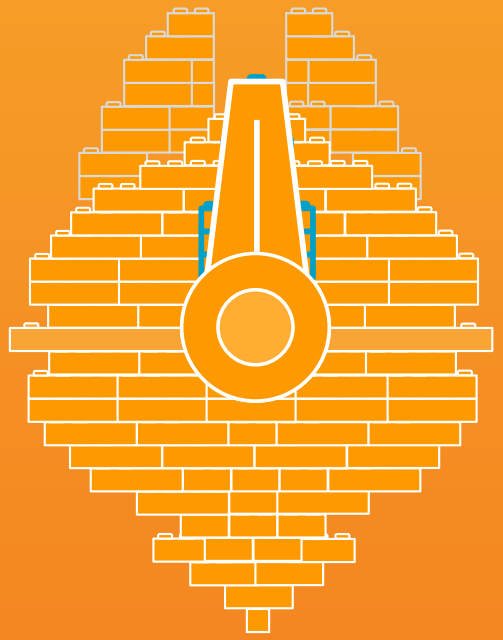


Lacks fine detail

Recognizable, but not exactly what was asked for

Easy to modify and extend

Optimization



Take a group of Lego bricks…

…and form a new custom brick

A more specialized common component

Traditional	Rapid Development
Full custom design	Building blocks assembly
Months of work	Hours of work
Custom components may be fragile and need to be debugged and integrated	Standard reliable components scale and are well understood and interoperable
Too many detailed choices	Need to adjust requirements to fit the patterns available
Long decision cycles	Constraints tend to reduce debate and speed up decisions

Containers	Serverless
Custom code and services	Serverless events and functions
Lots of choices of frameworks and API mechanisms	Standardized choices
Where needed, optimize serverless applications by also building services using containers to solve for anything serverless doesn't do well… yet.	Combine building blocks including: λ AWS Lambda API Gateway, EventBridge Amazon SNS, SQS Amazon DynamoDB AWS Step Functions

So… why doesn’t everyone use serverless first?

Objections and limitations

Note: See Re:Invent 2019 SVS343

Objections Summary

There are answers to all of these…

Patterns, Portability

Language support

Scalability, Resilience

Startup and network latency

Databases/storage interfacing

Too hard to get started

Security

State handling, event processing

Limited run duration

Complex configs

Adrian retired from Amazon in the middle of 2022 and is now working as an advisor, analyst and consultant via OrionX.net


DevSusOps
Adding sustainability concerns to development and operations
Adrian Cockcroft - OrionX.net
@adrianco - @DevSusOps - Oct 2022

Leave the world habitable for future generations

Market transition risks

Regulatory compliance

Physical risks to business assets

Why does sustainability matter?

“Green” market positioning

Employee enthusiasm

Reduced costs now or in the future

Social license to operate

What can we do about it?

Development

Optimize code

Choose faster languages and runtimes

Efficient algorithms

Faster implementations

Reduce logging

Reduce retries and work amplification

Operations

Higher utilization

Automation

Relax over-specified requirements

Archive and delete data sooner

Deduplicate data

Choose times and locations carefully

For workload optimization we need directional and proportional guidance:

Cloud Carbon Footprint tool - Open source, uses billing data as input
Maintains a set of reasonable estimates/guesses for carbon factors
https://www.cloudcarbonfootprint.org

Green Software Foundation Software Carbon Intensity - SCI
A model for reporting software impact per business operation
https://greensoftware.foundation/projects/

AWS Well Architected Pillar for Sustainability
Guidance on how to optimize development and operations for carbon
https://docs.aws.amazon.com/wellarchitected/latest/sustainability-pillar/sustainability-pillar.html

Where is all this going to be in a few years?

Monitoring tools will report carbon

Cloud providers will (all eventually) have detailed metrics

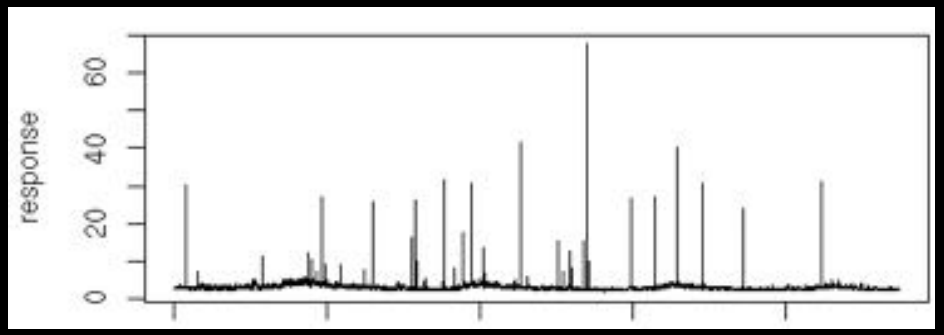
EU and US cloud regions are close to zero carbon now
Asian regions will move to zero carbon by 2025
(All providers have the same problem with regional policies)

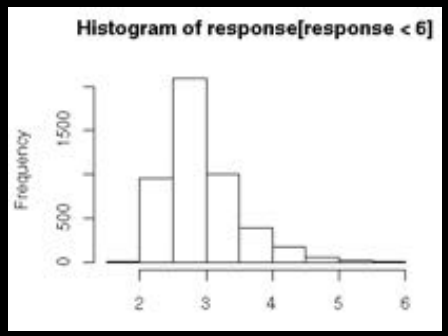

That’s all folks!
Unless there’s time for an encore
and some drinks?

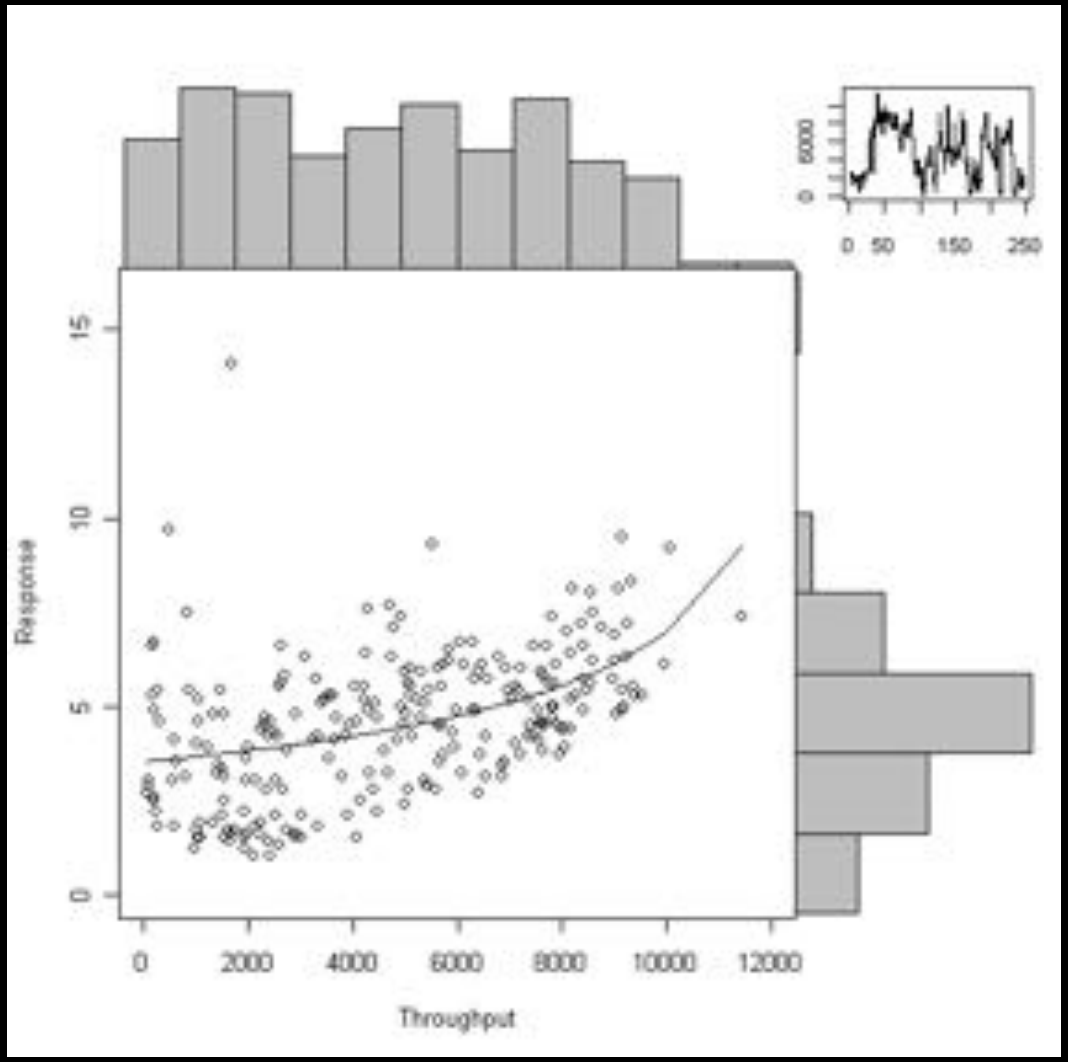

Bottleneck Analysis
July 2013
Adrian Cockcroft
@adrianco

Analysis

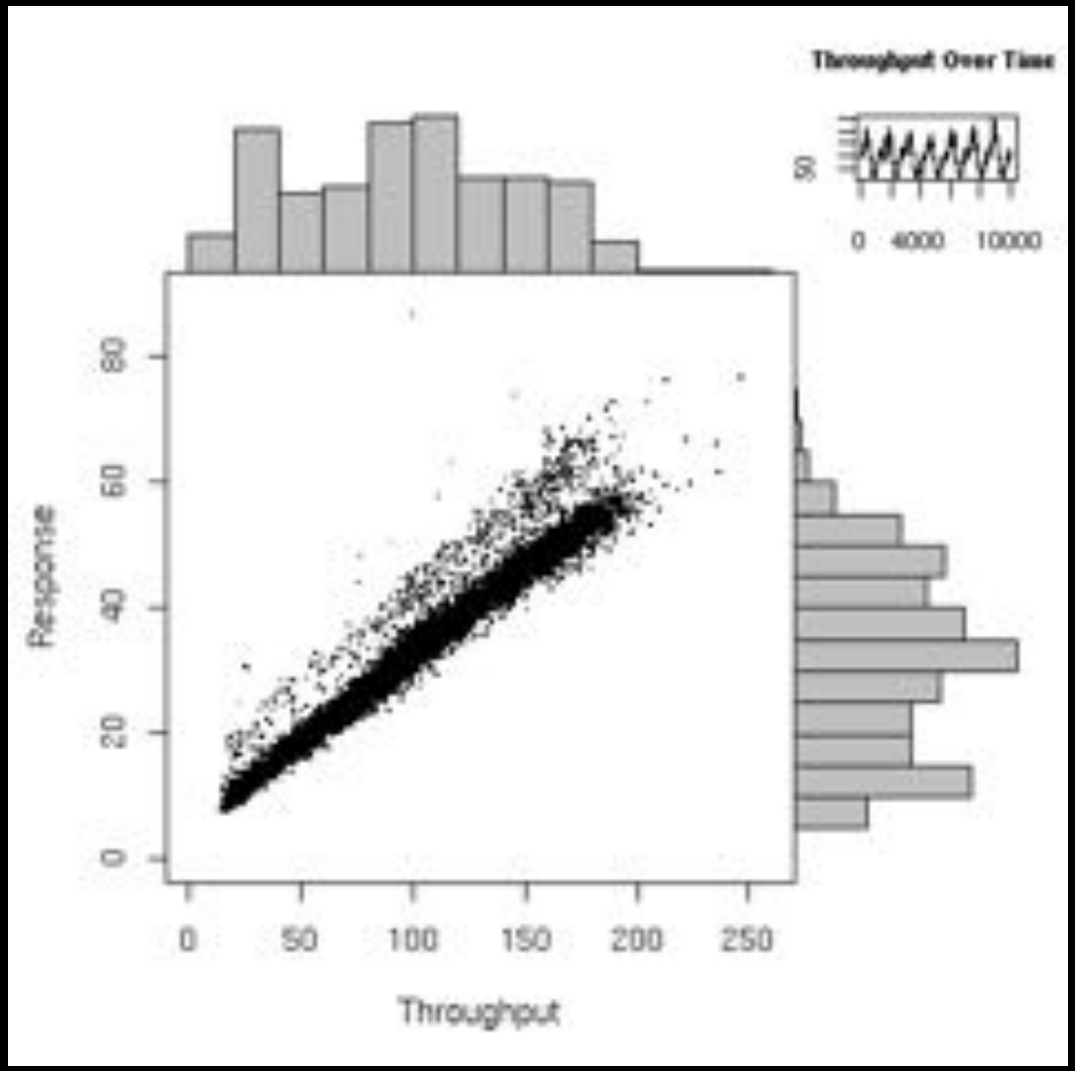
```
> summary(response)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.909   2.550   2.820   3.086   3.214  67.680
> quantile(response,c(0.95,0.99))
     95%      99%
4.149556 6.922115
> sd(response)
 1.941328
> mean(response) + 2 * sd(response)
 6.968416
```



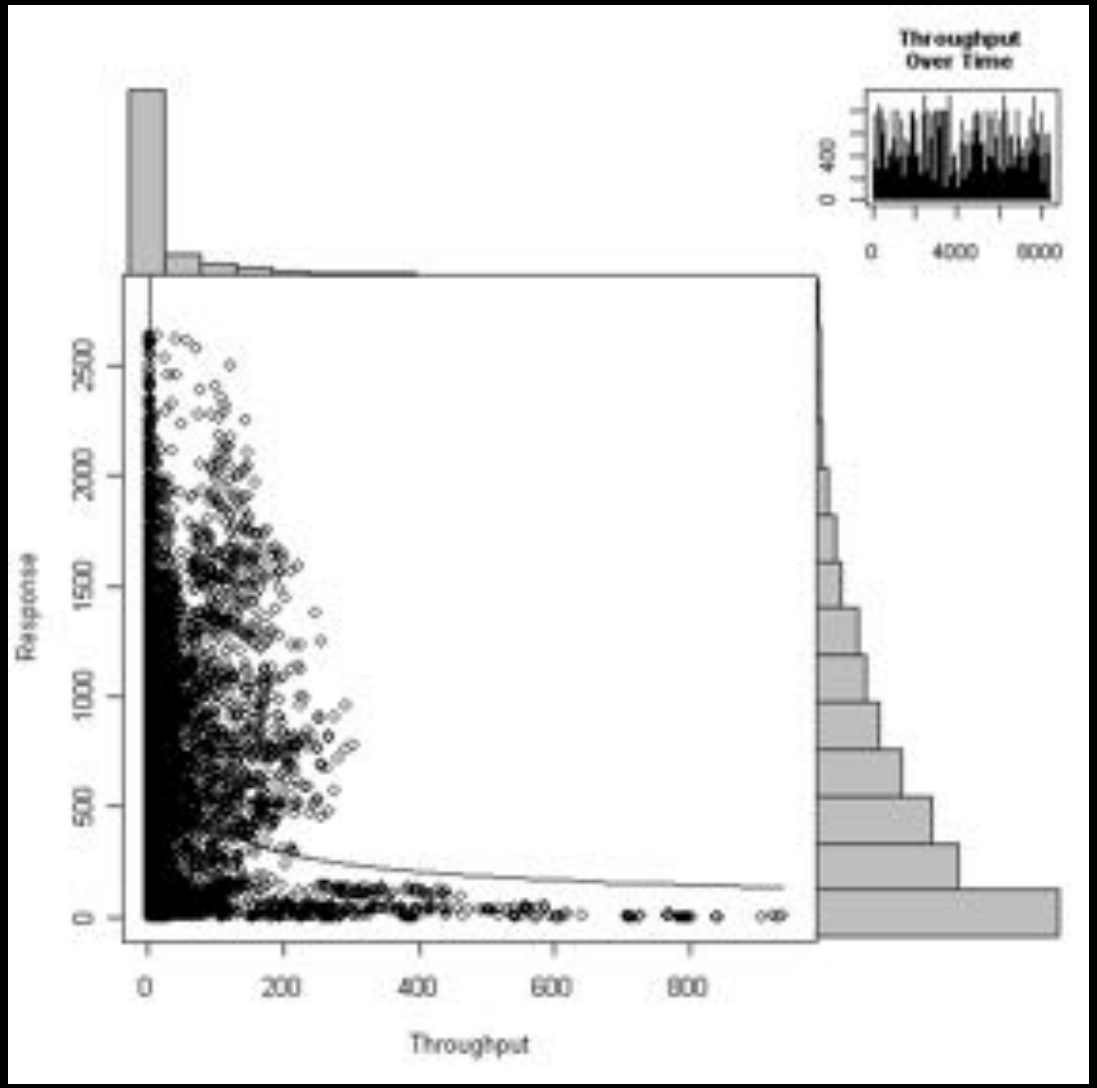


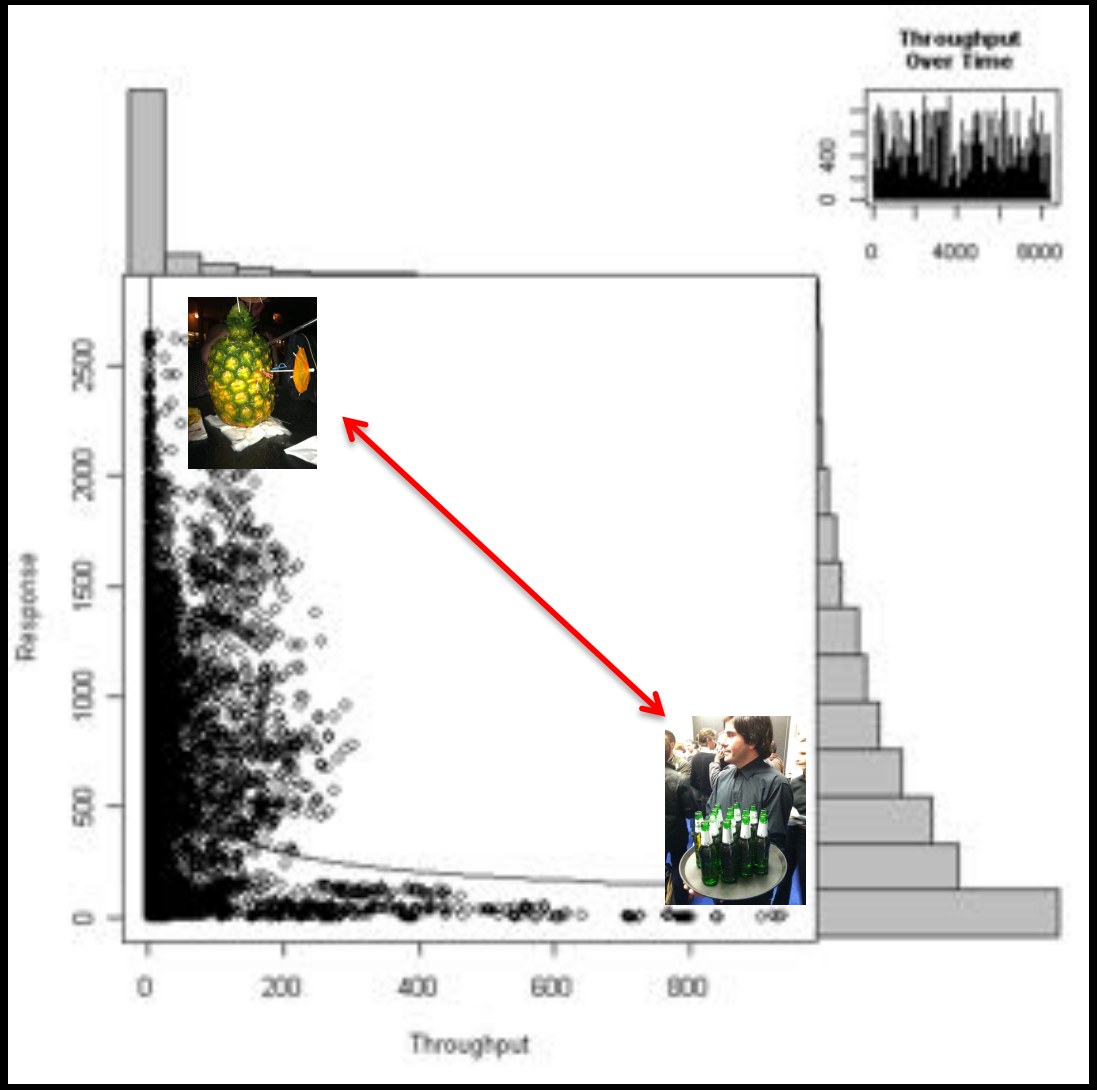


https://github.com/adrianco/chp

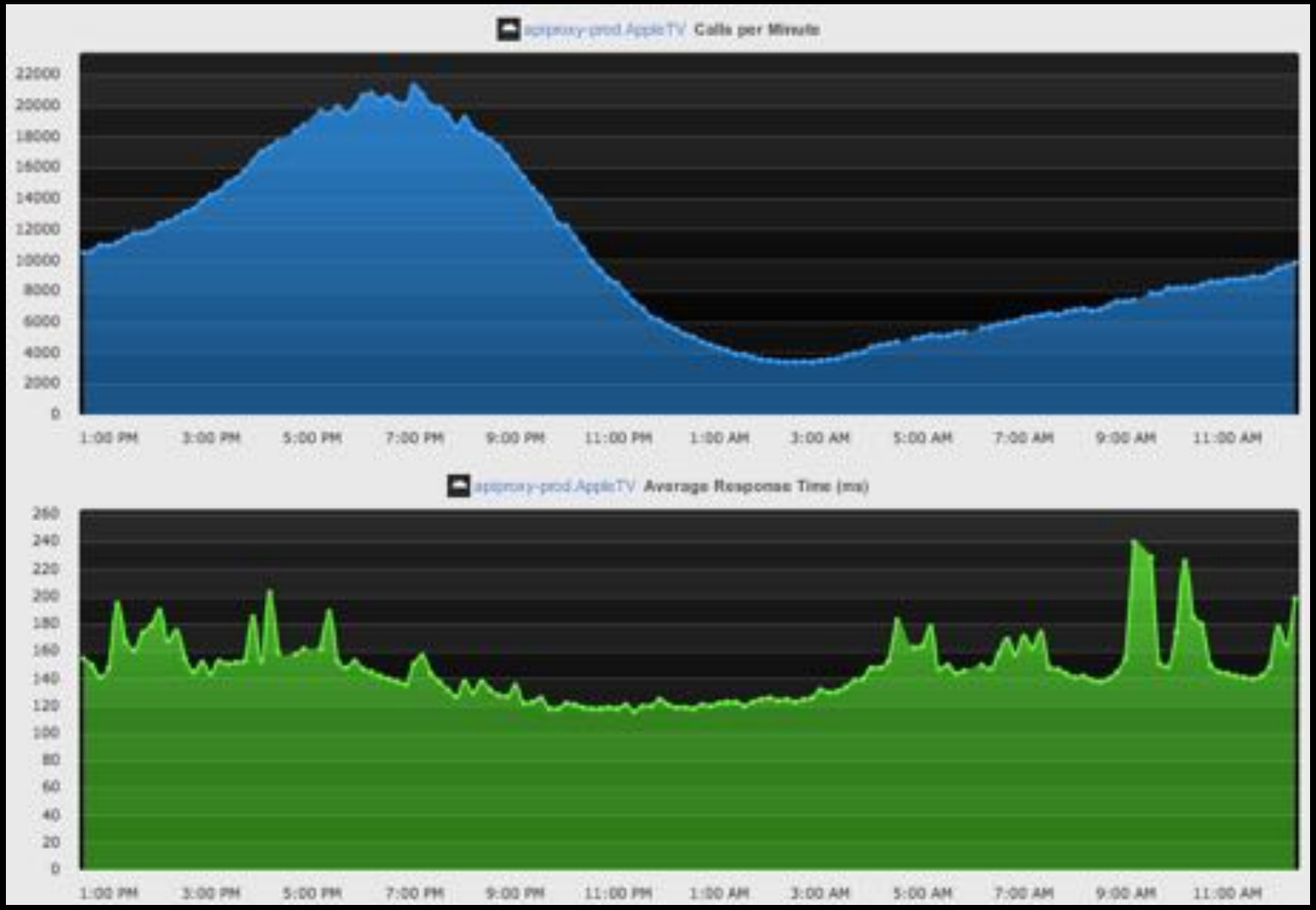

Throughput Over Time
50
0
4000
10000
Response
0
20
40
60
80
0
50
100
150
200
250
Throughput

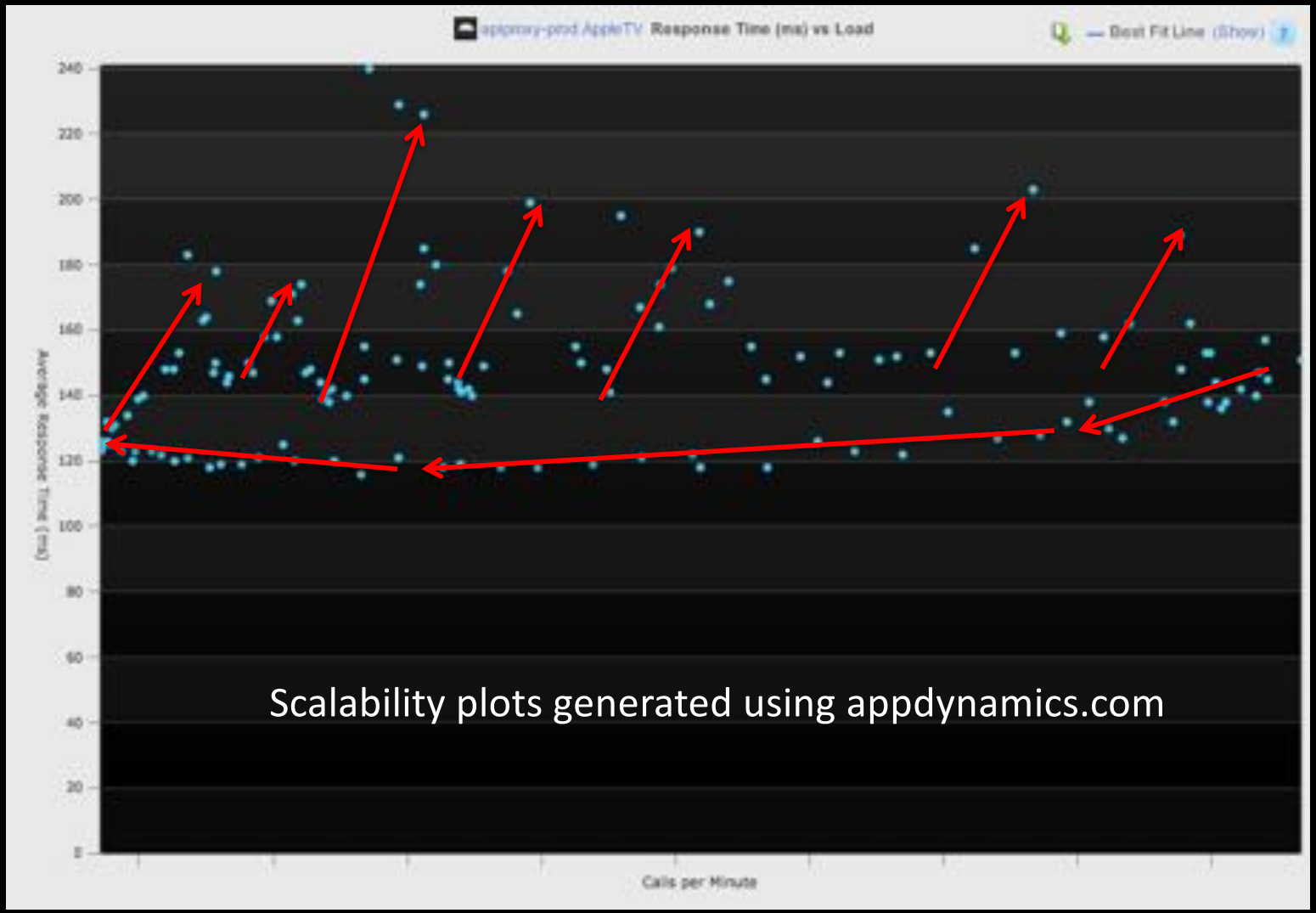
Pineapple Dalek says
INEBRIATE!
INEBRIATE!


Throughput
Over Time
0
400
0
4000
8000
Response
0
500
1000
1500
2000
2500
0
200
400
600
800
Throughput

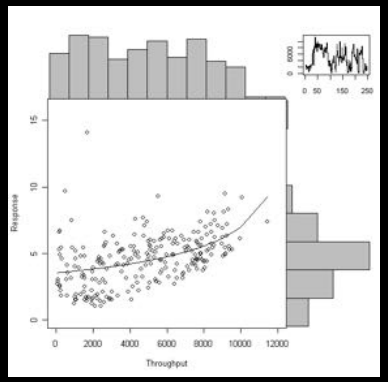

Throughput
Over Time
0
400
0
4000
8000
Response
0
500
1000
1500
2000
2500
0
200
400
600
800
Throughput


KONA
BREWING CO.

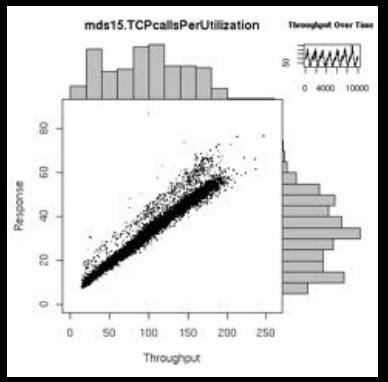

apiproxy-prod.AppleTV Calls per Minute
22000
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1:00 PM
3:00 PM
5:00 PM
7:00 PM
9:00 PM
11:00 PM
1:00 AM
3:00 AM
5:00 AM
7:00 AM
9:00 AM
11:00 AM
apiproxy-prod.AppleTV Average Response Time (ms)
260
240
220
200
180
160
140
120
100
80
60
40
20
0
1:00 PM
3:00 PM
5:00 PM
7:00 PM
9:00 PM
11:00 PM
1:00 AM
3:00 AM
5:00 AM
7:00 AM
9:00 AM
11:00 AM


appdemy-prod:AppleTV Response Time (ms) vs Load
— Best Fit Line (Show)
240
220
200
180
160
140
120
100
80
60
40
20
0
Average Response Time (ms)
Calls per Minute
Scalability plots generated using appdynamics.com

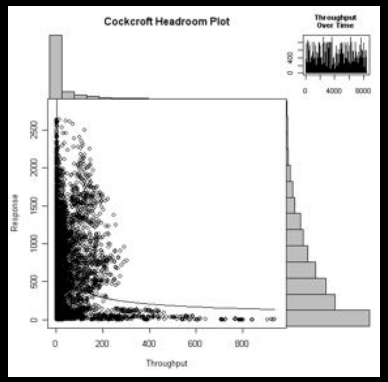
Bottlenecks



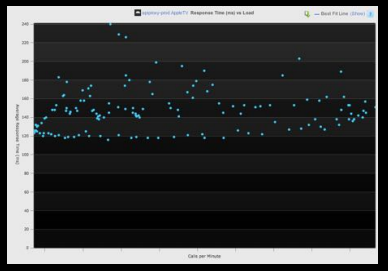
Well behaved



Lock Contention



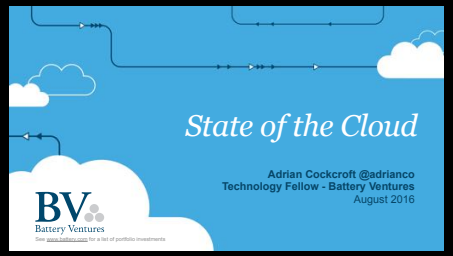
Oscillating, thread shortage

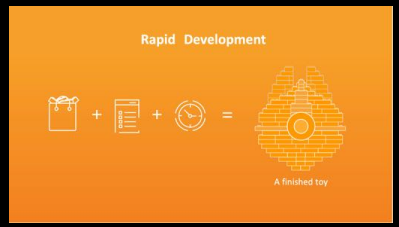


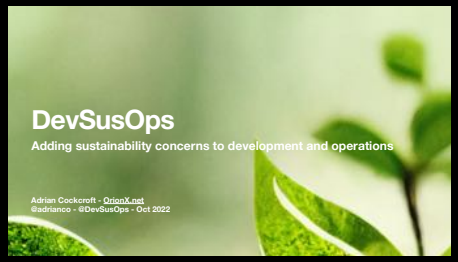
Looping autoscaled

https://github.com/adrianco/chp
Cockcroft Headroom Plot

https://soundcloud.com/adrian-cockcroft/black-tiger-dont-look-back
https://www.slideshare.net/adrianco - Netflix era decks
https://www.slideshare.net/adriancockcroft - Battery Ventures decks
https://github.com/adrianco/slides - AWS decks and later including a pdf of these slides!
https://www.youtube.com/@adriancockcroft








Adrian's Greatest Hits, B-Sides and Re-issues
Thanks! Any questions?
Adrian Cockcroft | YOW | December 2022
@adrianco@mastodon.social