Large-Scale Systems
The Unreasonable Effectiveness of Simplicity

Randy Shoup
@randyshoup

Background

ebay

Google

STITCH FIX

wework

@randyshoup

Goals

- From a systems perspective, what characterizes a scalable, well-engineered system?
- What can (application) systems designers learn from this community?
- Where are the biggest opportunities to improve (application) systems?

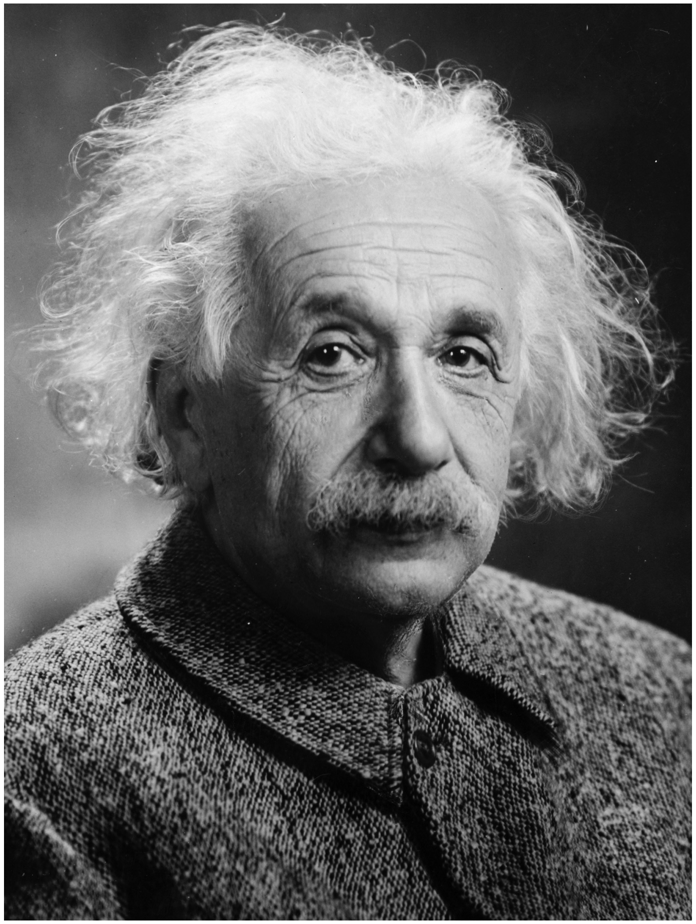
Evolving Systems

- eBay
  - 5th generation today
  - Monolithic Perl → Monolithic C++ → Java → microservices
- Twitter
  - 3rd generation today
  - Monolithic Rails → JS / Rails / Scala → microservices
- Amazon
  - Nth generation today
  - Monolithic Perl / C → C++ / Java services → microservices

No one starts with microservices

…

Past a certain scale, everyone ends up with microservices



"Make everything as simple as possible, but not simpler."

Large-Scale Systems

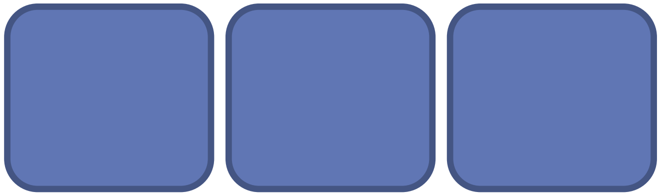
- Simple Components
- Simple Interactions
- Simple Changes

Large-Scale Systems

- Simple Components
- Simple Interactions
- Simple Changes

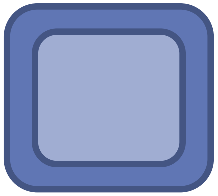
Modular Services

- Service boundaries match the problem domain
- Service boundaries encapsulate business logic and data
  - All interactions through published service interface
  - Interface hides internal implementation details
  - No back doors
- Service boundaries encapsulate architectural -ilities
  - Fault isolation
  - Performance optimization
  - Security boundary



Orthogonal Domain Logic

- Stateless domain logic
  - Ideally stateless pure function
  - Matches domain problem as directly as possible
  - Deterministic and testable in isolation
  - Robust to change over time



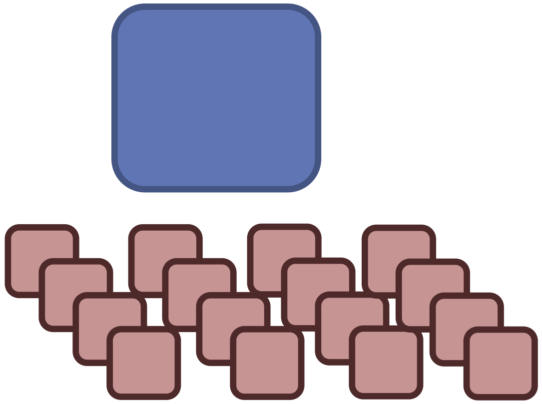
- "Straight-line processing"
  - Straightforward, synchronous, minimal branching
- Separate domain logic from I/O
  - Hexagonal architecture, Ports and Adapters
  - Functional core, imperative shell

Sharding

- Shards partition the service's "data space"
  - Units for distribution, replication, processing, storage
  - Hidden as internal implementation detail

- Shards encapsulate architectural -ilities
  - Resource isolation
  - Fault isolation
  - Availability
  - Performance

- Shards are autoscaled
  - Divide or scale out as processing or data needs increase
  - E.g., DynamoDB partitions, Aurora segments, Bigtable tablets

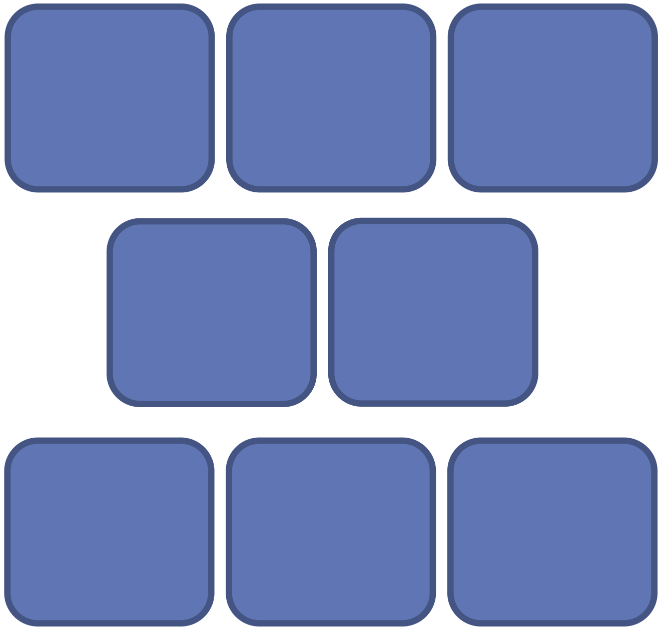


Service Layering

- Common services provide and abstract widely-used capabilities

- Service ecosystem
  - Services call others, which call others, etc.
  - Graph, not a strict layering

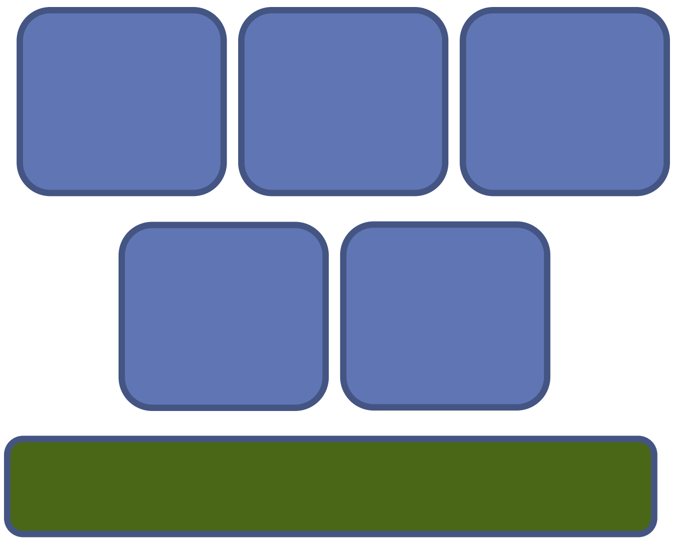
- Services grow and evolve over time
  - Factor out common libraries and services as needed
  - Teams and services split like "cellular mitosis"



Common Platform

- "Paved Road"
  - Shared infrastructure
  - Standard frameworks
  - Developer experience
  - E.g., Netflix, Google

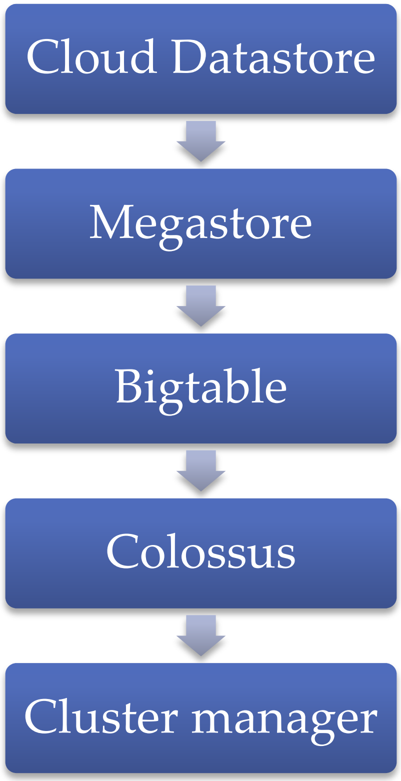
- Separation of Concerns
  - Reduce cognitive load on stream-aligned teams
  - Bound decisions through enabling constraints



Large-scale organizations often invest more than 50% of engineering effort in platform capabilities

Google Service Layering (2013)

- Cloud Datastore: NoSQL service
  - Strong transactional consistency
  - SQL-like rich query capabilities
- Megastore: geo-scale structured database
  - Multi-row transactions
  - Synchronous cross-datacenter replication
- Bigtable: cluster-level structured storage
  - (row, column, timestamp) -> cell contents
- Colossus: distributed file system
  - Block distribution and replication
- Borg: cluster management infrastructure
  - Task scheduling, machine assignment



Large-Scale Systems

- Simple Components
- Simple Interactions
- Simple Changes

Reactive Manifesto

The Reactive Manifesto

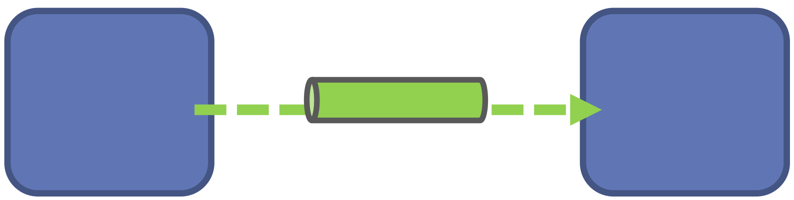
Published on September 16 2014. (v2.0)

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Event-Driven

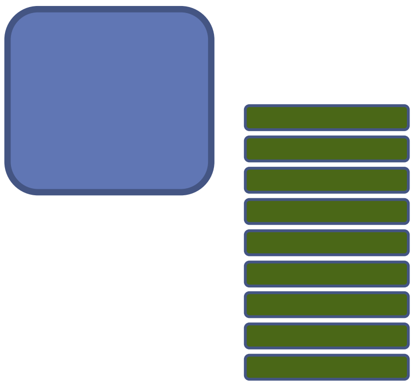


- Communicate state changes as stream of events
  - Statement that some interesting thing occurred
  - Ideally represents a semantic domain event

- Decouples domains and teams
  - Abstracted through a well-defined interface
  - Asynchronous from one another

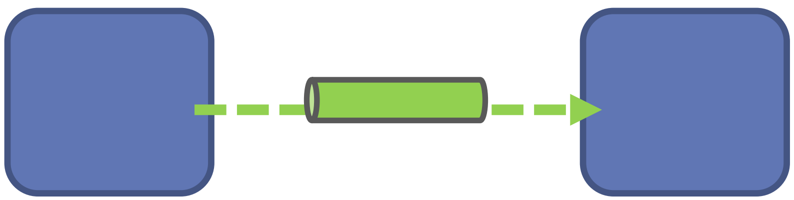
- Simplifies component implementation

Immutable Log



- Store state as immutable log of events
  - Event Sourcing
- Often matches domain
  - E.g., Stitch Fix package processing / delivery state
- Log encapsulates architectural –ilities
  - Durable
  - Traceable and auditable
  - Replayable
  - Explicit and comprehensible
- Compact snapshots for efficiency

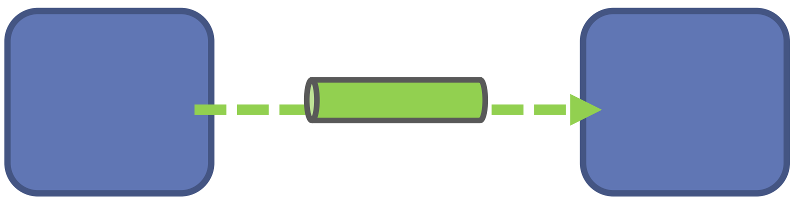
Embrace Asynchrony



- Decouples operations in time
  - Decoupled availability
  - Independent scalability
  - Allows more complex processing, more processing in parallel
  - Safer to make independent changes

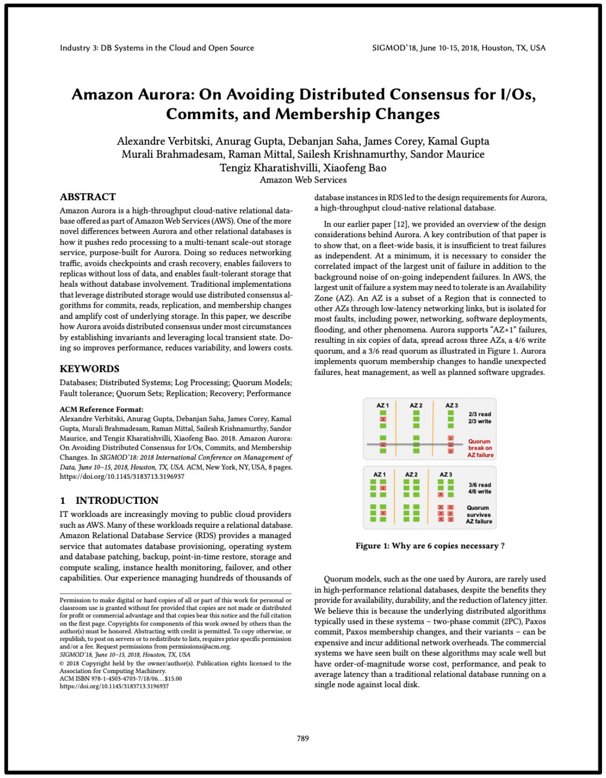
- Simplifies component implementation

Embrace Asynchrony



- Invert from synchronous call graph to async dataflow
  - Exploit asymmetry between writes and reads
  - Can be orders of magnitude less resource intensive

Amazon Aurora



Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta
Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice
Tengiz Kharatishvilli, Xiaofeng Bao
Amazon Web Services

ABSTRACT

Amazon Aurora is a high-throughput cloud-native relational database offered as part of Amazon Web Services (AWS). One of the more novel differences between Aurora and other relational databases is how it pushes redo processing to a multi-tenant scale-out storage service, purpose-built for Aurora. Doing so reduces networking traffic, avoids checkpoints and crash recovery, enables failovers to replicas without loss of data, and enables fault-tolerant storage that heals without database involvement. Traditional implementations that leverage distributed storage would use distributed consensus algorithms for commits, reads, replication, and membership changes and amplify cost of underlying storage. In this paper, we describe how Aurora avoids distributed consensus under most circumstances by establishing invariants and leveraging local transient state. Doing so improves performance, reduces variability, and lowers costs.

KEYWORDS

Databases; Distributed Systems; Log Processing; Quorum Models; Fault tolerance; Quorum Sets; Replication; Recovery; Performance

ACM Reference Format:
Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, and Tengiz Kharatishvilli, Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3183713.3196937

1 INTRODUCTION

IT workloads are increasingly moving to public cloud providers such as AWS. Many of these workloads require a relational database. Amazon Relational Database Service (RDS) provides a managed service that automates database provisioning, operating system and database patching, backup, point-in-time restore, storage and compute scaling, instance health monitoring, failover, and other capabilities. Our experience managing hundreds of thousands of database instances in RDS led to the design requirements for Aurora, a high-throughput cloud-native relational database.

In our earlier paper [12], we provided an overview of the design considerations behind Aurora. A key contribution of that paper is to show that, on a fleet-wide basis, it is insufficient to treat failures as independent. At a minimum, it is necessary to consider the correlated impact of the largest unit of failure in addition to the background noise of on-going independent failures. In AWS, the largest unit of failure a system may need to tolerate is an Availability Zone (AZ). An AZ is a subset of a Region that is connected to other AZs through low-latency networking links, but is isolated for most faults, including power, networking, software deployments, flooding, and other phenomena. Aurora supports "AZ+1" failures, resulting in six copies of data, spread across three AZs, a 4/6 write quorum, and a 3/6 read quorum as illustrated in Figure 1. Aurora implements quorum membership changes to handle unexpected failures, heat management, as well as planned software upgrades.

Figure 1: Why are 6 copies necessary ?

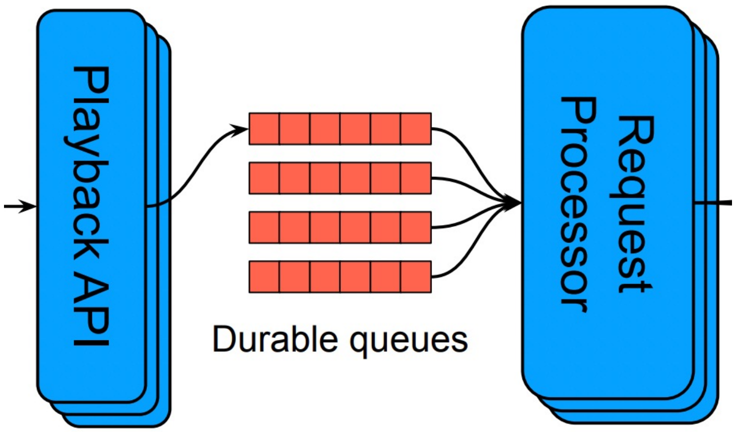
Quorum models, such as the one used by Aurora, are rarely used in high-performance relational databases, despite the benefits they provide for availability, durability, and the reduction of latency jitter. We believe this is because the underlying distributed algorithms typically used in these systems – two-phase commit (2PC), Paxos commit, Paxos membership changes, and their variants – can be expensive and incur additional network overheads. The commercial systems we have seen built on these algorithms may scale well but have order-of-magnitude worse cost, performance, and peak to average latency than a traditional relational database running on a single node against local disk.

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196937

- Asynchronous redo log writes
  - Sent asynchronously to Aurora storage nodes
  - Acknowledged asynchronously to database instance
  - No distributed consensus round
  - Idempotent, immutable, monotonic
- Quorum acknowledgement
  - Log progresses forward once quorum of nodes acknowledges
- Reestablish consistency on crash recovery

Verbitski, et al, 2018, Amazon Aurora: On Avoiding Distributed Consensus, SIGMOD '18.

Netflix Viewing History



- Store and process member's playback data
  - 1M requests per second
  - Used for viewing history, personalization, recommendations, analytics, etc.
- Original synchronous architecture
  - Synchronously write to persistent storage and lookup cache
  - Availability and data loss from backpressure at high load
- Asynchronous rearchitecture
  - Write to durable queue
  - Async pipeline to enrich, process, store, serve
  - Materialize views to serve reads

Sharma Podila, 2021, [Microservices to Async Processing Migration at Scale](), QConPlus 2021.

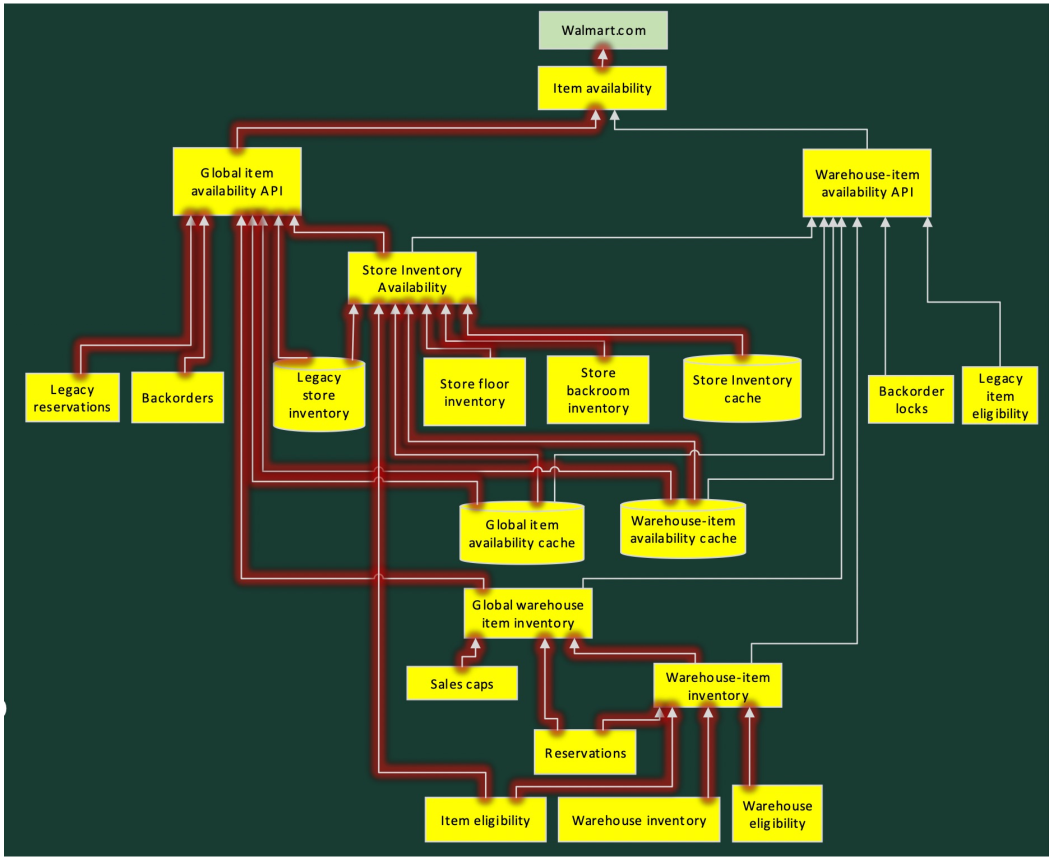
Walmart Item Availability

- Is this item available to ship to this customer?
  - Customer SLO 99.98% uptime in 300ms
- Complex logic involving many teams and domains
  - Inventory, reservations, backorders, eligibility, sales caps, etc.
- Original synchronous architecture
  - Graph of 23 nested synchronous service calls in hot path
  - Any component failure invalidates results
  - Service SLOs 99.999% uptime with 50ms marginal latency
  - Extremely expensive to build and operate

Walmart
Save money. Live better.

Scott Havens, 2019, Fabulous Fortunes, Fewer Failures, and Faster Fixes from Functional Fundamentals, DOES 2019.

Walmart Item Availability



@randyshoup

Scott Havens, 2019, Fabulous Fortunes, Fewer Failures, and Faster Fixes from Functional Fundamentals, DOES 2019.

Walmart Item Availability

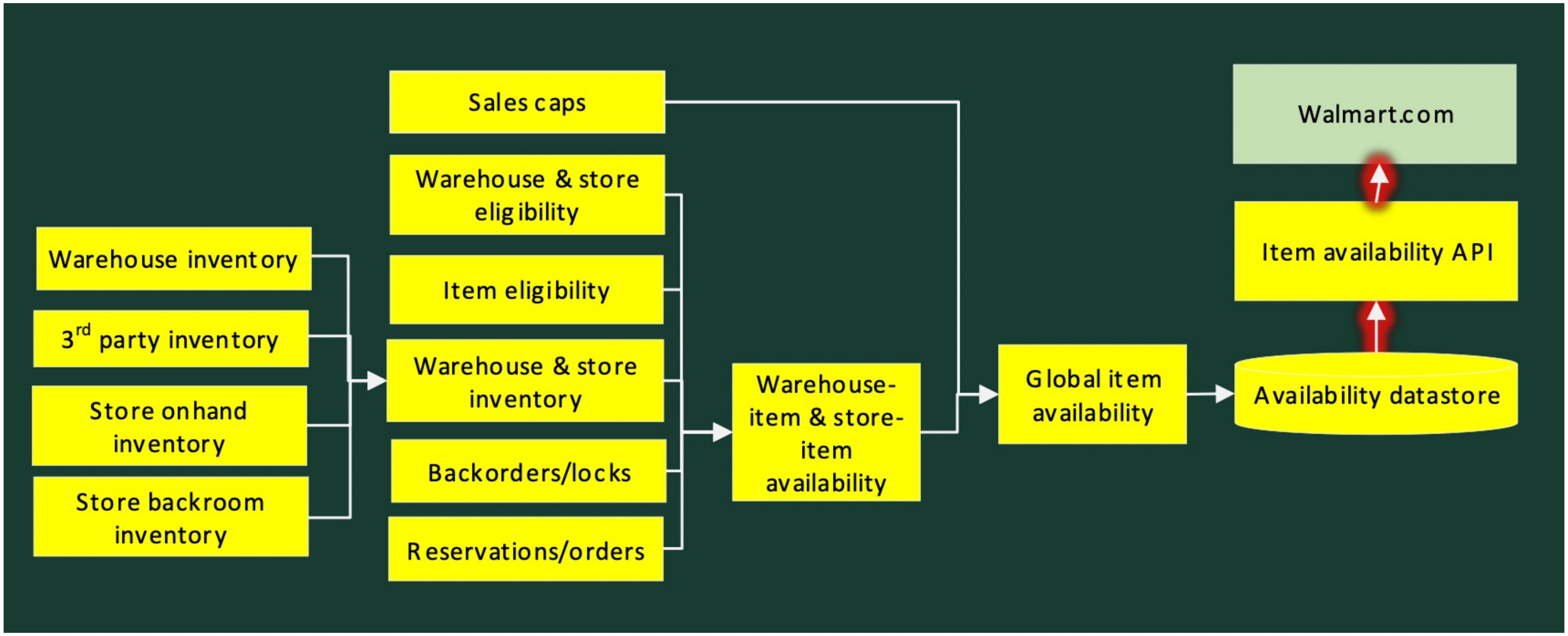
- Invert each service to use async events
  - Event-driven "dataflow"
  - Idempotent processing
  - Event-sourced immutable log
  - Materialized view of data from upstream dependencies

- Asynchronous rearchitecture
  - 2 services in synchronous hot path
  - Async service SLOs 99.9% uptime with latency in seconds or minutes
  - More resilient to delays and outages
  - Orders of magnitude simpler to build and operate

Walmart
Save money. Live better.

Scott Havens, 2019, Fabulous Fortunes, Fewer Failures, and Faster Fixes from Functional Fundamentals, DOES 2019.

Walmart Item Availability



Scott Havens, 2019, [Fabulous Fortunes, Fewer Failures, and Faster Fixes from Functional Fundamentals](), DOES 2019.

Large-Scale Systems

- Simple Components
- Simple Interactions
- Simple Changes

Incremental Change

- Decompose every large change into small incremental steps
- Each step maintains backward / forward compatibility of data and interfaces
- Multiple service versions commonly coexist
  - Every change is a rolling upgrade
  - *Transitional states are normal, not exceptional*

Continuous Testing

- Tests help us go faster
  - Tests are "solid ground"
  - Tests are the safety net
- Tests make better code
  - Confidence to break things
  - Courage to refactor mercilessly
- Tests make better systems
  - Catch bugs earlier, fail faster

Continuous Testing

- Tests make better designs
  - Modularity
  - Separation of Concerns
  - Encapsulation

"There's a deep synergy between testability and good design. All of the pain that we feel when writing unit tests points at underlying design problems."

-- Michael Feathers

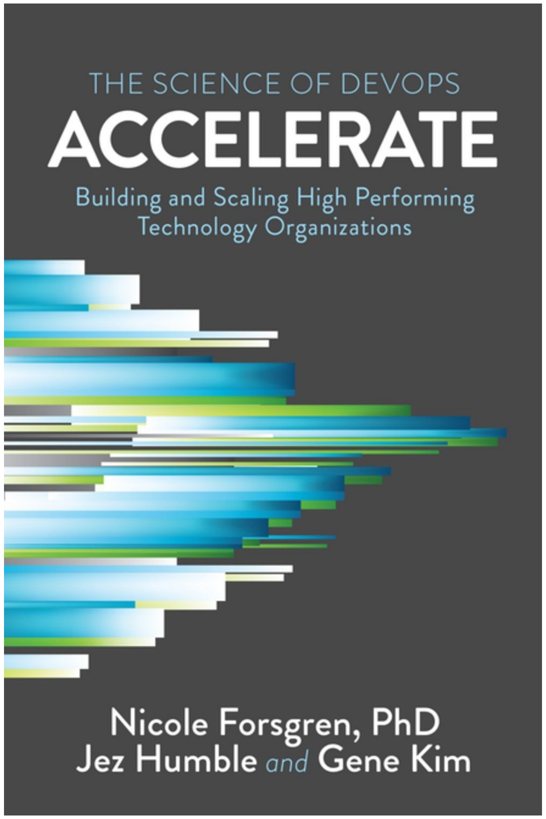
Continuous Delivery

- Deploy services multiple times per day
  - Robust build, test, deploy pipeline
  - Canary deployment
  - Feature flags
  - Dark launches
  - SLO monitoring
  - Synthetic monitoring

- More solid systems
  - Release smaller, simpler units of work
  - Smaller changes to roll back or roll forward
  - Faster to repair, easier to understand, simpler to diagnose
  - Increase rate of change and reduce risk of change

In the limit, production monitoring and software testing become the same thing

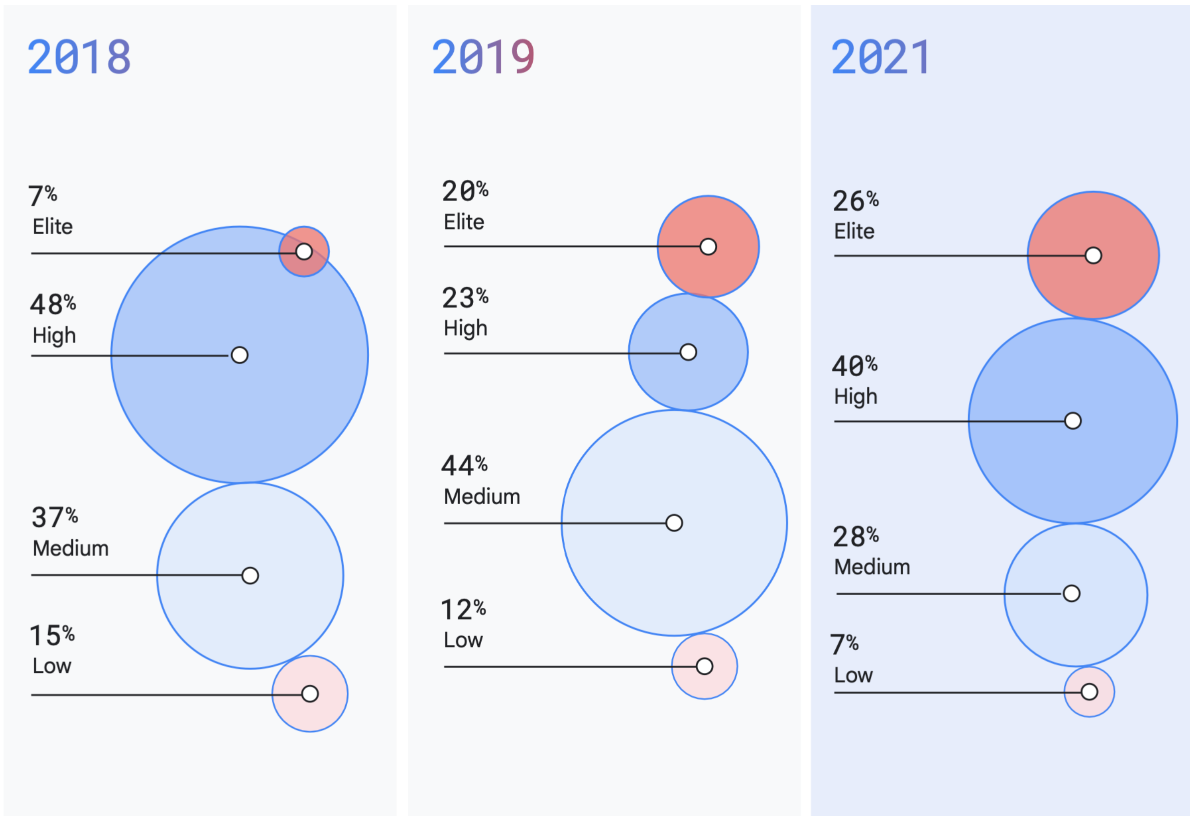
Software Delivery



- State of DevOps Surveys
  - 8 yearly surveys from 2014-2021
  - 31,000 survey responses
  - Rigorous scientific methodology
- Summarized in *Accelerate*

@randyshoup

Software Delivery



@randyshoup

State of DevOps Report, 2021

Continuous Delivery

- Cross-company *Velocity Initiative* to improve software delivery
  - Think Big, Start Small, Learn Fast
  - Iteratively identify and remove bottlenecks for teams
  - "What would it take to deploy your application every day?"
- Doubled engineering productivity
  - 5x faster deployment frequency
  - 5x faster lead time
  - 3x lower change failure rate
  - 3x lower mean-time-to-restore
- Prerequisite for large-scale architecture changes

Large-Scale Systems

- Simple Components
- Simple Interactions
- Simple Changes

Thank you!

@randyshoup

linkedin.com/in/randyshoup

medium.com/@randyshoup