Service Architectures at Scale
Lessons from Google and eBay

Randy Shoup
@randyshoup
linkedin.com/in/randyshoup

Architecture Evolution

- 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 C++ → Java / Scala → microservices

Service Architectures at Scale

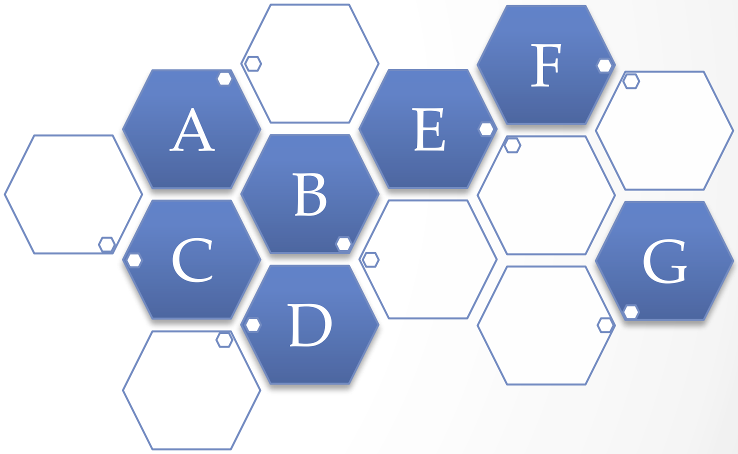
- Ecosystem of Services
- Building a Service
- Operating a Service
- Service Anti-Patterns

Service Architectures at Scale

- Ecosystem of Services
- Building a Service
- Operating a Service
- Service Anti-Patterns

Ecosystem of Services

- Hundreds to thousands of independent services
- Many layers of dependencies, no strict tiers
- Graph of relationships, not a hierarchy

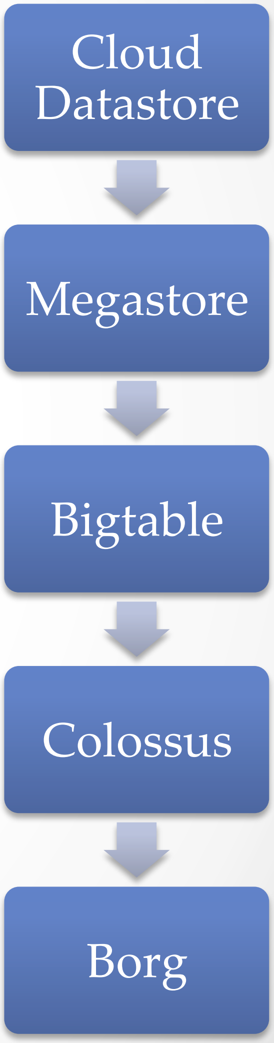


Evolution, not Intelligent Design

- No centralized, top-down design of the system
- Variation and Natural selection
  - Create / extract new services when needed to solve a problem
  - Deprecate services when no longer used
  - Services justify their existence through usage
- Appearance of clean layering is an emergent property

Google Service Layering

- Cloud Datastore: NoSQL service
  - Highly scalable and resilient
  - 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: next-generation clustered file system
  - Block distribution and replication
- Borg: cluster management infrastructure
  - Task scheduling, machine assignment



Architecture without an Architect?

- No “Architect” title / role
- (+) No central approval for technology decisions
  - Most technology decisions made locally instead of globally
  - Better decisions in the field
- (-) eBay Architecture Review Board
  - Central approval body for large-scale projects
  - Usually far too late in the process to be valuable
  - Experienced engineers saying “no” after the fact vs. encoding knowledge in a reusable library, tool, or service

Standardization

- Standardized communication
  - Network protocols
  - Data formats
  - Interface schema / specification

- Standardized infrastructure
  - Source control
  - Configuration management
  - Cluster management
  - Monitoring, alerting, diagnosing, etc.

Standards become standards by being better than the alternatives!

"Enforcing" Standardization

- Encouraged via
  - Libraries
  - Support in underlying services
  - Code reviews
  - Searchable code

The easiest way to encourage best practices is with *code*!

Make it really easy to do the right thing, and harder to do the wrong thing!

Service Independence

- No standardization of service internals
  - Programming languages
  - Frameworks
  - Persistence mechanisms

In a mature ecosystem of services, we standardize the arcs of the graph, not the nodes!

Creating
New Services

- Spinning out a new service
  - Almost always built for particular use-case first
  - If successful and appropriate, form a team and generalize for multiple use-cases
- Pragmatism wins
- Examples
  - Google File System
  - Bigtable
  - Megastore
  - Google App Engine
  - Gmail

Deprecating Old Services

- What if a service is a failure?
  - Repurpose technologies for other uses
  - Redeploy people to other teams

- Examples
  - Google Wave -> Google Apps
  - Multiple generations of core services

“Every service at Google is either deprecated or not ready yet.”

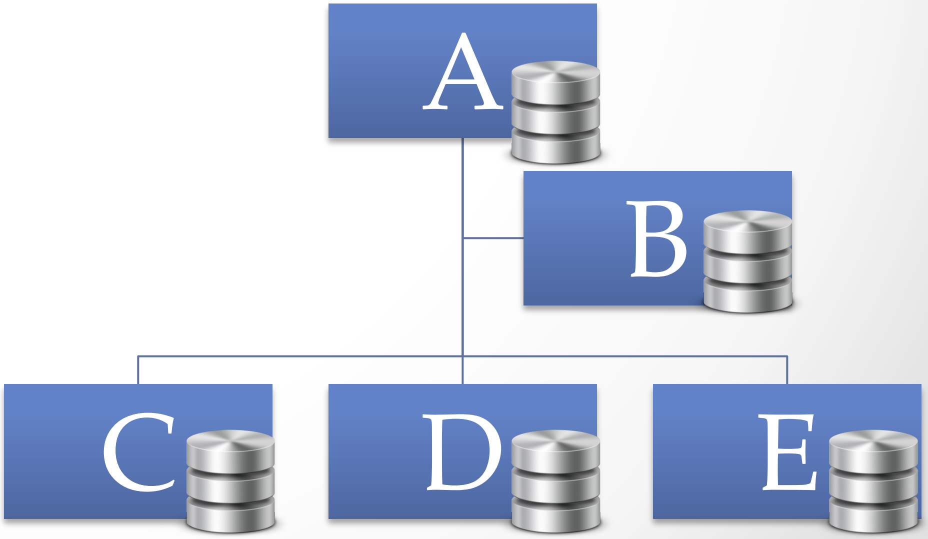
-- Google engineering proverb

Service Architectures at Scale

- Ecosystem of Services
- Building a Service
- Operating a Service
- Service Anti-Patterns

Characteristics of an Effective Service

- Single-purpose
- Simple, well-defined interface
- Modular and independent
- Isolated persistence (!)



Goals of a Service Owner

- Meet the needs of my clients …
  - Functionality
  - Quality
  - Performance
  - Stability and reliability
  - Constant improvement over time

- … at minimum cost and effort
  - Leverage common tools and infrastructure
  - Leverage other services
  - Automate building, deploying, and operating my service
  - Optimize for efficient use of resources

Responsibilities of a Service Owner

- End-to-end Ownership
  - Team owns service from design to deployment to retirement
  - No separate maintenance or sustaining engineering team
  - DevOps philosophy of "You build it, you run it"

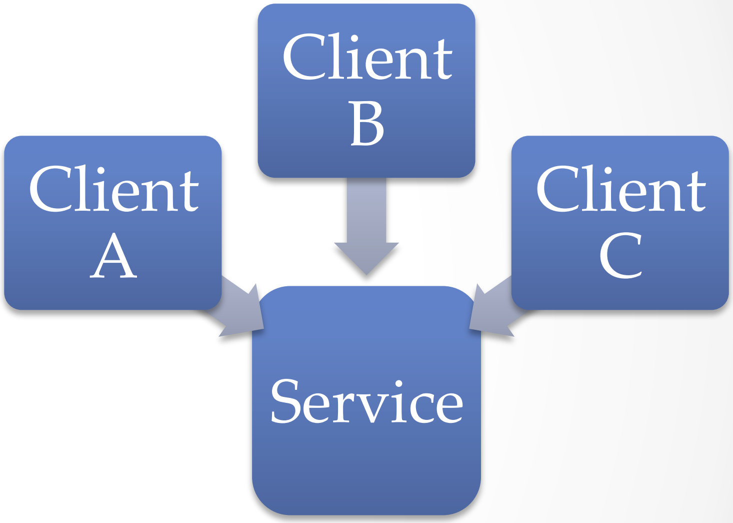
- Autonomy and Accountability
  - Freedom to choose technology, methodology, working environment
  - Responsibility for the results of those choices

Service as Bounded Context

- Primary focus on my service
  - Clients which depend on my service
  - Services which my service depends on
  - Cognitive load is very bounded

- Very little worry about
  - The complete ecosystem
  - The underlying infrastructure

- ➔ Small, nimble service teams



Service-Service Relationships

- Vendor – Customer Relationship
  - Friendly and cooperative, but structured
  - Clear ownership and division of responsibility
  - Customer can choose to use service or not (!)

- Service-Level Agreement (SLA)
  - Promise of service levels by the provider
  - Customer needs to be able to rely on the service, like a utility

Service-Service Relationships

- Charging and Cost Allocation
  - Charge customers for *usage* of the service
  - Aligns economic incentives of customer and provider
  - Motivates both sides to optimize for efficiency
  - (+) Pre- / post-allocation at Google

Maintaining Service Quality

- Small incremental changes
  - Easy to reason about and understand
  - Risk of code change is nonlinear in the size of the change
  - (-) Initial memcache service submission

- Solid Development Practices
  - Code reviews before submission
  - Automated tests for everything

- Google build and test system
  - Uses production cluster manager
  - Runs millions of tests per day in parallel
  - All acceptance tests run before code is accepted into source control

Maintaining Interface Stability

- Backward / forward compatibility of interfaces
  - Can *never* break your clients' code
  - Often multiple interface versions
  - Sometimes multiple deployments
  - Majority of changes don't impact the interface in any way

- Explicit deprecation policy
  - Strong incentive to wean customers off old versions (!)

Service Architectures at Scale

- Ecosystem of Services
- Building a Service
- Operating a Service
- Service Anti-Patterns

Predictable Performance

- Services at scale highly exposed to performance variability

- Imagine an operation …
  - 1ms median latency, but 1 second latency at 99.99%ile (1 in 10,000)
  - Service using one machine → 0.01% slow
  - Service using 5,000 machines → 50% slow

- Predictability trumps average performance
  - Low latency + inconsistent performance != low latency
  - Far easier to program to consistent performance
  - Tail latencies are *much* more important than average latencies

Google App Engine Memcache Service

- Periodic “hiccups” in latency at 99.99%ile and beyond
- Very difficult to detect and diagnose
- → Slab memory allocation

Service Reliability

- Systems at scale highly exposed to failure
  - Software, hardware, service failures
  - Sharks and backhoes
  - Operator “oops”

- Resilience in depth
  - Redundancy for machine / cluster / data center failures
  - Load-balancing and flow control for service invocations
  - Rapid rollback for “oops”

Service Reliability: Deployment

- Incremental Deployment
  - Canary systems
  - Staged rollouts
  - Rapid rollback

- eBay "Feature Flags"
  - Decouple code deployment from feature deployment
  - Rapidly turn on / off features without redeploying code
  - Typically deploy with feature turned off, then turn on as a separate step

Service Reliability: Monitoring

- Instrumentation
  - Common monitoring service
  - Machine / instance statistics: CPU, memory, I/O
  - Request statistics: request rate, error rate, latency distribution
  - Application / service statistics
  - Downstream service invocations

- Diagnosability
  - In-process web server with current statistics
  - Distributed tracing of requests through multiple service invocations

You can have too much alerting, but you can never have too much monitoring!

Service Architectures at Scale

- Ecosystem of Services
- Building a Service
- Operating a Service
- Service Anti-Patterns

Service Anti-Patterns

- The "Mega-Service"
  - Overbroad area of responsibility is difficult to reason about, change
  - Leads to more upstream / downstream dependencies
- Shared persistence
  - Breaks encapsulation, encourages "backdoor" interface violations
  - Unhealthy and near-invisible coupling of services
  - (-) Initial eBay SOA efforts

Thank You!

- @randyshoup
- linkedin.com/in/randyshoup
- Slides will be at slideshare.net/randyshoup