Pragmatic Microservices
Whether, When, and How to Migrate

Randy Shoup
@randyshoup
linkedin.com/in/randyshoup
Background

• Consulting CTO at Randy Shoup Consulting
  o Helping companies scale their organizations and technology
  o “CTO as a service”

• CTO at KIXEYE
  o Real-time strategy games for web and mobile

• Director of Engineering for Google App Engine
  o World’s largest Platform-as-a-Service

• Chief Engineer at eBay
  o Multiple generations of eBay’s infrastructure
“Tell us how you did things at Google and eBay.”

“Sure, I’ll tell you, but you have to promise not to do them! [... yet]”
Architecture Evolution

• eBay
  • 5th generation today
  • Monolithic Perl \(\rightarrow\) Monolithic C++ \(\rightarrow\) Java \(\rightarrow\) microservices

• Twitter
  • 3rd generation today
  • Monolithic Rails \(\rightarrow\) JS / Rails / Scala \(\rightarrow\) microservices

• Amazon
  • Nth generation today
  • Monolithic C++ \(\rightarrow\) Java / Scala \(\rightarrow\) microservices
Are microservices the right approach to every problem?
Pragmatic Principles

- Solve the problems you have, not the problems you might have
- Use the Right Tool for the Right Job (at the Right Time!)
Pragmatic Evolution

- Monoliths
- Microservices
- Deciding to Rearchitect
- Rearchitecture Tactics
Pragmatic Evolution

- Monoliths
- Microservices
- Deciding to Rearrange
- Rearrangement Tactics
The Monolithic Architecture

2-3 monolithic tiers

- Presentation
- Application
- Database
The Monolithic Application

Pros

- Simple at first
- In-process latencies
- Single codebase, deploy unit
- Resource-efficient at small scale

Cons

- Coordination overhead as team grows
- Poor enforcement of modularity
- Poor scaling (vertical only)
- All-or-nothing deploy (downtime, failures)
- Long build times
The Monolithic Database

Pros

- Simple at first
- Join queries are easy
- Transactions
- Resource-efficient at small scale

Cons

- Coupling over time
- Performance and scalability bottleneck
- Difficult to tune properly
- Single point of failure
Improving the Monolith

• Exploit “Natural Partitions”
  o Many domains cleanly partition into independent applications
  o E.g., Buyer / seller, rider / driver, etc.

• Modularity Discipline
  o Internal componentization bounds cognitive load
  o Easier to modify or replace

• Continuous Delivery
  o Rapid deployment allows rapid evolution
Pragmatic Evolution

- Monoliths
- Microservices
- Deciding to Rearchitect
- Rearchitecture Tactics
Microservices

- Single-purpose
- Simple, well-defined interface
- Modular and independent
- Fullest expression of encapsulation and modularity
- Isolated persistence (!)
Microservices are nothing more than SOA done properly.

-- me
# Microservices

**Pros**

- Each unit is simple
- Independent scaling and performance
- Independent testing and deployment
- Can optimally tune performance (caching, replication, etc.)

**Cons**

- Many cooperating units
- Network latencies
- No transactions
- Cross-cutting concerns and refactoring
- Requires more sophisticated tooling and dependency management
Prerequisites for Success

- Process Maturity
- Organizational Maturity
- Operational Maturity
Process Maturity: Continuous Delivery

• Repeatable Deployment Pipeline
  o Low-risk, push-button deployment
  o Rapid release cadence
  o Rapid rollback and recovery

• Automated Testing
  o Developers write tests and code together
  o Confidence to make risky changes

• Continuous Integration
  o System components assembled and tested together on each checkin
Organizational Maturity: Conway’s Law

• Organization constrains architecture
  o Design of a system will be a reflection of the communication paths within the organization

• Modular system requires modular organization
  o Small, independent teams lead to more flexible, composable systems
  o Larger, interdependent teams lead to larger, more monolithic systems

• To get the system we want, we need to engineer the organization
Organizational Maturity: Service Teams

• Teams Aligned to Domains
  o Clear, well-defined area of responsibility
  o Single service or set of related services
  o A given service is managed within one team

• Cross-functional Teams
  o Team has inside it all skill sets needed to do the job
  o Depends on other teams for supporting services, libraries, and tools

• End-to-end Ownership
  o Team owns service from design to deployment to retirement
  o No separate maintenance or sustaining engineering team
  o DevOps philosophy of “You build it, you run it”
Operational Maturity: Resilience

• Availability and Reliability as first-order concerns

• “Pets and Cattle”
  o We no longer care about individual machines or instances
  o We only care about the service in aggregate
Operational Maturity: Monitoring

- Strong emphasis on detailed, end-to-end monitoring of production systems
- Ability to detect and alert on system issues
- Sufficient monitoring to be able to do remote runtime diagnosis
“If you ever have to ssh into a machine, your monitoring has failed you.”

-- me
Pragmatic Evolution

- Monoliths
- Microservices
- Deciding to Rearrange
- Rearrangement Tactics
Why Rearchitect?

• Velocity
  - Time to market is severely constrained by coupling, complexity, and lack of isolation in the monolith
  - Teams are stepping on each others’ toes, and can no longer develop independently
  - Difficult for new engineers to be productive

• Scaling
  - Vertical scaling of the monolith no longer works
  - Parts of the system need to scale independently of others
Why Rearchitect?

• Isolation
  o Need to decompose for resource isolation
  o Need to decompose to avoid coordinated failures

• Deployment
  o Parts of the system need to deploy independently of others
  o Monolithic release cadence is too slow
  o Monolithic release is too complicated, too risky
When to Rearchitect

• Current architecture must be unsustainable in the near term
  o Need for rearchitecture outweighs significant investment
  o Need for rearchitecture outweighs opportunity costs

• Steady and growing usage
  o Organization should invest

• Resources to rearchitecture
  o Organization can invest
Getting to rearchitect a system is a sign of success, not failure.

-- me
If you don’t end up regretting your early technology decisions, you probably over-engineered.

-- me
Pragmatic Evolution

- Monoliths
- Microservices
- Deciding to Rearchitect
- Rearchitecture Tactics
“The only thing a Big Bang migration guarantees is a big *Bang*.”

-- Martin Fowler
Incremental Migration

• Step 0: Pilot Implementation
  o Choose initial end-to-end vertical experience to migrate / create
  o (+) Provide real customer value
  o (+) Opportunity to learn and adjust
  o (+) Bounded investment and risk
  o (+) Demonstrate feasibility and gain confidence

• Initial step is the hardest
  o Learning how to do things in the new way
  o Building out basic supporting capabilities
Incremental Migration

• Steps 1-N: Incremental Migration
  o Prioritize business value -- highest ROI areas first
  o Focus on areas with greatest rate of change
  o (+) Maximize near-term payoff from investment
  o (+) Confront and solve hard problems sooner rather than later

• New feature development in parallel
  o Typically cannot pause all feature work in all areas to migrate 😊
  o Within a particular area, try to separate feature work from migration work in distinct steps
Incremental Migration

- Residual monolith may remain indefinitely
  - Lowest business value
  - Most stable and least changing
  - Can migrate – or not – opportunistically
Carving up the Monolith

• Look for (or create) a “seam” in the monolith
  o This is often the hardest part (!)

• Wall it off behind an interface

• Write automated tests around the interface

• Replace implementation with a service

• ➔ Rinse and Repeat
Building a Microservice

• **Common Chassis / Framework**
  - Make it easy to build and maintain a service
  - E.g., NetflixOSS, GoKit

• **Define Service Interface (Formally!)**
  - Propose
  - Discuss with client(s)
  - Agree

• **Prototype Implementation**
  - Simplest thing that could possibly work
  - Limited investment of time and people
  - Client can integrate with prototype
  - Implementor can learn what is used, what works
Building a Microservice

- Real Implementation
  - Throw away the prototype (!)
- Rinse and Repeat
Maintaining Interface Stability

• Backward / forward compatibility of interfaces
  o Can *never* break your clients’ code
  o Often multiple interface versions
  o Sometimes multiple deployments

• Explicit deprecation policy
  o Strong incentive to wean customers off old versions (!)
Service Anti-Patterns

• The “Mega-Service”
  o Overbroad area of responsibility is difficult to reason about, change
  o Leads to more upstream / downstream dependencies

• “Leaky Abstraction” Service
  o Interface reflects provider’s implementation, not the consumer’s model
  o Consumer’s model is typically more aligned with the domain, simpler, more abstract
  o Leaking provider’s model in the interface constrains evolution of the implementation
Service Anti-Patterns

• Shared persistence
  o Breaks encapsulation, encourages “backdoor” interface violations
  o Unhealthy and near-invisible coupling of services
  o (-) Initial eBay SOA efforts
Microservice Persistence

• Option 1: Operate your own data store
  o Store to your own instance(s) of MySQL, etc., owned and operated by the service

• Option 2: Use a persistence service
  o Store to your own partition(s) of Dynamo, Bigtable, etc., operated as a service by another team

• ➔ Only external access to data store is through published service interface
Microservices may not be for you if …

• ... you have a simple system

• ... you have a small team

• ... you are not able to invest in continuous delivery, monitoring, etc.

• ... they don’t solve the problems you have
Thank You!

• @randyshoup

• linkedin.com/in/randyshoup

• Slides will be at slideshare.net/randyshoup