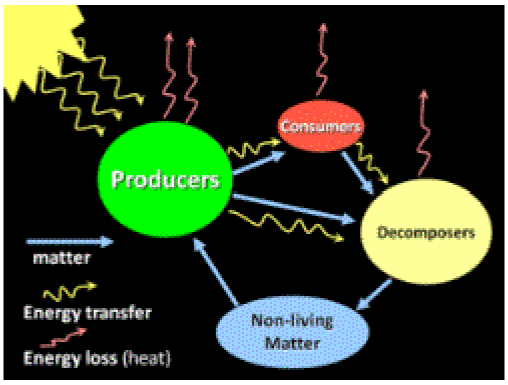
Deep Lessons From Google And EBay On Building Ecosystems Of Microservices

TUESDAY, DECEMBER 1, 2015 AT 8:56AM



When you look at large scale systems from Google, Twitter, eBay, and Amazon, their architecture has evolved into something similar: **a set of polyglot microservices**.

What does it looks like when you are in the polyglot microservices end state? Randy Shoup, who worked in high level positions at both Google and eBay, has a very interesting talk exploring just that idea: Service Architectures at Scale: Lessons from Google and eBay.

What I really like about Randy's talk is how he is self-consciously trying to immerse you in the experience of something you probably have no experience of: creating, using, perpetuating, and protecting a large scale architecture.

In the *Ecosystem of Services* section of the talk Randy asks: What does it look like to have a large scale ecosystem of polyglot microservices? In the *Operating Services at Scale*section he asks: As a service provider what does it feel like to operate such a service? In the *Building a Service* section he asks: When you are a service owner what does it look like? And in the *Service Anti-Patterns* section he asks: What can go wrong?

A very powerful approach.

The highlight of the talk for me was the idea of **aligning incentives,** a consistent theme that crosscuts the entire endeavour. While never explicitly pulled out as a separate strategy, it's the motivation behind why you want small teams to develop small clean services, why a charge back model for internal services is so powerful, how architecture can evolve without an architect, how clean design can evolve from a bottom up process, and how standards can evolve without a central committee.

My takeaway is **the deliberate aligning of incentives is how you scale both a large, dynamic organization and a large, dynamic code base**. Putting in the right incentives nudges things into happening without explicit control, almost in the same way more work in a distributed system gets done when you remove locks, don't share state, communicate with messages, and parallelize everything.

Let's see how large scale systems are built in the modern era...

Polyglot Microservices Are The End Game

- Large scale systems end up evolving into something that looks very similar: **a set of polyglot microservices**. Polyglot means microservices are can be written in more than one language.
- **eBay** started in 1995. Depending on how you count they are on the 5th generation of their architecture.

- Started as a monolithic Perl application that the founder wrote over a Labor Day weekend in 1995.
- Then it moved to a monolithic C++ application which ended up with 3.4 million lines of code in a single DLL.
- The previous experience spurred the move to a far more distributed partitioned system in Java.
- The eBay of today has quite a bit of Java, but a polyglot set of microservices.
- **Twitter**'s evolution looks very similar. Depending on how you count they are on the third generation of their architecture.
- Started as a monolithic Ruby on Rails application.
- Moved to a combination of Javascript and Rails on the frontend with a lot of Scala on the backend.
- Ultimately they've moved to what we'd call today a set of polyglot microservices.
- **Amazon** followed a similar path.
- Started with a monolithic C++ application.
- Then services written in Java and Scala.
- Ending up with a set of polyglot microservices.

Ecosystem Of Services

- What does it look like to have a large scale ecosystem of polyglot microservices?
- At eBay and Google hundreds to thousands of independent services all work together.
- Modern large scale systems compose services in a graph of relationships, not a hierarchy or set of tiers.
- Services depend on many other services while being depended on by many services.
- Older large scale systems were typically organized in strict tiers.

How Is An Ecosystem Of Services Created?

- These best performing systems are more the **product of evolution than intelligent design**. At Google, for example, there never has been a centralized top down design of the system. It has evolved and grown over time in a very organic way.
- Variation and natural selection. When a problem needs to be solved a new service is created, or more often extracted from an existing service or product. **Services survive as long as they are used**, as long as they provide value, otherwise they are deprecated.
- These large scale systems **develop from the bottom up**. **Clean design can be an emergent property rather than a product of top down design**.

- As an example consider some of the service layering for Google App Engine.
- The Cloud Datastore (a NoSQL service) is built on the Megastore (a geo-scale structured database) which is built on Bigtable (a cluster-level structured service) which is built on Colossus (a next-generation clustered file system) which is built on Borg (the cluster management infrastructure).
- The layering is clean. Each layer adds something that didn’t belong in the layer below. It was not the product of top down design.
- It was built from the bottom up. Colossus, the Google file system was built first. Several years later Bigtable was built. Several years later Megastore was built. And several years later Cloud Database migrated onto Megastore.
- You can have this wonderful separation of concerns without a top down architecture.
- **This is architecture without an architect**. Nobody at Google has the title of Architect. There is no central approval for technology decisions. Most technology decisions are made by individual teams locally for their own purposes, they are not made globally.
- Contrast with the eBay of 2004. There was an architecture review board, which had to approve all large-scale projects.
- Usually they only got involved in projects when it was far too late to change them.
- The centralized approval body became a bottleneck. Often its only influence was to say no at the last minute.
- A better way for eBay to have handled the situation was to **encode the knowledge** of the smart experienced people in the review board and **put it into something that’s reusable** by individual teams. Encode that experience into a library or a service or even a set of guidelines that people can use on their own rather only coming into the process at the last moment.

How Do Standards Evolve Without Architects?

- It is possible with **no central control to end up with standardization**.
- Standardization tends to occur **around communication** between services and common pieces of infrastructure.
- Standards become standards because they are **fitter than the alternative**s.
- The parts of communication that are usually standardized:
- **Network protocols**. Google uses a proprietary protocol called Stubby. eBay uses REST.
- **Data formats**. Google uses Protocol Buffers. eBay tends to use JSON.
- **Interface schema standard**. Google uses Protocol Buffers. For JSON there’s JSON schema.
- The pieces of common infrastructure that are usually standardized:

- Source code control.
- Configuration management.
- Cluster manager.
- Monitoring systems.
- Alerting systems.
- Diagnostic tools.
- All these components can evolve out of conventions.
- In an evolutionary environment **standards are enforced through**: code, encouragement, code reviews, and code search.
- The easiest way to **encourage best practices is through actual code**. It's not about top down review, or up front design, it's about someone producing code that makes it easy to get the job done.
- Encouragement is through **teams providing a library**.
- Encouragement is also through the services you want to depend on supporting X protocol or Y protocol.
- Google is famous for **every line of code** that is checked-in to source code control **being reviewed** by at least one other programmer. This is a good way of communicating common practices.
- With a few exceptions, every engineer at Google can **search the entire code base**. This is a huge value add for when a programmer is trying to figure out how to do something. With 10K engineers it's likely if you are trying to do something somebody has already done something like it before. This allows **best practices that start in one area to propagate through the code base**. It also allows mistakes to propagate.
- To encourage common practices and standardized conventions **make it really easy to do the right thing** and a lot harder to do the wrong thing.
- Individual **services are independent** from one another.
- At Google there's **no standardization of the internals of services**. Services are a black box to the outside.
- There are conventions and common libraries, but there are no programming language requirements. Four languages are used commonly: C++, Go, Java, Python. Lots of different services are written in various languages.
- There's no standardization around frameworks or persistence mechanisms.
- **In a mature ecosystem of services we standardize the arcs of the graph, not the nodes themselves.** Define a common shape, not a common implementation.

Creating New Services

- New services are created when their use has already been proven.
- Often a capability has been built for one particular use case. Then it’s discovered that capability is general and useful.
- A team is formed and the service is spun out to its own standalone unit.
- This happens only when a capability is successful and fits many different use cases.
- These **architectures grow through pragmatism**. Nobody is sitting on high and saying a service should be added.
- Google File System supported the search engine. No surprise that a distributed file system was more generally usable.
- Bigtable initially supported the search engine, but was much more broadly useful.
- Megastore was built as the storage mechanism for Google applications, but was more broadly useful.
- Google App Engine itself was started by a small group of engineers who recognized the need for help building web sites.
- Gmail came out of a side project that was extremely useful internally and then was externalized for other people.

Deprecating Old Services

- What happens if a service is not used anymore?
- Technologies that can be repurposed are reused.
- People can be fired or redeployed to other teams.
- Google Wave was not a market success, but some of the technologies ended up in Google Apps. For example, the ability for multiple people to edit documents comes from Wave.
- The more common case is for core services to go through multiple generations and the old generations are deprecated. This happens a lot at Google. There’s so much change it often seem like every service inside Google is either deprecated or not ready yet.

Building A Service

- When you are service owner what does it look like when you are building a service in a large scale system of polyglot microservices?
- A well performing service in a large scale architecture is:
- **Single-purpose**. It will have a simple well defined interface.
- **Modular and independent**. What we might call a microservice.
- **Does not share a persistence layer**. More on this later.

What Are The Goals Of The Service Owner?

- **Meet the needs of your clients**. Provide the necessary functionality, at the proper quality level, while meeting the negotiated performance levels, while maintaining stability and reliability, while constantly improving the service over time.
- **Meet the needs at a minimum cost and effort**.
- This **goal aligns incentives** in a way that encourages the **use of common infrastructure**.
- Each team has a limited set of resources so it’s in their interest to leverage common battle tested tools, processes, components, and services.
- It also incents good operational behavior. Automate the building and deploying of your service.
- It also incents optimizing for the efficient use of resources.

What Are The Responsibilities Of The Service Owner?

- **You build it you run it**.
- The team, typically a small teams, owns the service from design, through development, and deployment, all the way through to retirement.
- There’s no separate maintenance or sustaining engineering team.
- Teams have the freedom to make their own technology choices, methodologies, and working environment.
- Teams are accountable for the choices they make.
- **Service as a bounded context**.
- The **cognitive load on a team is bounded**.
- There’s no need to understand all the other services in the ecosystem.
- A team needs to understand their service in depth and the services they depend on.
- This means **teams can be extremely small and nimble**. A typical team is 3-5 people. (As an aside a US Marine Corps fireteam has four people.)
- The small team size means communication within the team is at a really high bandwidth and quality.
- Conway’s Law used to your advantage. By organizing in small teams you’ll end up with small individual components.

What Is The Relationship Between Services?

- Think about **relationships between services as vendor-customer relationships**, even though you are at the same company.
- Be very friendly and cooperative, but be very structured in the relationship.
- Be very clear about ownership.
- Be very clear how about who is responsible for what. In large part this is about defining a clear interface and maintaining it.
- **Incentives are aligned because customer can choose to use a service or not**. This encourages services to do right by their customers. This is one of the ways new services end up being built.
- Define SLAs. As service provider promise a certain level of service to their customers so customers can rely on the service.
- **Customer teams pay for services**.
- **Charging for a service aligns economic incentives**. It motivates both sides to be extremely efficient about the use of resources.
- When things are f**ree we tend not to value them and tend not to optimize them**.
- For example, an internal customer was using Google App Engine for free and they were using a lot of resources. Begging them to be more efficient about their use of resources turned out not to be a good strategy. A week after chargebacks kicked-in they were able to reduce their consumption of GAE resources by 90% with one or two simple changes.
- It’s not that the team using GAE was evil, they just had other priorities, so there was no incentive for them to optimize their use of GAE. It turns out they actually got better response times with the more efficient architecture.
- **Charging also incents a service provider to keep quality high**, otherwise an internal customer may go elsewhere. This directly incentivizes good development and management practices. Code reviews are one example. Google’s very large scale build and test system is another. Google runs millions of automated tests every day. Acceptance tests for all dependent code is run every time code is accepted into the repository, which helps all the small teams maintain the quality of their services.
- A charge back model **encourages small incremental changes**. Small changes are easier to understand. Also, the impact of code changes are non-linear. A thousand line change is not 10 times riskier than a 100 line change, it’s more like 100 times riskier.
- **Maintain full backward / forward compatibility of interfaces**.
- Never break client code.
- This means maintaining multiple interface versions. In some nasty situations it means maintaining multiple deployments, one for the new version and others for older versions.
- Usually because of the small incremental change model interfaces are not changed.

- Have an explicit deprecation policy. Then the service provider is strongly incented to moved all clients off version N and over to version N+1.

Operating Services At Scale

- As a service provider what does it feel like to operate a service in a large scale system of polyglot microservices?
- **Predictable performance is a requirement**.
- Services at scale are **very exposed to variability in performance**.
- **Predictability in performance is much more important** than average performance.
- Low latency with inconsistent performance is actually not low latency at all.
- It’s far easier for clients to program against a service when it provides consistent performance.
- Tail latencies dominate performance as services use many other services to carry out their work.
- Imagine a service that has 1ms latency at the median and at the 99.999%ile (1 in 10,000) the latency is one second.
- Making one call means you are slow .01% of the time.
- If you are using 5,000 machines, as many large scale services at Google do, then you are slow 50% of the time.
- For example, a one in a million problem with memcached was tracked down to a low level data structure reallocation event. This rare problem surfaced as at a higher level as latency spikes. Low level details like this turn out to be extremely important in a large scale system.
- Resilience in depth.
- Service disruptions are far more likely to occur from a mistake by a person rather than a hardware or software failure.
- Be resilient to machine, cluster and datacenter failures.
- Load balance and provide flow control when invoking other services.
- Be able to rapidly roll back changes.
- Incremental deployment.
- **Use a canary system**. Don’t deploy to all machines at once. Choose a system, put the new version of that software on that system, and see how it behaves in the new world.
- If it works **begin a staged rollout**. Start of with 10% of the machines, move to 20%, and so on through the rest of the fleet.

- If a problem happens at the 50% point in the deploy then you should be able to rollback.
- eBay made use of **feature flags to decouple code deployment from feature deployment**. Typically code is deployed with a feature turned off, then it can be turned on or off. This makes sure the code can be properly deployed before a new feature is turned on. It also means if the new feature has a bug, a performance issue, or a business failure, then the feature can be turned off without having to deploy new code.
- You can have too much alerting you can never have too much monitoring.

Service Anti-Patterns

- **The Mega-Service**
- A service that does too much. What you want is an ecosystem of very small clean services.
- A service that does too much is a **just another monolith**. It’s hard to reason about, it’s hard to scale, it’s hard to change, and it also creates more upstream and downstream dependencies than you want.
- **Shared Persistence**
- In the tiered model services are put in the application tier and the persistence layer is provided as a common service to the applications.
- They did this at eBay and **it didn’t work**. It **breaks the encapsulation** of the service. Applications can **back-door into your service** by updating the database. It ends up reintroducing coupling of services. Shared databases don't allow for loosely coupled services.
- Microservices prevent this problem by being small, isolated, and independent, which is how you keep your ecosystem healthy and growing.