Maximizing Velocity with SOA @ Yammer

Mike Ihbe
Senior Infrastructure Engineer
twitter.com/mikeihbe
github.com/mikejihbe

Thursday, May 24, 2012

Hi everyone, I'm Mike Ihbe, a senior infrastructure engineer at Yammer. Today we're going to be talking about Maximizing Velocity in a software engineering organization and how Service oriented architectures can help you manage that. Now, the title is pretty buzz-word filled so let's break it down. What do I mean by velocity?

Maximizing Velocity with SOA @ Yammer

Mike Ihbe
Senior Infrastructure Engineer
twitter.com/mikeihbe
github.com/mikejihbe

Thursday, May 24, 2012

Hi everyone, I'm Mike Ihbe, a senior infrastructure engineer at Yammer. Today we're going to be talking about Maximizing Velocity in a software engineering organization and how Service oriented architectures can help you manage that. Now, the title is pretty buzz-word filled so let's break it down. What do I mean by velocity?

What is Velocity?

$$\vec{V} = \frac{d}{t}$$

Thursday, May 24, 2012

Well, it's distance over time. From a software engineering perspective, velocity is the value created by your organization. This is a familiar concept, so it's easy to reason about. It's essentially about moving quickly, but more than that, velocity is a vector. Consistent direction is essential to effectively achieving your goals.

At Yammer we think of this as maximizing the production of business value we can create for our customers. Every time we release a new feature, fix a bug, or improve performance it adds to this value.

Things like refactoring and tackling technical debt count as friction. The forces your engineering organization can bring to bear must overcome this friction or you won't be producing any value. So, when I'm talking maximizing velocity, I really mean creating as much value as quickly as possible.

So, what is SOA?

What is SOA?

Technology

Software design paradigm enabling organizational flexibility through composable, loosely coupled, fault tolerant business services with clearly defined, published interfaces.

It's a software design paradigm enabling organizational flexibility through composable, loosely coupled, fault tolerant business services with clearly defined, published interfaces. That's a mouthful, so we'll talk more about these things, but I'd like to add a human perspective.

SOA is also about people and how we work. It's a decentralized organizational structure enabling small, autonomous, and extremely effective engineering teams.

What is SOA?

Technology

Software design paradigm enabling organizational flexibility through composable, loosely coupled, fault tolerant business services with clearly defined, published interfaces.

+

People

A decentralized human-organizational structure enabling small, directed, autonomous, extremely effective engineering teams.

It's a software design paradigm enabling organizational flexibility through composable, loosely coupled, fault tolerant business services with clearly defined, published interfaces. That's a mouthful, so we'll talk more about these things, but I'd like to add a human perspective.

SOA is also about people and how we work. It's a decentralized organizational structure enabling small, autonomous, and extremely effective engineering teams.

Technology + People

Thursday, May 24, 2012

Most of us in here are software engineers. We all know technology is hard. Human organization is even harder. Doing both well is nigh impossible. This talk is about some of the lessons we've learned at Yammer about how to organize your people and your technology to build a scalable organization that's laser focused on building value quickly.

That textbook definition of SOA wasn't terribly explanatory, so let's wrap our heads around it a little more. We can begin by contrasting it to the alternatives.

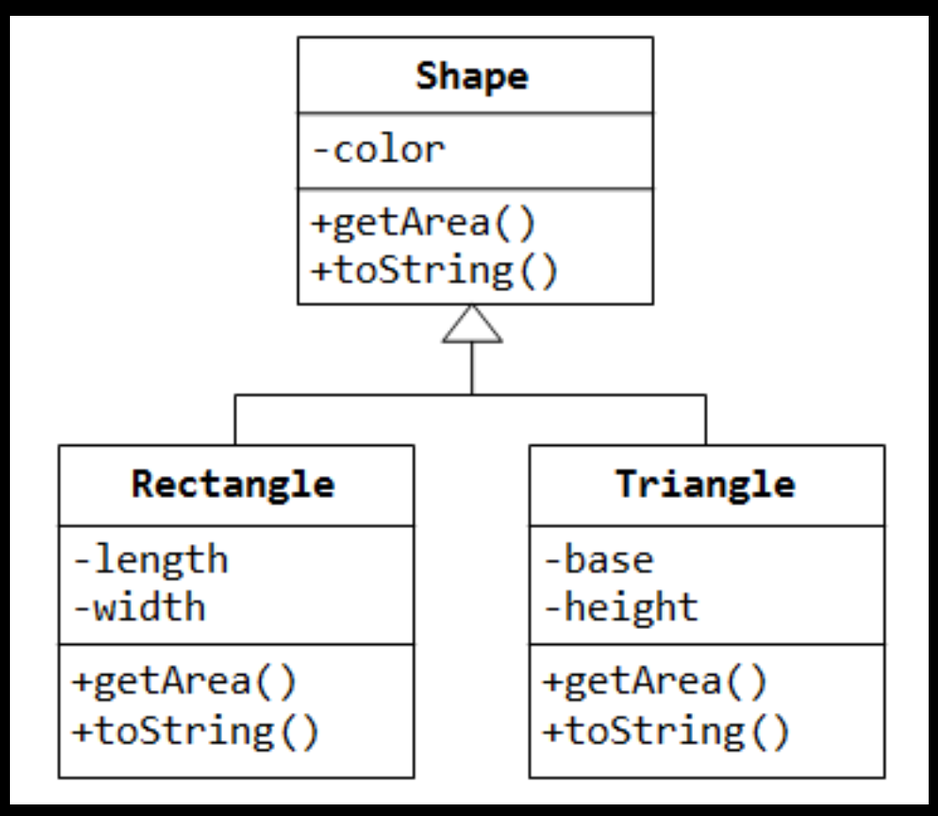
VS the monolith

- Large, complex, unfocused
- Difficult to reason about
- Shared state/data

Monolithic applications inevitably run into the same problems. They're large, unfocused and complicated because they try to solve every problem. It becomes impossible to keep the entire program in your head, so it becomes difficult to reason about. Developers end up stepping on each other's toes with code or data changes, because they're unsure of how each component is being used.

SOA provides an answer to these concerns by stealing some principles from object oriented programming.

OOP Similarities



- Encapsulation
- Focused functionality
- Clear interfaces
- Reusability

SOA does a better job encapsulating concerns. Each service has focused functionality which makes the pieces of a complex system easier to reason about. Clear interfaces hide implementation details and make it possible to change underlying implementations. Separated services become reusable components and can be composed to accomplish even more complex tasks.

The last way to think about SOA is the human-organizational component.

Organizationally

"Organizations...are
constrained to produce
designs which are copies of
the communication structures
of these organizations."

-Conway's law

This is Conway's law. "Organizations...are constrained to produce designs which are copies of the communication structures of these organizations."

Basically this is saying that if you hire a firm with 3 engineering teams, you'll end up with a product that has 3 subsystems and the quality of the interfaces between those subsystems will reflect the communication quality between those teams. Conway was a smart man. There are studies out of Microsoft Research and Harvard that help corroborate this.

SOA, when wielded properly, can help build an organization that takes advantage of Conway's law, rather than letting it dictate your design decisions.

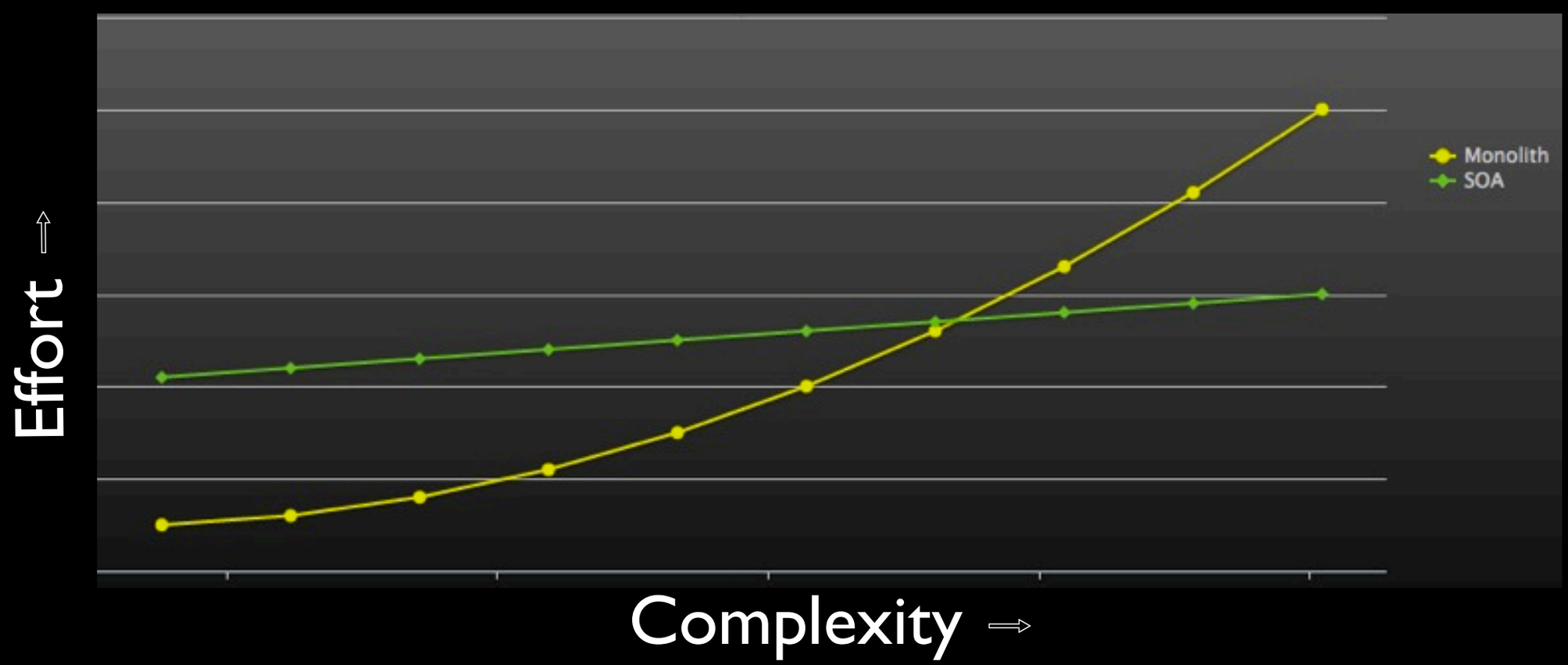
Panacea?

That makes it sound like SOA is all glitter, rainbow and unicorns. It's no panacea.

As with most engineering, there are very real costs behind all those benefits.
With a single, undivided code base you're able to dance around class boundaries, access data layers directly, share a lot of code easily, and avoid the overhead that managing many disparate production services entails.

There is value in that, but at some point the overhead of managing many services becomes less than the overhead of dealing with the overwhelming, complex, wild-west engineering that was happening in your monolithic application.

Costs



Thursday, May 24, 2012

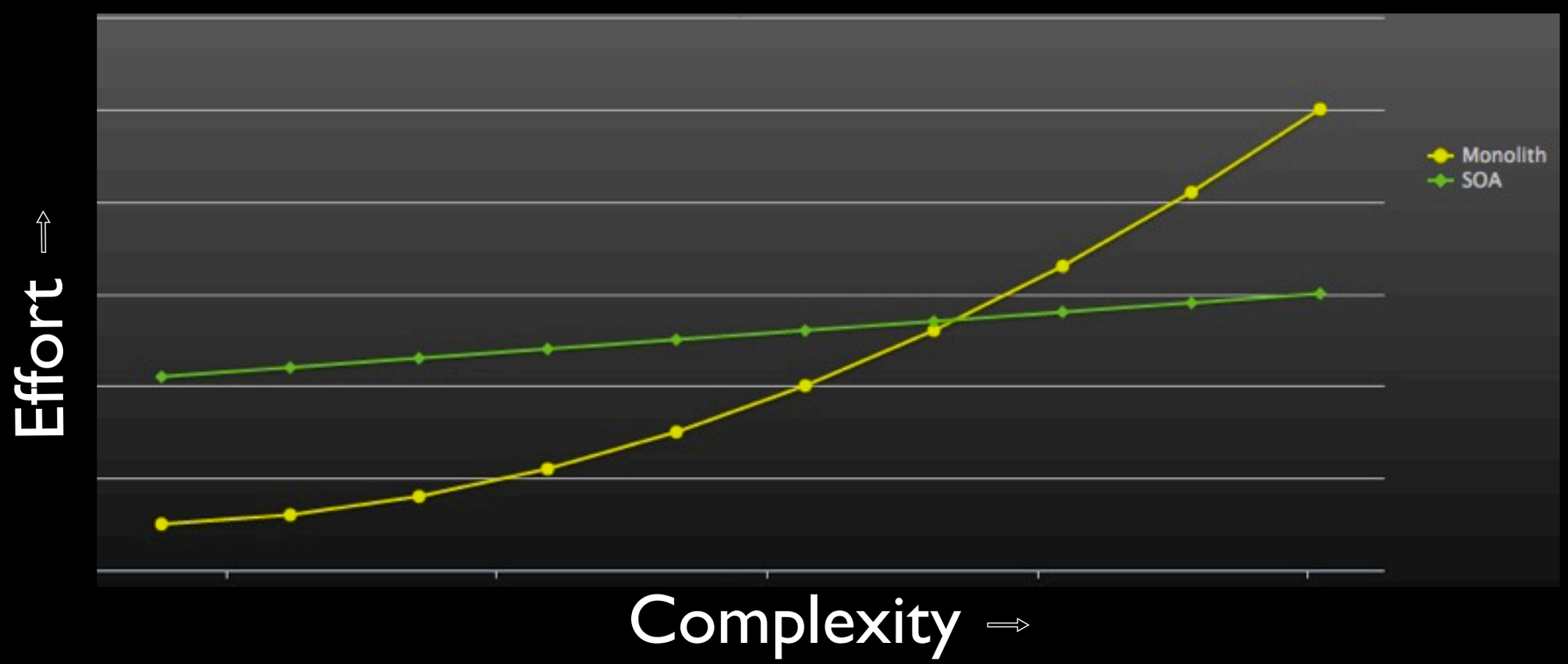
So, what do these costs look like? I've attempted to model them here. Complexity is my hand-wavy aggregate of the number of people working on a project and LOC. Effort, is the cost of building and maintaining a project.

We can see that the monolithic application's effort grows quadratically. The classic wisdom here is from Mythical Man Month - the effort needed to communicate within a team grows at a rate of n(n-1)/2, we also include the difficulty of the project as D.

To contrast that, building systems has some significant overhead. Systems programs are supposedly 3x more expensive than creating the product. Because we're building a small, focused service, the number of people and LOC are bounded, and that's the key to keeping effort growing linearly. It will always start higher than the monolithic approach, and it will always end lower.

In reality, there's a missing factor in this second equation to account for the complexity of the entire system of systems, but that factor would be based on the service dependency graph of a specific deployment and should be small for any given new project. The point is: as your System's complexity increases, SOA becomes much more viable.

Costs



Monolith: $\frac{x(x-1)}{2} + D$

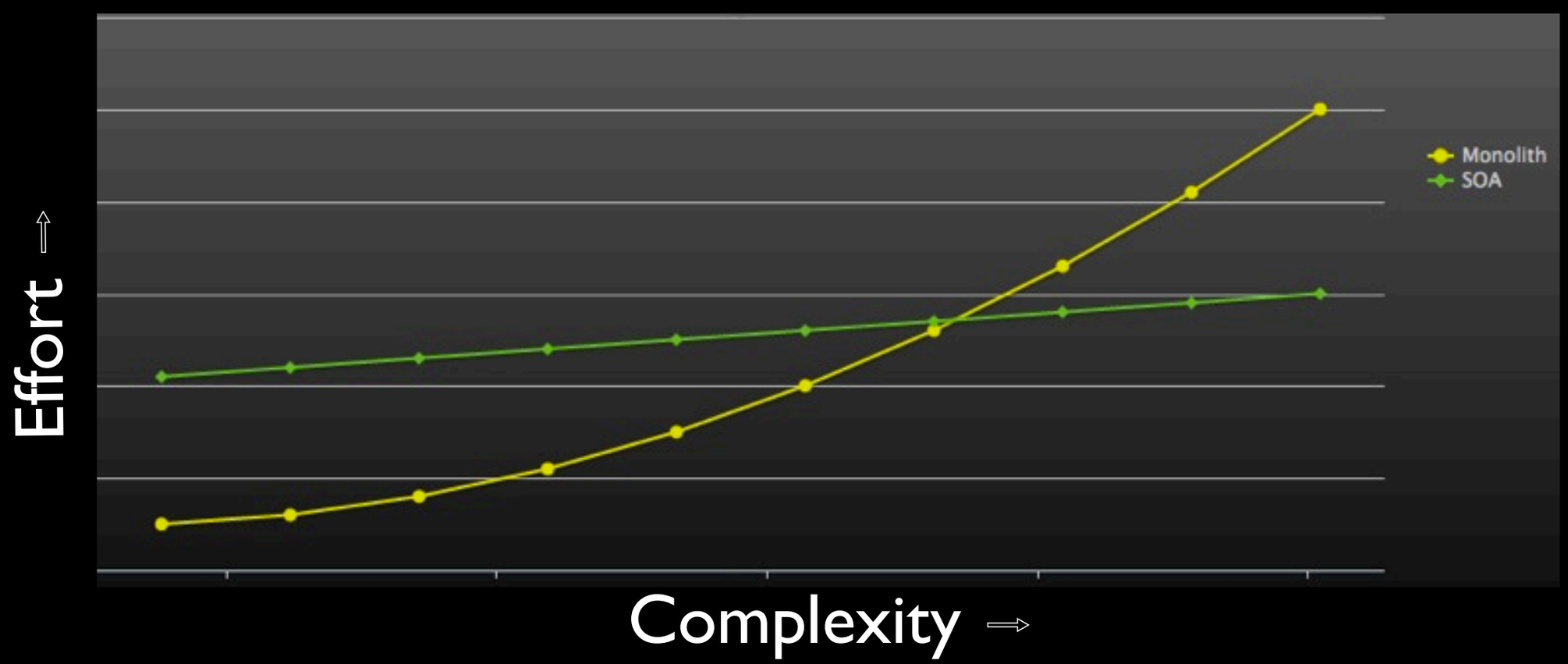
So, what do these costs look like? I've attempted to model them here. Complexity is my hand-wavy aggregate of the number of people working on a project and LOC. Effort, is the cost of building and maintaining a project.

We can see that the monolithic application's effort grows quadratically. The classic wisdom here is from Mythical Man Month - the effort needed to communicate within a team grows at a rate of n(n-1)/2, we also include the difficulty of the project as D.

To contrast that, building systems has some significant overhead. Systems programs are supposedly 3x more expensive than creating the product. Because we're building a small, focused service, the number of people and LOC are bounded, and that's the key to keeping effort growing linearly. It will always start higher than the monolithic approach, and it will always end lower.

In reality, there's a missing factor in this second equation to account for the complexity of the entire system of systems, but that factor would be based on the service dependency graph of a specific deployment and should be small for any given new project. The point is: as your System's complexity increases, SOA becomes much more viable.

Costs



Monolith: $\frac{x(x-1)}{2} + D$ SOA: $Cx + 3D$

So, what do these costs look like? I've attempted to model them here. Complexity is my hand-wavy aggregate of the number of people working on a project and LOC. Effort, is the cost of building and maintaining a project.

We can see that the monolithic application's effort grows quadratically. The classic wisdom here is from Mythical Man Month - the effort needed to communicate within a team grows at a rate of n(n-1)/2, we also include the difficulty of the project as D.

To contrast that, building systems has some significant overhead. Systems programs are supposedly 3x more expensive than creating the product. Because we're building a small, focused service, the number of people and LOC are bounded, and that's the key to keeping effort growing linearly. It will always start higher than the monolithic approach, and it will always end lower.

In reality, there's a missing factor in this second equation to account for the complexity of the entire system of systems, but that factor would be based on the service dependency graph of a specific deployment and should be small for any given new project. The point is: as your System's complexity increases, SOA becomes much more viable.

When is SOA worth it?

Thursday, May 24, 2012

The rest of this talk is about maximizing the velocity of a large software project - and at what points SOA becomes useful.
For many projects, becoming service oriented can be a natural progression, and we'll discuss Yammer's experience in this area. We've found that some of the ideas behind SOA, especially decentralization, make for extremely productive engineering teams.

<<pause>>

It's probably intuitive that maximizing velocity looks very different depending on the stage of your project and organization. So this is my perspective on how to maximize velocity at the different phases of the project development lifecycle.

Babies

Thursday, May 24, 2012

We all start as babies. This is the new project stage. The team is small, communication is easy, you're probably all in the same room and you can just talk to each other face to face. Everything is warm and soft and cuddly and people treat you nicely just because you're new to the world. And that’s despite the fact that you’re probably puking all over the place.

Baby Tech

- Web framework




- Database



- @Yammer
  - Rails
  - Postgres
- Normal Architecture
  - 1 Web / 1 Db

Thursday, May 24, 2012

Technologically speaking, things are also pretty ideal. Usually, you've picked your favorite web framework with a simple database backend. From an application standpoint, these frameworks can get you really far for a surprisingly long time. Velocity wise, they're absolutely worthwhile. Feature development is fast, development environments are pretty standardized. You actually save on communication costs by leaning on the convention over configuration paradigms that a lot of these frameworks offer. Yammer has leveraged the heck out of rails - I think we have one of the largest rails apps in existence.

Since we're just babies, we're mostly prototyping in this phase, so the architecture story is nothing to write home about.

Maximize Velocity

1. Get to Market
2. Find a fit

Thursday, May 24, 2012

Maximizing velocity at this stage is really the topic of another talk. It's usually a matter of getting something out into the world as quickly as possible and determining market fit. Some technical debt accumulates here and that's totally normal. You're going to try some things that will fail and collect dust in your code base. You're going to have other things that are hideous from a technology design perspective but end up as successful products. Survival is the only goal here. If somebody finds some disgusting code next year, that's a really good problem.

Once you've launched, you can move on to...the terrible twos

Terrible Twos

Thursday, May 24, 2012

Now, I’m not a father, but I hear this is actually a good thing. Our little kid is making an impression on the world, learning to express itself. At this point you've probably got customers, which is awesome. They might complain a lot, which sucks. They probably want more features. Maybe things are starting to get slow.

Two Tech

- Caching
- Feature services
  - Solr / Lucene / Redis / whatever
- @Yammer
  - Memcached
  - Solr/tsearch/Lucene
  - Redis
- Normal Architecture
  - n webs
  - Master-slave dbs

Thursday, May 24, 2012

Technology wise, Memcache or some other caching layer usually gets thrown in about here. You probably have a database slave. That's good - losing data is embarrassing.

As your feature set grows, it's likely that you'll diversify your technology stack a bit. This definitely happened at Yammer. We needed a search service, so we tried a few things. We juggled SOLR, postgres fulltext search and ended up with a Lucene application.

Velocity wise all this juggling was less than ideal. It was our first venture into services, and we learned a lot in the process. Prototyping your tools is incredibly important. Be sure you do your testing with production load, and make sure they can grow with you.

We also created another service to handle our "online now" presence feature that was backed by redis.

Organizationally, things still weren't terribly complicated. Communication wasn't that stressful, but we were definitely helped because we were building an internal collaboration tool.

Maximize Velocity

1. Best tool for job
2. Focused team & tools

Maximizing velocity during your twos is pretty pragmatic. Make sure you're using the best tools for the job. If that means you end up with a service or two, that's great. But make sure you stay focused on creating value -- We've started to approach the crossover point where SOA becomes more efficient, but we're definitely not there yet. Since we happen to have some services cropping up, let's talk about bootstrapping SOA.

Bootstrapping SOA

@Yammer -
Core Services team to own service creation & maintenance

Thursday, May 24, 2012

So far, we've been adding services only as our feature-set demands. This is the right approach and this is what I meant by SOA being a somewhat natural process. This may not look like it, but it's a fork in the road. You've started making services, and you've probably diversified your technology stack --
How are you going to back these services?
<<ENUNCIATE>>
Are you going to build monitoring infrastructure for them?
Are you going to hire dedicated people to maintain them?
Are those people going to build other services to help segment your quickly growing main application?

This was a turning point for Yammer. We decided to make a Core Services team that would own these things moving forward. The important thing to remember here is to be deliberate about your decision making. Don't over engineer too early -- be incremental and flexible in your changes.

The next stage of development is learning to speak...

"Reliability"

"Uptime"

"Disaster Recovery"

"MTTR"

"MTBF"

Learning to Speak

Thursday, May 24, 2012

We're starting to grow up at this stage. We're learning words like "Reliability", "Uptime", "disaster recovery". Things are actually starting to fail occasionally, so we're thinking about minimizing MTTR and maximizing MTBF. It's not that you didn't know these words before, it's just that you haven't had a lot of cycles to spend on them.

This step is about establishing and enabling operational processes. Your system is probably interesting enough that an operations team has come on board. So they're working on things like automating disaster recovery paths.

They're probably also spending time putting the beginnings of a complex monitoring system into place. Most companies do this, but it's not always with the intent of pursuing service oriented design.

Regardless of the reasoning, monitoring is a critical cornerstone in a service oriented world. It's obviously important to know that your services are running, but as teams and services establish SLA's there must be a system in place to ensure compliance and to immediately detect regressions or your system will inevitably degrade over time. Planning for these needs will improve your velocity as the organization and number of services grows.

Speaking Tech

- MTTR - (semi) Automated Recovery
- Monitoring
- Backups
- @Yammer
  - Smorgasbord :-(
- Normal Architecture
  - Ganglia / Nagios / Cacti / whatever
  - In-house recovery scripts

Thursday, May 24, 2012

From this point on, technological choices diverge significantly between organizations, so I'll be talking more about strategies and Yammer's implementations.

In any complicated system, there are going to be failures. You'll never be able to account for every failure mode, oftentimes it's just human error. So, rather than focus on mean time between failures, Yammer tries to optimize the mean time to recovery for our services. Generally speaking, optimizing MTTR will allow your organization to make changes more quickly and feel more confident about them. Being prepared for these failures by mitigating their impact and automating recovery is the best way to ensure a system under development keeps running smoothly.

It's becoming popular for developers to be on-call for the services that they build (at Amazon and Google among other places). At Yammer, developers script and optimize the recovery path as much as possible to enable our operations team to quickly recover service outages. Developers are still on the hook for bugs, but are typically the 2nd tier responders.

Monitoring is a complicated topic, and building a complete monitoring solution for a SOA ecosystem is a tall order. It needs to alert someone in the data center when a raid controller fails. It needs to notify the right developer when the p95 response time of their service is up today and then it has to help them track down the issue. If an upstream service is the problem, correlating those issues would be great. It may even need to be user friendly enough to be used by the support team or the public health site. There are a lot of open source tools out there -- ganglia, nagios, cacti, et al get close -- but none do everything that we'd like them to and they're all painful to look at.

Nevertheless, We use a smorgasbord of them at Yammer, and we're just starting to look into customizing them enough to make them into full-featured intuitive dashboards.

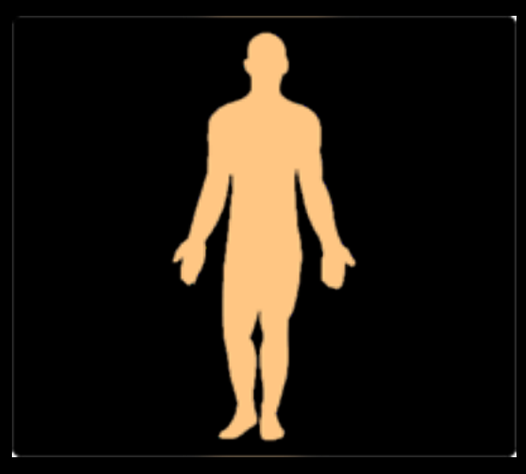
Maximize Velocity

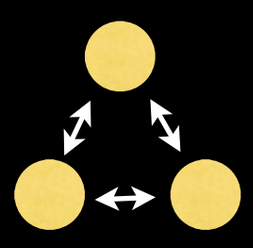
1. Monitor
2. Automate Recovery
3. Standard provisioning

As I said before, Monitoring is a cornerstone of SOA. It's super important. In order to do this effectively, you need to be able to enforce SLAs, detect regressions, and understand what your systems are doing to know where problems are coming from.

Automating recovery tasks just generally makes you more prepared to deal with inevitable failures and it will help you to overcome fear of changing production services.

The only other piece of operational advice I have to give here is to standardize your server setup. It will save you a world of hurt later as you build solutions for automated provisioning, deployments, monitoring, and sources of truth.





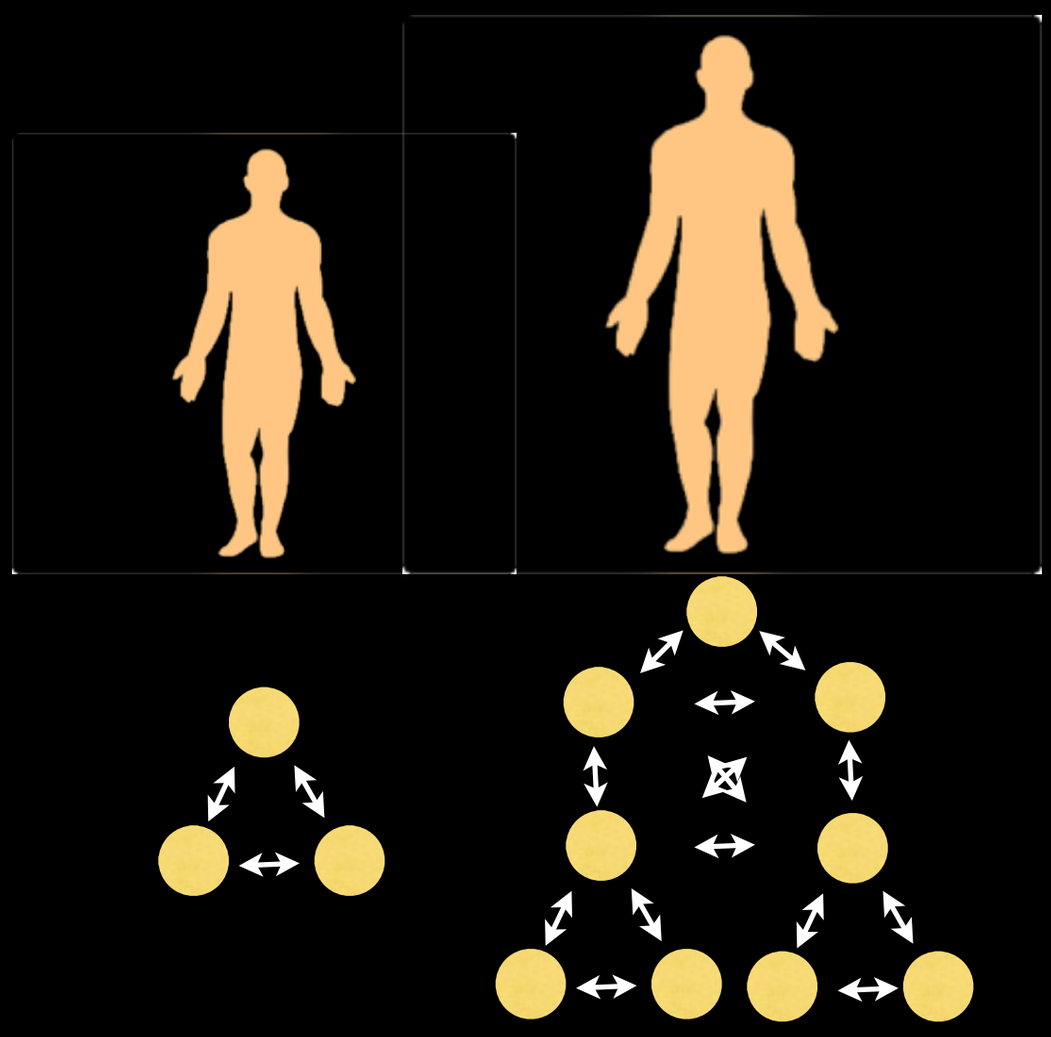
Adolescence

Thursday, May 24, 2012

At this stage you might start noticing some changes in your body. Hormones are raging through you and things are starting to get interesting. You're growing like crazy, things that ran smoothly before, like communication channels, are starting to get hairy, parts of your body/organization that you didn't spend much time thinking about before, like security and compliance, are becoming much more important to your day to day activities. Your outlook on the world is also probably changing. You realize the world doesn't revolve around you anymore, if you want to get laid and/or succeed with customers you need to start dealing with grown up issues. You also have to start planning for the future.

How an organization deals with these growing pains has an immense impact on how feasible SOA is going to be for them in the long term. This is a very good problem to have. This is the crossover point. This growth phase is an indicator that the business needs to be considering SOA - focusing too much on it beforehand will probably just slow you down and make it less likely that you'll even reach adolescence.

I would be remiss if I didn't mention the canonical SOA example, Amazon.



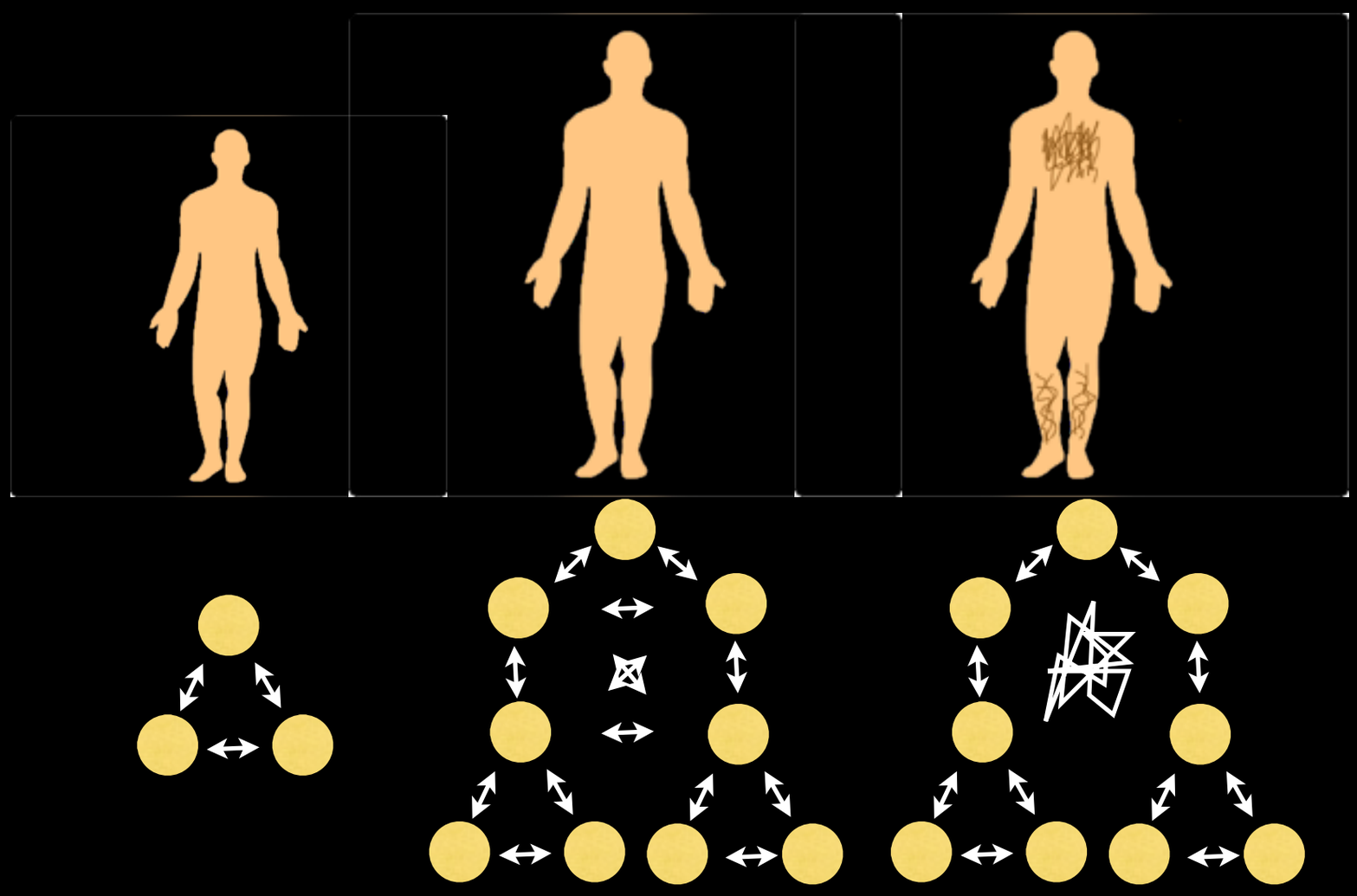
Adolescence

Thursday, May 24, 2012

At this stage you might start noticing some changes in your body. Hormones are raging through you and things are starting to get interesting. You're growing like crazy, things that ran smoothly before, like communication channels, are starting to get hairy, parts of your body/organization that you didn't spend much time thinking about before, like security and compliance, are becoming much more important to your day to day activities. Your outlook on the world is also probably changing. You realize the world doesn't revolve around you anymore, if you want to get laid and/or succeed with customers you need to start dealing with grown up issues. You also have to start planning for the future.

How an organization deals with these growing pains has an immense impact on how feasible SOA is going to be for them in the long term. This is a very good problem to have. This is the crossover point. This growth phase is an indicator that the business needs to be considering SOA - focusing too much on it beforehand will probably just slow you down and make it less likely that you'll even reach adolescence.

I would be remiss if I didn't mention the canonical SOA example, Amazon.



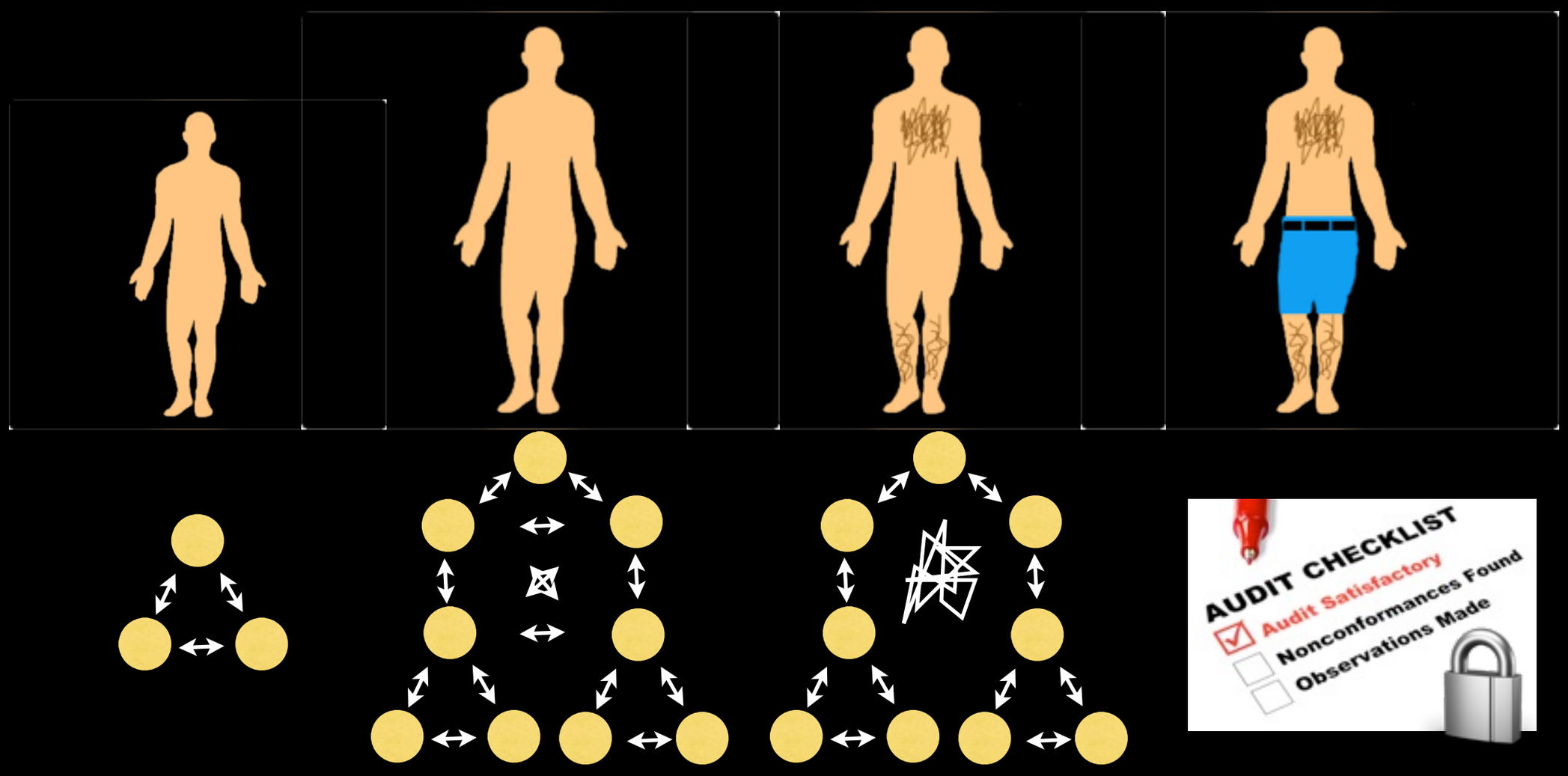
Adolescence

Thursday, May 24, 2012

At this stage you might start noticing some changes in your body. Hormones are raging through you and things are starting to get interesting. You're growing like crazy, things that ran smoothly before, like communication channels, are starting to get hairy, parts of your body/organization that you didn't spend much time thinking about before, like security and compliance, are becoming much more important to your day to day activities. Your outlook on the world is also probably changing. You realize the world doesn't revolve around you anymore, if you want to get laid and/or succeed with customers you need to start dealing with grown up issues. You also have to start planning for the future.

How an organization deals with these growing pains has an immense impact on how feasible SOA is going to be for them in the long term. This is a very good problem to have. This is the crossover point. This growth phase is an indicator that the business needs to be considering SOA - focusing too much on it beforehand will probably just slow you down and make it less likely that you'll even reach adolescence.

I would be remiss if I didn't mention the canonical SOA example, Amazon.



Adolescence

Thursday, May 24, 2012

At this stage you might start noticing some changes in your body. Hormones are raging through you and things are starting to get interesting. You're growing like crazy, things that ran smoothly before, like communication channels, are starting to get hairy, parts of your body/organization that you didn't spend much time thinking about before, like security and compliance, are becoming much more important to your day to day activities. Your outlook on the world is also probably changing. You realize the world doesn't revolve around you anymore, if you want to get laid and/or succeed with customers you need to start dealing with grown up issues. You also have to start planning for the future.

How an organization deals with these growing pains has an immense impact on how feasible SOA is going to be for them in the long term. This is a very good problem to have. This is the crossover point. This growth phase is an indicator that the business needs to be considering SOA - focusing too much on it beforehand will probably just slow you down and make it less likely that you'll even reach adolescence.

I would be remiss if I didn't mention the canonical SOA example, Amazon.

Thursday, May 24, 2012

They got to this stage and continued merrily building their massive C++/perl application. During adolescence most of us are strapped for personal wealth -- which in the computing business really means time -- so Amazon just bought some extravagantly huge databases to help them grow up. Eventually this became problematic, so they undertook the exceedingly painful process of switching to a SOA from their monolithic, single-database-backed application.

There's a lot of ways to learn from their mistakes and capitalize on their successes. From a human organizational standpoint they made small "2 pizza teams" to help keep communication overhead low. Yammer takes this a step farther. We spend a lot of time thinking about Conway's law and how to harness it, instead of letting it dictate poor design decisions.

A lot of organizations divide their departments vertically or horizontally, but we've found that this inevitably leads to silos within the organization of people working on the same thing day after day. This kind of arbitrary pigeon hole forces people to get attached to their "turf" and inhibits communication and decision making.

Thursday, May 24, 2012

They got to this stage and continued merrily building their massive C++/perl application. During adolescence most of us are strapped for personal wealth -- which in the computing business really means time -- so Amazon just bought some extravagantly huge databases to help them grow up. Eventually this became problematic, so they undertook the exceedingly painful process of switching to a SOA from their monolithic, single-database-backed application.

There's a lot of ways to learn from their mistakes and capitalize on their successes. From a human organizational standpoint they made small "2 pizza teams" to help keep communication overhead low. Yammer takes this a step farther. We spend a lot of time thinking about Conway's law and how to harness it, instead of letting it dictate poor design decisions.

A lot of organizations divide their departments vertically or horizontally, but we've found that this inevitably leads to silos within the organization of people working on the same thing day after day. This kind of arbitrary pigeon hole forces people to get attached to their "turf" and inhibits communication and decision making.

Thursday, May 24, 2012

They got to this stage and continued merrily building their massive C++/perl application. During adolescence most of us are strapped for personal wealth -- which in the computing business really means time -- so Amazon just bought some extravagantly huge databases to help them grow up. Eventually this became problematic, so they undertook the exceedingly painful process of switching to a SOA from their monolithic, single-database-backed application.

There's a lot of ways to learn from their mistakes and capitalize on their successes. From a human organizational standpoint they made small "2 pizza teams" to help keep communication overhead low. Yammer takes this a step farther. We spend a lot of time thinking about Conway's law and how to harness it, instead of letting it dictate poor design decisions.

A lot of organizations divide their departments vertically or horizontally, but we've found that this inevitably leads to silos within the organization of people working on the same thing day after day. This kind of arbitrary pigeon hole forces people to get attached to their "turf" and inhibits communication and decision making.

Cross-functional Teams

Thursday, May 24, 2012

When we're creating a new service or feature we put together a "cross-functional team" with representatives for every aspect of the project. This creates an autonomous, well-informed, decentralized design process. These teams have great discretion that yields well-designed, isolated, reusable systems. The key here is that these teams are ephemeral. They come together, solve a problem, then each team member moves on to something else. The result of this is that we have naturally occurring services whenever a cross-functional team comes together.

A good example of this is our search infrastructure.

We started with a large search application that tried to do everything. It took data from our rails app, built indexes, and made search results available. It was unwieldy and handling changing data schemas was a major pain point. The redesign of this system split up the components into 3 separate services. Flatterie for denormalization, Dexie for building indexes, and Query for the actual search interface.

By splitting up the search application into component parts, we limit complexity for each individual piece and we open the components up to reusability. Not long after the search rewrite, a different cross functional team came together to build autocomplete. Rather than expanding on Query or rebuilding all the pieces of this pipeline to satisfy their needs, they just leveraged the work Dexie was doing for the autocomplete indexes.

Later, when yet another cross functional team was working on a data export feature, they put together a service called Slurpie that leveraged Flatterie to handle the work of denormalization.

We've ended up with a SOA without anyone having to consciously design it, which is pretty elegant.

Cross-functional Teams

Search Infrastructure

Thursday, May 24, 2012

When we're creating a new service or feature we put together a "cross-functional team" with representatives for every aspect of the project. This creates an autonomous, well-informed, decentralized design process. These teams have great discretion that yields well-designed, isolated, reusable systems. The key here is that these teams are ephemeral. They come together, solve a problem, then each team member moves on to something else. The result of this is that we have naturally occurring services whenever a cross-functional team comes together.

A good example of this is our search infrastructure.

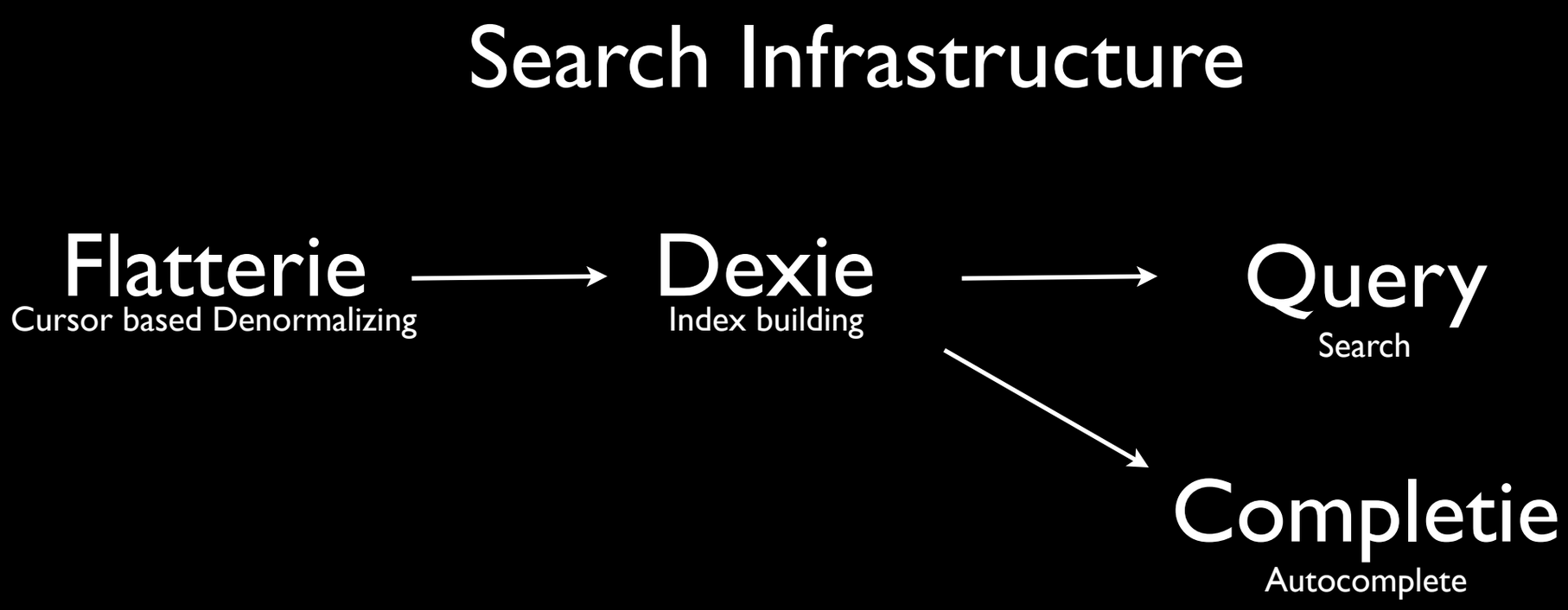
We started with a large search application that tried to do everything. It took data from our rails app, built indexes, and made search results available. It was unwieldy and handling changing data schemas was a major pain point. The redesign of this system split up the components into 3 separate services. Flatterie for denormalization, Dexie for building indexes, and Query for the actual search interface.

By splitting up the search application into component parts, we limit complexity for each individual piece and we open the components up to reusability. Not long after the search rewrite, a different cross functional team came together to build autocomplete. Rather than expanding on Query or rebuilding all the pieces of this pipeline to satisfy their needs, they just leveraged the work Dexie was doing for the autocomplete indexes.

Later, when yet another cross functional team was working on a data export feature, they put together a service called Slurpie that leveraged Flatterie to handle the work of denormalization.

We've ended up with a SOA without anyone having to consciously design it, which is pretty elegant.

Cross-functional Teams



Thursday, May 24, 2012

When we're creating a new service or feature we put together a "cross-functional team" with representatives for every aspect of the project. This creates an autonomous, well-informed, decentralized design process. These teams have great discretion that yields well-designed, isolated, reusable systems. The key here is that these teams are ephemeral. They come together, solve a problem, then each team member moves on to something else. The result of this is that we have naturally occurring services whenever a cross-functional team comes together.

A good example of this is our search infrastructure.

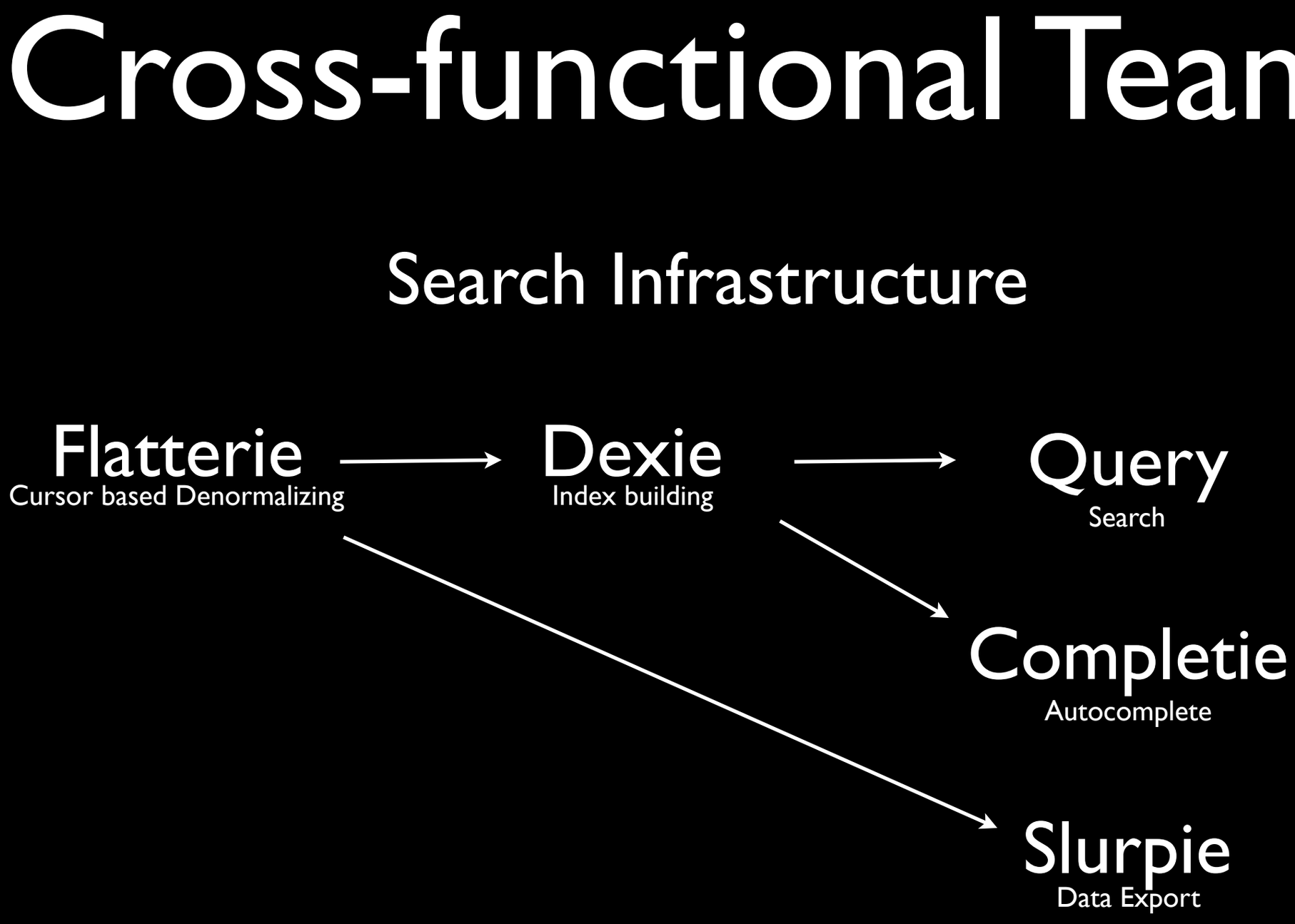
We started with a large search application that tried to do everything. It took data from our rails app, built indexes, and made search results available. It was unwieldy and handling changing data schemas was a major pain point. The redesign of this system split up the components into 3 separate services. Flatterie for denormalization, Dexie for building indexes, and Query for the actual search interface.

By splitting up the search application into component parts, we limit complexity for each individual piece and we open the components up to reusability. Not long after the search rewrite, a different cross functional team came together to build autocomplete. Rather than expanding on Query or rebuilding all the pieces of this pipeline to satisfy their needs, they just leveraged the work Dexie was doing for the autocomplete indexes.

Later, when yet another cross functional team was working on a data export feature, they put together a service called Slurpie that leveraged Flatterie to handle the work of denormalization.

We've ended up with a SOA without anyone having to consciously design it, which is pretty elegant.

Cross-functional Teams



Thursday, May 24, 2012

When we're creating a new service or feature we put together a "cross-functional team" with representatives for every aspect of the project. This creates an autonomous, well-informed, decentralized design process. These teams have great discretion that yields well-designed, isolated, reusable systems. The key here is that these teams are ephemeral. They come together, solve a problem, then each team member moves on to something else. The result of this is that we have naturally occurring services whenever a cross-functional team comes together.

A good example of this is our search infrastructure.

We started with a large search application that tried to do everything. It took data from our rails app, built indexes, and made search results available. It was unwieldy and handling changing data schemas was a major pain point. The redesign of this system split up the components into 3 separate services. Flatterie for denormalization, Dexie for building indexes, and Query for the actual search interface.

By splitting up the search application into component parts, we limit complexity for each individual piece and we open the components up to reusability. Not long after the search rewrite, a different cross functional team came together to build autocomplete. Rather than expanding on Query or rebuilding all the pieces of this pipeline to satisfy their needs, they just leveraged the work Dexie was doing for the autocomplete indexes.

Later, when yet another cross functional team was working on a data export feature, they put together a service called Slurpie that leveraged Flatterie to handle the work of denormalization.

We've ended up with a SOA without anyone having to consciously design it, which is pretty elegant.

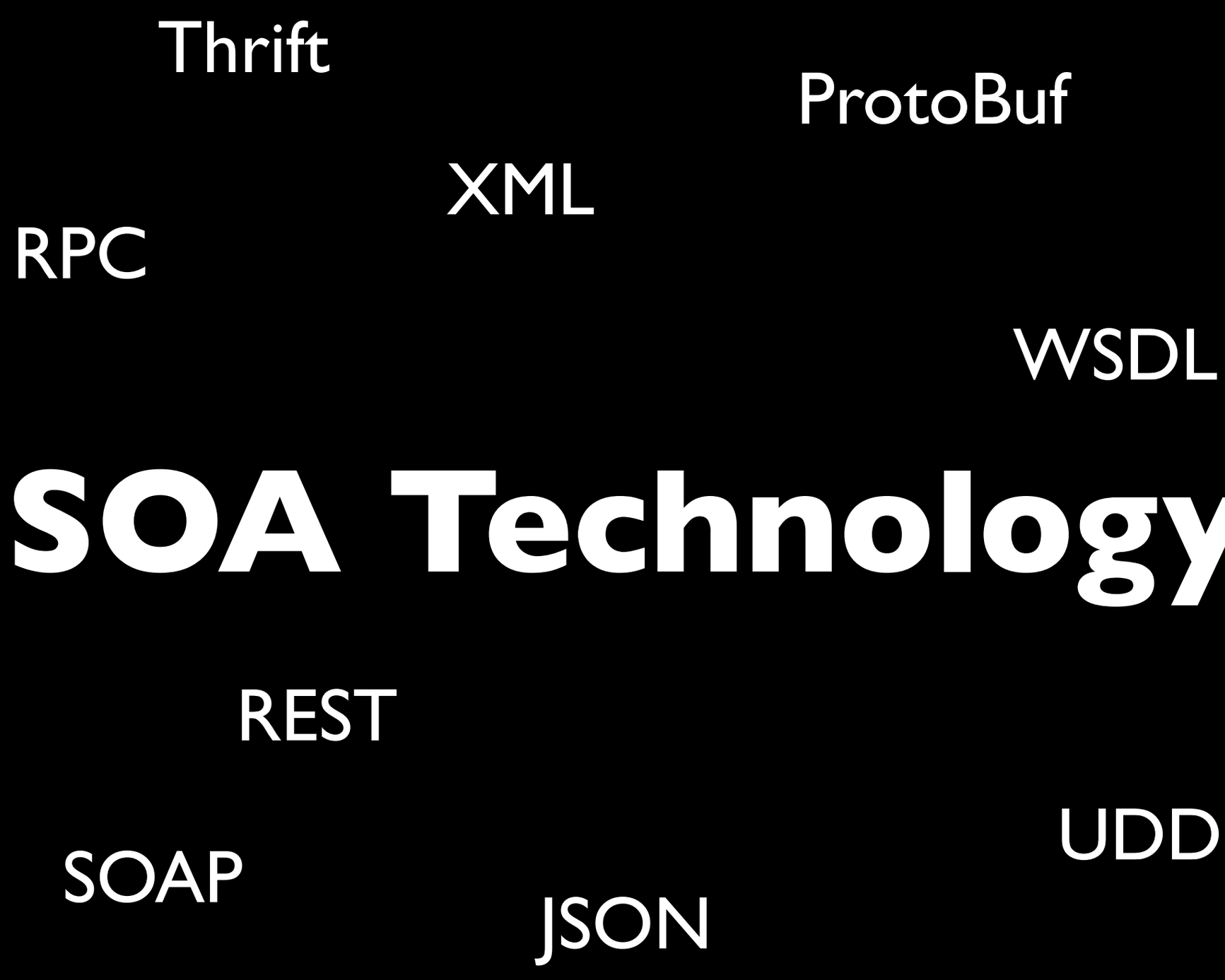
DISTRUST

Thursday, May 24, 2012

Speaking of trust, trust is a big deal in a service oriented world. As a developer, when you're dealing with a remote service, a certain distrust is healthy because it forces you to deal with error conditions more strictly than you otherwise might, which makes your system more robust.

From a velocity standpoint, trust is essential. You need to be able to trust your coworkers to make modifications to your code. Cross functional teams help us avoid some of the protectionism that is common in many organizations. At Yammer every engineer has completely unfettered access to any service. This helps to keep us moving quickly. Nobody is waiting on someone else to do something for them.

That level of trust also leads to happy, responsible engineering teams.



Thursday, May 24, 2012

This is also a good time to be thinking about the technology aspects of SOA. What are your services going to look like in production?

There's a lot of wisdom out there that says SOA is great with heterogeneous systems so services should be protocol independent and everyone can pick whatever protocols make them happy. That's a load of crap. If you can avoid that complexity, you should. Services can get the same velocity wins from convention over configuration that we saw with our early framework choices. There's no need to create multiple interface types for different services. That doesn't adding any value for the customers. Standardization here is great.

Having the same response formats, data protocols, monitoring interfaces, deployment stories, and dependency management systems will make your world infinitely easier than trying to manage tons of unique and special snowflakes that all speak different languages. Homogeneity is your friend. All of those tools will also help you maximize your long-term velocity by making it easier to get your new services to production quickly.

Adolescent Tech

- Varies widely - Tooling, Provisioning
- @Yammer
  - Dropwizard - https://github.com/codahale/dropwizard - Jetty, Jersey, Jackson, Metrics, Guava, Log4j, Hibernate Validator
    - Quickly deploy monitored, alerting, metrics reporting, logging fat Jars
  - Diploymacy - Deploy artifacts from CI
  - Partie - Framework for distributed BDB-HA KV store

Thursday, May 24, 2012

Adolescent technology varies widely, but it's mostly focused on tooling and provisioning.

At Yammer, we have an opinionated service framework called dropwizard. This is actually open source, it's available on Github and in use by a few other organizations. It combines many of the libraries and tools that we've found useful while building java sevices. It allows a new developer to deploy a production-ready, simple service that's monitored, alerting, reporting metrics, and logging in under 30 minutes.

We have a deployment tool for these services too, that we call Diploymacy. Any developer can deploy passing builds of any service from our continuous integration tool right into production.

We've also abstracted a distributed storage framework called Partie that's used by a number of different services. That's a distributed key value store based on a berkeley database high availability java edition.

Maximize Velocity

1. Standardize tools
2. Minimize time to production

Thursday, May 24, 2012

Maximizing Velocity during adolescence is really about investing in your future. Build tools that make it easy to build new services and you'll find that it becomes natural to create them. If it's not easy to build services, developers on a deadline will just add to your monolith, and that's not ideal.

Making this easy also improves your time to production. And you're not producing value if you're not in production.

And now we're on to the final stage, adulthood.

Adulthood

Thursday, May 24, 2012

Congratulations! You've made it! But adulthood isn't all fun and games. This is the longest, hardest part of the SOA lifecycle. It turns out that responsibility is a bitch. For the first time, you'll encounter problems that haven't been seen before that need custom solutions. You'll need to stay informed, pay the bills, and raise your family into well-monitored, independent functional units that maintain their service level agreements.

Adult Tech

- @Yammer - Specialized Domain Services
- Feedie - 10+ Billion messages, 45 node cluster
  - http://vimeo.com/41062751
- Streamie - Activity stream deliveries, Riak
  - http://basho.com/blog/technical/2011/03/28/Riak-and-Scala-at-Yammer/

Thursday, May 24, 2012

This is where the technology story tends to get very interesting. I won't even attempt to make sweeping generalizations about tech here because every company is different and will require different solutions for their domain.

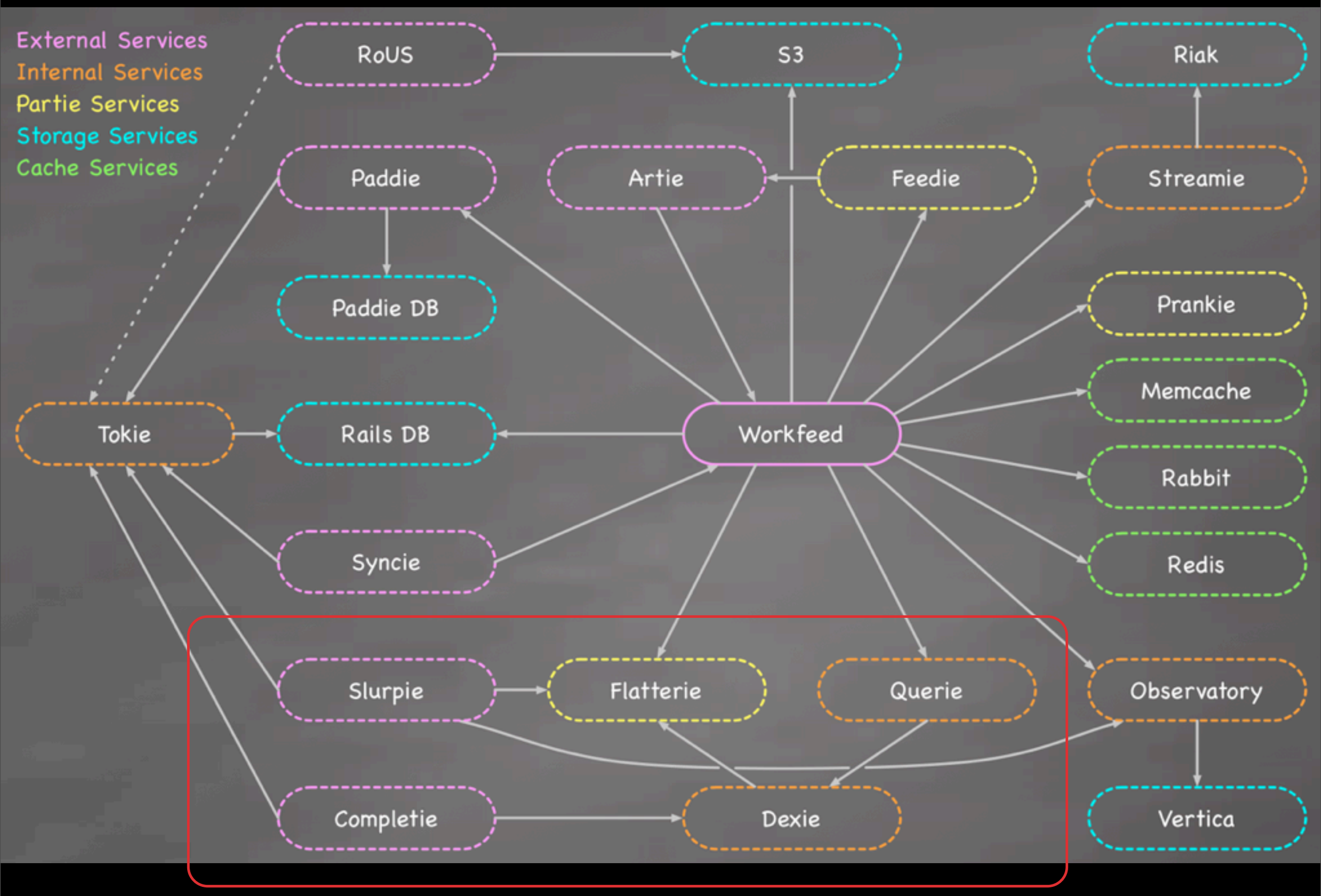
At Yammer, we still have a massive Rails app, but we've started componentizing, and continue to add new services constantly.

We have a couple of cools ones. Feedie is a distributed feed storage system that holds more than 11 billion messages. It uses our Partie framework on a 45 node cluster and distributes delivery of messages.

Streamie is our distributed activity stream store. It’s backed by Riak.

We’ve given public talks about both of these at Basho events, so check them out if you’re interested.

I don’t really want to go through the whole thing, but I thought this might be interesting -- This is a dependency diagram for services at Yammer.



I've highlighted the search stack we talked about earlier. Streamie and feedie are up in the upper right.

We have a few other services strewn about for authentication, ranking, realtime message deliveries, collaborative document editing, analytics, etc - And there are more being built constantly. If you want to take a closer look at this or discuss it in more detail just come find me later.

Tokie - OauthToken based authentication
Rous - file uploads
Paddie - realtime document editing
Syncie - directory sync
Slurpie - data export
Completie - autocomplete
Dexie - index building services
Flatterie - Cursor based data denormalization
Querie - Search
Observatory - event aggregation
Prankie - Ranking service
Artie - Realtime delivery
Feedie - Message feed delivery / storage

SOA Setup Tips

- Internalize SOA
- Direct, Decentralize, Divide & Conquer
- Monitor Everything
- Limit dependencies
- Handle Failure Modes
- Keep it simple

Thursday, May 24, 2012

I want to make sure you leave with some key points, so here they are.

When you're setting up a service oriented architecture it needs to be internalized. Make it part of the culture of the organization and try to make it as natural as possible.
Have a unified global direction and execute it in a trusting, distributed fashion. It's the only way to build a SOA ecosystem organically.
Monitor Everything. Know what's going on in your individual systems and in your ecosystem.
Limit dependencies wherever possible, it helps to keep your mental models manageable.
Handle failures and always, always try to return something useful to the user
Finally, and most importantly, make your systems as simple as possible, but not simpler.

<<PAUSE>>

is hiring

Mike Ihbe
Senior Infrastructure Engineer
twitter.com/mikeihbe
github.com/mikejihbe

Lastly, my obligatory shameless plug. Yammer is hiring and it's an incredible place to work.

That's me, find me online or come talk to me after, I'd be delighted to speak with you.

Thank you!

<<PAUSE>>