highscalability.com

Lessons Learned from Scaling Uber to 2000 Engineers, 1000 Services, and 8000 Git repositories - High Scalability -

23-29 minutes

For a visual of the growth Uber is experiencing take a look at the first few seconds of the above video. It will start in the right place. It's from an amazing talk given by Matt Ranney, Chief Systems Architect at Uber and Co-founder of Voxer: What I Wish I Had Known Before Scaling Uber to 1000 Services (slides).

It shows a ceaseless, rhythmic, undulating traffic grid of growth occurring in a few Chinese cities. This same pattern of explosive growth is happening in cities all over the world. In fact, Uber is now in 400 cities and 70 countries. They have over 6000 employees, 2000 of whom are engineers. Only a year and half a go there were just 200 engineers. Those engineers have

produced over 1000 microservices which are stored in over 8000 git repositories.

That's crazy 10x growth in a crazy short period of time. Who has experienced that? Not many. And as you might expect that sort of unique, compressed, fast paced, high stakes experience has to teach you something new, something deeper than you understood before.

Matt is not new to this game. He was co-founder of Voxer, which experienced its own rapid growth, but this is different. You can tell while watching the video Matt is trying to come to terms with what they've accomplished.

Matt is a thoughtful guy and that comes through. In a recent interview he says:

> And a lot of architecture talks at QCon and other events left me feeling inadequate; like other people-like Google for example - had it all figured out but not me.

This talk is Matt stepping outside of the maelstrom for a bit, trying to make sense of an experience, trying to figure it all out. And he succeeds. Wildly.

It's part wisdom talk and part confessional. "Lots of

mistakes have been made along the way," Matt says, and those are where the lessons come from.

The scaffolding of the talk hangs on WIWIK (What I Wish I Had Known) device, which has become something of an Internet meme. It's advice he would give his naive, one and half year younger self, though of course, like all of us, he certainly would not listen.

And he would not be alone. Lots of people have been critical of Uber (HackerNews, Reddit). After all, those numbers are really crazy. Two thousand engineers? Eight thousand repositories? One thousand services? Something must be seriously wrong, isn't it?

Maybe. Matt is surprisingly non-judgemental about the whole thing. His mode of inquiry is more questioning and searching than finding absolutes. He himself seems bemused over the number of repositories, but he gives the pros and cons of more repositories versus having fewer repositories, without saying which is better, because given Uber's circumstances: how do you define better?

Uber is engaged in a pitched world-wide battle to build a planetary scale system capable of capturing a winner-takes-all market. That's the business model. Be

the last service standing. What does better mean in that context?

Winner-takes-all means you have to grow fast. You could go slow and appear more ordered, but if you go too slow you'll lose. So you balance on the edge of chaos and dip your toes, or perhaps your whole body, into chaos, because that's how you'll scale to become the dominant world wide service. This isn't a slow growth path. This a knock the gate down and take everything strategy. Think you could do better? Really?

Microservices are a perfect fit for what Uber is trying to accomplish. Plug your ears, but it's a Conway's Law thing, you get so many services because that's the only way so many people can be hired and become productive.

There's no technical reason for so many services. There's no technical reason for so many repositories. This is all about people. mranney sums it up nicely:

> Scaling the traffic is not the issue. Scaling the team and the product feature release rate is the primary driver.

A consistent theme of the talk is *this or that is great, but there are tradeoffs, often surprising tradeoffs that you really only experience at scale*. Which leads to two of

the biggest ideas I took from the talk:

- **Microservices are a way of replacing human communication with API coordination**. Rather than people talking and dealing with team politics it's easier for teams to simply write new code. It reminds me of a book I read long ago, don't remember the name, where people lived inside a Dyson Sphere and because there was so much space and so much free energy available within the sphere that when any group had a conflict with another group they could just splinter off and settle into a new part of the sphere. Is this better? I don't know, but it does let a lot of work get done in parallel while avoiding lots of people overhead.
- **Pure carrots, no sticks**. This is a deep point about the role of command and control is such a large diverse group. You'll be tempted to mandate policy. *Thou shalt log this way*, for example. If you don't there will be consequences. That's the stick. Matt says don't do that. Use carrots instead. Any time the sticks come out it's bad. So no mandates. The way you want to handle it is provide tools that are so obvious and easy to use that people wouldn’t do it any other way.

  This is one of those talks you have to really watch to understand because a lot is being communicated along

dimensions other than text. Though of course I still encourage you to read my gloss of the talk :-)

Stats (April 2016)

- Uber is in 400 cities worldwide.
- 70 countries.
- 6000+ employees
- 2000 engineers (a year and half ago there were 200)
- 1000 services (number is approximate)
- Number of different services changes so rapidly it's hard to get an accurate count of the actual number of services in production. Lots of teams are building lots of things. Don't even care about the actual number.
- Over 8000 repos in git.

Microservices

- It's great to break up all your monoliths into smaller things. Even the name monolith sounds bad. But microservices have their bad side.
- The time when things are most likely to break is when you change them. Uber is most reliable on the weekends when engineers aren't making changes even

though that's when Uber is the busiest.

- Everytime you go to change something you risk breaking it. Does it make sense at some point to never touch a microservice? Maybe.
- The Good
- It's worth questioning, why are we throwing so much effort into microservices? It's not by accident, there are a lot of good results.
- Microservices allow teams to be formed quickly and run independently. People are being added all the time. Teams need to be formed quickly and put to work on something where they can reason about the boundaries.
- Own your own uptime. You run the code that you write. All the service teams are on call for the services they run in production.
- Use the best tool for the job. But best in what way? Best to write? Best to run? Best because I know it? Best because there are libraries? When you dig into it, best doesn't mean a lot.
- The Obvious Costs
- What are the costs of running a big Microservices

deployment?

- Now you are running a distributed system, which is way harder to work with than a monolith.
- Everything is an RPC. You have to handle all these crazy failure modes.
- What if breaks? How do you troubleshoot? How do you you figure out where in the chain of services the break occurred? How do make sure the right people get paged? The right corrective actions are taken fix the problem?
- These are still all the obvious costs.
- The Less Obvious Costs
- Everything is a tradeoff, even if you don't realize you are making it. In exchange for all these microservices you get something, but you give up something too.
- By super-modularizing everything we introduce some subtle and non obvious problems.
- You might choose to build a new service instead of fix something that's broken. At some point the cost of always building around problems and cleaning up old problems starts to be a factor.
- **You find yourself trading complexity for politics**.

Instead of having awkward conversations, laden with human emotions, you can just write more software and avoid talking.

- **You get to keep your biases**. If you like Python and the team you are interfacing with likes node and instead of working in another code base you can just build new stuff in your own preferred language. You get to keep doing what you think is best even though that might not be the best thing for the organization or the system as a whole.

The Cost of Having Lots of Languages

- Prehistory: At first Uber was 100% outsourced. It didn't seem like a technology problem so some company wrote the first version of the mobile app and the backend.
- Dispatch: When development was brought in house it was written in Node.js and is now moving to Go.
- Core Service: the rest of the system, was originally written in Python and is now moving to Go.
- Maps was eventually brought in house and those teams are using Python and Java.
- The Data Engineering team writes their code in Python

and Java.

- The in-house Metric system is written in Go.
- You start to notice that's a lot of languages. Microservices allow you to use lots of languages.
- Teams can write in different languages and still communicated with each other. It works, but there are costs:
- Hard to share code.
- Hard to move between teams. Knowledge built up on one platform doesn't transfer to another platform. Anyone can learn of course, but there's a switching cost.
- **What I Wish I Knew**: having multiple languages can fragment culture. By embracing the microservices everywhere you can end up with camps. There's a node camp, a Go camp, etc. It's natural, people organize around tribes, but there's a cost to embracing the strategy of having lots of languages everywhere.

The Cost of RPC

- Teams communicate with each other using RPCs.
- At scale with lots and lots of people joining really

quickly the weaknesses of HTTP start to show up. Like what are status codes for? What are headers for? What goes in the query string? Is this RESTful? What method is it?

- All these things that seem really cool when doing browser programming, but become very complicated when doing server programming.
- What you really want to say is run this function over there and tell me what happened. Instead, with HTTP/REST you get all these subtle interpretation issues, all this is surprisingly expensive.
- JSON is great, you can look at it with your eyeballs and read it, but without types it's a crazy mess, but not right away, the problems pop up later. When someone changes something and a couple of hops downstream they were depending on some subtle interpretation of empty string versus null, or some type coercion in one language versus another, it would cause a huge mess that would take forever to sort out. Types on interfaces would have fixed all these of the problems.
- RPCs are slower than procedure calls.
- **What I Wish I Knew**: servers are not browsers.
- When talking across a datacenter it makes a lot more

sense to treat everything like a function call rather than a web request. When you control both sides of a conversation you don't need all the extra browsery stuff.

Repositories

- How many repos are best? He thought one was best, but many disagreed.
- Many people think many repos is best. Maybe one per project or even multiple per project.
- Having many repos follows the industry trend of having many small modules. Small modules are easy to open source or swap out.
- One repo is great because you can make cross cutting changes. It you want to make a change it's easy to hit all the code that needs to be changed. It's also easy to browse the code.
- Many is bad because it's going to stress out your build system. It hurts your ability to navigate the code. It's painful to make sure cross cutting changes are done correctly.
- One is bad because it's going to get so big you won't be able to build or even checkout your software unless

you have some crazy elaborate system on top. One repo is probably not usable without special tooling. Google has one repo but it uses a virtual file system to make it seem like you have the whole repo checked out.

- Over 8000 repos in git. One month ago there were 7000.
- Some individuals have their own repos.
- Some teams track service configuration separately from the service itself using a separate repo.
- But the majority are production repos.
- That's a lot repos.

Operational Issues

- What happens when things break? There are some surprising issues that come up with a large microservices deployment.
- If other teams are blocked on your service and that service isn't ready to be released, is it OK for the other teams to put a fix into your service and release it?
- Is owning your own uptime compatible with other teams releasing your service, even if all your tests pass? Is

the automation good enough that teams can release each other's software?

- At Uber it depends on the situation. Sometimes yes, but usually the answer is no, teams will just have to blocked on the fix.
- Great, small teams, everyone is moving fast, all releasing features super quickly, but sometimes you have to understand the whole system as one thing connected together as one giant machine. That’s hard when you’ve spent all this time breaking it up into microservices.
- This is a tricky problem. Wishes more time was spent keeping that context together.
- Should still be able to understand the whole system working as one.

Performance Issues

- Performance is definitely going to come up given how dependent microservices are on one another.
- RPCs expensive, and especially when there are multiple languages the answer for how you understand your performance totally depends on the language tools and the tools are all different.

- You've let everyone program in their own language now understanding performance across those languages is a real challenge.
- Trying to have all languages have a common profiling format using Flame Graphs.
- As you want to understand the performance of the system a big point of friction when trying to chase down a performance problem is how different the tools are.
- Everyone wants a dashboard, but if the dashboards aren't generated automatically teams will just put on dashboards what they think is important so when you want to chase a problem down one team's dashboard will look completely different from another.
- Every service when created should have a standard dashboard with the same set of useful data. Should be able to create a dashboard with no work at all. Then you can browse other team's services and it will all look the same.
- **What I Wish I Knew**: Good performance is not required but you need to know where you stand.
- A big debate is if you should even care about performance. The "premature optimization is the root of all evil" type thinking has spawned a very weird

subculture of people who are against optimization. It doesn't matter, services aren't that busy. We should always optimize for developer velocity. Buying computers is cheaper than hiring engineers.

- A certain truth to this. Engineers are very expensive.
- The problem is performance doesn't matter until it does. One day you will have a performance problem and if a culture of performance doesn't matter has been established it can be very difficult to have performance suddenly matter.
- You want to have some sort of SLA that's performance based on everything that gets created just so there is a number.

Fanout Issues - Tracing

- Fanout causes a lot of performance problems.
- Imagine a typical service that 99% of the time responds in 1ms. 1% of the time it responds in one second. Still not so bad. 1% of the time users will get the slow case.
- Now let's say services start having a big fanout, calling lots of other services. The chance of experiencing slow response times goes up quickly. Use 100 services and 63% of response time ar at least 1 second (1.0 -

.99^100 = 63.4%).

- Distributing tracing is how you track down fanout problems. Without a way to understand a requests journey through the architecture it will be difficult to track down fanout problems.
- Uber is using OpenTracing and Zipkin.
- Another approach is to use logs. Each log entry has a common ID that threads all the services together.
- Example is given of a tricky case. The top level had a massive fanout all to the same service. When you look at the service it looks good. Every request is fast and is consistent. The problem was the top level service got a list of ID and was calling the service for each ID. Even concurrently that will take too long. Just use a batch command. Without tracing it would have been very hard to find this problem.
- Another example is a service that made many thousands of service calls. Though each call was fast the large number of them made the service slow. It turns out when traversing a list and changing a property it magically turned into a database request. Yet the database team says the database is working great because each operation is fast, but they will wonder

why there are so many operations.

- The overhead of tracing can change the results. Tracing is a lot of work. One option is to not trace all requests. Trace a statistically significant portion of the requests. Uber traces about 1% of requests.
- **What I Wish I Knew**: Tracing requires cross-language context propagation.
- Because all these different languages are used with all these different frameworks, getting context about the request, like what user it is, are they authenticated, what geo-fence are they in, that becomes very complicated if there's no place to put this context that will get propagated.
- Any dependent requests a service makes must propagate context even though they may not understand it. This feature would have saved so much time if it had been added a long time ago.

Logging

- With a bunch of different languages and bunch of teams and lots of new people, half the engineering team has been around for less than 6 months, everyone might tend to log in very different ways.

- Mandate is a tricky word, but that's what you really want to do, mandate a common logging approach. The more acceptable way to say it is provide tools that are so obvious and easy to use that people wouldn't do any other way in order to get consistent and structured logging.
- Multiple languages makes logging hard.
- When there are problems logging itself can make those problems worse by logging too much. Need backpressure in the log to drop log entries when overload. Wish that had been put in the system earlier.
- **What I Wish I Knew**: some notion of accounting for log message size so it can be traced who is producing too much log data.
- What happens to the all logs is they get indexed by some tool so people can search through them and learn things.
- The amount of data that gets logged, if logging is free, is quite variable. Some people will log a lot and overwhelm the system with data.
- With an accounting system when log data is sent to the cluster for indexing a bill could be sent to the service to pay for it.

- The idea is to put pressure back on developers to log smarter, not harder.
- Uber created uber-go/zap for structured logging.

Load Testing

- Want to load test before putting a service in production but there's no way to build as big a test environment as the production environment.
- Also a problem generating realistic test data to test all parts of the system.
- Solution: run tests on production during off-peak hours.
- Causes lots of problems.
- It blows up all the metrics.
- You don't want people to think there's more load than there really is. To fix the problem it gets back to the context propagation problem.
- Make sure all test traffic requests have some context that says this a test request so handle your metrics differently. That has to plumb all the way through the system.
- **What I Wish I Knew**: what we really want to do is run load through all the services all the time because a lot

of bugs only show up when traffic hits its peak.

- Want to keep systems near their peaks and the back off as real traffic increases.
- Wishes the system was long ago built to handle test traffic and account for it differently.

Failure Testing

- **What I Wish I Knew**: Not everybody likes failure testing, like Chaos Monkey, especially if you have to add it in later.
- What we should have done is made failure testing happen to you if you liked it or not. It's just part of production. Your service just has to withstand random killings and slowings and perturbations of operations.

Migrations

- All our stuff is legacy. It's all migrations. Mostly people who work on storage all they are doing is migrations from one piece of legacy to another not quite so legacy thing.
- Someone is always migrating something somewhere. That's what everyone is doing no matter what people tell you at conferences.

- Old stuff has to be kept working. The business still has to run. There's no such thing as a maintenance window any more. There's acceptable downtime.
- As you become a global business there are no off-peak times. It's always peak time somewhere.
- **What I Wish I Knew**: mandates to migrate are bad.
- Nobody wants to be told they have to adopt some new system.
- Making someone change because the organization needs to change versus offering a new system that is so much better that it will be obvious that you need to get on this new thing.
- Pure carrots no sticks. Anytime the sticks come out it's bad, unless it's security or compliance related, then it's probably OK to force people to do stuff.

Open Source

- Nobody agrees on the build/buy tradeoff. When should you build? When should you buy?
- Anything that's part of the infrastructure, anything that seems like it's part of the platform and not part of the product, at some point it's on its way to become an undifferentiated commodity. Amazon (or someone) will

offer it as a service.

- Eventually that thing you are spending all your time working on, someone else will do it for cheaper and do it better.
- **What I Wish I Knew**: if people are working on some kind of platform type feature it doesn't sound good to hear Amazon has just released your thing as a service.
- You still try to rationalize why you should use your own private thing as a service.
- It turns out there are people on the other end of those text editors. There are vast differences in how people make judgements about the build/buy tradeoff.

Politics

- By breaking everything up into small services it allows people to play politics.
- Politics happen whenever you make a decision that violates this property: Company > Team > Self.
- You put the values of your self above the team.
- The values of the team are put above the company.
- Politics isn't just a decision you don't like.
- By embracing highly modularized rapid development,

hiring people very quickly, and releasing features as quickly as you can, there's a temptation to start violating this property.

- When you value shipping things in smaller individual accomplishments it can be harder to prioritize what is better for the company. A surprising tradeoff.

Tradeoffs

- Everything is a tradeoff.
- **What I Wish I Knew**: how to better make these tradeoffs intentionally.
- When things are happening without an explicit decision because it seems like this is just the way things are going, think about what tradeoffs are being made, even if the decisions are not explicitly being made.

Related Articles

- On HackerNews
- Excellent discussions of the video on HackerNews and on Reddit
- Design Decisions For Scaling Your High Traffic Feeds
- Google On Latency Tolerant Systems: Making A

Predictable Whole Out Of Unpredictable Parts

- How Uber Scales Their Real-Time Market Platform
- Uber Goes Unconventional: Using Driver Phones As A Backup Datacenter
- How Uber Manages A Million Writes Per Second Using Mesos And Cassandra Across Multiple Datacenters
- Scaling Uber with Matt Ranney
- The InfoQ Podcast: Uber's Chief Systems Architect on their Architecture and Rapid Growth
- Distributed Web Architectures: Matt Ranney, Voxer
- Riak at Voxer - Matt Ranney, RICON2012