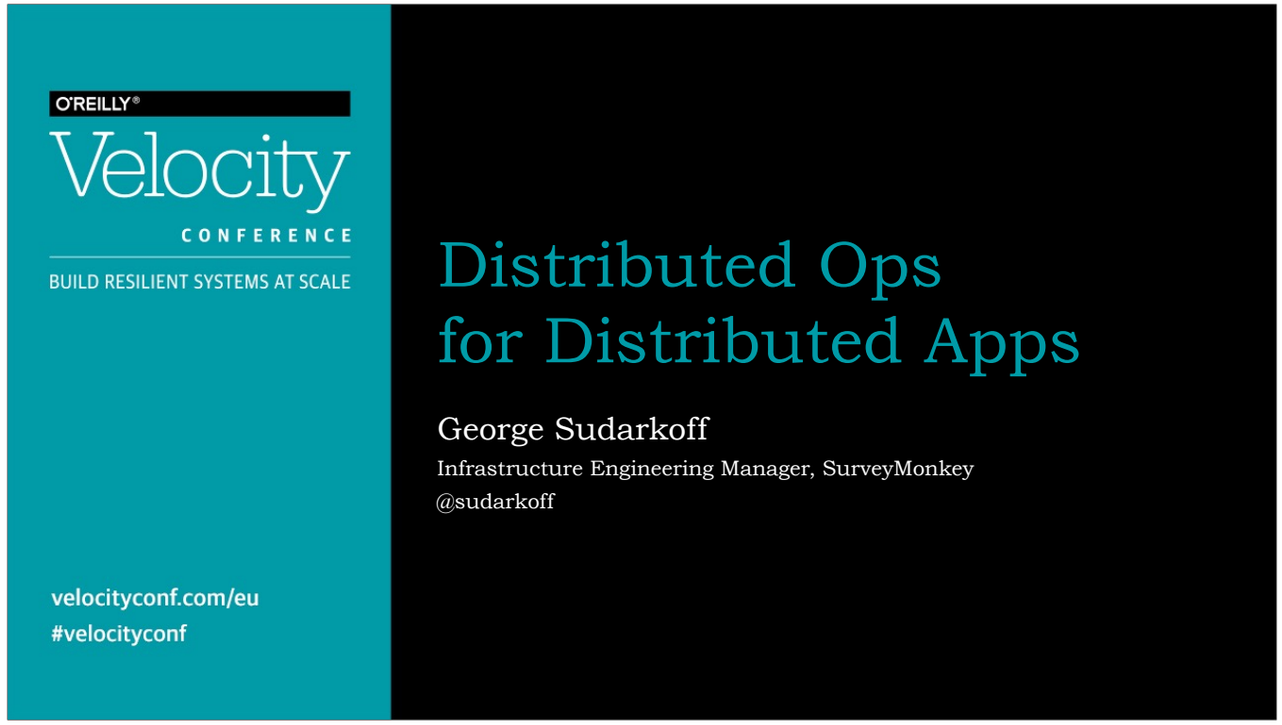
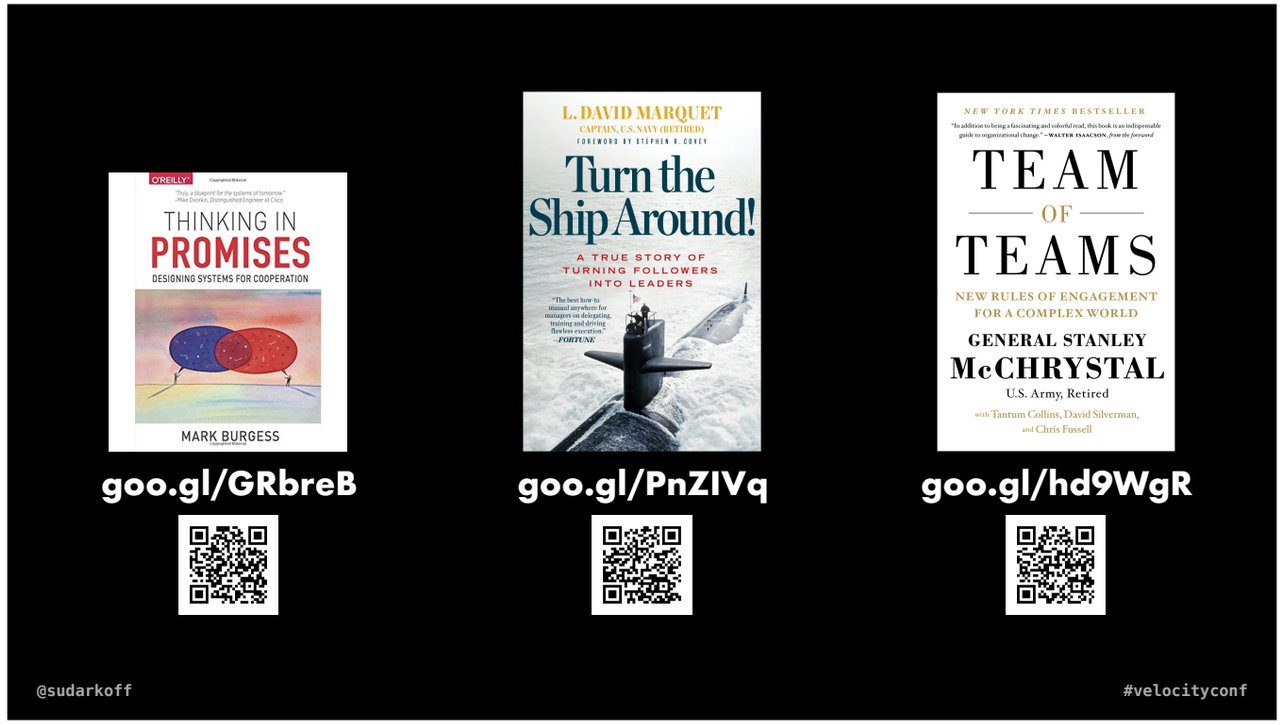


A few years ago I metamorphed into a manager and what I discovered along the way is that scaling software systems and scaling organizations is remarkably similar. The same math applies whether we're talking about distributed software or distributed organizations.

When I joined SurveyMonkey in 2014, we had ~15 teams developing ~30 Python service. To deploy those Python services we used a home-grown tool cleverly called Doula (get it? Continuous Delivery, Doula?). Doula allowed teams to build the Python package, and deploy it along with all of its Python dependencies into a virtualenv on a test or staging server. Then, another artisanal bash script was used to gzip that virtualenv from the staging server and rsync it to a handful of production servers.

This process had a number of problems. Virtualenv incapsulates the Python environment fairly ok, but any system-level dependencies, such as database drivers, had to be manually installed on the right servers at just the right time. And because production deployments relied on the staging environment being in a stable state, we had to institute deploy and code freezes. Which created a bottleneck and slowed down the development. On average teams were releasing once or twice a month.

Today, we have just over 100 services. And we're in the process of migrating them to a completely distributed pull-based configuration management system that scales sub-linearly. Engineering teams own every step of the service delivery - from coding, to provisioning the machines, deploying, responding to alerts and troubleshooting in production.



I want to point out these three books that in particular have influenced the way I think about scaling operations:

Thinking in Promises describes in plain English the Promise Theory developed by a physicist Mark Burgess. Promise Theory, according to Wikipedia, studies "voluntary cooperation between individual, autonomous actors or agents who publish their intentions to one another in the form of promises." If you're building distributed systems (or scaling an organization) and haven't read this book yet, I highly recommend that you do it as soon as possible.

Turn the Ship Around is a story of a nuclear submarine commander David Marquet, who succeeded to turn a consistently underperforming ship into one that received the highest scores ever seen in the history of the NAVY. And he has done that in less than a year by giving away control, and by creating leaders instead of followers out of his crew. It's a fascinating story! If you're going to read only one book on leadership make it this one.

Team of Teams is a book by an ex-US Army General Stanley McChrystal. In 2004 he took command of the Joint Special Operations Task Force where he quickly realized that the Army couldn't fight Al Qaeda - the highly distributed and decentralized network of guerrilla fighters - with the conventional military tactics based on the century-old command-and-control model. McChrystal and his colleagues had to abandon the old ways and to embrace the transparent communication with decentralized decision-making authority.

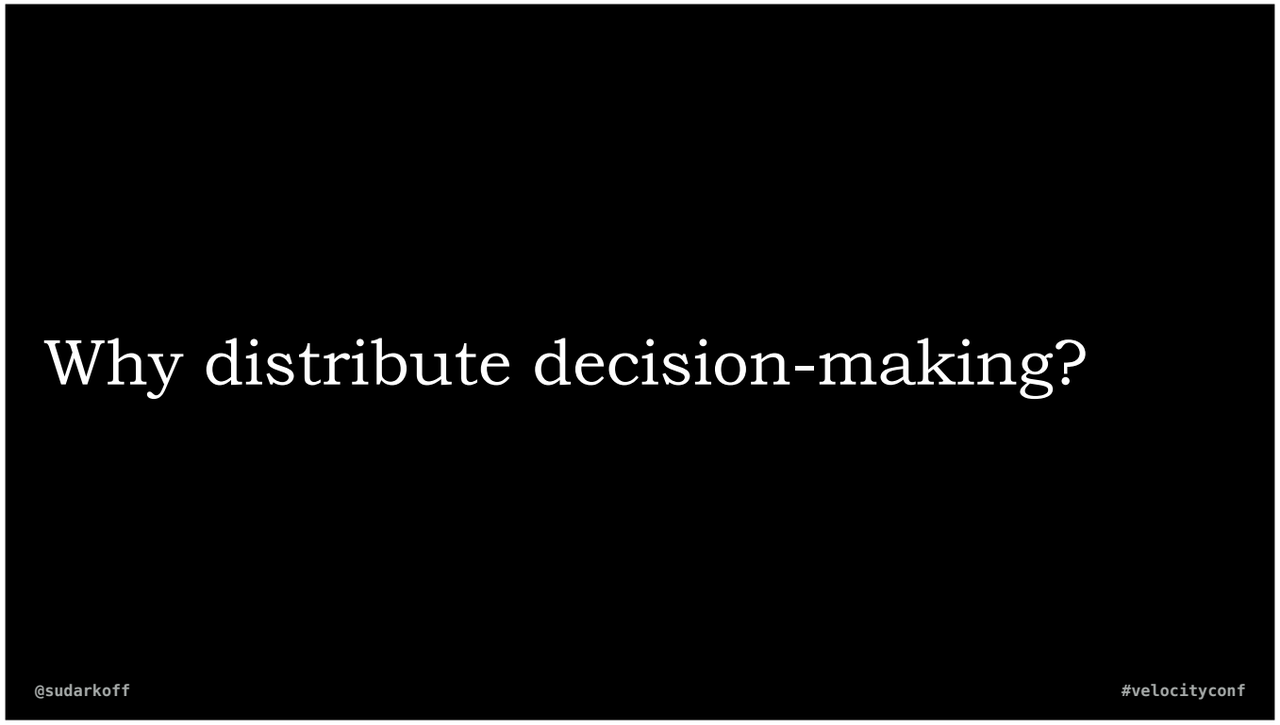
Distributed Operations

Distribution of **decision-making authority** across a number of small highly-skilled teams directly involved in service development and delivery.

@sudarkoff #velocityconf

Distributed operations is <read the slide>.

US Marines developed the concept of Distributed Operations after they recognized that they are no longer capable of fighting the highly distributed and adaptive enemies using the good old centralized command and control methods. In the world of tech we're dealing with our own highly distributed and adaptive enemies today - it's micro-services. Micro-services come with enormous benefits - otherwise we wouldn't be embracing them with such enthusiasm. But they also introduce a lot of challenges that the traditional Ops are unable to meet.

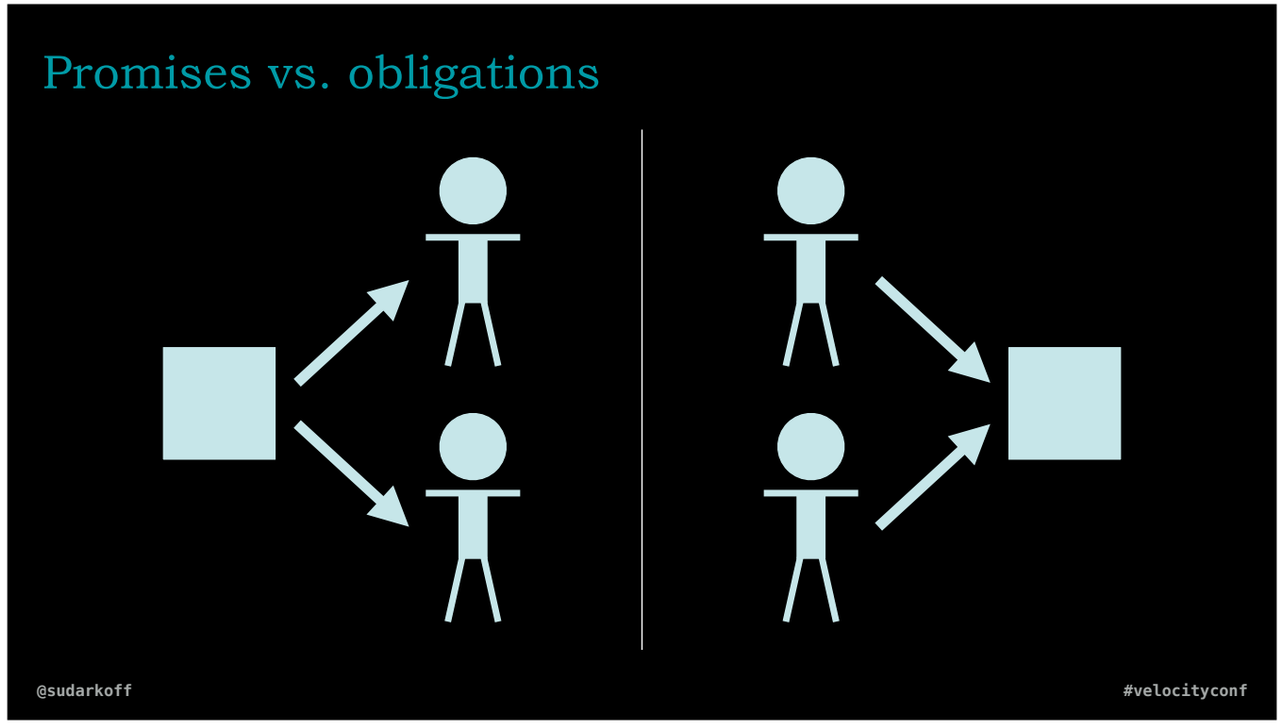


Why do we want to distribute the decision-making across many teams? Why is it better than centralizing it in a single team that lives and breathes operations and, thanks to training and daily practice, is extremely good at it.

A lot could be said about the efficiency and consistency of centralized operations. But a lot has also been said about their inability to scale. Centralized systems do not scale.

<long pause>

Because I'm going to be using the ideas from the Promise Theory in this talk, I thought I'd make a brief introduction for those of you who are not familiar with it. And at the same time hopefully demonstrate why distributed is better than centralized.



Promise Theory was proposed by Mark Burgess in 2004. It came out of realization that "command and control" model of IT management struggles to keep up as our systems grow to a massive scale often spanning multiple data centers and continents. He proposed that instead of looking at systems from the traditional and familiar obligations point of view, we look at them as systems of autonomous agents promising each other certain behavior or outcomes.

The two basic tenets of the Promise Theory is that:

1. every agent promises only its own behavior
2. independently changing parts of the system exhibit separate agency and therefore should be modeled as separate agents

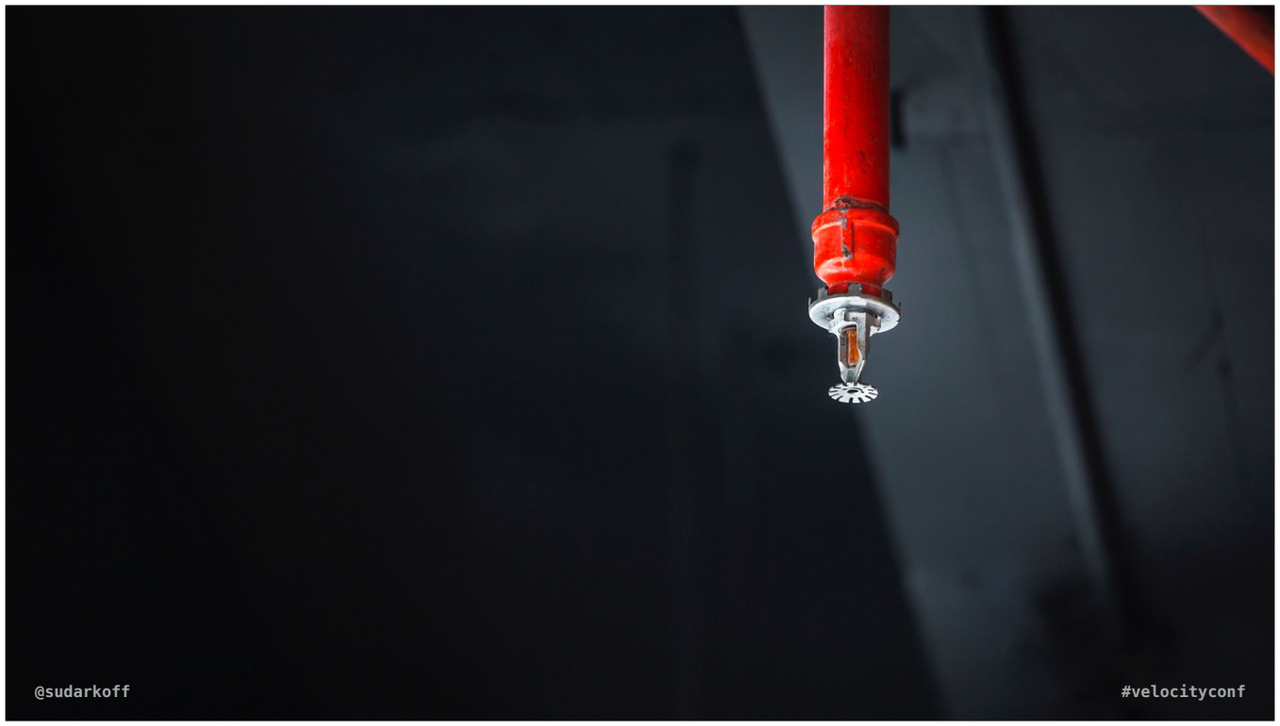
In other words, it studies distributed systems.

In contrast, the traditional obligations model is an imperative, command-and-control model, that describes a system that executes each step of a process to achieve a desired result.

One of the biggest problems of this model is that it separates the intent from the implementation. Imagine if two instructors start sending conflicting instructions to the same component. The component doesn't know which set of instructions is the correct one. The instructors are not aware of each other. The outcome of the situation is entirely uncertain.

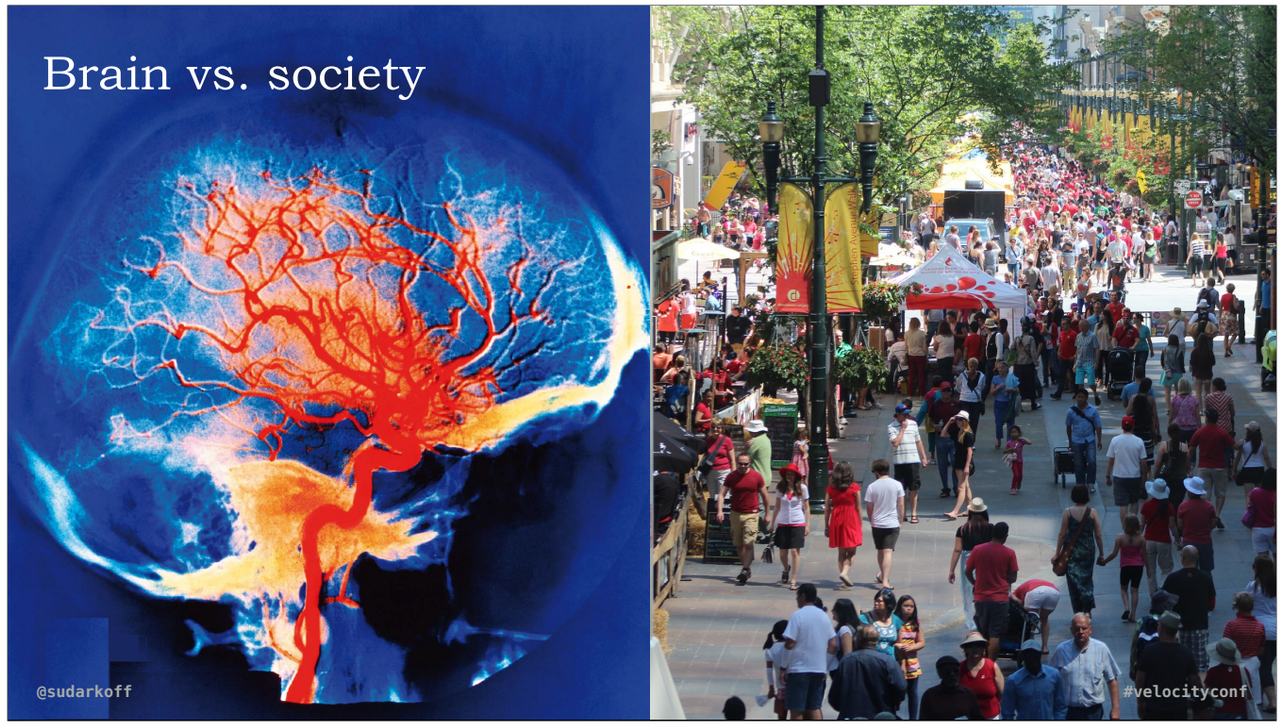
Those of you who had the dubious privilege of working in traditional ops can probably recognize this pattern. Multiple teams submit potentially conflicting requests. Ops team doesn't have enough visibility into the business priorities, so everything is presented as high priority. So they reacts by dropping any planned work and throwing all available resources onto those requests. But they still miss the deadlines because it's impossible to plan other people's work. Other teams are frustrated with slow response because Ops is now a bottleneck and a SPOF. None of their own projects get completed, technical debt grows, the team becomes overworked, burned out and everybody rage-quits.

The alternative is to distribute operations (i.e. the implementation) to where the intent is - service delivery teams (i.e. the developers).



In the model of Distributed Operations each team makes their own operational decisions. But it's important to note that decision-making autonomy does not necessarily mean that the team performs all the tasks on their own. They can still rely on another team to do a specific job. BUT! Since they cannot make promises on behalf of other teams, **in order to keep their own promises**, they better have a contingency plan in case another team breaks their promise. Either have an agreement with yet another team that performs a similar function. Or have their own perhaps scaled down version of the service.

For example, at SurveyMonkey we have CI system that continuously builds, tests and packages artifacts for all teams. But each team is capable of performing the same steps outside of our CI system, in case it becomes unavailable for whatever reason.



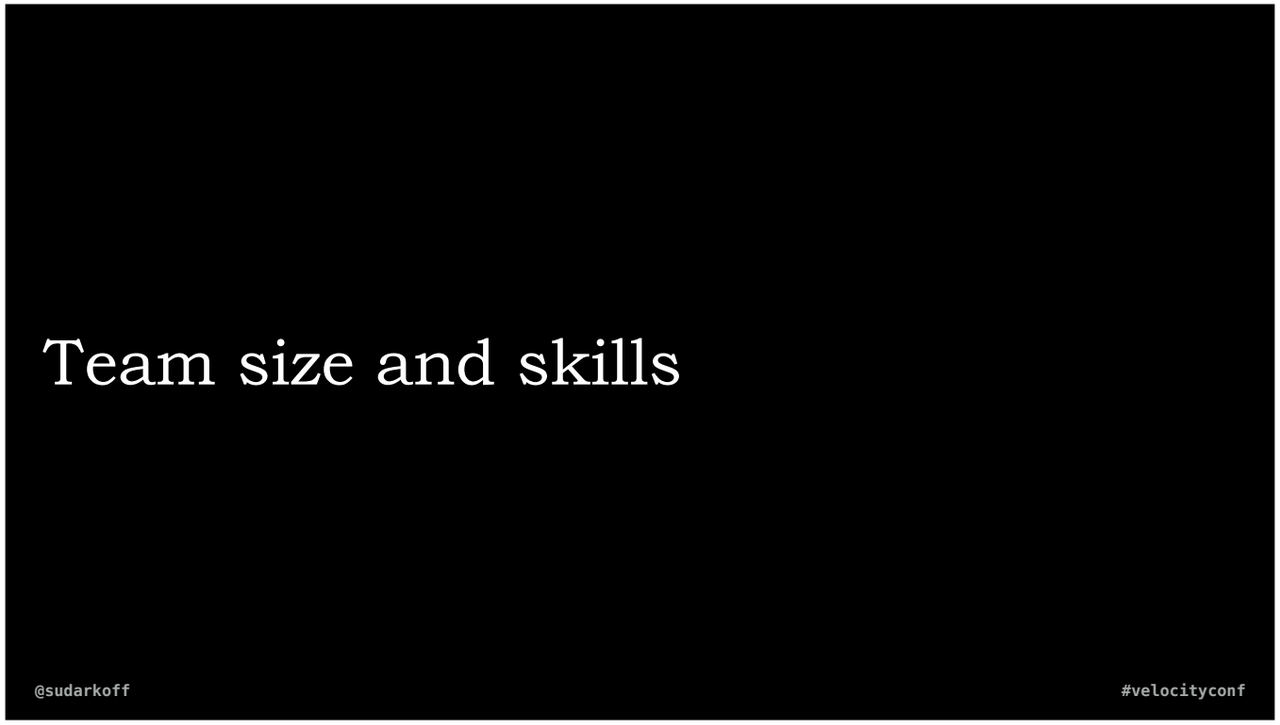
I like the analogy that Mark Burgess uses to talk about centralized vs. distributed systems.

In his analogy brain represents the centralized system: it receives information from a number of sensors through the nervous system, analyses the information and sends commands to various actuators in response. Centralized systems are typically viewed as "smart", but as they grow the time it takes for signals to reach the brain, get processed and for commands to be sent back goes up and the system becomes "slow". Slow is the opposite of smart, slow is dumb.

On the other hand, society represents a distributed system - it is a number of loosely coupled autonomous agents that cooperate to achieve their goals. Societies might include centralized components (for example, a library), but the society is capable of surviving even when a centralized component fails. In distributed systems "smart" behavior emerges at the system level. One of the examples of a distributed system slash society is our immune system. It's thousands of millions of defender cells - different types of white blood cells - that constantly patrol your body, destroying germs as soon as they enter. Each individual immune cell is fairly dumb - it "recognizes" the germs it specializes in by matching their shape like a puzzle piece. But it is a blazingly fast process. And when you combine hundreds of millions of dumb autonomous cells, you get a very "fast" and "smart" system.

How to distribute ops?

So, how do you distribute operations at your company?



One of the most common objections I hear is - does it mean that front-end developers now need to understand how to tune their linux kernels? Wouldn't that increase the scope of the engineers work and prevent them from focusing on product and satisfying their customers' needs? Wouldn't it make them less effective? You don't expect a dentist know how to do brain surgery? How about instead of requiring engineers to learn more skills we could embed specialists in the teams?

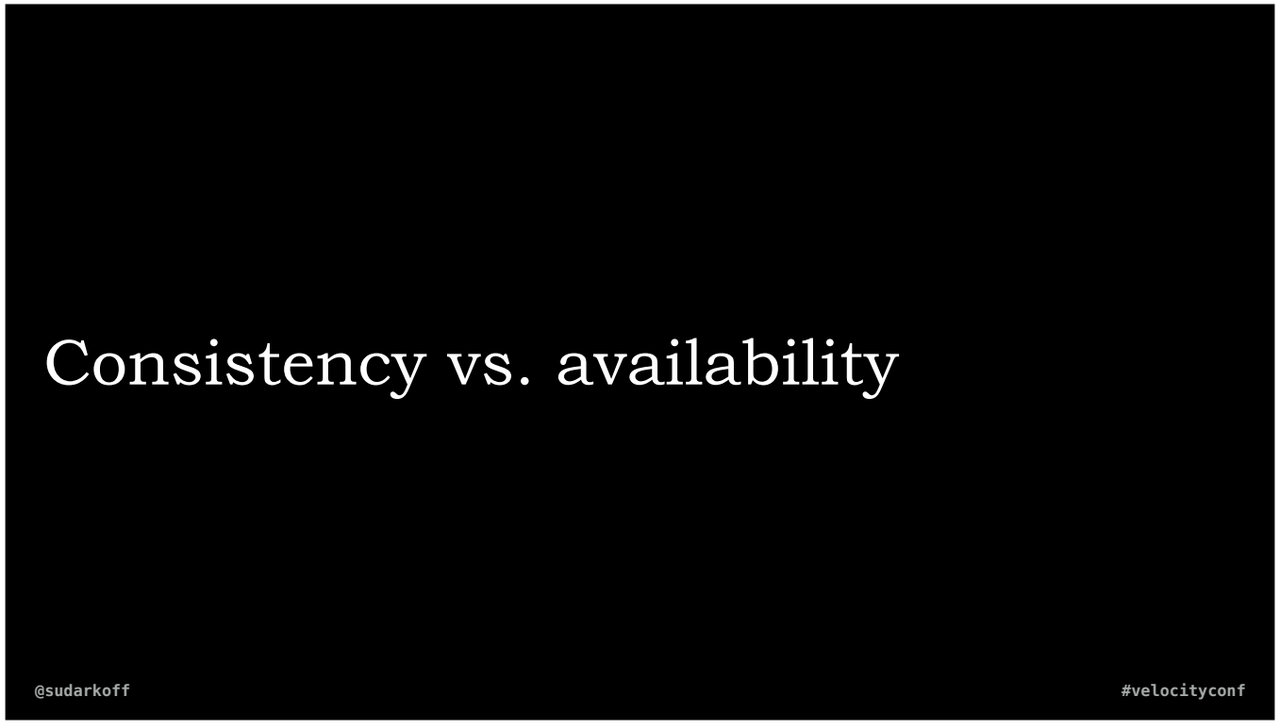
What I hear when people ask me these questions is that their services (and teams) are likely too broad and too deep. Maybe you've embraced the idea of the Full Stack engineer? Maybe you partition your micro-services vertically instead of horizontally? For example, is your front-end team responsible for data persistence as well? You might be making your life more difficult than necessary. Not only the development will be more complex, but maintainability, resilience and availability all will suffer as well.

First of all, remember the immune cells? Each cell has a very limited functionality, so to speak. The complex behavior emerges at the system level. Micro-services have the same promise. And by splitting your app into very small and very specialized services you simplify operations as well.

Second, operations is not so much about tuning the kernels as it is about caring about maintainability, resilience and availability. Whether it's a backend or a front-end - it doesn't matter. Operating the front-end does not mean your engineers have to rack the servers, it just means they need to worry about availability and resilience.

And worry they must, because in order to really **deliver the service** (i.e. continuously satisfy a customer need despite network instabilities and whatnot) the teams must understand and control not only their code (i.e. the semantics), but how that code performs in production as well (i.e. the dynamics).

So keep your teams small and micro-services ... well, micro.



Consistency vs. availability is another obvious place to explore when we're talking about any distributed system. CAP theorem states that it is impossible to provide both consistency and availability at the same time. (I'm ignoring partition tolerance, as one normally does).

Consistency of operations is kind of an obvious one. It mean that the tools and processes are uniform and consistent across the entire company. There's ruthless standardization! For example, all of your services might be written in the same programming language, using the same frameworks and libraries. Then built and packaged with the same tools. And finally, the same process is used to deploy and run it in production. When any part of this process is updated, it's consistently applied to all services. But if anything breaks, it's broken for everybody and nobody can release.

A perfectly consistent operations model will have to come with a religiously followed process for vetting and approving new technologies, which will introduce delays and make adopting to change harder.

Availability of operations is a bit less obvious. First of all, for example, making your CI server redundant does not make operating the services available. Because it's the build **process** that we care about. If everybody's using the same process and it's broken, it's still broken for everybody. But now redundantly so! So, then availability of operations means that each team uses their own tools and processes to build, package, deploy and run their code. And as a consequence, there's no consistency in how things are done across the teams. And it makes moving between the teams and collaborating harder.

Standardize

One way to deal with the lack of consistency when distributing operations is to create policies, standards, guidelines or best practices – whatever you wanna call it.

Completely distributed systems lack a centralized body that receives, processes and redistributes the information. Each agent is fully independent and has only loose ties to other agents. In a sense, policy is just another autonomous agent that promises a best practice or a guideline to other agents.

"Policy" is a lot like that tape that guides people through long lines at the airport. It's not physically stopping you from stepping over it, it's not an obligation or an imposition. But you wouldn't normally cut through the line unless you have a very compelling reason. Policies come with consequences for violating them, but it's best to keep them natural – you don't want to completely discourage experimentation. A team might discover a better way of doing things – a new guideline could be created to propagate that knowledge to other teams.

What to standardize

- Copy of Item
- Item (2)
- Item copy (2)

I’ve been told, that in Autodesk Revit – an construction industry application – there exist three naming conventions for creating duplicate items.

Naming conventions, style guides, coding standards and best practices – all of that exists to allow the teams to be creative where it counts. While producing consistent results and experience for their customers – external AND internal.

What to standardize

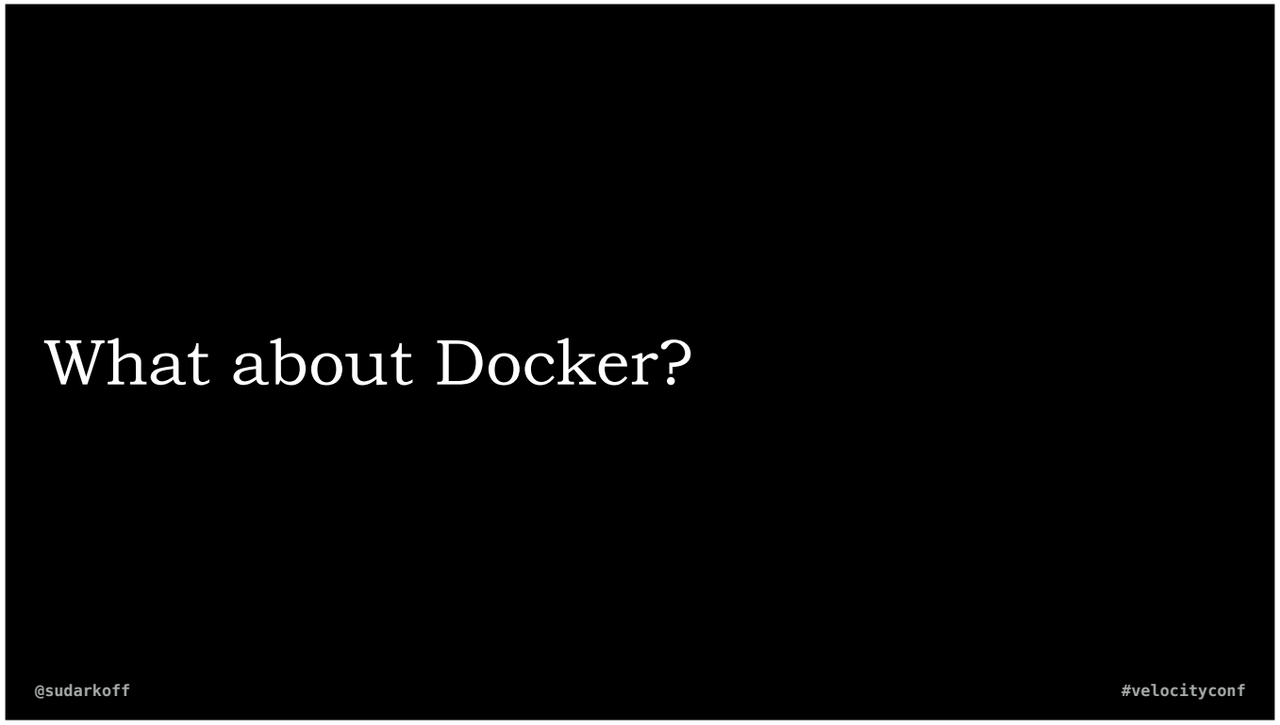
- racking and provisioning servers
- configuration management
- package management
- secrets management
- monitoring

There's a sizable subset of problems in operations that could be resolved in a predictable amount of time - this is what we call synchronous model in distributed systems. Problems such as racking and provisioning servers could be solved quite efficiently through capacity planning and other well understood and standard practices.

<read the slide> - all of these problems have one thing in common, they could be resolved in a predictable amount of time. And therefore probably should be standardized.

Communication

Communication is a big one. One of the potential dangers of distributing operations is creating multiple silos where only two previously existed. Open, candid, and respectful communication is how you break down the silos.



What about Docker? No talk would be complete nowadays without talking about Docker.

This is an interesting one. You hear often that Docker allows engineers to not talk to Operations. A big red flag, in my books!

I see containers as simply a way to break tight coupling between the application stack and the underlying infrastructure making it possible to evolve them independently. But you still care about resilience, availability, performance, and so on and so forth – whether it's inside or outside the container.

What NOT to standardize

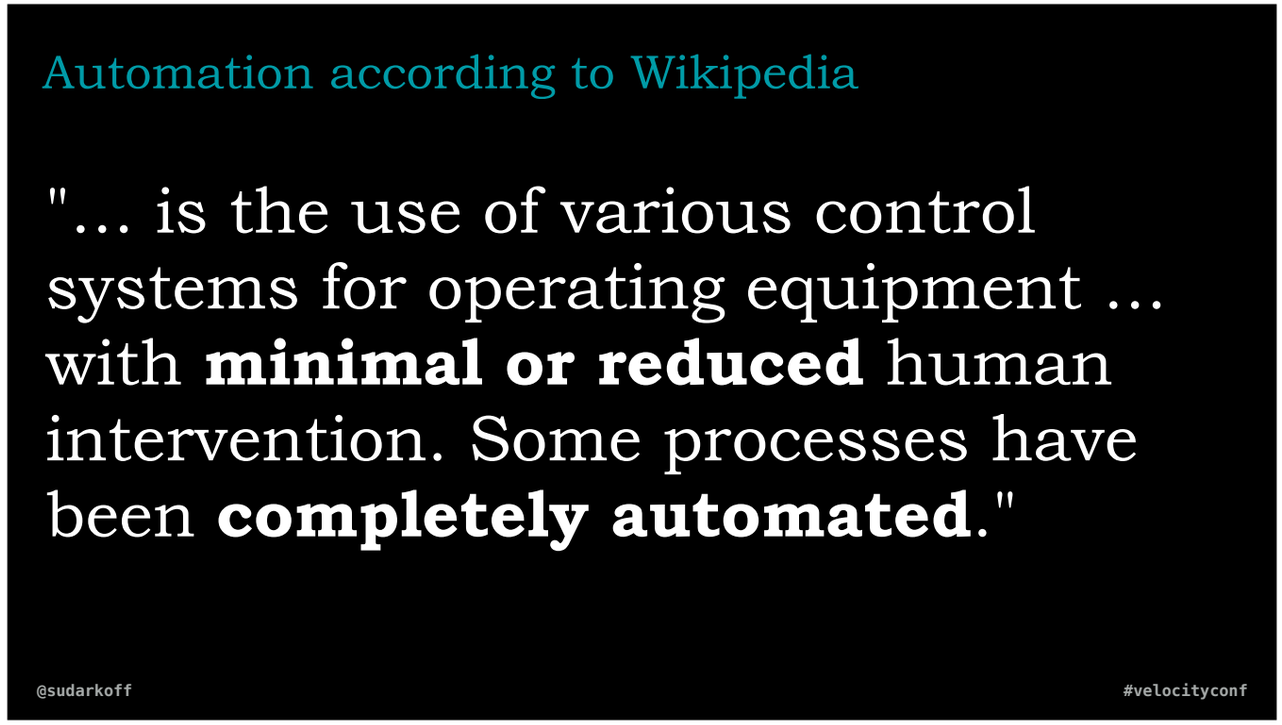
Premature standardization is the root of all evil.

So what would you NOT standardize? The short answer is - "everything else". Remember, standardization is a sort of communication mechanism. If you have nothing to communicate, if institutionally you have not yet learned enough about a thing, standardization is like premature optimization.

...

Automate

And last, but not least, is automation. This is probably the most obvious and the least understood answer to the "how do we scale human operations" question.



First of all, let's agree on what "automation" means. If we look it up on the Wikipedia, we'll find this definition:

"Automation is the use of various control systems for operating equipment ... with minimal or reduced human intervention. Some processes have been completely automated."

I personally see two separate meanings here. It talks about minimizing or reducing human intervention. And than tacks this "completely automated" bit at the end almost as an afterthought. I believe the words we use to describe things matter and in this case it would be useful to separate the two meanings.

Very often when we talk about automation we imagine something like this.

Imagine you are driving a Formula 1 car. You enter a corner and break hard as you downshift, then just before you exit the turn you, you step on the gas and the car accelerates into the distance. You feel exhilaration, you're in control, it's addictive! The scene fades to black.

The Formula 1 steering wheel might look complicated, but it's just the tip of an iceberg. Behind the scenes, there's an incredible amount of automation, the gearbox is almost entirely automated. A computer calculates the optimal moment for engaging the gears. This car would be nearly impossible to drive without automation. And yet, even with all this automation, the car still requires a highly skilled human to drive. And if you make a mistake you lose control and find yourself on the side of the track. I call this type of automation – "augmentation" (or "scripting" if we're talking about infrastructure automation).

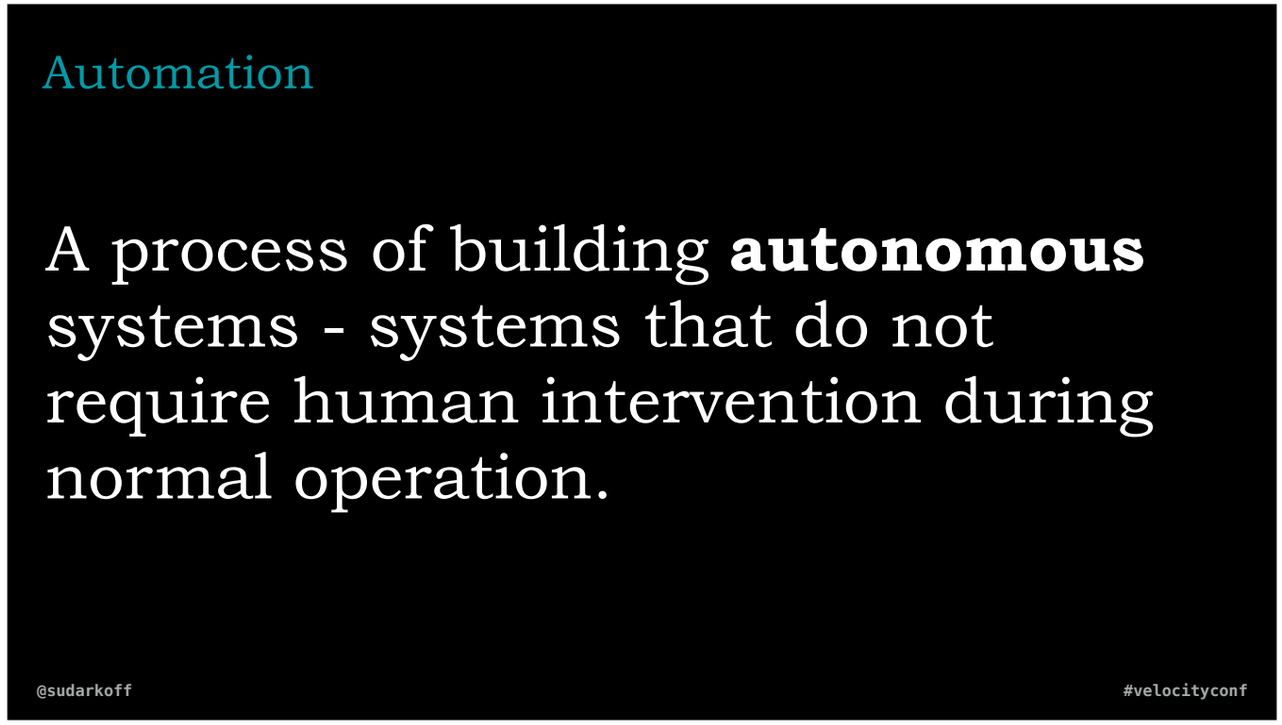
Augmentation (or Scripting)

A process of **reducing human effort** through integrating various tools, processes and tasks into a single process typically invoked with a **single command**.

<read the slide>

The true automation looks more like this. It's a boring looking car. It's boring to drive. In fact, there's no driving involved at all. You just enter the destination and the car takes you there while you're reading a book. Or more likely playing with your phone.

True automation means giving up control. Which is difficult to do, requires trust, and makes us feel uncomfortable. That's why it's so hard. That's why there are so many traditional ops engineers that resist true automation and instead write scripts.



<read the slide>

Of course, you might object, that humans are still needed to deal with equipment breakages, maintenance, and so on and so forth. And that's why I added "during normal operation" there.

Now if you compare these two definitions, hopefully you'll see that even intuitively there's a vast difference between automation and augmentation (or scripting). Augmentation increases our abilities. It allows you to get from point A to point B much faster, but you're still driving. In other words, it reduces the time necessary to perform a task, but the task is still there. While automation completely eliminates the task.

Automation is also directly related to the concept of self-healing. Because if you continue with the above analogy, you'll see that maintaining a self-driving car is probably quite a bit more difficult than maintaining a Formula 1 car. And we've been seeing this happening in our industry. Automated systems work with zero human intervention until they fail. And then they very often fail catastrophically requiring specialists to deal with the problem. That's another reason individual components of the system must be kept very simple (remember immune cells). One of the main ideas behind micro-services fits here very well. Each micro-service does only one thing and does it well. And each micro-service is in charge of its own resilience and availability (because you can't ask somebody else to exercise for you to stay healthy).

So, you know what, maybe a self-driving car is STILL not the best idea? Maybe we should invest in self-driving trains instead?

If you squint a bit, you might see the Consistency vs. Availability trade-off here. A self-driving train leaves on a schedule. It takes everybody riding it along the same route. You can't divert it to a different destination. And if the train breaks, everybody on the train gets to their destination late. The experience for everybody on the train is consistent.

A self-driving car, on the other hand, is optimized for availability. If you and I need to go to different places, we take separate cars. I can leave whenever it is convenient for me. And I can change my mind and go somewhere else and make extra stops. If you car breaks down on the way to the office? I'm still on time in my own car.

There's no scaling without giving away control