The InfoQ eMag / Issue #72 / May 2019

Operationalizing Microservices

InfoQ

Mature Microservices and How to Operate Them

Microservices have allowed The Financial Times to release code to production 250 times more frequently than for their previous monolithic platform. There are many steps involved when optimizing for speed, with continuous delivery providing the foundation.

Design Microservice Architectures the Right Way

Michael Bryzek believes investing time and effort into good tools, practices, and automation up front is critical to ensure that teams and systems remain productive and scale.

Lessons from 300k+ Lines of Infrastructure Code

Becoming an expert in anything requires repetitive practice. Writing hundreds of thousands of lines of infrastructure code should therefore provide quite a bit of expert guidance. Yevgeniy Brikman shares some of those lessons through a fourpart infrastructure cookbook.

Operationalizing Microservices

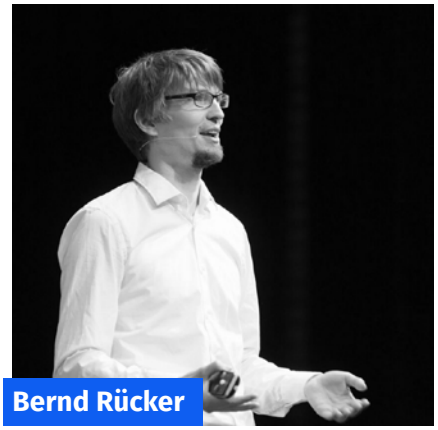
IN THIS ISSUE

GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

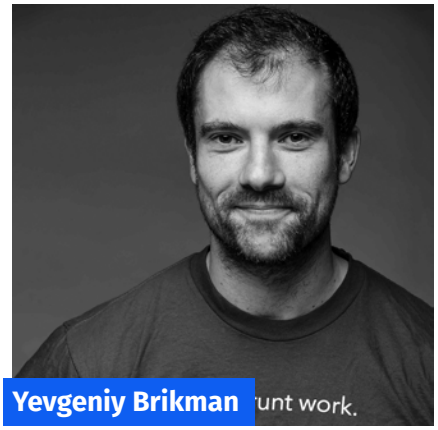
Kathryn Jonas

currently works as a Software Engineer with Teachers Pay Teachers and previously worked at The Economist as Tech Lead for the Content Platform. Jonas has lead projects for organizations in Beijing, London, and New York, applying technology to diverse challenges such as mission impact evaluation, editorial transparency and trust, and online learning and collaboration. She thrives on engaging software architecture debates and working with animated, empowered teams.



Bernd Rücker

is co-founder and technologist at Camunda. Previously, he has helped automating highly scalable core workflows at global companies including T-Mobile, Lufthansa, Zalando. He is currently focused on new workflow automation paradigms that fit into modern architectures around distributed systems, microservices, domain-driven design, event-driven architecture and reactive systems.



Yevgeniy Brikman

is the co-founder of Gruntwork, a company that provides DevOps as a Service. He's also the author of two books published by O'Reilly Media: “Hello, Startup” and “Terraform: Up & Running”. Previously, he worked as a software engineer at LinkedIn, TripAdvisor, Cisco Systems, and Thomson Financial.

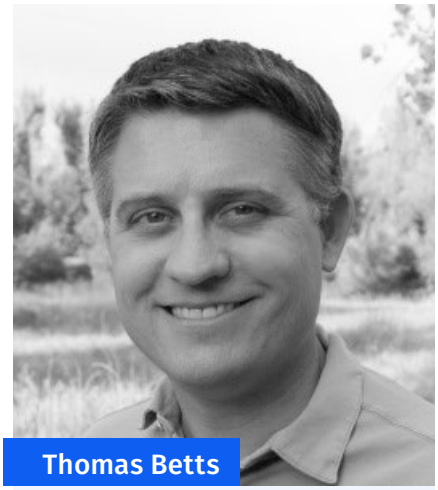
Sarah Wells

has been a developer for 15 years, leading delivery teams across consultancy, financial services and media. Over the last few years she has developed a deep interest in operability, observability and devops, and this has recently led to her taking over responsibility for Operations and Reliability at the Financial Times.

Michael Bryzek

is the CTO and co-founder of Flow Commerce, an enterprise SAAS platform that is the world's most advanced solution for global ecommerce. Prior, he was the cofounder and CTO of Gilt Groupe, an innovative online shopping destination offering.

A LETTER FROM THE EDITOR



Thomas Betts

is a Principal Software Engineer at IHS Markit, with two decades of professional software development experience. His focus has always been on providing software solutions that delight his customers. He has worked in a variety of industries, including retail, finance, health care, defense and travel. Thomas lives in Denver with his wife and son, and they love hiking and otherwise exploring beautiful Colorado.

Over the past few years, most companies have started using microservices architecture practices to some degree. However, scaling up from a proof-of-concept project to a production-grade, enterprise-scale software platform built on microservices requires serious planning, dedication, and time. The companies that have invested heavily in creating stable microservices architectures have learned many lessons on overcoming the challenges involved in operating complex, distributed systems.

Mature Microservices and How to Operate Them

Microservices have allowed The Financial Times to release code to production 250 times more frequently than for their previous monolithic platform. There are many steps involved when optimizing for speed, with continuous delivery providing the foundation. But faster delivery is only one-third of the problem. In her role overseeing operations and reliability at FT, Sarah Wells has learned that the other two puzzles to solve are how to operate microservices, and what to do when people move on, leaving behind legacy microservices to be maintained.

Design Microservice Architectures the Right Way

A handful of key decisions directly impact the quality and maintainability of a microservice architecture. Michael Bryzek believes investing time and effort into good tools, practices, and automation up front is critical to ensure that teams and systems remain productive and scale. At Flow Commerce, this philosophy covers infrastructure, continuous deployment, communication, event streaming, language choice and more.

Monitoring and Managing Workflows across Collaborating Microservices

Proponents of microservices like to cite loose coupling as a major benefit of the architecture. But, as with any architectural decision, loose coupling comes with trade-offs. By allowing for flexible choreography between services, you lose some ability to ensure those services are interacting according to established business processes. Bernd Rücker describes the challenges of balancing this trade-off, and covers solutions ranging from monitoring to full orchestration.

Lessons from 300k+ Lines of Infrastructure Code

Becoming an expert in anything requires repetitive practice. Writing hundreds of thousands of lines of infrastructure code should therefore provide quite a bit of expert guidance. Yevgeniy Brikman shares some of those lessons through a four-part infrastructure cookbook. He believes too many teams only focus on the first part, involving configuration and deployment, and fail to realize how much additional, time-consuming work is still required to create production-grade infrastructure. The checklist includes everything from security and backups to documentation and tests.

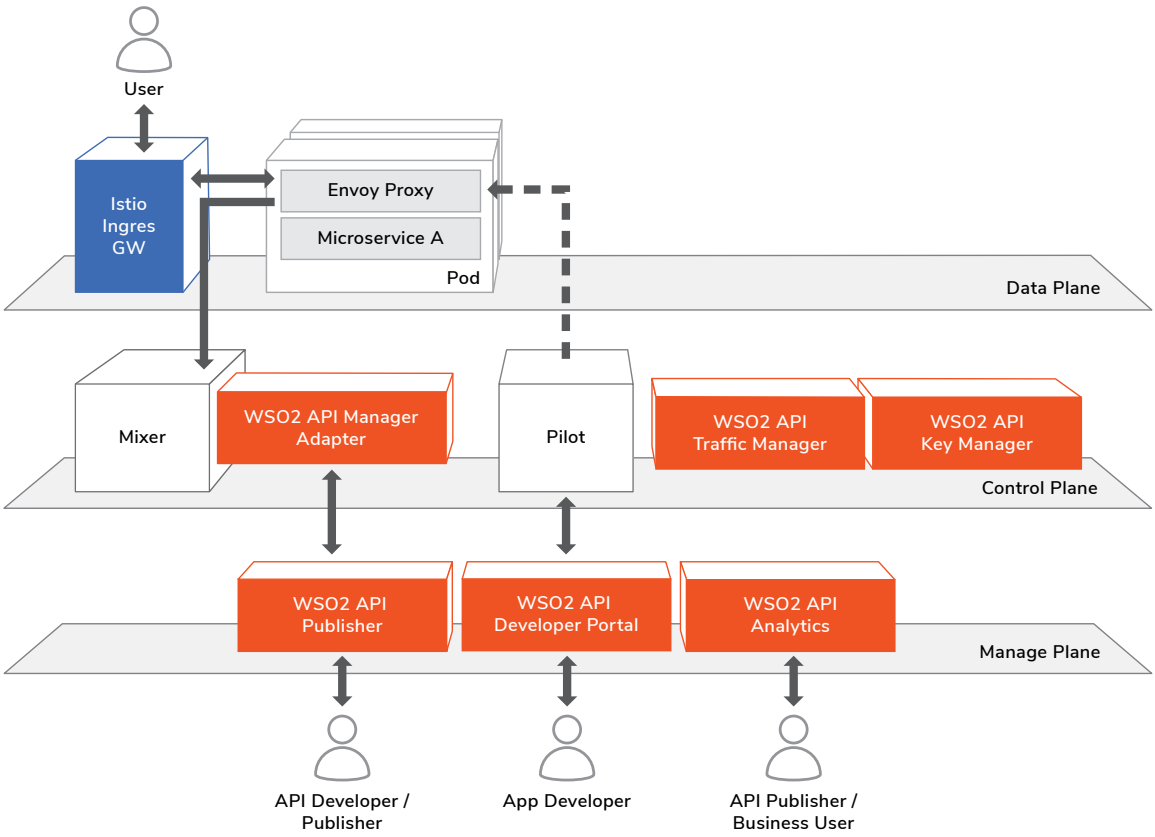
Using Golang to Build Microservices at The Economist: A Retrospective

While it is useful to understand some of the high-level challenges of operating microservices, it can be equally useful to do a deep-dive into the day-to-day process of creating microservices. K Jonas provides a retrospective over three years of using Golang to build microservices at The Economist. Go was selected from a handful of possible languages because it closely aligned with the platform goals of moving quickly, helped enforce best practices around fast-failing services, and facilitated a platform that could perform at scale.

API MANAGEMENT for ISTIO

Augment your service mesh functionality with API management capabilities

We provide a fully open source end-to-end solution for your entire business — from microservices to APIs to the end consumer.



In a world of disaggregated API-based architectures, developers are increasingly adopting microservices — and service mesh is being used to manage service-to-service communications. To gain value from the business logic contained within a microservice, it needs to be securely exposed via APIs.

Multiple implementations of service meshes exist today, but none of them address the concern of how the exploding number of APIs can be managed and securely exposed to their API consumers. With WSO2 API Management for Istio, you get the best of both worlds. Istio manages your services while WSO2 API Manager manages your APIs.

It's the only fully open source solution in the market so you can install and try it out for free at https://wso2.com/api-management/microservices/istio/

wso2.com

Mature Microservices and How to Operate Them

by **Sarah Wells,** Technical Director for Operations and Reliability at Financial Times
Adapted by **Thomas Betts** from a presentation at QCon London 2019

Microservices at The Financial Times

At The Financial Times, we have about five years' experience building microservices. I was the principal engineer on the content platform, which was the first system built using microservices at the FT. In that time, we've learned lessons about operating microservices that are essential to building them successfully.

When one of our early microservices, a URL management tool, failed in production, we realized it's quite possible to have a microservice you haven't touched in several years, and no one doing the trouble-shooting has any idea how it works. When that happens, you're really relying on two things: does the monitoring tell you there's a problem, and does the documentation give you what you need?

Another factor when trouble-shooting microservices is just working out where the problem is. For example, between an editor clicking a button to publish an update to a list on our homepage, and that update appearing on the website, there are probably ten or more microservices. You need to be able to identify the source of the problem.

Microservice architectures are more complicated to operate and maintain than a monolith. So why bother doing them, and why not just stick with the monolith? The answer to that, as with all technical decisions, really should come down to business reasons, so I need to start with a little explanation about the business of The Financial Times.

The Business of The Financial Times

At the Financial Times, we don't describe ourselves as a newspaper. We describe ourselves as one of the world's leading business and financial news organizations. That's because most people don't read the FT on print; they read it online, on the phone,

on a tablet, or on a computer. We have a paywall, and that's where we make most of our money. What's clear is that in the near future it's going to be really important for us to be able to experiment and try things out and see what's going to work for us.

I saw a really interesting talk from Linda Rising last year where she said, "It isn't an experiment if there's no hypothesis and if you can't fail." She also said that *experiment* for most organizations actually just means *try*. If an organization invests quite a bit of money in a product or feature, it is really unlikely that they're going to decide that it wasn't worth doing and get rid of it. But the only way you get a culture of experimentation is if you can do experiments quickly and cheaply. Then you have a chance that someone will say, "This is what constitutes success," and you'll be able to say, "Oh, we didn't meet it, we're not going to roll it out."

A/B testing is built into ft.com and we do hundreds of experiments a year. We start by saying, "What are we measuring, and what would be the criteria for its success?" There are plenty of things that don't go live because we've decided that they didn't give us enough benefit. We have this A/B testing built in, but you still need to release code to do any kind of tests. We need to be able to do it very quickly, which we do. We release code to ft.com thousands of times a year, which means that we do have a culture of experimentation.

But releasing changes that frequently doesn't just happen. You have to do a lot of work and it had a massive impact on the way we work and the culture of our company. Microservices has been a thing that has enabled it for us; a combination of continuous delivery, microservices and DevOps. Together, they've been the foundation for being able to release things quickly. That also means that the team that builds the system has to operate it too.

We've now been doing microservices for five years, so teams are starting to get smaller, and people are moving on to new projects. The people who currently work on the website or on the content platform aren't the same people who made a lot of these decisions. What do you do when teams move on? How do you make sure that systems don't become unmaintained, unloved, and risky? Every bit of software eventually becomes legacy, and your next legacy system is going to be made of microservices, not a monolith.

There are three things about our journey that are important. First is optimizing for speed – why we've done it, and how we've done it. Second is the things that we've found essential to build in to operate microservices. Finally are some of the ways we're attempting to make sure that when people move on we still have systems that work.

Optimizing for Speed

Optimizing for speed is really essential for us because it's part of the ability to let us experiment. I was really happy when the book Accelerate came out, by Nicole Forsgren, Jez Humble, and Gene Kim. It is about how you build high-performing software development organizations. Those high-performing software development organizations have an impact on organization performance. They found that there are four measures that have the most impact on the business that are highly correlated with high-performing teams. High-performing organizations release changes frequently, they're small changes, they fix things quickly, and they have a low change fail rate.

Continuous Delivery is the Foundation

Continuous delivery is the foundation of optimizing for speed. The thing I take from the book Continuous Delivery by Jez Humble and David Farley, is, if it hurts, do it more frequently. Our old build and deployment process was very manual. We could only release a maximum of 12 times a year because we had to freeze the website, and we couldn't publish new stories while we were doing a release. The manual process was documented in a 54-line Excel spreadsheet, but

those steps weren't always correct, and they often went wrong. Once we recognized that it was painful and we should fix it, then we adopted continuous delivery.

Also, you can't experiment when you're only doing 12 releases a year. The feedback cycle isn't fast enough. Because so many changes are deployed together, you can't tell the impact of one thing or another.

The first step for our continuous delivery was creating an automated build and release pipeline. It needs to be stored as code, and it needs to be version controlled. You need to be able to recreate your pipeline from scratch, if you have to. Effectively, your aim is to make it so incredibly easy to release code that anyone can do it at any time. In our previous process, people were scared to do releases. Now, no one at the FT is scared to do a release. It doesn't mean you don't evaluate the risk, but you know that the release is going to work and you know that you can fall back to the previous version easily if it goes wrong. That is not something we used to have.

The second thing is you can't stop doing manual regression testing if you want to move fast. It just takes too long. You can't regression test against your entire system, so you need automated testing built into the pipeline. When you do use manual testing, you need to target it so that it's focused on the change you've just made.

Finally, you need continuous integration, which people sometimes skip when doing continuous deployment. You need to be putting changes out regularly. It's good to have an automated pipeline, but if you're still only releasing once a week, you're not benefiting from it. If you aren't releasing multiple times a day, you need to work out what's stopping you. You need to decide whether it's worth adopting a microservice architecture, with all of the costs that come with it.

Often, the architecture is the thing that's stopping you. For us, the 12 releases a year were because we couldn't do zero-downtime deployments. That led to our new systems being built for zero-downtime deployments from the beginning. A great benefit of zero-downtime releases is that in-hours releases becomes the normal thing that you do. When something goes wrong, anyone that you need to help you is going to be there, instead of having to track them down or wake them up.

If you want to move fast, you need to be able to test and deploy your changes independently. The authors of *Accelerate* say it actually doesn't really matter what your architectures are, as long as the systems, and teams, are loosely coupled. For us, microservices are loosely coupled. You can keep your monolith loosely coupled, but it's much harder work because it's much easier to get things dependent on each other without realizing it.

"Process Theater"

It isn't enough to change your architecture and to build a continuous delivery pipeline. You need to look at your other processes as well. When I joined the FT, we had a change approval board and we also created change requests. This is just process theater because it's pretending to be making things less risky, but really, it's not. There's research in *Accelerate* on this which says that CABs and change requests actually have no impact on the change failure rate, but what they do is slow down everything else; your change failure rate is similar, but you deploy less frequently, it takes you longer to fix stuff, and generally you tend to do bigger releases, which are inherently more risky. Because CABs don't reduce the risk of failure, you can remove the process theater.

We don't have change approval boards anymore. Our theory is, if you're doing small changes, the best person to make the decision about whether this change should go is the person who just made that development change. We rely on the peer review – the pull request – as the approval, and we just log that we've made a change, automatically through a call to an API. All we need is the ability to ask, "What has changed?" when something goes wrong.

It isn't enough to change your architecture and to build a continuous delivery pipeline. You need to look at your other processes as well.

Speed and Benefits

We're now moving much faster at the FT. In 2017, the content delivery platform (just one group within the FT) did 2,500 releases, which works out about 10 releases per working day. That's about 250 times as often as we released our monolith, which is a massive difference.

We've also seen our failure rate drop. When we did 12 releases a year, one or two of them would fail and when they failed they were incredibly painful. That's about a 16% failure rate. In 2017, we had less than 25 releases fail, which is less than a 1% failure rate. The kinds of failures we got were much less significant, with a greatly reduced blast radius, and they're easy to reverse because we've got this automated pipeline.

The Cost of Operating Microservices

While there are benefits, there are costs that come with microservices, and operating them is harder than a monolith. An old, but totally true tweet by Al Davidson said "[microservices] are an efficient device for transforming business problems into distributed transaction problems." Everything's over network traffic now and things are unreliable. All your transactions being distributed means that things can partially fail or partially succeed. You're hoping for eventual consistency, but quite often you end up with just inconsistency that you have to fix up somehow. Luckily there are patterns and approaches that can help.

I think DevOps is a good thing to do regardless of your architecture, but it's absolutely required for microservices. You want your team to all have the same goal, focused on the value you can give your customers. You can't hand things off to another team if they're changing multiple times a day. As soon as you are coupled to another team, you slow down.

Decisions about Tools and Technology

High-performing teams get to make their own decisions about tools and technology, for two reasons. The first benefit is that empowering your teams makes people happy, which is not a bad thing. Secondly, it speeds you up again. You can't be waiting for the result of an architecture review board to make decisions. Because teams will make different decisions, it becomes extremely hard for any central team to support everything. You have to have those teams supporting the decisions that they've made, including paying the price for bad decisions.

You can mitigate this a bit by making things someone else's problem. Don't install and run queues and databases if someone else can do that better. Lots of the FT runs on Heroku. Where we run on AWS, we want to use Kinesis, or Aurora, or other services to avoid installing our own software onto EC2 instances.

Simon Wardley, another speaker at QCon London, describes how all successful technologies go through a cycle: from innovators and early adopters building their own solutions, to eventually becoming ubiquitous commodities. Just as you wouldn't build your own power station for your electricity needs, you should just be using AWS or its equivalents rather than installing your own data center. If a commodity is available, use it, unless it's critical to your business. The only time where you want to be doing stuff yourself is when it is a differentiator for you, and the customization that you can do is critical for your business.

Grey Failure

With microservices or any distributed architecture, you have to accept that you're generally be in a state of a “grey failure” where something's not working. The only thing that matters is, is it having an impact on the business functionality?

Good advice when a failure occurs is “Mitigate now, fix tomorrow." This is something developers find quite difficult because I think we naturally want to fix things. We want to work out what went wrong and then fix it. Mitigation is normally quicker. If a release just went out, roll it back and see if it fixes the problem. Then you can indulge in your excitement at investigating the problem.

Monitoring

You need to make sure you know when something's wrong. It is extremely easy with a microservice architecture to get inundated with alerts if you just start monitoring all your systems. Instead, concentrate on the business capabilities, so you can determine any impact on your customers.

For us, we need to know if we're able to publish content, and

we use synthetic monitoring to check that. We combine monitoring, which is happening in production all the time, with synthetic, not real, events. Every minute, we have a service that re-publishes an old article. It sends a publish event using exactly the same mechanism as all our content management systems do. Then we just read the same API's that our website reads and check that the update made it through. We monitor it the way we monitor anything else. It's got a health check endpoint, and basically it's healthy as long as publishing is working.

This has a secondary benefit of letting us know whether publishing is working, even if no one is actually currently publishing real content. News publishing can have peaks and troughs. Previously, if we had an alert in place to say, "Has anything not been published for a couple of hours?", the chances are that alert is going to fire on Christmas day because no one publishes anything on Christmas Day. You really don't want alerts that are most likely to bother you on a bank holiday.

Synthetic monitoring is great, but it's not enough because you also need to make sure that you know whether real things are working in production. We need to decide, "What do we mean by publish being successful?" When you publish an article, it goes into MongoDB, Neo4J, ElasticSearch, and S3. Because we have two regions, a successful publish has to get to eight different destinations. That's complicated to do with normal monitoring, so we wrote a specialized service. It listens to notifications of publish events and it just checks everything where that event should be and it waits for two minutes. If it doesn't happen in that time, effectively it says, "I'm unhealthy. You need to republish this bit of content." Because publishing an article is idempotent, we can just republish it. Basically, we end up with eventual consistency, even if sometimes we have to publish it a few times.

Observability, Log Aggregation and Metrics

You need to build observability into your system, because the way that you debug microservices is completely different from how you debug a monolith. With a monolith, you can attach a debugger and step through to reproduce a bug. And your logs were probably pretty shocking because you never really had to rely on them. With microservices, you probably can't run your entire stack locally. You become reliant on what you've got in production to work out what was going on. Observability is being able to infer what was going on in the system by looking at its external outputs, such as logs, metrics, and monitoring.

Log aggregation is important because the events that you've

got in a microservices architecture are all over the place, and you need to aggregate them all together somehow. To keep the size manageable, you can keep all the error logs, but sample the successful ones. We also remove all the health check logs from the monitoring.

You need the ability to find the logs for a particular event, and we do that using a transaction ID. We expect there to be a header on a request that has a unique identifier. Every service has to output that in the logs and pass it on to any other services. If a service doesn't find a transaction ID, it will generate one. We have a library for doing this, so it's pretty easy to add it to any new service, and it's essential. When you're trying to debug something, you find the transaction ID, then you can find all the logs. We wrote this ourselves because service meshes and tracing services weren't around when we were doing it.

Metrics measure something over a period of time. When you start with microservices, it can be easy to measure a lot of metrics and then get completely overwhelmed by how much you've got. Focus on metrics for services closest to your user, and keep it simple. You want to know how many requests you're getting, how long it's taking for those requests to respond and what the error rate is.

Always Migrating

With any system, you're always doing migrations and upgrades. But we have 150 microservices in the content platform, and we don't want to have to release 150 services to do an upgrade. Our deployment pipelines use templates, which provides a centralized way to easily update how we deploy our services. If our security team needs to add new security scanning into all our deployment pipelines, we add it in one place and all of the pipelines get updated.

As I mentioned earlier with the transaction ID, we don't use a service mesh because they just didn't exist when we started with this. I would totally use that if I were building microservices now, because the service mesh can take on lots of things like back off and retry, routing, load balancing, log aggregation, all the things that we've built into our systems with individual libraries. The problem for us is if we have to update a library, we have to release a lot of services. A service mesh can do that for you. This means your microservices focus on your business functionality, not all these other things that need to be there just to have a network of systems.

Any codebase has bits that don't change very much. The difference between a monolith and microservices is that that can mean that there's a service that you haven't actually released for years. If you haven't done it, are you sure you could when you needed to, such as when there's a security vulnerability that you need to patch urgently? We have one team at the FT that builds (but doesn't deploy) all of the services overnight. If a build is not working, they can do small fixes along the way.

What Happens When People Move On?

The final thing I want to cover is what happens when people move on, when the team gets smaller, when people move on to new challenges and the people you've got maybe weren't involved in a lot of the decisions? I think it's critical to make sure that every system is owned, and owned by a team, not an individual. While some people would like to believe that systems just run with no effort for years, it isn't true. Particularly with microservices, you need to maintain a level of knowledge of the system, and the ability to fix it. If you won't invest enough to keep a system running, you should shut it down.

When you've got thousands of systems, it's really hard to keep operational documentation up to date. If you have a searchable runbook library, as we do, you need to work out what's important in that runbook. Troubleshooting steps probably won't help you much in the future, because they tend to be a record of past failures, but every failure is different in some way. Instead,

what you do want to know is: who owns this system? Where is the code? How do I look at the logs? How do I find the metrics? How do I do a restore? That stuff is going to be helpful and useful if I look at it six months after I last worked on it, or I'm looking at it for the first time.

We're currently working on a new monitoring dashboard system at the FT that we're calling Heimdall, because Heimdall is the god that can see everything, everywhere. We look at a graph of system data, and provided you've created everything correctly, you just get a dashboard for free. In the past, teams had to maintain their own dashboards. We found that as we started putting information onto this, people would see something incorrectly reported, such as a system being decommissioned, and they'd self-correct the data.

Going back to the theme of continuous delivery, you need to practice stuff, especially the basics like failovers and database restores. We practice failing over ft.com weekly, and different people do it every time, so we know that the documentation is correct. All your operational activities should not be scary; they should be something you just know how to do. You do not want to be trying something at 2:00 in the morning when you've just been woken up and discover that the documentation is wrong or you don't have access rights to do this thing.

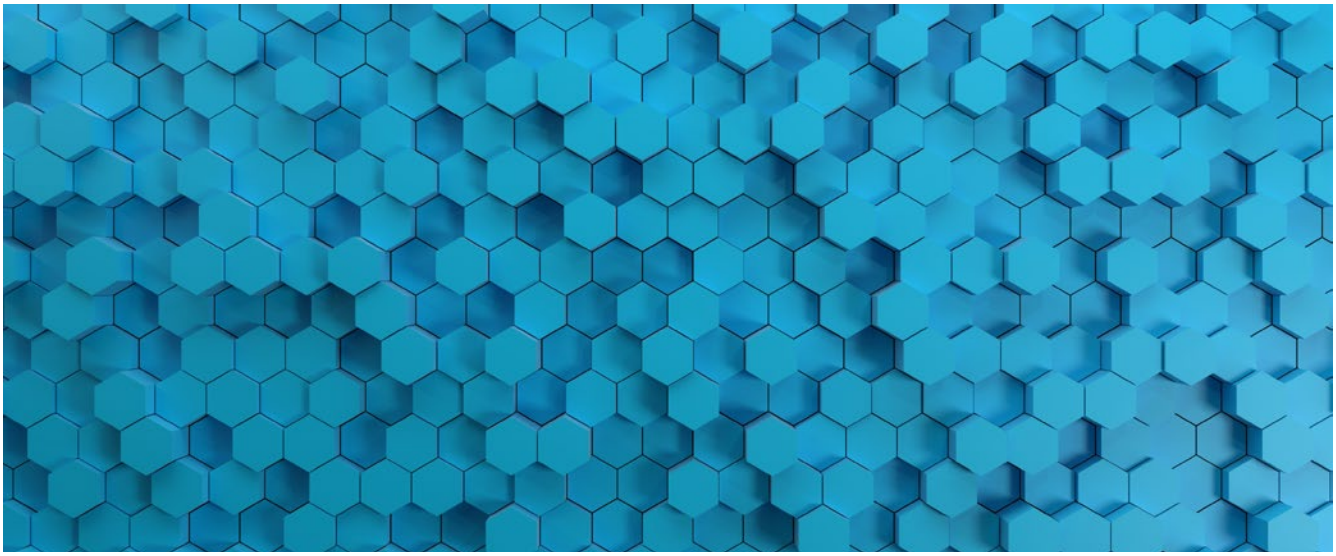
Conclusion

Building and operating microservices is hard work. It's much harder than the monolith, but I think it is worth it. You have to prepare for legacy by maintaining knowledge of the services that are live. You have to invest in it. Like gardening, you have to get rid of the weeds and keep things clear.

You need to plan now. If you're building microservices, think about the case in three or four years' time where you've got a skeleton team trying to maintain 150 microservices. That could be extremely difficult, so plan for that and build things in. And remember that it's about the business value, so keep an eye on whether you're still able to move fast. If you stop moving fast, you're losing the benefits, but you're still paying the cost.

TL;DR

- Because microservice architectures are more complicated to operate and maintain than a monolith, there must be good business reasons that justify the investment.
- Loosely-coupled services and teams are able to move fast and respond to change. Continuous delivery provides the foundation for this increased speed.
- In addition to a continuous delivery pipeline, you must evaluate and change business processes, and eliminate those that artificially slow you down.
- Operating a distributed system means you have to get comfortable with being in a state of "grey failure." A good approach to handling problems that arise is to "mitigate now, fix tomorrow."
- The original developers will not always be around to support the microservices they created. You must plan for how others will still be able to support your system long-term.
- You always have to continue moving fast. Otherwise, you're paying for the cost of microservices, without getting the benefits.



Design Microservice Architectures the Right Way

by **Michael Bryzek,** Co-founder / Chairman / CTO at Flow.io
Adapted by **Thomas Betts** from a presentation at QCon New York 2018

What is Great Architecture?

Great architecture allows us to scale development teams and deliver higher quality software. It also enables business reasons to drive the choice between high performance and low cost. Finally, a defining attribute of great architecture, and the hardest one, is the ability to support future features, naturally. Ultimately, the way we to know if you have a good design is to look back after three or four years, and see if people like to use it, and if you made good decisions today.

Not-so-great architecture, which can look like spaghetti, trades near-term velocity for future paralysis. We've seen many examples of this in the microservices space, where we're tempted by the benefits of microservices, and we underestimate, or underinvest in, what is needed to build a great architecture.

Today, I'd like to discuss how you avoid the spaghetti, and create a great, layered mille-feuille, where the whole is greater than the sum of its parts.

I'm currently the co-founder and CTO of an enterprise SaaS company, Flow Commerce, where we help brands expand internationally. From day one, we built our company on microservices, based on the many lessons we learned in our prior experience. Prior to Flow, I was the co-founder and CTO of Gilt.com, which had a large-scale microservices architecture of over 400 applications at the time we sold the company to Hudson's Bay. There, we learned a lot about the benefits of microservice architectures, in terms of scaling teams, delivering quality, isolation, and performance. We also learned many of the challenges, and, in hindsight, areas where we wished we had invested more.

Misconceptions

We need to start with a few misconceptions.

Misconception #1: Microservices enable our teams to choose the best programming languages and frameworks for their tasks.

This is often cited as one of the major benefits of microservices, where I can build one service in Go, one service in Rust, one service in Node, and one service in whatever language gets invented tomorrow. The reality is it is extremely expensive to adopt new languages and frameworks. Really, the bigger bar is team size and the level of investment into the architecture.

Google has about twenty- or thirty-thousand engineers, and they have eight programming languages. I like to say one programming language for every 4,000 engineers is a good metric.

Misconception #2: Code generation is evil

I used to think code generation was evil, but in reality, code generation is just a technique. What's really important, especially in microservices, is creating a defined schema that is 100% trusted. Later, I'll demonstrate one technique where Flow is leveraging a significant amount of code generation in our software development process.

Misconception #3: The event log must be the source of truth

When we were starting Flow, I reached out to Jay Kreps, who wrote the definitive paper on logging (pdf). I told him, "I don't get it. I've got a REST service, and I'm creating a user. What am I supposed to do? Publish an event, then wait for that event to come back to my service, so that I can respond to my client? How do I guarantee I have that user's details?" From the horse's mouth he said, "No, you just throw that in a database. It's fine." So that's what we did.

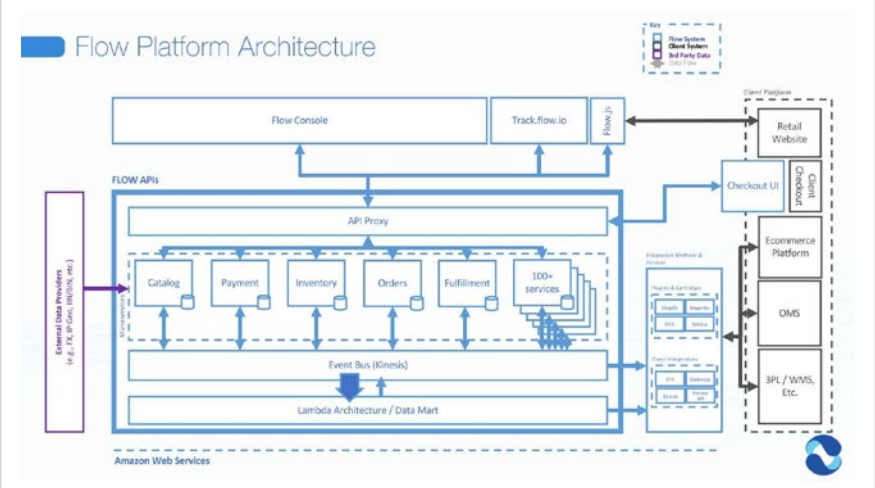
Resources are stored in databases that belong to the microservices. But then we guarantee absolutely, with at-least-once semantics, that those messages will end up on the event stream. It's okay.

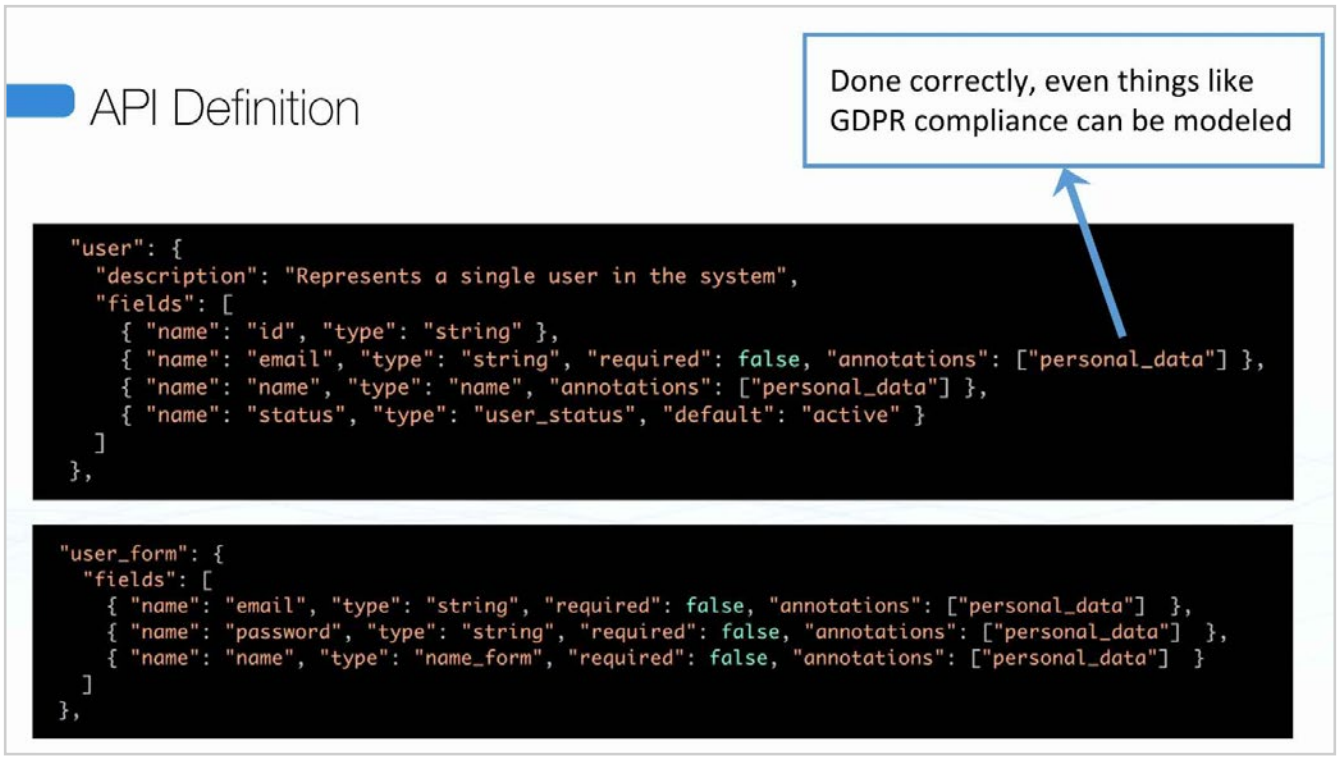
Misconception #4: Developers can maintain no more than three services each

A few folks at Netflix have shared that three services per developer is a magic ratio, and when you get to that number, you stop all feature development, and just babysit those services. I think this is the wrong metric to focus on. If you're having conversations around this metric, I think that's a clear sign that you need to invest in automation and tooling. In our third year at Flow, the ratio is about five services per engineer, and they spend less than five-percent of their time maintaining services.

Flow Platform Architecture

The Flow architecture consists of over 100 distributed microservices. Each service defines a REST API, and they all commu-





nicate via APIs and events. One thing that's unique is we don't have a private network, which means all the products we develop are built on the same APIs we offer our clients.

One of the key practices and the first critical decision to make when going into a microservice architecture is, "How are you going to manage and define your APIs?" The very first artifact and step of software development at Flow is the design of the API. Critically, this is not written in code, using annotations, because it has to be language-neutral. We use JSON.

Everything is resource-first. As shown in the example above, a user has an id, email, name, and status. By convention, to create a resource, you use an object called "{resourceName}_form." To create a user, you use user_form.

In an API-first world, we were able to handle GDPR requirements by simply adding an annotation at a field level. For example, email is considered "personal_data." From that, we can automatically generate a complete trace through every single service at Flow for anything that may contain email. We don't have to guess. That's possible because we start with the API definition.

To interact with the user, we create operations on the API. Again, in JSON, we define the user and a GET operation to fetch the user by ID, and a POST operation to create a user, and it accepts a user_form.

Don't Break APIs

Because the API is first-class, the API definitions are not in the microservice repos – they're in a dedicated repo called API. Making a change to an API means editing the JSON file and submitting a pull request. This runs continuous integration and automated tests on the definition of our API. We also run linters over the API definitions. One of the big goals with CI on the API definition is to make it feel like one person wrote the entire API, and automated linters help make that possible.

Continuous delivery is a prerequisite to managing microservice architectures.

From a policy perspective, we're empowered to make a decision that says, "We don't break things." While this is a common practice with databases and schema design, we can make the same decision with APIs. It's a critical decision to make, but you have to build practices around that. You have to know when you're about to break an API.

An open-source tool we started at Gilt, called API Builder, is a free, hosted solution for end-to-end API design. One native feature is detailed annotation of every change to the API, including things that are breaking. This makes it trivial to add a CI step that checks if you broke the API, and can then fail the build. This happens at the API phase, very early in the process, before any implementation, so it becomes easy to course correct.

Implementing Services

Now that we have our API, with user resource and user_form, we want to create something. This is the first time we get into code generation. There are opportunities to say, "The specification is the first-class thing that we built; don't duplicate that." Anything that can be driven off the specification is an opportunity to either do it dynamically, using reflection, or with code generation. I've become a fan of code generation, because it makes it really easy for anyone to read the code. We invest heavily in the generators we write to make them readable.

An an example, we run apibuilder update --app user. This generates three things: a routes file, a client, and a mock client.

All our services at Flow are written in Scala and Play. In Play, you respond to an http request by declaring a route. Based on our API, we automatically generate two routes: GET /users/:id, and POST /users. This guarantees that our implementation has these routes defined. When I first run the compiler, it complains because these routes don't exist, so the definition in our API guarantees that the implementation includes the required operations.

Two additional benefits are user-friendly routes and consistent naming for methods. If you create a new resource named company, you can probably guess that to create a company it will be POST /companies, and to get a company it's GET /companies/:id.

The client library that's generated is really easy for a developer to use. For our internal use, it creates the client library in Scala, and that means a developer doesn't have to write the code to call the new API. As you build microservices, you'll have lots of them, and that leads to a lot of time writing client libraries. Also, consider if you introduce a second language. While we use Scala, not all of our clients do. If a client wants to use Ruby, or Go, or whatever, you have to recreate

all the client libraries in every single language. While that work is valuable, it competes with other priorities. That's why this is an important place for code generation.

When using code generation, I think a lot of people just optimize to make things possible, but that's not the intent. The goal is to make the generated client so nice that a developer will love using it, and not write their own, hand-crafted library.

Testing has to be thought about from the start, particularly in a microservice architecture, which is why we also generate a mock client. Because our generated mock clients come from the same API specification, they allow us to do high-fidelity, fast testing.

The code that we write is simple and consistent. Leveraging the generated code, controllers just have a few lines to take a request body, submit it to the data layer, and handle either success or an error. This means it's easy to read, and less likely to contain bugs.

Databases

Each microservice application owns its own, private database, and no other service is allowed to connect to that database. All services communicate with each other through the API or events. This is really important because once you let someone connect to your database, you lose the ability to know if a change is safe. That is a very insidious form of tech debt, and the easy solution is to just not let anyone in.

This is another place where we've invested in tooling. We have a CLI called dev, intended for developers, and it handles all of our common infrastructure tasks. To create a database for your service, the statement is just dev rds --app test. If you don't know how to do something, you can just type dev and get a menu of all the things people have done and are now automated.

This is super important, because it means everything is done in a consistent way. If you logged into Amazon, you would see all our database names follow the same naming convention. The only reason that is true is because someone took the time to make it automated. It also means the experts in databases created that part of the CLI, and everyone else benefits from that expertise.

Because we like code generation, we describe our database needs using metadata, and code generate our way to a solution, including the data access object (ex. usersDao). It's worth noting that the data tier is independent from the API; it just uses the same toolchain.

One very useful benefit of generating your database schema is consistent rules around data

formatting. From the beginning, we came up with a convention that we're not going to allow empty strings, and we have constraints that cause an error to be thrown if you try to submit an empty string. Code generation allows us to enforce a policy like this across the company.

We also have a hashcode field added on every table. When we receive a large batch of data to process, we can quickly compare the hashcode and then only perform an update if the hashcode has changed. This reduced the writes on our database by 100x. Adding that field, in every database, across every microservice, without developers having to think about it, was only possible because of code generation.

Another really powerful benefit of driving the database through the metadata is the creation of indexes. Traditionally, if someone finds a slow query, then adds an index to speed it up, they look like a hero. But the real hero is the person who can prevent you from ever writing that slow query in the first place. The metadata-first approach means if you specify an index on email, then you have a query for users by email. If that index wasn't specified in the metadata, you cannot query by email.

Testing

Automated end-to-end integration testing is really easy, because we can use the generated mock client. For example, the following code blocks show positive and negative tests for the user object.

```
"GET /users/:id" in {
  val user = createUser()
  await(
    identifiedClient().users.getById(user.id)
  ) must equal(user)
}
```

```
"GET /users/:id w/ invalid id returns 404" in {
  expectNotFound(
    identifiedClient().users.getById(UUID.randomUUID.toString)
  )
}
```

Because of our focus on code generation based on the spec, and the ability to create these verification tests, our team now expects that the code will just work in production. That drives quality and team velocity.

Deployment

Continuous delivery is a prerequisite to managing microservice architectures. If your team is spending hours babysitting releases, good luck if you have a hundred services. At Gilt, we made a big investment in a software delivery system. It took about nine months to make it reliable, but it was definitely the right first decision.

Continuous delivery can mean many things. At Flow, it means that a deploy is triggered by a Git tag. We automatically create a tag when there's a change on master.

A continuous delivery system must be 100% automated and 100% reliable. Rarely do systems behave that way. It's a red flag if deploys keep failing, and it means developers must log in to systems to determine why the deploy failed. That needs to be fixed, because it affects velocity across the platform.

Our continuous delivery is an open source project we created called Delta. The entire system that deploys software thousands of times each week was written by one developer in a week-and-a-half. This doesn't need to be a massive investment; it just needs to be focused on reliably deploying software into the cloud.

I'm a developer, and I don't like describing my infrastructure in thousands of lines of JSON.

Instead, we need to get it down to the most basic elements, and let the people who understand infrastructure make the recommendations for the company. Our configuration for deploying a service is just six lines of YAML.

Every service implements a standard health check on a well-known URL. They implement it by pulling in the specification of the health check from the API spec. This is really important to implement early on, even if it's just returning a 200, because at least you've got the placeholder to add a more robust health check later on.

Events

Everything we do at Flow is API-first, and we're very proud of our APIs. But, we'd be perfectly happy if you never use our APIs. We'd prefer you to use our event streams instead. In our own internal network, we almost never use the APIs, except for rare cases where you need synchronous operations.

Events, just like the API, really take an investment to make them work. You have to have a first-class schema for all events. Everyone who's using binary formats, like gRPC, is in great shape; everyone who's using Swagger is in terrible shape. The big difference is binary formats force developers to use the code generation to produce and consume events, guaranteeing that the schema is correct.

Regardless of the tooling that's used, it is critical to ensure correctness of events and APIs. If you find a place in production where behavior differs from the declared spec, I think that's the point when to pull the Andon Cord, stop everyone, and fix the process so it can never happen again. Once developers lose trust in the specification, it turns into a huge bottleneck.

The semantics we chose mean all of our producers guarantee at-least-once delivery, and all of our consumers must assume multi-delivery, and therefore must implement idempotency.

For the producers, we create a journal of all the operations on the table. The journal stores every insert, update, and delete along with the operation on that table, so we have a complete history of everything that happened. For example, the users table contains the latest view of every user, and the `journal.users` table has a record of every operation. This means replay becomes quite simple because you can just replay the entire record.

Consumers read off of Kinesis and get a batch of records, then insert them into their local database, in temporary storage, which is partitioned for fast removal. On event arrival, we queue that there is a record to be consumed, and process them in micro batches. Any failures are recorded locally and published to a monitoring system.

Our event schemas are defined using API Builder, because it's one tool, so there's less to learn. We create one model per event, and group multiple related events into a union type, with the naming convention of "`{microservice}_event.`" So so user_event will contain the `user_upserted` and `user_deleted` events. We use a schema linter to enforce the naming conventions. We then map one union type to a single stream in Kinesis. A stream can only be owned by one service and most services define exactly one stream. This means only the user's service can publish user events, so you can always determine the source.

Event Producers and Consumers

Writing application code starts with getting a stream. We invested a lot of time in building our library for eventing, named queue, so that developers had a simple experience when working with events. Creating a stream is a single line of code: `val stream = queue.producer[UserEvent]()`. The developer doesn't need to provide a stream name, because we use reflection to figure out what stream it should be publishing to.

Producing an event again leverages code generation, with a UserVersion object based on the journal.users table. This guarantees that all code in all services

looks the same. An unspoken benefit is that any backend developer can drop in to any microservice and be productive. The same is true on the consumer side, where an event is received as JSON, cast to a type such as `UserEvent`, then pattern matched to handle specific cases of `UserUpserted` or `UserDeleted`.

Like other testing, end-to-end tests for events on both the producer and consumer sides are also easy to write. Because of the investment in the streaming library, when a test passes locally, it will also work in production.

Dependencies

At this point, it may seem like everything's great and we're done. Our service works in production, we have a database, and can create users. But the dependencies is where things get really interesting.

With microservices, you have to choose to handle dependencies, and there are two broad extremes. The first extreme is once a service is deployed, you never touch it; if you need to make an improvement, you might as well rewrite it. The other extreme is to pay a tax as you go, and keep your dependencies up-to-date. We've chosen to pay that tax.

Our goal with dependency management is to automatically update all of our services to the latest dependencies. I believe this is the right way to go. Although there's a tax you pay, it means a critical bugfix or security patch can be deployed in hours, not weeks or months.

We typically update all of our dependencies once a week, sometimes twice. I think our investment in the tooling around dependency management is one of the best things we've done. https://dependency.flow.io/ is an open source project where you can add your github project, and it will crawl your project and automatically extract all the libraries and binary dependencies. It crawls all the resolvers around the world, and keeps track of every version of every library. It turns that into an event stream of recommendations for you, the human, to decide which dependencies you want to update. More details are at https://github.com/flowcommerce/dependency.

Critical Decisions

In summary, I think there are three critical decisions. Number one is to design your schema first for all your APIs and events. While you need the API, you should focus on consuming the events.

Second is a high level of investmentment in automation, across the board. Automation should really be leveraged for code generators, the deployment system, and dependency management.

Third, enable teams to write amazing and simple tests. This drives quality, streamlines maintenance, and enables continuous delivery.

TL;DR

- A great architecture is able to support future features, naturally. The only way to know you have a good design is to wait a few years and see if people like using it.
- Design your schema first, for all your APIs and events. While you need the API, you should focus on consuming the events.
- Automation should be leveraged for code generators, the deployment system, and dependency management.
- Enable teams to write amazing and simple tests. This drives quality, streamlines maintenance, and enables continuous delivery.
- Continuous delivery is a prerequisite to managing microservice architectures.

SPONSORED ARTICLE

Augmenting Your Service Mesh with API Management

Software architects and developers today are increasingly adopting microservice development for faster innovation. Key drivers of this adoption include the need for smaller teams, agile software development life cycles, freedom to use heterogeneous technologies, and early feedback cycles.

But microservices come with their own challenges. Because of the disaggregation of architectures, the number of endpoints is exploding and communication between these endpoints has become a key challenge. Now, development teams have to think about service discovery, network resiliency, and secure communication in addition to solving the real business problem at hand.

On the other hand, DevOps teams need to tune their deployment engine to roll out frequent releases without disrupting the end users. Also, increasing the number of endpoints means tracing a runtime issue will be harder and requires extra effort in enabling observability on all distributed components. As a solution to these challenges, software architects have introduced service mesh: a promising, future proof software architecture.

To gain value from the business logic contained within a microservice, it needs to be exposed via APIs. Even though multiple implementations of service meshes, such as Istio, exist today, none of them address the concern of how these APIs can be exposed in a controlled and secure manner to their API consumers. API management goes beyond merely managing the development, deployment, and resilience of microservices to provide broader business value for organizations in all stages of the API lifecycle. This includes designing, publishing, documenting, analyzing, and monetizing APIs in a secure environment.

It's a common misconception that implementing a service mesh means there is no requirement for API management. But the key difference is that the service mesh manages services and API management manages APIs.

Istio: An Open Source Service Mesh

Istio is an open source service mesh that reduces the complexity of managing microservices by providing a uniform way to connect, secure, control, and monitor microservices. It is deployed as a sidecar proxy that intercepts all communication between microservices. Its functionality includes:

- Automatic load balancing
- Fine-grained control of traffic
- Support for access controls, rate limits, and quotas
- Metrics, logs, and traces for all traffic
- Secure service-to-service communication

WSO2 API Manager: The Complete Open Source Platform for API Management

WSO2 API Manager is an open source approach to addressing any spectrum of the API lifecycle, monetization and policy enforcement. As part of the larger WSO2 Integration Agile Platform, it is a central component used to deploy and manage API-driven ecosystems. It's hybrid integration capabilities further simplify projects that span traditional as well as microservice environments.

How Istio and WSO2 API Manager Work Together

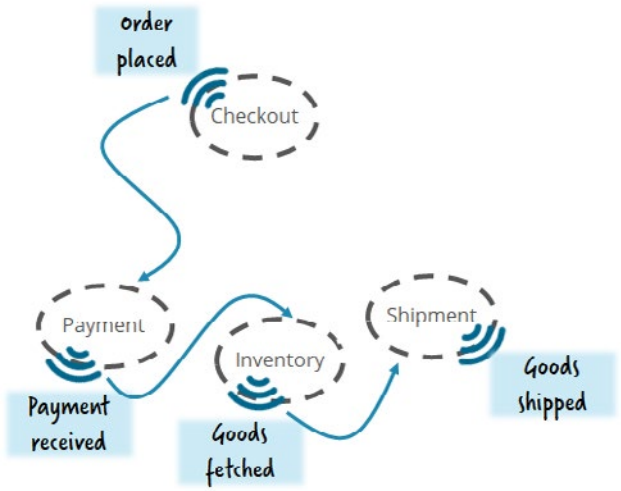
Whenever a service developer deploys a microservice, Istio injects an Envoy sidecar as a service proxy. For each request sent to the microservice, the sidecar proxy will capture a set of data and publish it to the Mixer (see Figure).

READ THE FULL ARTICLE

Monitoring and Managing Workflows across Collaborating Microservices

by **Bernd Rücker,** Co-Founder and Technologist at Camunda

Last year I met an architect from a big e-commerce organisation. He told me that they do all the right things and divide their functionality into smaller pieces along domain boundaries, even if they don't call this architectural style “microservices”. Then we talked about how these services collaborate to carry out business logic that crosses service boundaries, as this is typically where the rubber meets the road. He told me their services interact via events published on an event bus, which is typically known as “choreography” (and this concept will be explained in greater detail later). They considered this to be optimal in terms of decoupling. But the problem they face is that it becomes hard to understand what's happening and even harder to change something. “This is not like that choreographed dances you see in slides of some microservices talks; this is unmanageable pogo jumping!”



Choreographed event flow

This matches what other customers tell me, e.g. Josh Wulf from Credit Sense said, "the system we are replacing uses a complex peer-to-peer choreography that requires reasoning across multiple code-bases to understand."

Let's investigate this further using a simplified example. Assume you build an order fulfillment application. You choose to implement the system using an event-driven architecture and you use, for example, Apache Kafka as an event bus. Whenever somebody places an order, an event is fired from the checkout service and picked up by a payment service. This payment service now collects money and fires an event which gets picked up by an inventory service.

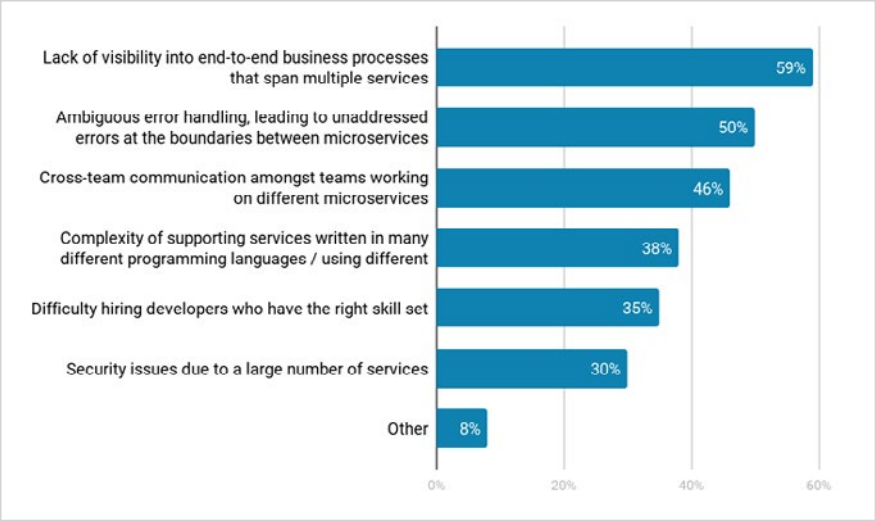
The advantage of this way of working is that you can easily include new components into the system. Assume you want to build a notification service to send emails to your customer; you can simply add a new service and subscribe to relevant events, without touching any of the other services. You can now manage communication settings and handle all complexity of GDPR compliant customer notifications in that central place.

This architectural style is called choreography, as there is no orchestrator required which tells others what to do. In contrast, every component emits events and others can be reactive to them. The assumption is that this style reduces coupling between the components and systems become easier to develop and change, which is true for the sketched notification service.

Losing Sight of the Flow of Events

In this article I want to concentrate on the most-often asked question whenever I discuss this architecture: How do we avoid losing sight (and probably control) of the flow of events? In a recent survey, Camunda (the company I work for) asked about the adoption of microservices. 92% of all respondents at least consider microservices, and 64% already do microservices in some form. It is more than just hype. But in that survey we also asked about challenges, and found a clear confirmation of this risk; the top answer was about lack of visibility into end-to-end business processes that span multiple services.

Remember architectures based on a lot of database triggers? Architectures where you never exactly knew what would happen if you did this - and wait - why did that happen now? Challenges with reactive microservices sometimes remind me a bit of this, even if this comparison is clearly misplaced.



Establishing Visibility

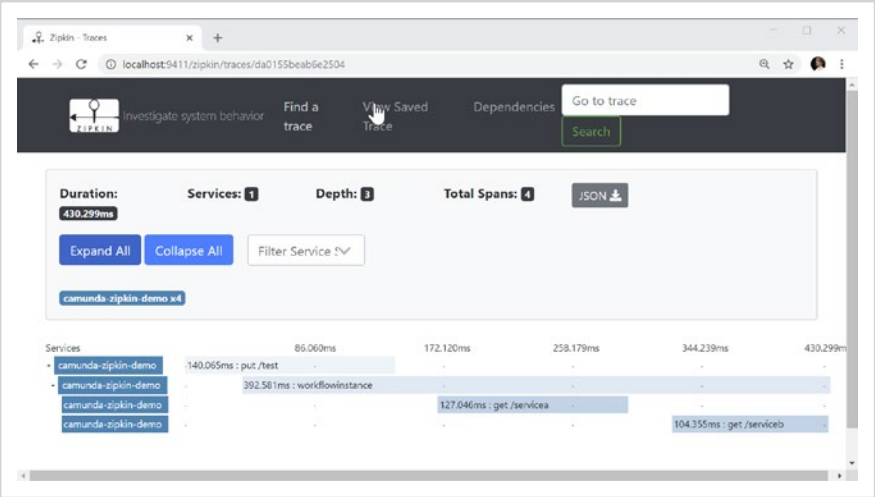
But what can we do about it? The following approaches can help you get back visibility, but each have their different pros and cons:

1. Distributed tracing (e.g. Zipkin or Jaeger)
2. Data lakes or analytic tools (e.g. Elastic)
3. Process mining (e.g. ProM)
4. Tracking using workflow automation (e.g. Camunda)

Please be aware that all approaches observe a running system and inspect instances flowing through it. I do not know any static analysis tool that yields useful information.

Distributed Tracing

Distributed tracing wants to trace call-stacks across different systems and services. This is done by creating unique trace ids that are typically added to certain headers generically (e.g. HTTP or messaging headers). If everybody in your universe understands or at least forwards these headers, you can leave breadcrumbs while a request hops through different services.



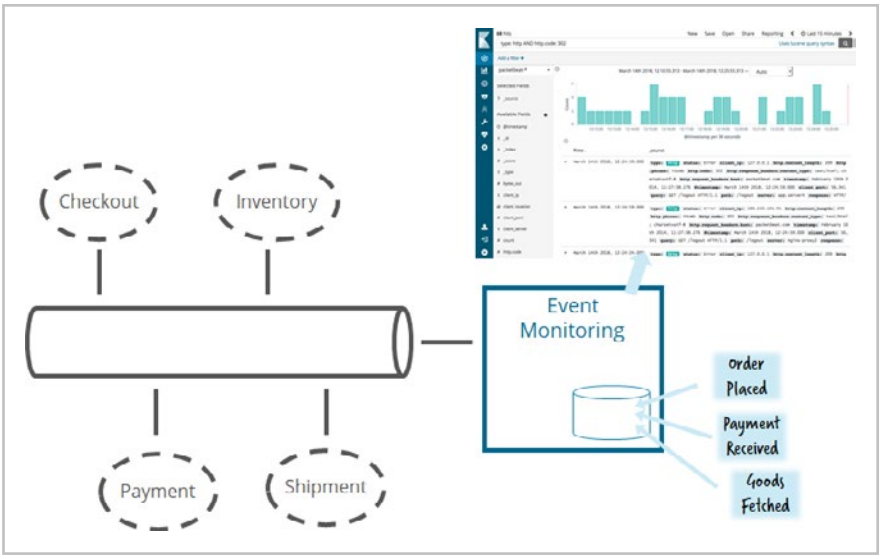
Distributed tracing is typically used to understand how requests flow through the system, to pinpoint failures or to investigate the root of performance bottlenecks. The great thing about distributed tracing is that there are mature tools with a lively ecosystem around. So it is relatively easy to get started, even if you typically have to (potentially invasively) instrument your applications or containers.

So why not use this to actually understand how business processes emerge by events flowing through our system? Well, basically two reasons make it hard to apply distributed tracing for this use case:

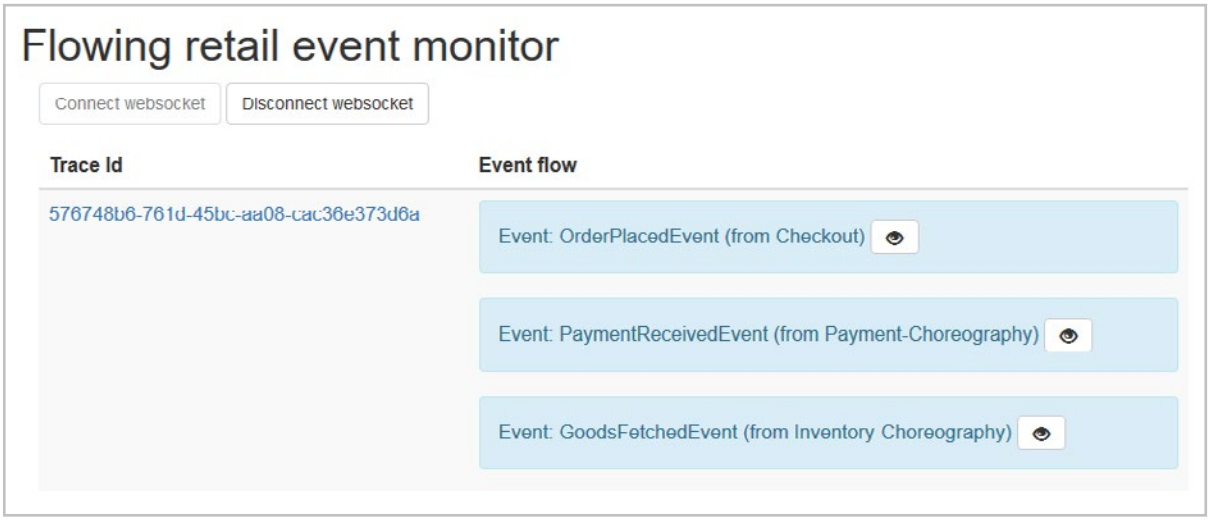
- Traces are hard to understand for non-engineers. My personal experiments aiming to show traces to non tech people failed miserably. It was far better to invest some time to redraw the same information with boxes and arrows. And even if all the information about method calls and messages is totally useful to understand communication behaviors, it is too fine-grained to understand the essence of cross-service business processes.
- To manage the overwhelming mass of fine-grained data, distributed tracing uses so-called sampling. This means only a small portion of all requests are collected. Typically, more than 90% of the requests are never recorded. A good take on this is Three Pillars with Zero Answers - towards a New Scorecard for Observability. So you never have a complete view of what's happening.

Data Lakes or Analytic Tools

So, out-of-the-box tracing will probably not be the way to go. The logical next step is to do something comparable, but bespoke to the problem at hand. That basically means not collecting traces, but instead collecting meaningful business or domain events that you might have flowing around already anyway. This often boils down to building a service to listen to all events and storing them in a data store that can take some load. Currently a lot of our customers use Elastic for this purpose.



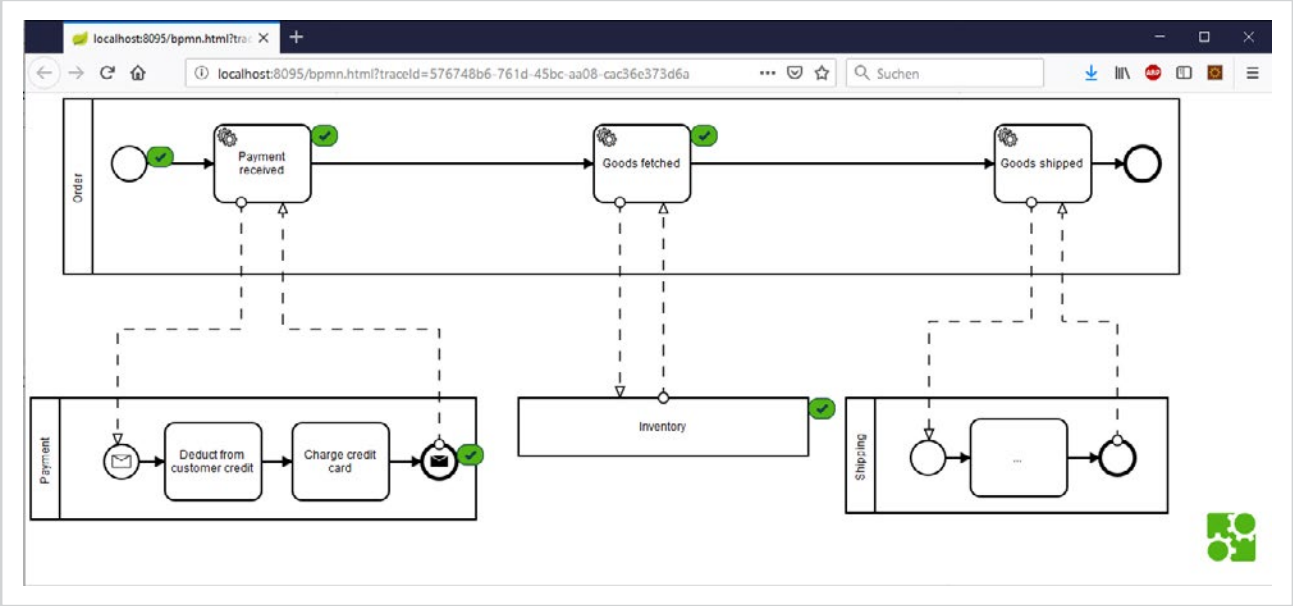
This is a powerful mechanism which is relatively easy to build. Most customers who work in an event-driven manner have this setup already. The biggest barrier for introduction is often the question of who will operate such a tool within a large organization, as it definitely needs to be managed by some centralized facility. It is also easy to build your own user interfaces on top of this to find relevant information for certain questions easily.



Example event monitor UI

One shortcoming is the lack of graphics to make sense out of a list of events. But you could build that into this infrastructure by for example projecting events onto some visualisation like BPMN.

Lightweight frameworks like bpmn.io allow you to add information to such a diagram in simple HTML pages (an example can be found here) which could also be packaged into a Kibana Plugin.)



This model is not executed by some workflow engine; it is a diagram used to visualize the captured events in a different way. In that sense you also have some freedom as to what granularity it will show, and it is also OK to have models that show events from different microservices in one diagram, as this is what you are especially interested in: the big picture. The good news is that this diagram does not stop you from deploying changes to individual services, so it does not hinder agility in your organization, but the tradeoff is that that introduces the risk of diagrams becoming outdated, when compared with the current state of the system operating in production.

Process Mining Tools

In the above approach you have to explicitly model the diagram you use for visualization, but if the nature of the event-flow is not known in advance, it needs to be discovered first.

This process discovery could be done by process mining tools. They can derive the overall blueprint and show that graphically, often even allowing to dig into a lot of detailed data, especially around bottlenecks or optimization opportunities.

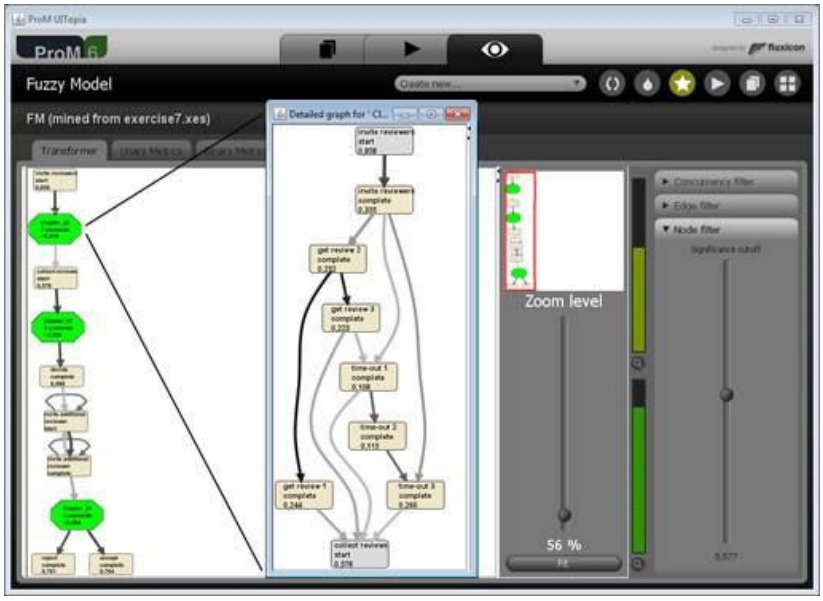


image source

This sounds like a perfect fit for our problem. Unfortunately, the tools are most often used to discover process flows within legacy architectures, so they focus on log file analysis and are not really good at ingesting live event streams. Another issue with these tools is that they are either very scientific and hard to use (like ProM) — or very much heavyweight (like Celonis). So in our experience it is often impractical to introduce these tools into typical microservice endeavors.

Nevertheless, process discovery and mining add interesting capabilities into the mix in order to get visibility into your event-flows and business processes. I hope that there will be technology emerging soon that offers comparable functionality but is also lightweight, developer-friendly and easily adoptable.

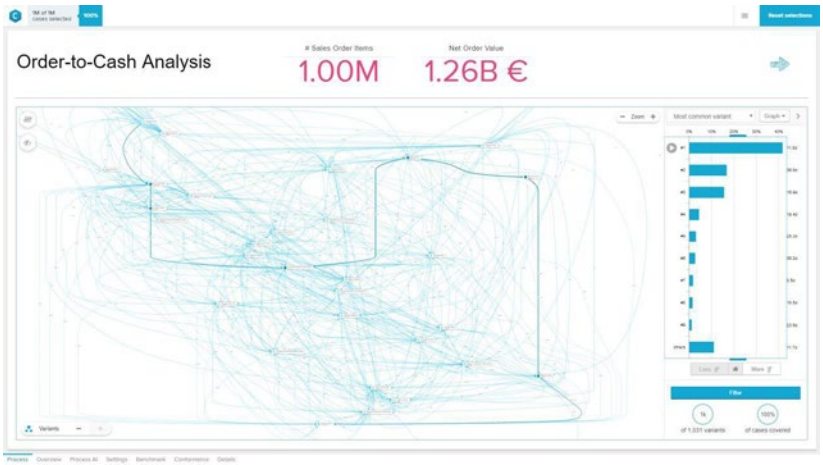
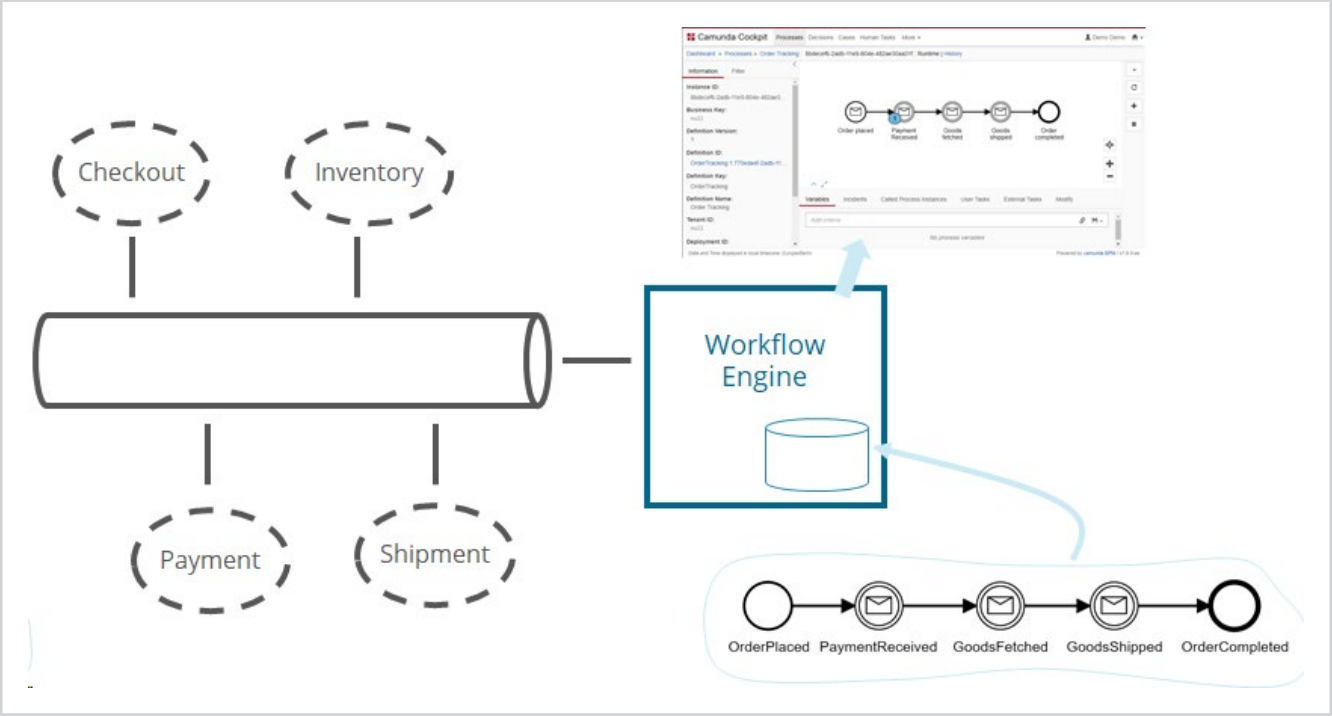


image source



Tracking via Workflow Automation

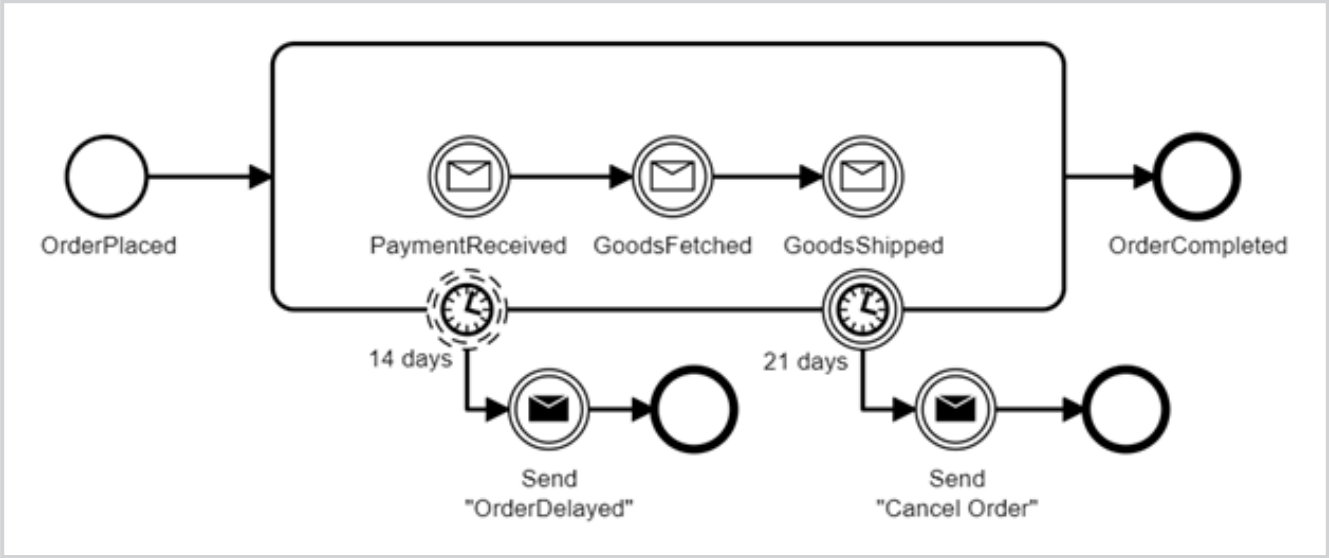
Another interesting approach is to model the workflow, but then deploy and run it on a real workflow engine. The workflow model is special in a sense, in that it is only tracking events, and not doing anything actively itself. So it does not steer anything -- it simply records. I talked about this at the Kafka Summit San Francisco 2018, and the recording includes a live demo using Apache Kafka and the open source workflow engine Zeebe.

This option is especially interesting as there is a lot of innovation in the workflow engine market, which is resulting in the emergence of tools that are lightweight, developer-friendly and highly-scalable. I wrote about this in Events, Flows and Long-Running Services: A Modern Approach to Workflow Automation. The obvious downside is that you have to model the workflow upfront. But in contrast to event monitoring, this model is executed on a workflow engine - in fact you start workflow instances for incoming events or correlate events to that workflow instance. This also allows conformance checks - does the reality fit into what you modeled?

And this approach allows you to leverage the complete tool chain of workflow automation platforms, which allows to see what's currently going on, monitor SLA's and detect stuck instances or do extensive analysis on historical audit data.

When I validated this approach with customers it was easy to set up. We just had to build a generic component picking up events from the bus and correlate it to the workflow engine. Whenever an event could not be correlated, we used a small decision table to decide if it could be ignored

Sample workflow monitoring (from camunda.com)



or would yield in an incident to be checked later. We also instrumented workflow engines used within microservices to execute business logic to generate certain events (e.g. workflow instance started, ended or milestone reached) to be used in the big picture.

This workflow tracking is a bit like event-monitoring, but with a business process focus. Unlike tracing, it can record 100% of your business events and provide a view that is suitable for the various stakeholders.

The Business Perspective

One big advantage of having the business process available in monitoring is that you understand the context. For a certain instance you can always see how and why it ended up in the current state, which enables you to understand which path it did not take (but other instances often do) and what events or data lead to certain decisions. You can also get an idea of what might happen in the near future. This is something you miss in other forms of monitoring. And even if it is often not currently hip to discuss the alignment between business and IT, it is absolutely necessary that non-engineers also understand business processes and how events flow through various microservices.

A Journey from Tracking to Managing

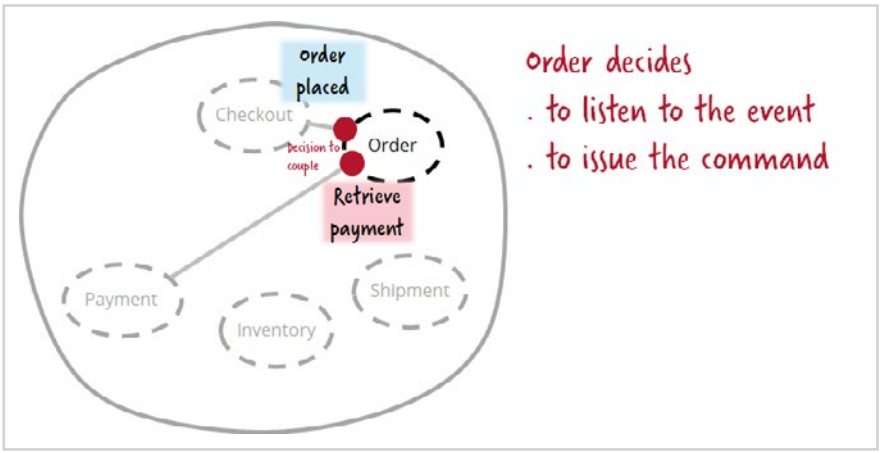
Process tracking is fine, as this gives you operational monitoring, reporting, KPIs and visibility as an important pillar to keep agility. But in current projects this tracking approach is actually just the first step in a journey towards more management and orchestration in your microservice landscape.

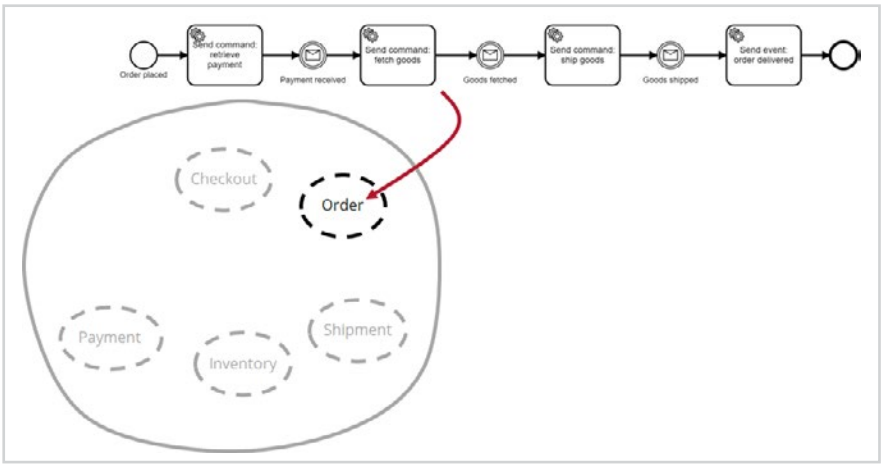
A simple example could be that you start to monitor timeouts for your end-to-end process. Whenever this timeout is hit, some action is taken automatically. In the following example we would inform the customer of a delay after 14 days — but still keep waiting. After 21 days we give up and cancel the order.

One interesting aspect in the above picture is the sending of the command cancel order. This is orchestration -- and this is sometimes discussed in a controversial fashion.

Orchestration

I often hear that orchestration should be avoided with the argument that it introduces coupling or violates autonomy of single microservices. And of course it is true that orchestration can be done badly, but it also can be done in a way that aligns with microservices principles while adding a lot of value to the business. At InfoQ New York 2018 I talked specifically about this misconception.





In its essence, orchestration for me means that one service can command another to do something. That's it.That's not tighter coupling, it is just coupled the other way round. Take the order example. It might be a good idea that the checkout service just emits an order placed event but does not know who processes it. The order service listens to that order placed event. The receiver knows about the event and decides to do something about it; the coupling is on the receiving side.

It is different with the payment, because it would be quite unnatural that the payment service knows what the payment is for. But it would need that knowledge in order to react on the right events, like order placed or order created. This also means it has to be changed whenever you want to receive payments for new products or services. Many projects work around this unfavorable coupling by issuing payment required events, but these are not events, as the sender wants somebody else to do something. This is a command! The order service commands the payment to retrieve money. In this case the sender knows about the command to send and decides to use it; the coupling is on the sending side.

Every communication between two services involves some degree of coupling in order to be effective, but depending on the problem at hand it may be more appropriate to implement the coupling within one side over the other.

The order service might even be responsible of orchestrating more services and keeping track of the sequence of steps in order fulfillment. I discussed the advantages in the above mentioned talk in detail. The tricky part is that a good architecture needs to balance orchestration and choreography, which is not always easy to do.

But for this article I wanted to focus on visibility. And there is an obvious advantage of orchestration using a workflow engine; the model is not only the code to execute the orchestration, it can also be used directly for visibility of the flow.

Summary

It is essential to get visibility into your business processes, independently of the way they are implemented. I discussed multiple possibilities, and in most real-world situations it typically boils down to some event-monitoring using Elastic-like tools or process tracking using workflow engines. This might depend slightly on the use case and role involved, so business analysts need to understand the data collected from all instances on the correct granularity, whereas operations need to look into one specific instance in varying granularity and probably want to have tools to resolve system-level incidents quickly.

If you are a choreographed shop, process-tracking can lead you to a journey towards more orchestration, which I think is a very important step in keeping control of your business processes in the long run. Otherwise, you might "make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus set yourself up for trouble in future years", as Martin Fowler puts it. If you are working on a more greenfield system, you should find a good balance for orchestration and choreography right from the beginning.

However, regardless of the implementation details of your system, make sure you have a business-friendly view of your business processes implemented by collaborating services.

TL;DR

- Peer-to-peer communication between components can lead to emergent behavior, which is challenging for developers, operators and business analysts to understand.
- You need to make sure to have the overview of all of the backwards-and-forwards communication that is going on in order to fulfill a business capability.
- Solutions that provide an overview range from distributed tracing, which typically misses the business perspective; data lakes, which require some effort to tune to what you need to know; process tracking, where you have to model a workflow for the tracking; process mining, which can discover the workflow; all the way through to orchestration, which comes with visibility built-in.
- In this article we argue that you need to balance orchestration and choreography in a microservices architecture in order to be able to understand, manage and change the system.

Lessons from 300k+ Lines of Infrastructure Code

by **Yevgeniy Brikman**, Co-founder of Gruntwork
Adapted by **Thomas Betts** from a presentation at QCon London 2019

A Confession about DevOps

At Gruntwork, we've learned some lessons about managing the ugly layer of infrastructure beneath your microservices. Although it's not a well-defined term, I'll refer to this as DevOps. And I have to start with a confession about DevOps.

The DevOps industry is very much in the stone ages, and I don't say that to be mean or to insult anybody. I just mean, literally, we are still new to this. We have only been doing DevOps for a few years, and we're still figuring out how to do it. What's scary about that is we're being asked to build amazing, cutting edge infrastructures, but I feel like the tooling we're using to do it looks like duct tape and paper clips.

You wouldn't know that if all you did was read blog posts where everything sounds really cutting edge. But my day-to-day experience as a developer feels far removed from all those headlines. Nothing seems to fit together quite right. Everything's just weirdly connected. You probably think, "Okay, I guess that works, but why are we doing it like this?"

We don't admit the fact that building things for production is hard. It's really hard. It actually takes a lot of work to go to production. It's really stressful. It's really time-consuming. From our experience working with many companies, and at previous jobs, we're able to quantify the time it takes to build production-grade infrastructure.

Production-grade infrastructure

Project	Examples	Time estimate
Managed service	ECS, ELB, RDS, ElastiCache	1 – 2 weeks
Distributed system (stateless)	nginx, Node.js app, Rails app	2 – 4 weeks
Distributed system (stateful)	Elasticsearch, Kafka, MongoDB	2 – 4 months
Entire cloud architecture	Apps, DBs, CI/CD, monitoring, etc.	6 – 24 months

If you are trying to deploy infrastructure for production use cases using a managed service in a cloud provider, you should expect to spend around two weeks. If you're building a stateless, distributed system on top of those services, with several apps and microservices, the time is doubled, to between two to four weeks.

For a stateful, distributed system that needs to write data to disk (and not lose that data), we go up an order of magnitude, to two to four months. Any large, complicated system is going to take you months before it's operational.

Finally, the entire cloud architecture: if you want to go build an entire production system on AWS, Azure, or Google Cloud, it will take six to 24 months. Six months, that's your tiny little startup, and 24 months and up is much more realistic for larger companies.

These are best case scenarios. This stuff takes a long time. As an industry, we don't talk about this enough. When you hear the story of some company's success, they don't tell you they spent three years working to get there.

Infrastructure as Code

There are some things that are getting better in this industry, though. Personally, I'm very happy to see more and more infrastructure managed as code. I think that's a game changer. Code gives you a tremendous number of benefits. It facilitates automation. You get version control. You can code review changes. You can write automated tests for your infrastructure. You can't code review or write automated tests for somebody manually deploying something. Your code acts as documentation, so how your infrastructure works doesn't just exist inside some CIS admin's head. And you can reuse code that you, or others, have written earlier.

I work at a company called Gruntwork, and we've built a reusable library of infrastructure code using a variety of technologies. We basically have pre-built solutions for many different types of infrastructure. Along the way, we've deployed this infrastructure for hundreds of companies; they're using it in production, and it's over 300,000 lines of code. My goal here is to share the things we got wrong, along with

the lessons we've learned along the way. Hopefully, as you're starting to deploy your microservices, you can benefit from some of these lessons, and not make the same mistakes that we did.

To do that, first I'll cover "The Checklist," which explains why infrastructure work takes as long as it does. Then I'll discuss some of the tools that we use and the lessons we've learned from them. I'll finish with how to build reusable infrastructure modules, how to test them, and how to release them.

The Checklist

I've generally found that there are really two groups of people. There are the people who have gone through the process of deploying a whole bunch of infrastructure, have suffered the pain, have spent those six to 24 months and understand it. And then there are the people who haven't, who are newbies. When they see these numbers, when they see that it's going to take them six to 24 months to get live for production infrastructure, they tend to say, "You've got to be kidding me. It's 2019. It can't possibly take that long." You have a lot of overconfident engineers who think, "Ah, maybe other people take that long. I'll get this done in a couple of weeks." No, you won't. It's just not going to happen. It's going to take a long time if you're doing it from scratch.

The real question is, why? Many engineers can relate to expecting to deploy something in a week, and it took three months. Where did that time go? There are really two main factors, and the first one is something called "yak shaving."

Yak shaving is basically a long list of tasks that you do before the thing you actually want to do. The best explanation I've seen comes from a story on Seth Godin's blog, and I encourage you to read it if you're not familiar with the term. Programmers run into this all the time. All you wanted to do was change the button color on some part of your product, and for some reason, you're dealing with some TLS certificate issue, and then you're fixing some CI/CD pipeline thing, and you seem to be moving backwards and sideways, rather than the direction you want to go. That's yak shaving.

In the DevOps space, I would argue that this is incidental complexity. The tools we have are all really tightly coupled, and not well-designed. Remember, we're still in the Stone Age. We're still learning how to do this. As a result, when you try to move one little piece that's stuck to everything else, you seem to just get into these endless yak shaves.

The second reason why deploying infrastructure takes so long is the essential complexity of infrastructure. This is the part of the problem that's actually real, and you have to solve, and I think most people aren't aware of what most of it is. What we've learned in this space is described in what we call our production-grade infrastructure checklist. Production-grade means infrastructure that you're willing to bet your company on. If you put your company's data in some database, you want to know that it won't be gone tomorrow and take your company out of business.

The first part of the checklist is the part that most people are very aware of. You want to deploy some piece of infrastructure, Kafka, ELK, microservices, whatever it is. You realize, "Okay, I've got to install some software. I have to configure it, tell it what port numbers to use, what paths to use on the hard drive. I have to get some infrastructure, provision it, might be virtual infrastructure in a cloud, and then I have to deploy that software across my infrastructure."

When you ask somebody to estimate how long it'll take to deploy something, these are the obvious things 99% of developers are thinking of. But this is page one out of four, and the other three pages are just as important, and most people don't take them into account when they're doing their estimates.

Page two of the checklist has things most people would agree are equally important. Security

Production-grade infrastructure checklist, part 1/4

Task	Description	Example tools
Install	Install the software binaries and all dependencies.	Bash, Chef, Ansible, Puppet
Configure	Configure the software at runtime: e.g., configure port settings, file paths, users, leaders, followers, replication, etc.	Bash, Chef, Ansible, Puppet
Provision	Provision the infrastructure: e.g., EC2 Instances, load balancers, network topology, security groups, IAM permissions, etc.	Terraform, CloudFormation
Deploy	Deploy the service on top of the infrastructure. Roll out updates with no downtime: e.g., blue-green, rolling, canary deployments.	Scripts, Orchestration tools (ECS, K8S, Nomad)

includes authentication, encrypting data, managing secrets, and server hardening. Each of these things can take weeks of work. Monitoring a new piece of infrastructure means deciding what metrics to gather and what alerts to trigger when those metrics are not where they should be. Logs have to be rotated on disc and aggregated to some central endpoint. If you're going to put some data in a database, you need backups and restore so you're confident that data isn't going to disappear tomorrow. This stuff takes time.

Production-grade infrastructure checklist, part 2/4

Task	Description	Example tools
Security	Encryption in transit (TLS) and on disk, authentication, authorization, secrets management, server hardening.	ACM, EBS Volumes, Cognito, Vault, CiS
Monitoring	Availability metrics, business metrics, app metrics, server, metrics, events, observability, tracing, alerting.	CloudWatch, DataDog, New Relic, Honeycomb
Logs	Rotate logs on disk. Aggregate log data to a central location.	CloudWatch Logs, ELK, Sumo Logic, Papertrail
Backup and restore	Make backups of DBs, caches, and other data on a scheduled basis. Replicate to separate region/account.	RDS, ElastiCache, ec2-snapper, Lambda

Page three starts with networking. This is especially important if you're thinking about microservices, which means thinking about service discovery and other features of a service mesh. It also includes handling IP addresses, subnets, SSH access, and VPN access to your infrastructure. These aren't optional, and this is where the time ends up going when you're deploying a piece of infrastructure. You also need to address high availability, scalability, and performance.

Production-grade infrastructure checklist, part 3/4

Task	Description	Example tools
Networking	VPCs, subnets, static and dynamic IPs, service discovery, service mesh, firewalls, DNS, SSH access, VPN access.	EIPs, ENIs, VPCs, NACLs, SGs, Route 53, OpenVPN
High availability	Withstand outages of individual processes, EC2 Instances, services, Availability Zones, and regions.	Multi AZ, multi-region, replication, ASGs, ELBs
Scalability	Scale up and down in response to load. Scale horizontally (more servers) and/or vertically (bigger servers).	ASGs, replication, sharding, caching, divide and conquer
Performance	Optimize CPU, memory, disk, network, GPU and usage. Query tuning. Benchmarking, load testing, profiling.	Dynatrace, valgrind, VisualVM, ab, Jmeter

Production-grade infrastructure checklist, part 4/4

Task	Description	Example tools
Cost optimization	Pick proper instance types, use spot and reserved instances, use auto scaling, nuke unused resources	ASGs, spot instances, reserved instances
Documentation	Document your code, architecture, and practices. Create playbooks to respond to incidents.	READMEs, wikis, Slack
Tests	Write automated tests for your infrastructure code. Run tests after every commit and nightly.	Terratest

Finally, page four has tasks which most people do not get to. Cost optimization is important to make sure your system doesn't bankrupt your company. Almost nobody gets to documentation, writing down what was done and why. Automated testing of infrastructure code is something few people even think of doing.

When you think, "how long is it going to take me to deploy something?" you're thinking of page one. Basically install, configure, deploy. I'm done. And you're forgetting about three other pages of really important stuff.

Not every piece of infrastructure needs every single one of these items, but you need to do most of them. When you go to build something in the future, go through that checklist and make very conscious, explicit decisions about the items you will or won't do. Your time estimate to deploy will at least be directionally correct, although you'll still be off by quite a bit because of yak shaving.

You can find a more complete version of the checklist at https://www.gruntwork.io/devops-checklist/. Use it. We use it all the time. It's really valuable.

Tools

People often ask what tools we use to implement the checklist. While I'll go over the tools that we use, it's important to understand that this is not a recommendation to you, because you will have your own requirements and use cases.

At Gruntwork, we like tools that let us manage our infrastructure as code, of course. We like using open source tools that have large communities, support multiple providers and multiple clouds, and that have strong support for reuse and composition of the code. Composable pieces are important at the infrastructure layer as well. Tools shouldn't require additional infrastructure and tooling just to use them. We like immutable infrastructure, so our tools also need to support that idea.

As of 2019, the toolset we've found most effective starts with Terraform and Packer, both of which are open-source tools that let you manage infrastructure as code. At the basic layer, all of our general networking, load balancers, integrations with services, and all the virtual servers are deployed and managed using Terraform, with HCL as the coding language. On top of that are virtual machine images, and we define and manage those as code using Packer.

Some of those virtual machine images run a Docker agent, with Kubernetes, ECS, or some other Docker cluster. Docker lets us define how to deploy and manage those services as code using a Dockerfile.

There's a hidden layer under the hood that most people don't tell you about, but it's always there. It's the glue to bind everything together. For that, we use a combination of bash scripts, Go for binaries, and Python scripts.

While this is our stack, and these are great tools, that's not really the takeaway. The real takeaway is that whatever toolset you end up with, it isn't going to be enough. The toolset you use won't matter at all unless you also change the behavior of your team and give them the time to learn these tools. Infrastructure-as-code is only is useful when combined with a change in how your team works on a day-to-day basis.

Changing Behavior

There's a simple example of where changing behavior is absolutely critical.

Imagine you need to make a change to the infrastructure. You do it manually, by directly connecting to a server and running some command. But when you introduce any of those tools, you're introducing a layer of indirection. Now, to make a change, you have to check out some code, change the code, and then run some tool or some process, which causes a change to the infrastructure. That's great, but the thing to remember is that these tools and processes take time to learn, understand, and internalize. And not just five minutes

Infrastructure-as-code is only useful when combined with a change in how your team works on a day-to-day basis.

of time. It takes weeks, months, potentially, for your team to get used to this. It takes much longer than doing it directly.

What happens when you have an outage? When something goes wrong, your sysadmin or DevOps person will have to make a choice. They can either spend five minutes making a fix directly, which they know how to do, or they can spend two weeks, or maybe two months, learning those tools. Unless you have prevented it upfront, they're going to make the change by hand.

Once you start making changes manually, then the infrastructure as code – that you worked so hard to create – no longer matches the reality of what is actually deployed and running. As a result, the next person who tries to use your code is going to get an error. Then they're going to say, "This infrastructure as code thing doesn't work. I'm going to just make a change manually." That propagates the problem until it never works for anyone. You might've spent three months writing the most amazing code, and in a week of outages, all of it becomes useless and no one's using it.

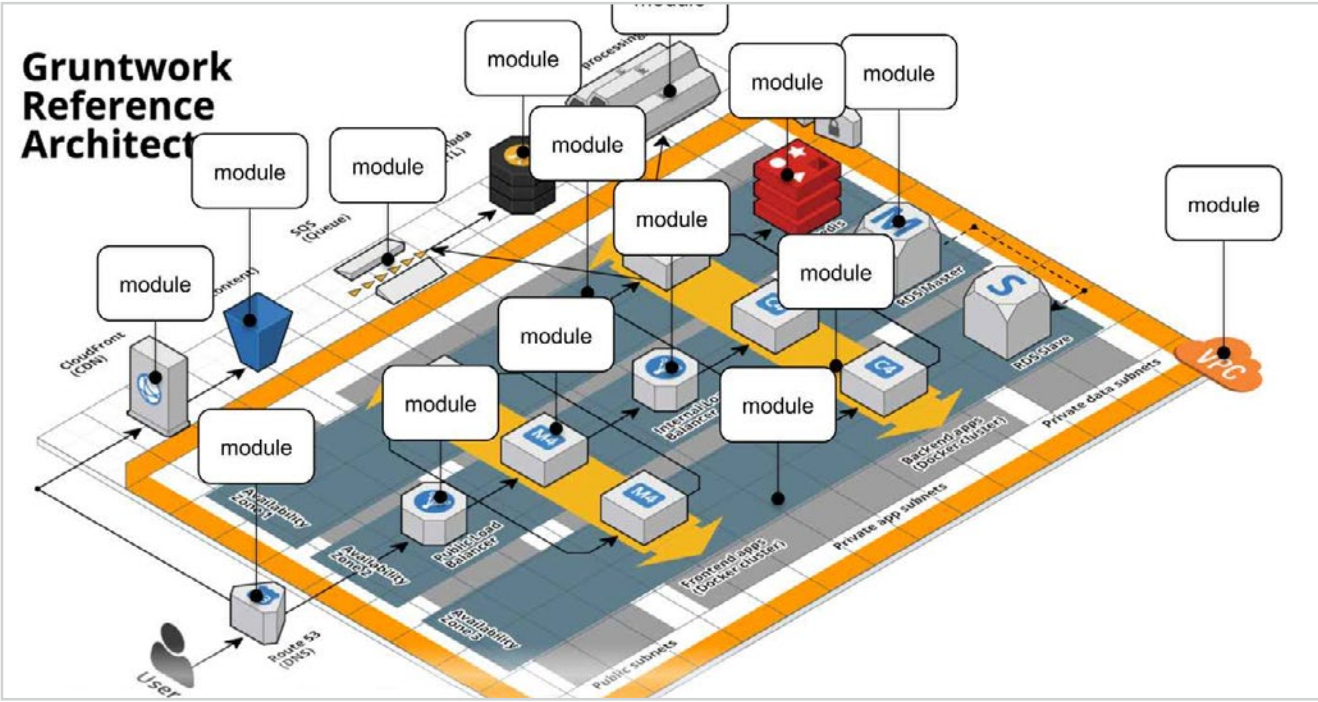
Changing things by hand does not scale, whereas code does. It's worth the time to use these tools *only* if you can also afford the time to let everybody learn them, internalize them, make them part of their process. If you don't do that, don't bother with the tools, because there are no silver bullets.

Modules

The third lesson we learned as we wrote this huge library of code is how to build reusable modules.

When people start using infrastructure as code, they often try to define all of their infrastructure in a single file, or maybe a single set of files that are all deployed and managed together, for all environments, Devs, Stage, QA, and Prod. There are many downsides to this approach. It's going to run slower just because there's more data to fetch and update. It's going to be harder to understand, code review, and test. It also prevents reuse and limits concurrency. There's also the possibility that one little typo anywhere will break everything, so a change in stage could accidentally take down prod.

The idea that large modules are a bad idea is not a novel concept. Infrastructure as code is still code. If you wrote a single function in Java that was 200,000 lines long, that probably wouldn't get through your code review process. But for some reason, when people go to infrastructure code, they assume that it's somehow different, they forget all the practices that they've learned, and they shove everything into one place.



At the very least, you want to make sure that your environments are isolated from each other, to avoid the accidental impact problem. But even within an environment, you want to isolate the different pieces from each other. A good rule of thumb is to isolate different components of your infrastructure based on how often they are deployed. Typically, this means your networking, data stores, and apps are each deployed separately. Because your microservices and individual apps might be deployed ten times per day, you don't want to risk breaking your networking because of a silly typo by deploying everything that frequently.

While that fixes the issue of, “Well, I broke everything,” it does not address the other concerns, such as testing and reuse. For that, I like to compare infrastructure code to functional programming.

Instead of writing one function, you write a bunch of individual little functions, each responsible for one thing, and then you compose them altogether. In the infrastructure, those “functions” are generically referred to as modules, but they behave the same as functions. They take in some inputs, they produce some output, and you should be able to compose them together. In this case, the output could be a server, a load balancer, a database, or whatever.

A good approach is to start by breaking down your environments, with dev, stage, and prod in separate folders. Within each environment, you separate the major components, with your networking layer separate from your database layer, separate from your app layer. Under the hood, those components are implemented with reusable modules, stored in a shared library that you can build out. That library itself should be made, again, from even simpler pieces. This is nothing new, it's just following good programming practices for any type of code.

If you want a more in-depth example of how to do this, check out the the Terraform module for how to run Vault on AWS using Terraform and Packer, available at https://github.com/hashicorp/terraform-aws-vault. The code is broken into standalone pieces. In the examples folder, we show you the different ways to put them together. Think of it as executable

documentation. The repo also has automated tests that deploy the examples.

Tests

One concern with infrastructure code is that it rots very, very quickly, because all of the underlying pieces are constantly changing. The cloud providers, tools like Terraform and Chef, and even Docker, are all changing all the time. This means that code doesn't last very long before it starts to break down. Put another way, any infrastructure code that does not have automated tests is broken.

Every single time that we have written a nontrivial piece of infrastructure, tested it to the best of our ability manually, even sometimes run it in production, taking the time to go and write an automated test for it, it has revealed nontrivial bugs. Those bugs may be in your code or in some dependency, but you won't catch them without automated tests. I mean this very literally. If you have a pile of infrastructure code that doesn't have tests, I absolutely guarantee that code is broken.

Testing infrastructure code is not like unit testing with general purpose languages. You can't run everything on localhost, and you can't mock all your outside dependencies, because most of what the code does is talk to a cloud provider in the outside world. If you try to mock the outside world, there's nothing left to test. That means you have to do end-to-end testing.

The test strategy is straightforward: deploy your infrastructure for real; validate that it works the way you expect it to; and then un-deploy it.

We use an open-source library for writing tests in Go, called Terratest. The name comes from Terraform, but it's now a general purpose tool for integration testing that works with Packer, Docker, Kubernetes, Helm Charts, AWS, Google Cloud, etc.

Using Terratest, start with thinking through the steps you'd go through to manually test something, then implement exactly those same steps as code. It's not magic, nor is it necessarily easy. But it's incredibly effective, because it will verify that your infrastructure actually does what it should after every single commit, rather than waiting until it hits production.

Testing Advice

Your tests are going to create a lot of stuff. You don't want to run those tests in production because that would cause issues, but you don't want to run them in your existing staging or dev accounts either. You actually want to create a completely isolated account just for automated testing. Then you can use cloud-nuke, or a similar tool, to completely delete an entire account. This needs to be on run as a cron job, because occasionally tests will fail and leave resources behind due to a bug in the test.

Testing infrastructure code should follow the classical test pyramid, which has unit tests at the bottom, integration tests on top of unit tests, and end-to-end tests at the very top. The only slight gotcha is that we don't have pure unit tests. The unit test equivalent in the infrastructure world is to test individual modules. This is why having small little pieces is so advantageous, because you can run a test for an individual module pretty quickly and easily, whereas running a test for your entire infrastructure will take hours.

The equivalent layer to integration tests in the pyramid combines multiple modules. Finally, at the top, you might be testing your entire infrastructure, but that's pretty rare because of the time involved. Testing infrastructure code just takes a lot longer than testing general purpose code. Instead of sub-second unit tests, an individual module can take between one to twenty minutes. Integration tests can take from five to 60 minutes. End-to-end tests depend completely on what your architecture looks like, and could take several hours to run. For your own sanity, you want to be as far down the pyramid as you can. That means creating small building blocks so you can test them in isolation.

Releases

Hopefully the next time your boss asks you to deploy something, you'll follow the process I've described. Start by going through a checklist to make sure you actually know what it is that you need to build, and you don't forget critical things like data backups and restores. Then you can write some code, in whatever tools make sense for your team. Hopefully, you've given your team the time to learn, adapt, and internalize those tools. Write tests for the deployment code to make sure it works. Peer review all code changes, just as you'd use pull requests for any code, before you release a new version of your library.

Creating a version of your library isn't anything fancy. It can literally be a git tag. The point of a version is to be a pointer to some immutable bit of code. In this case, it might be the code used to deploy a microservice for version 1.0, then 1.1, etc. You deploy that immutable artifact to your pre-prod environments. If it works well, you take the same immutable artifact and deploy it to staging and then production environments. Since it's the same code, it should work the same way.

By creating small, reusable modules, testing them and code reviewing them, you'll have a much more reliable and repeatable deployment process.

TL;DR

- Building production-grade infrastructure is hard, and it takes time – more time than most people expect.
- The DevOps industry is still in the stone age. The tools we use are not very sophisticated, but they're constantly improving.
- Using a checklist that covers all the major tasks involved in creating production infrastructure can help plan all the work involved, and how long it might take.
- Infrastructure code should be treated like any general-purpose code. Source control, code reviews, and versioning are all important.
- Infrastructure code that does not have automated tests should be considered broken.

Using Golang to Build Microservices at The Economist: A Retrospective

by **Kathryn Jonas,** Senior Software Engineer at Teachers Pay Teachers

I joined The Economist engineering team with the job title of Drupal Developer. However, my real task was to engage in a project that would fundamentally reshape the technology delivering Economist content. My first few months were spent learning Go, working with an external consultant for several months on building an MVP, and then rejoining the team to guide them on their journey to Go.

This shift in technology was driven by The Economist's mission to reach a broader digital audience as news consumption moved away from print. The Economist needed more flexibility to deliver content to increasingly diverse digital channels. To achieve this goal of flexibility and maintain a high level of performance and reliability, the platform transitioned from a monolith to microservice architecture. Services written in Go was a key component of the new system that would enable The Economist to deliver scalable, high performing services and quickly iterate new products.

Working with distributed data means wrestling with the guarantees promised to consumers.

Implementing Go at The Economist:

- Allowed engineers to quickly iterate and develop new features
- Enforced best practices on fast-failing services with smart error handling
- Provided robust support for concurrency and networking in a distributed system
- Lacked some maturity and support in some areas required for content and media
- Facilitated a platform that could perform at scale for digital publishing

Why did The Economist choose Go?

To answer this question, it's helpful to highlight the overall architecture for the new platform. The platform, called the Content Platform, is an event based system. It responds to events from different content authoring platforms and triggers a stream of processes run in discrete worker microservices. These services perform functions such as data standardization, semantic tagging analysis, indexing in ElasticSearch, and pushing content to external platforms like Apple News or Facebook. The platform also has a RESTful API, which combined with GraphQL, is the main entryway for front end clients and products.

While designing the overall architecture, the team investigated what languages would fit the platform needs. Go was compared against Python, Ruby, Node, PHP, and Java. While every language had its strengths, Go best aligned with the platform's architecture. Go's baked in concurrency and API support along with its design as a static, compiled language would enable a distributed eventing systems that could perform at scale. Additionally, the relatively simple syntax of Go made it easy to pick up and start writing working code, which was a quick win for a team going through so much technology transition. Overall, it was determined that Go was the language best designed for usability and efficiency in a distributed, cloud-based system.

Three years later, did Go meet these ambitious goals?

Several elements of the platform design were well aligned with the Go language. Failing Fast was a critical part of the system since is was composed of distributed, independent services. Aligning with the Twelve Factor App principles, applications needed to start and fail quickly. Go's design as a static, compiled language enables fast start up times and the performance of the compiler has continually improved and never been an issue for engineering or deployments. Additionally, the Go error handling design allowed applications to not only fail faster, but fail smarter.

Error Handling

A difference engineers quickly notice in Go is that is does not have exceptions, rather it has an Error type. In Go, all errors are values. The Error type is predeclared and is an interface. An interface in Go is essentially a named collection of methods, and any other custom type can satisfy the interface if it has those same methods. The Error type is an interface that can describe itself with a string.

```
type error interface {
    Error() string
}
```

This provides engineers with greater control and functionality around error handling. By adding an Error method that returns a string in any custom module, you can create custom errors and generate them like with the New function below, which comes from the Errors package.

```
type errorString struct {
    s string
}

func (e *errorString)
Error() string {
    return e.s
}
```

What does this mean in practice? In Go, functions allow multiple

return values, so if your function can fail, it will likely return an error value. The language encourages you to explicitly check for errors where they occur (as opposed to throwing and catching an exception), so you code will commonly have an "if err != nil" check. This frequent error handling can seem repetitive at first. However, Error as a value enables you to use the error to simplify your error handling. In a distributed system for example, one can easily enable retries by wrapping errors.

Network issues are always going to be encountered in a system, whether sending data to other internal services or pushing to third party tools. This example from the Net package highlights how you can take advantage of error as a type to distinguish temporary network errors from permanent ones. The Economist team used similar error wrapping to build in incremental retries when pushing content to external APIs.

```
package net

type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}

if nerr, ok := err.(net.Error); ok && nerr.Temporary()
{
    time.Sleep(1e9)
    continue
}

if err != nil {
    log.Fatal(err)
}
```

The Go authors believe not all exceptions are exceptional. Engineers are encouraged to sensibly recover from errors rather than let the application fail. Additionally, the Go error handling allows you to have more control over errors, which can improve things like your debugging or the usability of errors. Within the Content Platform, this design feature of Go enabled developers to make thoughtful decisions around errors, which resulted in stronger reliability of the system as a whole.

Consistency

Consistency is a critical factor in the Content Platform. At The Economist content is the core of business and the Content Platform's goal is to ensure content can be published once and read everywhere.

Therefore it's essential that every product and consumer has consistency from the Content Platform API. Products primarily use GraphQL to query the API, which requires a static schema that serves as a contract between consumers and the Platform. Content processed by the Platform needed to consistently align with this schema. A static language helped enforce this and enabled an easy win in ensuring data consistency.

Testing with Go

Another feature that improved consistency was Go's testing package. Go's fast compile times combined with testing as a first class feature enabled the team to embed strong testing practices into engineering workflows and quick failures in build pipelines. The Go tooling for tests makes them easy setup and run. Running "go test" will run all tests in a current directory and the test command has several helpful feature flags. The "cover" flag provides a detailed report on code coverage. The "bench" test runs benchmark tests, which are denoted by starting the test function name with the word "Bench" rather than "Test". The TestMain function provides methods for extra test setup, such as a mock authentication server.

Additionally, Go has the ability to create table test with anonymous structs and mocks with interfaces, improving test coverage. While testing is nothing new in terms of a language feature, Go makes it easy to write robust tests and embed them seamlessly into workflows. From the start, The Economist engineers were able to run tests as part of build pipelines with no special customization and even added Git Hooks to runs tests before pushing code to Github.

However, the project wasn't without struggles in achieving consistency. The first major challenge for the platform was managing dynamic content from unpredictable backends. The platform consumes content from source CMS systems primarily via JSON endpoints where the data structure and types were not guaranteed. This meant the platform couldn't use Go's standard encoding/json package, which supports unmarshalling JSON into structs, but panics if the struct field and incoming data field types do not match.

To overcome this challenge, a custom method for mapping backends to a standard format was needed. After a few iterations on the approach, the team implemented a custom unmarshalling process. While this approach felt a bit like rebuilding a standard lib package, it gave engineers fine grained control of how to handle source data.

Networking Support

Scalability was a focus for the new platform and this was supported with Go's standard libraries for networking and APIs. In Go, you can quickly implement scalable HTTP endpoints with no frameworks needed. In the below example, the standard library net/http package is used to setup a handler that takes a request and response writer. When the Content Platform API was first implemented, it used an API framework. This was eventually replaced with the standard library as the team recognized it meet all of their networking needs without additional bloat. Golang HTTP handlers scale because each request on a handler is concurrently run in a Goroutine, a lightweight thread, with no customization needed.

```
package main

import (
    “fmt”
    “log”
    “net/http”
)

func handler(w http.ResponseWriter, r
*http.Request) {
    fmt.Fprintf(w, “Hello World!”)
}

func main() {
    http.HandleFunc(“/”, handler)
    log.Fatal(http.ListenAndServe(“:8080”,
nil))
}
```

Concurrency Model

Go's concurrency model provided several performance gains across the platform. Working with distributed data means wrestling with the guarantees promised to consumers. As per the CAP theorem, it is impossible to simultaneously provide more than two out of the following three guarantees: Consistency. Availability. Partition tolerance. In The Economist platform, Eventual Consistency was accepted, meaning reads from data sources will eventually be consistent, and moderate delays in all data sources reaching a consistent state is tolerated. One of the ways this gap was minimized is by taking advantage of Goroutines.

Goroutines are lightweight threads managed by the Go runtime to prevent thread exhaustion. Goroutines enabled optimizing asynchronous tasks across the platform. For example, one of the Platform's data stores is Elasticsearch. When content is updated in the system, content referencing that item in Elasticsearch is updated and reindexed. By implementing Goroutines, reprocessing time was reduced, ensuring items are consistent faster. This example illustrates how items eligible for reprocessing are each reprocessed in a Goroutine.

```
func reprocess(searchResult *http.Response) (int, error) {
      responses := make([]response, len(searchResult.Hits))
      var wg sync.WaitGroup
      wg.Add(len(responses))

      for i, hit := range searchResult.Hits {
            wg.Add(1)
            go func(i int, item elastic.SearchHit) {
                  defer wg.Done()
                  code, err := reprocessItem(item)
                  responses[i].code = code
                  responses[i].err = err
            }(i, *hit)
      }
      wg.Wait

      return http.StatusOK, nil
}
```

Designing systems is more than simply programming, engineers have to understand what tools work where and when. While Go was a strong tool for most of The Economist's Content Platform needs, certain limitations required other solutions.

Dependency Management

When Go was released it had no dependency management system. Within the community several tools were developed to meet this need. The Economist used Git Submodules, which made sense at the time as the community was actively pushing for a standard dependency management tool.
As of today, while the community is closer to an aligned approach for dependency management, it's not there yet. At The Economist, the submodules approach didn't pose significant challenges, but it has been challenging for other Go developers and is something to consider when transitioning to Go.

There were also requirements for the Platform that Go's features or design was not best suited for.
As the Platform added support for audio processing, the Go tooling for metadata extraction at the time was limited, and so the team chose Python's Exiftool instead. Platform services run in docker containers, which enabled installing Exiftool and running it from the Go application.

```
func runExif(args []string) ([]byte,
error) {
      cmdOut, err := exec.
Command("exiftool", args...).Output()
      if err != nil {
            return nil, err
      }
      return cmdOut, nil
}
```

Another common scenario for the Platform is ingesting broken HTML from source CMS systems, parsing the HTML to be valid, and sanitizing the HTML. Go was initially used for this process, but because the Go standard HTML library expect a valid HTML input, a large amount of custom code was required to parse the HTML input before sanitation. This code quickly became brittle and missed edge cases, and so a new solution was implemented in Javascript. Javascript provided more flexibility and adaptability for managing the HTML validation and sanitation process.

Javascript was also a common choice for the event filtering and routing in the Platform. Events are filtered with AWS Lambdas, which are light weight functions spun up only when called. One use case is to filter events into different lanes, such as fast and slow lanes. This filtering is done based on a single metadata field in an event wrapper JSON object. The filtering implementation took advantage of the Javascript JSON pointer package to grab an element in a JSON object. This approach was far more efficient compared to the full JSON unmarshalling that would be required with Go. While this type of functionality could have been achieved with Go, using Javascript was easier for engineers and provided simpler Lambdas.

Go Retrospective

After implementing the Contact Platform and supporting it in production, if I was to hold a Go and the Content Platform retro, my feedback would be the following:

What Went Well?

- Key language design elements for distributed systems
- Concurrency model that's relatively easy to implement
- Enjoyable to write and fun community

What Could be Improved?

- Further progress on versioning and vendoring standards
- Lacks maturity in some areas
- Verbose for certain use cases

Overall, it's been a positive experience and Go is one of the critical elements that has allowed the Content Platform to scale. Go will not always be the right tool, and that's fine. The Economist has a polyglot platform and uses different languages where it makes sense. Go is likely never going to

be a top choice when messing around with text blobs and dynamic content, so Javascript is in the toolset. However, Go's strengths are the backbone that allows the system to scale and evolve.

When considering if Go would be the right fit for you, review the key questions for system design:

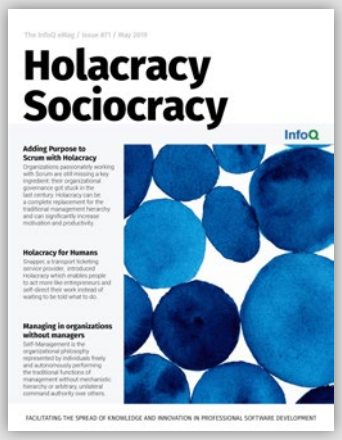
- What are your system goals?
- What guarantees are you providing your consumers?
- What architecture and patterns are right for your system?
- How will your system need to scale?

If you're designing a system that aims to tackle the challenges of distributed data, asynchronous workflows, and high performance and scaling, I encourage you to consider Go and how it can accelerate your system goals.

TL;DR

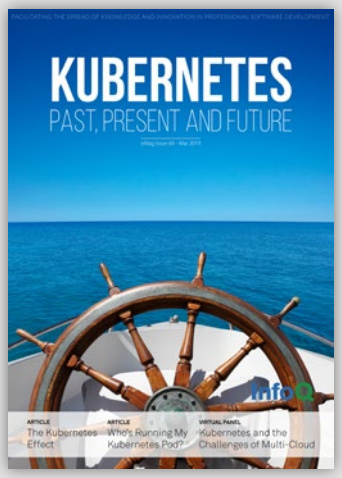
- The Economist needed more flexibility to deliver content to increasingly diverse digital channels. To achieve this goal of flexibility and maintain a high level of performance and reliability, the platform transitioned from a monolith to microservice architecture
- Services written in Go was a key component of the new system that would enable The Economist to deliver scalable, high performing services and quickly iterate new products.
- Go's baked in concurrency and API support along with its design as a static, compiled language would enable a distributed eventing systems that could perform at scale. Testing support is also excellent
- Overall, The Economist team's experience with Go has been positive experience, and this has been one of the critical elements that has allowed the Content Platform to scale.
- Go will not always be the right tool, and that's fine. The Economist has a polyglot platform and uses different languages where it makes sense.

Curious about previous issues?



In this eMag, we explore the real-world stories of organizations that have adopted some of these new ways of working: sociocracy, Holacracy, teal organizations, self-selection, self-management, and with no managers.

In this eMag, we present you expert security advice on how to effectively integrate security practices and processes in the software delivery lifecycle, so that everyone from development to security and operations understands and contributes to the overall security of the applications and infrastructure.



This eMag explores how Kubernetes is moving from a simple orchestration framework to a fundamental cloud-native API and paradigm that has implications in multiple dimensions, from operations to software architecture.

InfoQ