Container Solutions

The CLOUD NATIVE ATTITUDE

Move Fast Without Breaking Everything

Anne Currie

PART 2

Next Steps With Cloud Native

ABOUT THIS BOOK/BLURB

This is a small book with a single purpose, to tell you all about Cloud Native - what it is, what it's for, who's using it and why.

Go to any software conference and you'll hear endless discussion of containers, orchestrators and microservices. Why are they so fashionable? Are there good reasons for using them? What are the trade-offs and do you have to take a big bang approach to adoption? We step back from the hype, summarize the key concepts, and interview some of the enterprises who've adopted Cloud Native in production.

Take copies of this book and pass them around or just zoom in to increase the text size and ask your colleagues to read over your shoulder. Horizontal and vertical scaling are fully supported.

The only hard thing about this book is you can't assume anyone else has read it and the narrator is notoriously unreliable.

What did you think of this book? We'd love to hear from you with feedback or if you need help with a Cloud Native project
email info@container-solutions.com

This book is available in PDF form from the Container Solutions website at www.container-solutions.com

First published in Great Britain in 2017 by Container Solutions Publishing, a division of Container Solutions Ltd.

Chapter 7 "Distributed Systems Are Hard" first appeared in The New Stack on 25 Aug 2017

Design by Remember to Play / www.remembertoplay.co

ABOUT THE AUTHORS

Anne Currie

Anne Currie has been in the software industry for over 20 years working on everything from large scale servers and distributed systems in the '90's to early ecommerce platforms in the 00's to cutting edge operational tech on the 10's. She has regularly written, spoken and consulted internationally. She firmly believes in the importance of the technology industry to society and fears that we often forget how powerful we are. She is currently working with Container Solutions.

Container Solutions

As experts in Cloud Native strategy and technology, Container Solutions support their clients with migrations to the cloud. Their unique approach starts with understanding the specific customer needs. Then, together with your team, they design and implement custom solutions that last. Container Solutions' diverse team of experts is equipped with a broad range of Cloud Native skills, with a focus on distributed system development.
Container Solutions have global perspective and their office locations include the Netherlands, United Kingdom, Switzerland, Germany and Canada.

CONTENT

06

WHERE TO START - THE MYTHICAL BLANK SLATE?

A company of any size might start a project that appears to be an architectural blank slate. Hooray! Developers like blank slates. It's a chance to do everything properly, not like those cowboys last time. A blank slate project is common for a start-up, but a large enterprise can also be in this position.

However, **even a startup with no existing code base still has legacy.**

- The existing knowledge and experience within your team is a valuable legacy, which may not include microservices, containers or orchestrators because they are all quite new concepts.
- There may be existing third-party products or open source code that could really help your project but which may not be Cloud Native.
- You may possess useful internal code, tools or processes from other projects that don't fit the Cloud Native model.

Legacy is not always a bad thing. It's the abundance and reuse of our legacy that allows the software industry to move so quickly. For example, Linux is a code base that demonstrates some of the common pros and cons of legacy (e.g. it's a decent OS and it's widely used, but it's bloated and hardly anyone can support it). We generally accept that the Linux pros outweigh the cons. One day we may change our minds, but we haven't done so yet.

Using your valuable legacy might help you start faster, but push you away from a Cloud Native approach. So, what do you do?

What's Your Problem?

Consider the problems that Cloud Native is designed to solve: fast and iterative delivery, scale and margin. Are any of these actually your most pressing problem? Right now, they might not be. Cloud Native requires an investment in time and effort and that effort won't pay off if neither speed (feature velocity), scale nor margin are your prime concern.

Thought Experiment 1 - Repackaging a Monolith

Imagine you are an enterprise with an existing monolithic product that with some minor tweaks and repositioning could be suited to a completely new market. Your immediate problem is not iterative delivery (you can tweak your existing product fairly easily). Scale is not yet an issue and neither is margin (because you don't yet know if the product will succeed). Your goal is to get a usable product live as quickly and cheaply as possible to assess interest.

Alternatively, you may be a start-up who could rapidly produce a proof-of-concept to test your market using a monolithic framework like Ruby on Rails with which your team is already familiar.

So, you potentially have two options:

1. Develop a new Cloud Native product from scratch using a microservices architecture.

2. Rapidly create a monolith MVP, launch the new product on cloud and measure interest.

In this case, the most low-risk initial strategy might be option 2, even if it is less fashionable and Cloud Nativey. If the product is successful then you can reassess. If it fails, at least it did so quickly and you aren't too emotionally attached to it.

Thought Experiment 2 - It Worked! Now Scale.

Imagine you chose to build the MVP monolith in thought experiment 1 and you rapidly discover that there's a huge market for your new product. Your problem now is that the monolith won't scale to support your potential customer base.

Oh no! You're a total loser! You made a terrible mistake in your MVP architecture just like all those other short-termist cowboys! Walking the plank is too good for you!

What Should You Do Next?

As a result of the very successful MVP strategy you are currently castigating yourself for, you learned loads. You understand the market better and know it's large enough to be worth making some investment. You may now decide that your next problem is scale. You could choose to implement a new version of your product using a scalable microservices approach. Or you may not yet. There are always good arguments either way and more than one way to scale. Have the discussions and make a reasoned decision. Ultimately, having to move from a monolith to a Cloud Native architecture is not the end of the world, as we'll hear next.

The Monolithic Legacy

However you arrive at it, a monolithic application is often your actual starting point for a Cloud Native strategy. Why not just throw it out and start again?

What if the Spaghetti is Your Secret Sauce?

It's hard to successfully re-implement legacy products. They always contain more high-value features than is immediately apparent. The value may be years of workarounds for obscure field issues (been there). Or maybe the hidden value is in undocumented behaviours that are now taken for granted and relied upon by users (been there too).

Underestimated, evolved value increases the cost and pain of replacing older legacy systems, but it is real value and you don't want to lose it. If you have an evolved, legacy monolith then converting it to microservices is not easy or safe. However, it might be the correct next step.

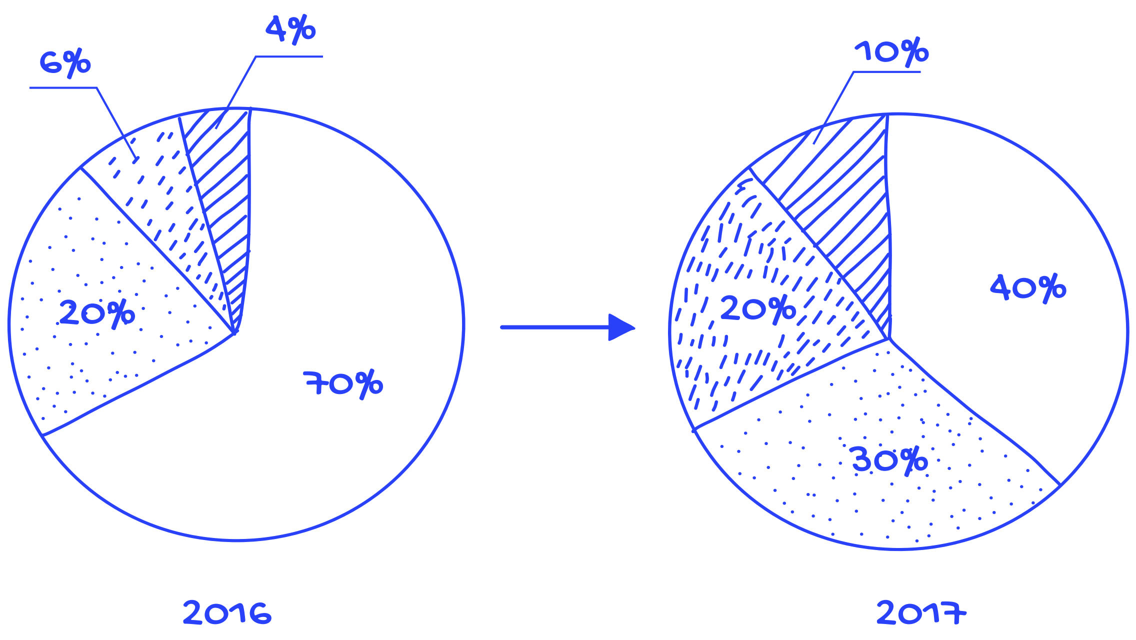
So what are folk doing? How do they accomplish the move from monolith to microservice?

Can a Monolith Benefit From Cloud Native?

To find out more about what folk are doing in real life I interviewed the charming engineer Daniel Van Gils of the DevOps-as-a-Service platform Cloud66 [9] about how their customers are working with Cloud Native. The data was very interesting.

All Cloud66 hosting is container-based so their customers are already containerized. They have over 500 users in production so the data is reasonably significant. How those clients are utilizing the service and how that has progressed over the past year draws a useful picture.

In June 2016:
- 70% of Cloud66's 500+ business users ran a containerized monolith.
- 20% had taken an "API-first" architectural approach and split their monolith into 2 or 3 large subservices (usually a front-end and a back-end) with a clear API between them. Each of these subservices was containerized and the front end was usually stateless.
- 6% had evolved their API-first approach further, often by splitting the back-end monolith into a small, distributable, scalable API service and small distributed back-end worker services.
- 4% had a completely native microservice architecture.



In January 2017, Cloud66 revisited their figures to see how things had progressed. By then:
- 40% were running a single containerized monolith, down from 70% six months earlier
- 30% had adopted the API-first approach - described above (separated services for back-end and front-end with a clear API), up from 20% in June 2016.
- 20% had further split the back-end monolith (> 3 different services), up from 6%.
- 10% were operating a native microservice architecture (> 10 different services), up from 4% the previous year.

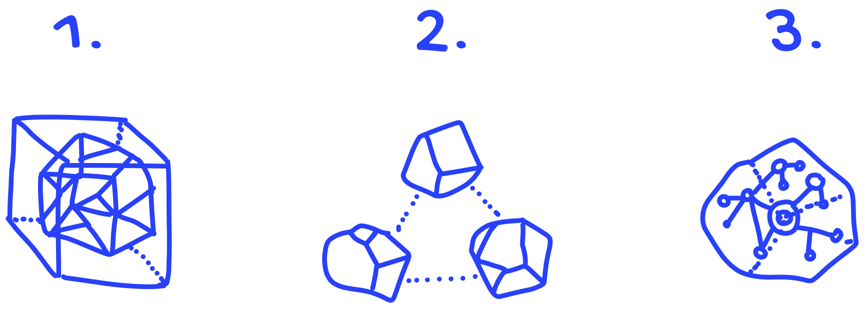
So, in 2016 96% of those who had chosen to containerize on the Cloud66 platform were not running a full microservice-based Cloud Native architecture. Even 6 months later, 90% were still not fully Cloud Native. However, Cloud66's data gives us some idea of the iterative strategy that some folk with monoliths are following to get to Cloud Native.

- First, they containerize their existing monolithic application. This step provides benefits in terms of ease of management of the containerized application image and more streamlined test and deploy. Potentially there are also security advantages in immutable container image deployments.
- Second, they split the monolithic application into a stateless and scalable front-end and a stateful (fairly monolithic) back-end with a clear API on the back-end. Being stateless the front-end becomes easier to scale. This step improves scalability and resilience, and potentially margin via orchestration.
- Third, they break up the stateful and monolithic back-end into increasingly smaller components, some of which are stateless. Ideally they split out the API at this point into its own service. This further improves scale, resilience and margin. At this stage, businesses might be more likely to start leveraging useful third-party services like databases (DBaaS) or managed queues (QaaS).

The Cloud66 data suggest that, at least for their customers, businesses who choose to go Cloud Native often iteratively break up an existing monolithic architecture into smaller and smaller chunks starting at the front and working backwards, and integrating third party commodity services like DBaaS as they go.

Iterative break-up with regular deployment to live may be a safer way to re-architect a monolith. You'll inevitably occasionally still accidentally lose important features but at least you'll find out about that sooner when it's relatively easier to resolve.

So, we can see that even a monolith can have an evolutionary strategy for benefitting from a microservice-oriented, containerized and orchestrated approach – without the kind of big bang rewrite that gives us all nightmares and often critically undervalues what we already have.



Example Cloud Native Strategies

So, there are loads of different Cloud Native approaches:

- Some folk start with CI and then add containerization.
- Some folk start with containerization and then add CI.
- Some folk start with microservices and add CI.
- Some folk slowly break up their monolith, some just containerize it.
- Some folk do microservices from a clean slate (as far as that exists).

Many enterprises do several of these things at once in different parts of the organization and then tie them together - or don't.

So is only one of these approaches correct? I take the pragmatic view. From what I've seen, for software the "proof of the pudding is in the eating". Software is not moral philosophy. The ultimate value of Cloud Native should not be intrinsic ("it's on trend" or "it's more correct"). It should be extrinsic ("it works for us and our clients").

If containers, microservices and orchestration might be useful to you then try them out iteratively and in the smallest, safest and highest value order for you. If they help, do more. If they don't, do something else.

Things will go wrong, try not to beat yourself up about it like a crazy person. Think about what you learned and attempt something different. No one can foresee the future. A handy alternative is to get there sooner.

In this chapter, I've talked a lot about strategies for moving from monolith to microservice. Surely just starting with microservices is easier? Inevitably the answer is yes and no. It has different challenges. In the next chapter I'm going to let out my inner pessimist and talk about why distributed systems are so hard. Maybe they obey Conway's Law, but they most definitely obey Murphy's Law - what can go wrong, will go wrong.

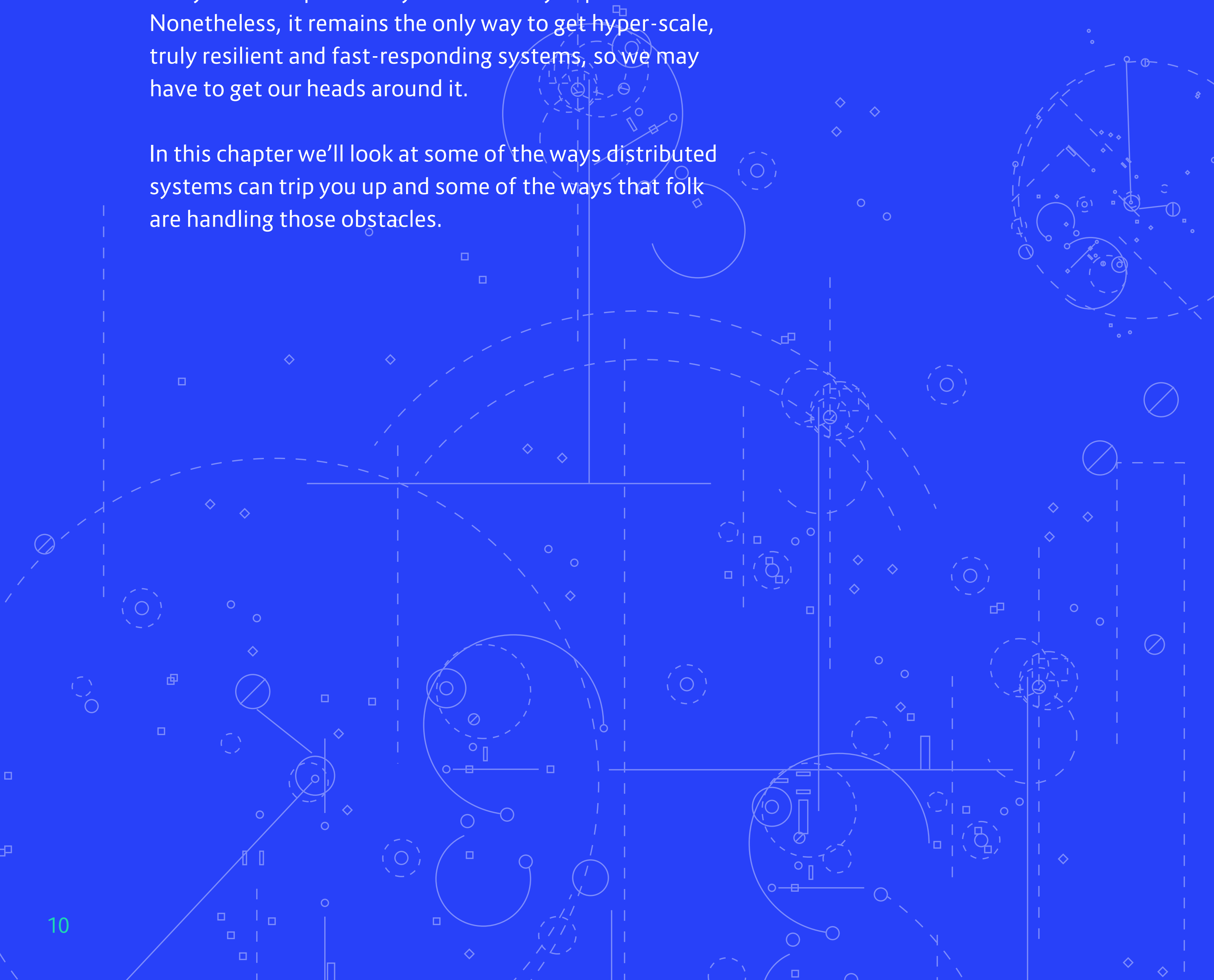
But does that matter?

07

DISTRIBUTED SYSTEMS ARE HARD

Nowadays I spend much of my time singing the praises of a Cloud Native (containerized and microservice-ish) architecture. However, most companies still run monoliths. Why? It's not merely because those folk are wildly unfashionable, it's because distributed is really hard and potentially unnecessarily expensive. Nonetheless, it remains the only way to get hyper-scale, truly resilient and fast-responding systems, so we may have to get our heads around it.

In this chapter we'll look at some of the ways distributed systems can trip you up and some of the ways that folk are handling those obstacles.



Anything That Can Go Wrong, Will Go Wrong

Forget Conway's law, distributed systems at scale follow Murphy's Law: "anything that can go wrong, will go wrong".

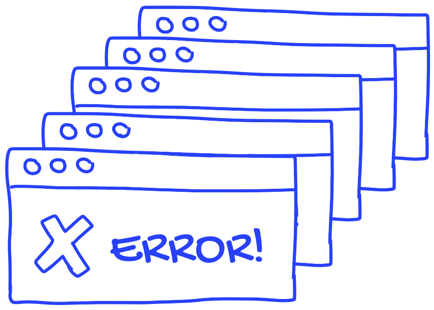
At scale, statistics are not your friend. The more instances of anything you have, the higher the likelihood one or more of them will break. Probably at the same time.

Services will fall over before they've received your message, while they're processing your message or after they've processed it, but before they've told you they have. The network will lose packets, disks will fail, virtual machines will unexpectedly terminate.

There are things a monolithic architecture guarantees that are no longer true when we've distributed our system. Components (now services) no longer start and stop together in a predictable order. Services may unexpectedly restart, changing their state or their version. The result is that no service can make assumptions about another - the system cannot rely on 1-to-1 communication.

A lot of the traditional mechanisms for recovering from failure may make things worse in a distributed environment. Brute force retries may flood your network and restores from backups are no longer straightforward. There are design patterns for addressing all of these issues but they require thought and testing.

If there were no errors, distributed systems would be pretty easy. That can lull optimists into a false sense of security.



Distributed systems must be designed to be resilient by accepting that "every possible error" is just business as usual.

What We've Got Here is Failure to Communicate

There are traditionally two high-level approaches to application message passing in unreliable (i.e. distributed) systems:

- Reliable but slow: keep a saved copy of every message until you've had confirmation that the next process in the chain has taken full responsibility for it.
- Unreliable but fast: send multiple copies of messages to potentially multiple recipients and tolerate message loss and duplication.

The reliable and unreliable application-level comms we're talking about here are not the same as network reliability (e.g. TCP vs UDP). Imagine two stateless services that send messages to one another directly over TCP. Even though TCP is a reliable network protocol this isn't reliable application-level comms. Either service could fall over and lose a message it had successfully received, but not yet processed, because stateless services don't securely save the data they are handling.

We could make this setup application-level-reliable by putting stateful queues between the services to save each message until it had been completely processed. The downside to this is it would be slower, but we may be happy to live with that if it makes life simpler, particularly if we use a managed stateful queue service so we don't have to worry about the scale and resilience of that.

The reliable approach is predictable but involves delay (latency) and work: lots of confirmation messages and resiliently saving data (statefulness) until you've had sign-off from the next service in the chain that they have taken responsibility for it.

A reliable approach does not guarantee rapid delivery but it does guarantee all messages will be delivered eventually, at least once. In an environment where every message is critical and no loss can be tolerated (credit card transactions for example) this is a good approach. AWS Simple Queue Service (Amazon's managed queue service) [10] is one example of a stateful service that can be used in a reliable way.

The second, unreliable, approach involves sending multiple messages and crossing your fingers. It's faster end-to-end but it means services have to expect duplicates and out-of-order messages and that some messages will go missing. Unreliable service-to-service communication might be used when messages are time-sensitive (i.e. if they are not acted on quickly it is not worth acting on them, like video frames) or later data just overwrites earlier data (like the current price of a flight). For very large scale distributed systems, unreliable messaging may be used because it is faster with less overhead. However, microservices then need to be designed to cope with message loss and duplication - and forget about order.
Within each approach there are a lot of variants (guaranteed and non-guaranteed order, for example, in reliable comms), all of which have different trade-offs in terms of speed, complexity and failure rate. Some systems may use multiple approaches depending on the type of message being transmitted or even the current load on the system.

This stuff is hard to get right, especially if you have a lot of services all behaving differently. The behaviour of a service needs to be explicitly defined in its API and it often makes sense to define constraints or recommended communication behaviours for the services in your system to get some degree of consistency. There are framework products that can help with some of this like Linkerd, Hysterix or Istio.

What Time Is It?

There's no such thing as common time, a global clock, in a distributed system. For example, in a group chat there's usually no guaranteed order in which my comments and those sent by my friends in Australia, Colombia and Japan will appear. There's not even any guarantee we're all seeing the same timeline - although one ordering will generally win out if we sit around long enough without saying anything new.

Fundamentally, in a distributed system every machine has its own clock and the system as a whole does not have one correct time. Machine clocks may get synchronized loads but even then transmission times for the sync messages will vary and physical clocks run at different rates so everything gets out of sync again pretty much immediately.

On a single machine, one clock can provide a common time for all threads and processes. In a distributed system this is just not physically possible.

In our new world then, clock time no longer provides an incontrovertible definition of order. The monolithic concept of “what time is it?” does not exist in a microservice world and designs should not rely on it for inter-service messages.

The Truth is Out There?

In a distributed system there is no global shared memory and therefore no single version of the truth. Data will be scattered across physical machines.

In addition, any given piece of data is more likely to be in the relatively slow and inaccessible transit between machines than would be the case in a monolith. Decisions therefore need to be based on current, local information.

This means that answers will not always be consistent in different parts of the system. In theory they should eventually become consistent as information disseminates across the system but if the data is constantly changing we may never reach a completely consistent state short of turning off all the new inputs and waiting. Services therefore have to handle the fact that they may get “old” or just inconsistent information in response to their questions.

Talk Fast!

In a monolithic application most of the important communications happen within a single process, between one component and another. Communications inside processes are very quick so lots of internal messages being passed around is not a problem. However, once you split your monolithic components out into separate services, often running on different machines, then things get trickier.
To give you some context:

- In the best case it takes about 100 times longer to send a message from one machine to another than it does to just pass a message internally from one component to another [11].
- Many services use text-based RESTful messages to communicate. RESTful messages are cross-platform and easy to use, read and debug but slow to transmit and receive. In contrast, Remote Procedure Call (RPC) messages paired with binary message protocols are not human-readable and are therefore harder to debug and use but are much faster to transmit and receive. It might be 20 times faster to send a message via an RPC method, of which a popular example is gRPC, than it is to send a RESTful message [12].

The upshot of this in a distributed environment is:

- Send fewer messages. You might choose to send fewer and larger messages between distributed microservices than you would send between components in a monolith because every message introduces delays (aka latency).
- Consider sending messages more efficiently. For what you do send, you can help your system run faster by using RPC rather than REST for transmitting messages. Or even just go UDP and handle the unreliability. That will have tradeoffs, though, in terms of developer productivity.

Status Report?

If your system can change at sub-second speeds, which is the aim of a dynamically managed, distributed architecture, then you need to be aware of issues at that speed. Many traditional logging tools are not designed to track that responsively. You need to make sure you use one that is.

Testing to Destruction

The only way to know if your distributed system works and will recover from unpredictable errors is to continually engineer those errors and continually repair your system. Netflix uses a Chaos Monkey to randomly pull cables and crash instances. Any test tool needs to test your system for resilience and integrity and also, just as importantly, test your logging to make sure that if an error occurs you can diagnose and fix it retrospectively - i.e. after you have brought your system back online.

All This Sounds Difficult. Do I Have To?

Creating a distributed, scalable, resilient system is extremely tough, particularly for stateful services. Now is the time to decide if you need it, or at least need it immediately.

Can your customers live with slower responses or lower scale for a while? That would make your life easier because you could design a smaller, slower, simpler system first and only add more complexity as you build expertise.

The cloud providers like AWS, Google and Azure are also all developing and launching offerings that could do increasingly large parts of this hard stuff for you, particularly resilient statefulness (managed queues and databases). These services can seem costly but building and maintaining complex distributed services is expensive too.

Any framework that constrains you but handles any of this complexity (like Linkerd or Istio or Azure's Service Fabric) is well worth considering.

The key takeaway is don't underestimate how hard building a properly resilient and highly scalable service is. Decide if you really need it all yet, educate everyone thoroughly, introduce useful constraints, start simple, use tools and services wherever possible, do everything gradually and expect setbacks as well as successes.

08

REVISE!

The past chapters have, in true tech style, been bunged full of buzzwords. We've tried to explain them as we went along but probably poorly so let's step back and review them with a quick Cloud Native Glossary.

Container Image - A package containing an application and all the dependencies required to run it down to the operating system level. Unlike a VM image a container image doesn't include the kernel of the operating system. A container relies on the host to provide this.
Container - A running instance of a container image (see above). Basically, a container image gets turned into a running container by a container engine (see below).
Containerize - The act of creating a container image for a particular application (effectively by encoding the commands to build or package that application).
Container Engine - A native user-space tool such as Docker Engine or rkt, which executes a container image thus turning it into a running container. The engine starts the application and tells the local machine (host) what the application is allowed to see or do on the machine. These restrictions are then actually enforced by the host's kernel. The engine also provides a standard interface for other tools to interact with the application.
Container Orchestrator - A tool that manages all of the containers running on a cluster. For example, an orchestrator will select which machine to execute a container on and then monitor that container for its lifetime. An orchestrator may also take care of routing and service discovery or delegate these tasks to other services. Example orchestrators include Kubernetes, DC/OS, Swarm and Nomad.
Cluster - the set of machines controlled by an orchestrator.
Replication - running multiple copies of the same container image.
Fault tolerance - a common orchestrator feature. In its simplest form fault tolerance is about noticing when any replicated instance of a particular containerized application fails and starting a replacement one within the cluster. More advanced examples of fault tolerance might include graceful degradation of service or circuit breakers. Orchestrators may provide this more advanced functionality or delegate it to other services.
Scheduler - a service that decides which machine to execute a new container on. Many different strategies exist for making scheduling decisions. Orchestrators generally provide a default scheduler which can be replaced or enhanced if desired with a custom scheduler.
Bin Packing - a common scheduling strategy, which is to place containerized applications in a cluster in such a way as to try to maximize the resource utilization in the cluster.
Monolith - a large, multipurpose application that may involve multiple processes and often (but not always) maintains internal state information that has to be saved when the application stops and reloaded when it restarts.
State - in the context of a Stateful Service, state is information about the current situation of an application that cannot safely be thrown away when the application stops. Internal state may be held in many forms including entries in databases or messages on queues. For safety, the state data needs to be ultimately maintained somewhere on disk or in another permanent storage form (i.e. somewhere relatively slow to write to).
Microservice - a small, independent, decoupled, single-purpose application that only communicates with other applications via defined interfaces.
Service Discovery - mechanism for finding out the endpoint (e.g. internal IP address) of a service within a system.

There's a lot we haven't covered here but hopefully these are the basics.

09

FIVE COMMON CLOUD NATIVE DILEMMAS

Adopting Cloud Native still leaves you with lots of tough architectural decisions to make. In this chapter we are going to look at some common dilemmas faced by folk implementing CN.

Dilemma 1 - Does Size Matter?

A question I often hear asked is “how many microservices should I have?” or “how big should a microservice be?” So, what is better, 10 microservices or 300?

300!

If the main motivation for Cloud Native is deploying code faster then presumably the smaller the microservice the better. Small services are individually easier to understand, write, deploy and debug.

Smaller microservices means you’ll have lots. But surely more is better?

10!

Small microservices are better when it comes to fast and safe deployment, but what about physical issues? Sending messages between machines is maybe 100 times slower than passing internal messages. Monolithic internal communication is efficient. Message passing between microservices is slower and more services means more messages.

A complex, distributed system of lots of microservices also has counter-intuitive failure modes. Smaller numbers are easier for everyone to grok. Have we got the tools and processes to manage a complicated system that no one can hold in their head?

Maybe less is more?

10,000!

Somewhat visionary Cloud Native experts are contemplating not just 300 microservices but 3000 or even 30,000. Serverless platforms like AWS Lambda could go there. There’s a cost for proliferation in latency and bandwidth but some consider that a price worth paying for faster deployment.
However, the problem with very high microservice counts isn’t merely latency and expense. In order to support thousands of microservices, lots of investment is required in engineer education and in standardization of service behaviour in areas like network communication. Some expert enterprises have been doing this for years, but the rest of us haven’t even started.

Thousands of daily deploys also means aggressively delegating decisions on functionality. Technically and organizationally this is a revolution.

Compromise?

Our judgment is distributed systems are hard and there’s a lot to learn. You can buy expertise, but there aren’t loads of distributed experts out there yet. Even if you find someone with bags of experience, it might be in an architecture that doesn’t match your needs. They might build something totally unsuited to your business.
The upshot is **your team’s going to have to do loads of on-the-job learning**. Start small with a modest number of microservices. Take small steps. A common model is one microservice per team and that’s not a bad way to start. You get the benefit of deployments that don’t cross team boundaries, but it restricts proliferation until you’ve got your heads round it. As you build field expertise you can move to a more advanced distributed architecture with more microservices.
I like the model of gradually breaking down services further as needed to avoid development conflicts.

Dilemma 2 - Live Free or Die!

Freedom vs Constraints
The benefit of small microservices is they're specialized and decoupled, which leads to faster deployment. However, there's also cost in the difficulty of managing a complex distributed system, and many diverse stacks in production. **Diversity is not without issues.**

The big players mitigate this complexity by accepting some operational constraints and creating commonality across their microservices. Netflix uses its Hystrix as a common connectivity library for its microservices. Linkerd from Buoyant serves a similar purpose of providing commonality, as does Istio from Google and Lyft. Some companies who used containerization to remove all environmental constraints from developers have begun reintroducing recommended configurations to avoid fixing the same problem in 20 different stacks.

Our judgement is this is perfectly sensible. Help your developers use common operational tools where there's benefit from consistency. Useful constraints free us from dull interop debugging.

Dilemma 3 - What Does Success Look Like Anyway?

Moving fast means quickly assessing if the new world is better than the old one. Devs must know what success looks like for a code deploy: better conversions, lower hosting costs or faster response times, for example?

Ideally, all key metrics would be automatically monitored for every deploy. Any change may have an unforeseen negative consequence (faster response times but lower conversions). Or an unexpected positive one (it fails to cut hosting costs but does improve conversion). We need to spot either.

If checking is manual that becomes the bottleneck in your fast process. So, assessing success is another thing that eventually needs to be encoded. At the moment, however, there's no winning product to do metric monitoring or A/B testing. Most of the folk we talk to are still developing their own tools.

Dilemma 4 - Buy, Hire or Train?

If you want feature velocity, then a valuable engineer is one who knows your product and users and makes good judgments about changes.

At the extreme end, devs might make changes based only on very high level directions (CTO of the UK's Skyscanner, Bryan Dove, calls this "radical autonomy"). Training existing staff is particularly important in this fast-iteration world. **If you go for radical autonomy then devs will be making decisions and acting on them. They'll need to understand your business as well as your tech.**

Folk can be bought or hired with skills in a particular tool, but you may need to change that tool. Your hard skills requirements will alter. **You'll need engineers with the soft skills that support getting new hard skills (people who can listen, learn and make their own judgments)**. In the Cloud Native world, a constructive attitude and thinking skills are much more important than familiarity with any one tool or language. You need to feel new tools can be adopted as your situation evolves.

Dilemma 5 - Serverless or Microservice?

Serverless aka Function-as-a-Service (like AWS Lambda or Google Cloud Functions or Azure Functions) sounds like the ultimate destiny of a stateless microservice? If a microservice doesn't need to talk directly with a local database (it's stateless) then it could be implemented as a function-as-a-service.

So why not just do that and let someone else worry about server scaling, backups, upgrades, patches and monitoring? You'd still need to use stateful products like queues or databases for handling your data but they too could be managed services provided by your cloud provider. Then you'd have no servers to worry about. This world has a high degree of lock-in (con) but little or no ops work (pro).

That is pretty attractive. Most folk are trying to reduce their ops work. Serverless plus managed stateful services could do that.

However, it's still early days for Functions-as-a-Service. At the moment, I suspect there's a significant issue with this managed world, which is the lack of strong tooling. In the same way that western civilization rests on the dull bedrock of effective sanitation, modern software development depends on the hygiene factors of code management, monitoring and deployment tools. With Serverless you'll still need the plumbing of automated testing and delivery. Tools will appear for Serverless environments but **I suspect there isn't a winning toolchain yet to save us from death by a thousand code snippets.**

Modern team-based software development needs plumbing. Most folk will have to create their own right now for Function-as-a-Services, so it's probably still for creative pioneers.

10

AFTERWORDS - SHOULD SECURITY BE ONE?

This chapter is an interview with the brilliant Sam Newman, author of “Building Microservices”, where we discussed the unique challenges of securing Cloud Native systems and microservice architectures. Sam’s book is a great read for more microservice-meatiness after this book, which is a mere taster. All the intelligent thought in this chapter I entirely attribute to Sam.

Are Microservices Very Secure or Very Insecure?

Unfortunately, the answer to this question is "yes".

The first thing that struck me when talking to Sam was that I'd written a whole chapter on Microservices architecture and, indeed, a whole book on Cloud Native, but I hadn't once mentioned security. That wasn't because I don't care about security or it's an innate mystery to me, it's just that it didn't strike me as a big issue to talk about. How wrong I was! Of course it is! And Sam very succinctly told me why.

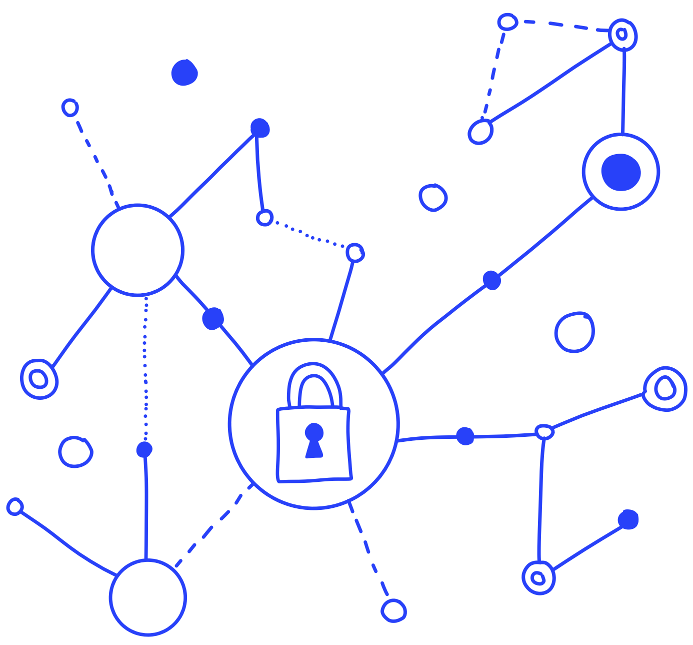
In a Cloud Native world probably **the biggest security challenge is microservices or, more accurately, how to secure a distributed system.**

What are Microservices Again?

As Sam puts it, microservices are independently deployable processes. That means in a system of microservices you can start, stop or replace any of them at any time without breaking everything. That's great for reducing clashes between developers, increasing resilience and improving feature velocity, but for security it's a double-edged sword. It can enable you to make everything more secure with better defense in depth (hurray!) but if you don't make a significant effort it can leave you in a much more exposed position than a monolith (damn!).

Hurray, Microservices are Secure!

Security-wise, the good thing about microservices, according to Newman, is that by dividing your system up you can separate data and processes into "highly sensitive or critical" and "less sensitive" groups and put more energy, focus and expenditure into protecting your high sensitivity and critical stuff. In the olden days of a monolith, everything was together in one place so it all had to be highly protected (or not, as the case may be). Your eggs were all in one basket, which colloquially we tend to disapprove of, although it is not actually an unknown security strategy - WW2 Atlantic convoys very successfully made use of a heavily defended single basket.



Microservices give you more opportunity to layer your defenses (defense in depth) but also more opportunities to fail to do so.
I'm sure you're getting the picture that this advantage isn't entirely clear cut.

Boo, Microservices are Insecure!

However, Sam also told me the downside of microservices is that by spreading your system out over multiple containers and machines you increase the attack surface. You have more to protect.

What kind of attack surfaces are we talking about?

- More machines means more OSs to keep patched for vulnerabilities.
- More containers means more images to refresh for vulnerability patches.
- More inter-machine messages means more communications need to be secured against sniffing (people reading your stuff on the wire) or changing the message payload (man-in-the middle attacks).
- More service-to-service comms means more opportunity for bad players to start talking to your services masquerading as you.

Basically, microservices are very powerful but also hard. They can improve your security but without careful thought they will probably reduce it. In Sam's correct judgment, microservice security needs to be considered and planned in from the start.

OK, so what can we do about it?

Threat Modelling

Sam recommends we use a process called "threat modelling", which helps us analyze potential points of weakness or likely attacks that our distributed, microservice system will have to withstand.

One useful technique for threat modelling is thinking up "attack trees" that cover every (often multistep) way a baddie could possibly attack your system and then putting a cost/difficulty against each attack.

For example: breaking into my house. The lowest attacker-cost way in would be climbing through an open window while I was out (easy). The highest cost way in might be fighting the sabre-toothed tiger on my doorstep (hard).

The idea is not to make every attack impossible but to make every attack too costly. Apparently my sabre-toothed tiger was complete overkill, I should just remember to close my windows.

Some attacks are physical (like breaking a window) and some are social (like persuading me to let you in to read a meter). The first you usually battle with tools and code, the second with processes.

Defend, Detect, Respond, Recover

According to Sam, a useful way to think about security and how to handle the attack points you've just uncovered with your attack tree is as a 4 step process:

1. Defend
2. Detect
3. Respond
4. Recover

Defend

So, what tools does he say we have that can secure microservices?

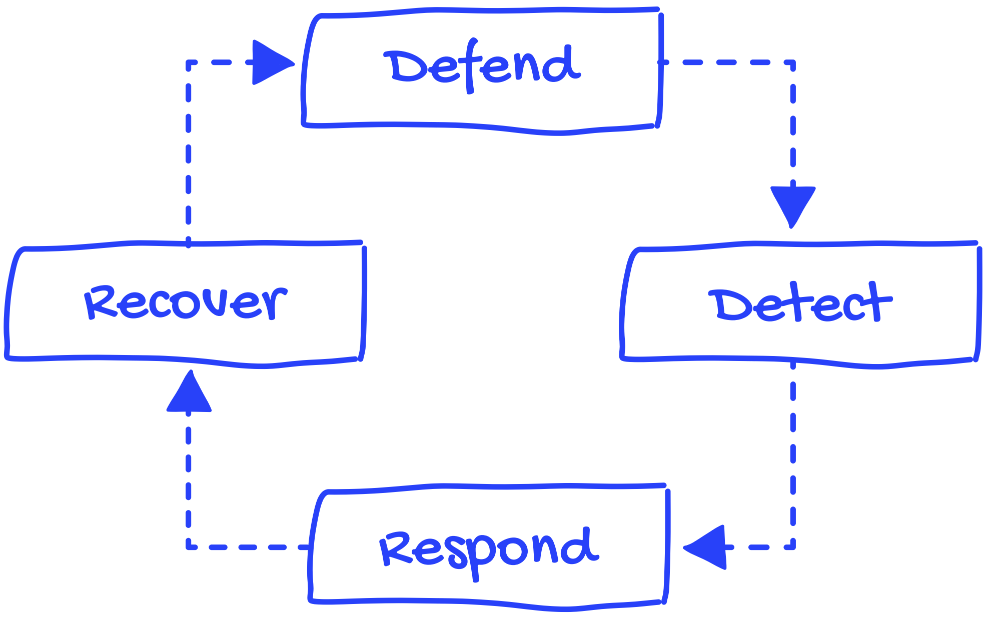
HTTPS

The first and easiest is HTTPS. If any of your microservices communicate over HTTP then stop. Move them to HTTPS. Just because a connection is inside your system perimeter that doesn't mean we can assume it's safe from snooping. The good news is HTTPS is not as hard as it used to be. There are now great tools and free certificates from Let'sEncrypt, amongst others. HTTPS also doesn't slow things down anymore because most servers are optimized for encryption.
Using HTTPS verifies the data hasn't been read or tampered with and verifies the callee, but it doesn't verify the caller. For that you'll need some form of client-side auth, such as client-side certificates. Don't have a heart attack, those are also easier than they used to be. Sam says take a look at Lemur from Netflix.

If you are using other forms of communication rather than REST/HTTP then there are ways to secure that too but that's too, complicated for this chapter so you'll have to read more of Sam's work to find out about that.

Authentication and Authorization

That covers service-to-service authentication, but what about user auth? What is a specific individual user allowed to do within the perimeter of your product? You still need to use OAuth or equivalent to cover that. You'll also have to consider whether or not services further downstream need to revalidate what a logged in user can do.



Networking

You'll probably also want to use SDN/network security and policy enforcement to make sure that traffic only ever comes at your services from other services they are allowed to talk to. Defense in depth folks! Policy AND encryption!

Patching

Everyone's security "open window", however, is usually patching. You've got to keep all your machines and containers patched for vulnerabilities. In a microservice environment you are probably going to end up with too many units to do this manually. You'll quickly need to automate this process. Look at tools that can help you do so.

Polyglot?

Microservices lend themselves to a best-of-breed or polyglot approach where everyone runs their dream stack. That has security advantages and disadvantages. Commonality is easier to secure until you've got your head round everything and automated loads of it. Keeping 5 stacks secure and patched is easier than 500. The benefit of diversity, however, is if your hackers do find an exploit then maybe they can take it less far, just compromise one microservice. Pros and cons abound but Sam recommended that you start with a smaller number of stacks and patch them carefully.

Detect

Logs! And keep your logs for a very long time. Sam points out that the usual demand for logs is from developers diagnosing a field issue from maybe a few days or weeks ago. Intrusion detection might involve investigating problems from a long time earlier than that so you need to keep logs longer. Look at the ELK stack: Logstash, Elasticsearch and Kibana for example.

IP-based security appliances or tools that detect unusual behaviour inside or at your perimeter are also very useful.

Respond

The success of your immediate response to an attack is less about tools and more about processes. Knowing what to do and then actually doing it.

1. **Don't panic!**
2. Don't ignore it!

Have processes that are pre-defined, carefully thought-through and tested for acting on attack detection. Don't wait until the problem happens to work out what to do next because in the heat of the moment you'll make mistakes.

Recover

This is the bread and butter stuff. Recovery from a security alert is actually just best practice for recovering from any disaster:

- Already have all your data backed up, in multiple locations with restore from backup tested.
- Already have your whole system recreatable at will (ideally automated build and deploy).
- In the event of an attack, patch as necessary and then burn it all down and restore everything from scratch.

That's a lot of stuff you have to get in place in advance. Tough, you're going to have to do it ;-) (that's me BTW, I'm sure Sam would not be so bossy).

So, Sam's overall conclusion was microservices are a hugely powerful tool for letting you build defense in depth, but also they also give you loads more opportunities to screw up and leave a window open so you need to think and plan.
I suspect the general advice is "Don't Panic". But also "Don't Ignore it!"

11 BONUS CHAPTER!

CLOUD NATIVE DATA SCIENCE

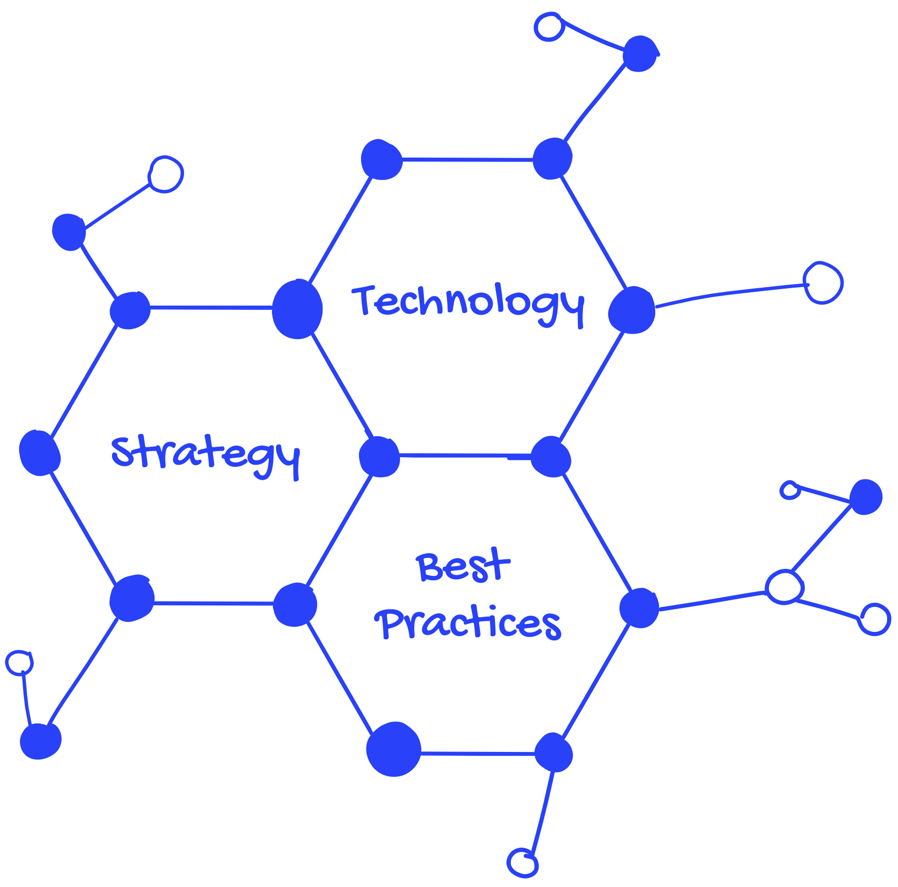
I know very little about Data Science, but this book felt like it was missing something rather important without covering how Cloud Native and data fit together. It's another area where the increased scale and speed of the cloud have revolutionized what we can do. So, Phil Winder has kindly contributed his expert thoughts on the subject in this bonus chapter, which explains the key challenges and opportunities introduced to Data Science by CN tools.

Cloud Native Meets Data

Data Science encapsulates the process of engineering value from data. Other people use different terms like Machine Learning or Artificial Intelligence or Big Data and usually they all mean the same thing. Given data, make a decision that adds value. Cloud Native Data Science (CNDS) is an emerging trend that combines Data Science with the benefits of being Cloud Native. This chapter intends to delve into three aspects of CNDS: strategy, technology and best practices.

Strategy

Data Science has become an important part of any business because it provides a competitive advantage. Very early on, Amazon's data on book purchases allowed them to deliver personalized recommendations whilst customers were browsing their site. Their main competitor in the US at the time was Borders, who mainly operated in physical stores. This physicality prevented them from seamlessly providing customers with personalized recommendations [21]. This example highlights how **strategic business decisions and data science are inextricably linked.**
The context and direction of a business also have an impact on CNDS. Smaller businesses, where offerings are small but scopes fluctuate dramatically, can benefit from the flexibility that Cloud Native delivers. Enterprise businesses, with larger projects and longer lead times, will benefit from reduced coupling and decreased feedback cycles. Irrespective of size, **all businesses should tend towards empowering teams by integrating data into their business.**



DataDevOps

Devops is an integration philosophy that popularized the idea that the people who build a system should run the system. Teams are responsible for the ongoing success of a component of the business. This goes against traditional ideas of functional roles, where entire teams would be dedicated to one technical niche. This is a particular problem in enterprises. A "wall" between people developing and operating products negatively affects productivity and makes it a worse place to work.[22]
When data-driven products are added to the mix, the problems compound; another "wall" is created. Teams of mathematicians and scientists would "throw" models and algorithms to Developers to implement. Software would then be given to operators to run in production. The result is blindingly obvious. Operators blame the Developers for providing them with buggy code. Developers blame the scientists saying that they've provided an inefficient model, or the model doesn't work. And the scientists blame both for not understanding their work.
The only solution, which can be approached in several different ways, is that these walls must not exist. **Everyone is responsible for delivering a product. This should be stated explicitly in the way that teams are organized.**

Team Organization

A practical solution depends again on the scale and strategy of a business. Typical solutions include creating cross-functional teams or dedicated "Data Science Consultants" within the business. Cross-functional teams, dedicated to one or more products or customers, benefit from the cross-pollination of experience and learning. Developers learn Data Science. Operators learn Software Engineering. Scientists learn how to operate a product. Whereas dedicated teams of data science consultants within a business can be more efficient.

The most interesting part of organizing a team which includes Data Scientists is that the level of expertise required is flipped upside down. In conventional Software Engineering projects (i.e. software that is easy to plan), it pays dividends to have experienced engineers up front. Good software designs and architectures make it easier to create good products.
But Data Science projects are often harder at the end. It is very easy to come up with a quick model given some data. It is very hard to run a robust model in production. It is often said that Data Science only makes up about 10% of the effort of a data-driven product. The other 90% consists of other typical Cloud Native components: monitoring and alerting, continuous integration and delivery, stateful storage, user interfaces, business logic and various other supporting tools and technologies. For this reason, it pays dividends to have experienced Data Scientists at the end of a project; where they can help grow the long-term viability of a product rather than the short-term proof of concept.

Technology

Technology choices in data-driven products are, as you would expect, largely directed by the type and amount of data. **The first and most crucial decision to make is whether the data will be processed in a batch or streaming fashion.**

Stream or Batch?

The theoretical distinction between streaming and batching is less important than the practical implications. Streaming data implies a constant flow of new data and batch implies at least semi-static data when viewed from smaller timescales.

But the distinction is fuzzy; you can stack streaming data and treat it like a batch process or use streaming methods even when update rates are slow. The real distinction is between the tools and technologies that have been developed to handle these two types of data. It is generally considered that problems are quicker and easier to solve if you can solve them in batch, but you lose the granularity of receiving rapid updates. [23]

You can infer the regime from the problem definition. For example, detection automation problems (e.g. fraud detection) usually handle their data as a stream to allow for rapid response times. But application automation problems (e.g. customer churn, loan applications) can be handled in batch because this matches the underlying task.

Data Storage

The second key decision is the handling and storage of data. It can be hard to move data once it has come to rest, due to the size. So it makes sense to think carefully about storage requirements upfront. This is usually called Data Warehousing and has interesting implications for a Cloud Native architecture.

Many of the benefits of being Cloud Native are focused on the consumer-facing parts of an application. We want it to be resilient because we don't want our consumers to see our broken code. We want it to be scalable so that we can meet demand. The main way in which Cloud Native techniques achieve this is through immutability. Scalability and resiliency is the result of replicating a small amount of code many times (e.g. containers). But data, by definition, is mutable. It is constantly changing. This means we have to repeatedly move new data.

Therefore the real challenge is how to best move the data from one place to another. And the key decision to be made is when, how and in what form this data is moved.

If your data is highly structured and it needs to be accessed often, databases are your best bet. Note that even unstructured data can often be coerced into a database. The primary benefit of doing so is that we offload the complex task of storing, managing and exposing the data efficiently. If you have highly unstructured data, for example data of many differing types, then it is usually better to use an object or file store. Object stores such as S3 have become very popular for storing binary blobs of any size, at the expense of performance. High-performance blob storage systems often revert to clusters of machines that expose the performance of SSDs.

Monitoring

Creating software is easy. You decide what you want it to do, make it do what you want it to do, then assert that it does it. This is all made possible because the code is deterministic. For any given input you can make sure it generates the expected output. But algorithms used to make decisions about data are often developed as black boxes. We create models based on the data that we see at the time; this is called induction.

The problem with inductive reasoning is when we observe data that we haven't seen before, we don't precisely know how the black box is going to behave.

There are techniques that we can use to improve the level of determinism, but a good form of validation is to constantly reassure ourselves that the application is working. Like Cloud Native software, we need to instrument our Data Science code in order to monitor their operation. For Data Science applications we can:

- generate statistics about the types of decisions being made and verify that they are "normal",
- visualize distributions of results or inputs and assert their validity,
- assert that the input data is as expected and isn't changing over time or has invalid values, and
- instrument feature extraction to ensure performance.

This is the feedback that the Data Scientists and Engineers need to iteratively improve their product. **Through monitoring, we can gain trust in models and applications and prove to others that they work. Ironically, implementing monitoring and alerting in your CNDS application is the best way to avoid being woken up at 3 AM.**

Best Practices

Following the Cloud Native best practices of immutability, automation and provenance will serve you well in a CNDS project. But working with data brings its own subtle challenges around these themes.

Provenance

Affirming the provenance of a model is most important when things go wrong. For example, unexpected new data or attacks [24] can cause catastrophic errors in the predictions of live systems. In these situations, we need to be able to fully reconstruct the state of the model at the time the error occurred. This can be achieved by snapshotting:

- the model, along with its parameters and hyperparameters,
- the training data,
- the results at the end of training,
- the code that trained and ran the model, and
- by fixing seeds.

When failures do occur it's really important to make sure you cover the basics first. All failures should be surprising and due to some misunderstanding of the data. For example, there have been reports of Tesla's parking mode ramming customers garage doors. If it can't avoid hitting a door, how can people have confidence whilst driving at 70 mph.
When failures do occur, make sure they don't happen again. Always inspect the largest failures and ensure automated tests are up to scratch. Retrain the model if necessary. If online and offline results are different, then you have a bug. If it makes sense, add the data to a regression test set to make sure it never happens again. And finally, keep raising the baseline; people don't accept a decrease in performance unless it's for a very good reason.

Automation

Automation is a central tenet of being Cloud Native. Models are updated often throughout early development. Automated delivery pipelines to ensure quality are vital. Making it easy to push new developments into production is important to reduce the friction and the feedback cycle. Jupyter Notebooks [26], a common method of communicating research in the Data Science community, are not to be used in production code, testing or within pipelines. Production code needs the rigours of dedicated Software Engineering. Dedicated graph-based pipeline tools like Luigi [25] are helpful here.

People shouldn't be spending time repeatedly writing boilerplate to read from different data sources. Invest effort to create abstractions of your data sources and give engineers easy access to data over APIs.
Furthermore, your feature extractors shouldn't care where data is coming from; this will make it easier to add new data sources and reuse code.

Rapid and clear feedback is vital for gaining confidence in a model and a product. Switching between environments or data should be avoided (e.g. throwing models over a "wall" to software engineers are a sure way to introduce subtle bugs that fail silently). Ideally, the exact same code should be used on and offline to be confident that offline results should match online. And again, monitoring is often the first and last line of defence.

Finally, deployment of models can be particularly tricky, since you can never be 100% happy that a model is working as expected. Once adequate monitoring has been implemented, you can feed this information back into the deployment pipeline. For example, a simple but efficient strategy is to shadow the new model against a model in production. Once you have enough data over enough time to prove your model is working, you can manually switch it over.

You can also run multiple models in this way to compare performance. This is known as the champion-challenger approach. The model with the best production performance is moved to the front. Other standard deployment strategies like A/B testing, blue-green, etc. also apply here. But make sure your monitoring is up to scratch first.

Conclusion

In this chapter we talked about strategy, technology and best practices. Developing a Cloud Native strategy, focused towards data science, can have profound effects on a business or a product. There can be speed bumps, both with the product direction and team organization, but the result is reliable, flexible products that will help you compete.
The important early technology choices are, unsurprisingly, focused on the data. You must consider the type of data you will be gathering and the use case that you are trying to fulfil. Non-functional requirements are also very helpful. An incorrect choice can often mean throwing away significant amounts of work.

During our time developing CNDS projects we have found that the best practices of immutability, provenance and automation have notable differences when compared to process automation projects. Keeping complete copies of the state of models is very useful for post-mortem analysis. And many day to day tasks are highly repetitive; it is worth spending effort to automate these to reduce mistakes and improve efficiency.
Doing Data Science in a Cloud Native world can have its difficulties. The development cycle of a Data Science project can be very different to a Software project; at least in the early stages of development. But **being Cloud Native yields robust, performant products. Being confident that your models are operating as expected will help you sleep at night.**

THE END

THE STATE OF THE CLOUD NATION?

Finally, you've reached the end. Well done. Finding time to read tech books (even short ones) is surprisingly difficult!

Way back in my somewhat facetious blurb I said this book was horizontally and vertically scalable. Would you scale up a book horizontally by making more copies or vertically by sizing up the text? Actually the tradeoffs (for tradeoffs there are) are surprisingly analogous to those for distributed systems vs monoliths.

There are genuine difficulties with just making more copies of a book (aka horizontal scaling). It requires more resources, there may be licensing issues, you can't be sure if anyone you gave a copy to actually read the book, how far through it they are or whether they understood it. That's the reason we don't teach kids to read by handing them a copy of a book and walking away. To share a book with young children, we choose a big font and read together.

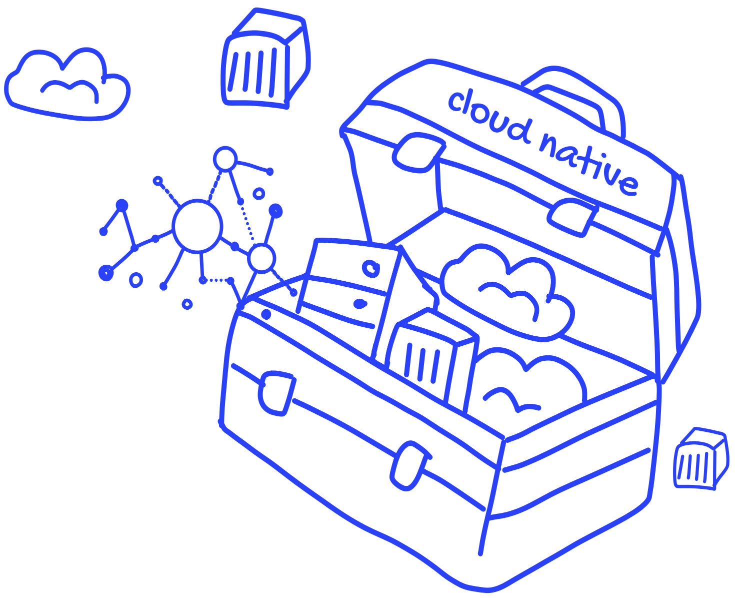
Vertically sizing up text is clearly neither a fast nor a scalable approach to group reading. Usually we distribute copies of the book. What I'm saying, however, is there are always some use cases for monolithic (vertical) scaling approaches and some for distributed (horizontal) ones. Even the absurd example of vertically scaling a book by increasing the text size and reading en masse has a vital use in teaching literacy to kids. **There is no one true way to solve every scaling problem.**

In the introduction, we said our goal for this book was to understand Cloud Native (CN) - what it is, what it's being used for and whether it's actually effective. We did this by talking to companies, thinking about what they told us and considering our own experiences. We tried to show both what we learned and our thought processes so the other half of this partnership (you, dear reader) can form your own judgment.

Our initial definition for Cloud Native came from the Cloud Native Computing Foundation who say that, ideologically, CN systems are container-packaged, dynamically managed, and microservice-oriented (or orientated if you're a Brit like me). We'd add another two characteristics: they need to be hosted on flexible, on-demand infrastructure (aka "cloud"), and plumbed in with a high degree of automation (automated testing and continuous integration, delivery and deployment).

So, should we conclude that Cloud Native is a five-point checklist?

- If you're containerized and orchestrated and in the cloud, but not microserviced (like 40% of Cloud66's customers) then are you not really Cloud Native?
- Or if you're microserviced, CI/CD and cloud hosted, but not containerized (like ASOS) then are you not genuine?
- If you can't deploy 10,000 times a day, like Skyscanner will eventually be able to do, are you a Cloud Native failure?



Will you only win at CN if you check all the boxes? We don't believe so.
We think Cloud Nativeness is a spectrum not a value system.

Infrared is neither superior nor inferior to ultraviolet, there just happen to be use cases for each (inside a microwave or a discotheque you might have a distinct preference). CN is merely a toolbox of architectural approaches that can be very effective at delivering speed (aka feature velocity), scale and reduced hosting costs. You can use some of the tools, or all of them, or none of them, depending on what you need.

For example, containers are less useful to you on Windows where the tech is less mature, but you might still want to use microservices to get better dev team concurrency. Microservices are less useful to you where a quick Ruby-on-Rails MVP will suffice, but you might still want to containerize and orchestrate to speed up your deployments.

You don't have to adopt all of Cloud Native for it to be useful (although we suspect all successful CN does rely on automation of testing, code management, and delivery processes).

A Philosophy

Cloud Native may not be a value system but there does appear to be a philosophy to it.
Everyone we met using CN urgently wanted to move fast and be adaptive to change in their industry, but they didn't want to break everything they already had.

Their old tools and processes depended on moving slowly to manage risk but they wanted to move quickly, so they had to use new ones. They then often used those same tools to cut their hosting bills and to scale but, critically, that was less vital to them than speed and improving their ability to respond and adapt.

We saw that **Cloud Native was more of an attitude than a checklist.** It was a rejection of the slow, visionary, utopian big bang. It was **about embracing an iterative mindset, taking it one small, low-risk step at a time but taking those steps quickly.** Cloud Native solutions were often distributed and scalable but that was not generally the point. The point was delivery speed - getting features out faster.

Adopting a Cloud Native attitude seems to mean evolving into a flexible business that embraces new technology, trusts its employees' judgment and is culturally able to move quickly, be experimental and grasp opportunities.

A Cloud Native attitude doesn't sound bad to me.

REFERENCES

1 - Cloud Native Computing Foundation charter https://www.cncf.io/about/charter/ The Linux Foundation, November 2015
2 - Wikipedia, https://en.wikipedia.org/wiki/Orchestration_(computing) September 2016
3 - The Register 'EVERYTHING at Google runs in a container' https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/ May 2014
4 - Ross Fairbanks Microscaling Systems Use Kubernetes in Production https://medium.com/microscaling-systems/microscaling-microbadger-8cba7083e2a February 2017
5 - Forbes David Williams, The OODA loop https://www.forbes.com/sites/davidkwilliams/2013/02/19/what-a-fighter-pilot-knows-about-business-the-ooda-loop/#30e3c4963eb6 February 2013
6 - The Skeptical Inquirer, Prof Richard Wiseman The Luck Factor http://www.richardwiseman.com/resources/The_Luck_Factor.pdf June 2003
7 - Skyscanner Stuart Davidson http://codevoyagers.com/2016/05/02/continuous-integration-where-we-were-where-we-are-now/ May 2016
8 - ASOS public revenue data from https://www.asosplc.com/
9 - About Cloud66 http://www.cloud66.com
10 - AWS SQS provides reliable message queuing https://ndolgov.blogspot.co.uk/2016/03/aws-sqs-for-reactive-services.html
11 - Peter Norvig Moving data between machines http://norvig.com/21-days.html#answers

12 - Husobee Restful vs RPC https://husobee.github.io/golang/rest/grpc/2016/05/28/golang-rest-v-grpc.html) May 2016
13 - Cloud Native early mentions InformationWeek http://www.informationweek.com/cloud/platform-as-a-service/cloud-native-what-it-means-why-it-matters/d/d-id/1321539 July 2015
14 - Greenpeace report on inefficient use of energy within the IT sector http://www.greenpeace.de/sites/www.greenpeace.de/files/publications/20170110_greenpeace_clicking_clean.pdf January 2017
15 - Sam Newman -The Principles of Microservices http://samnewman.io/talks/principles-of-microservices/ 2015
16 - Wikipedia - Test Automation https://en.wikipedia.org/wiki/Test_automation
17 - Upgard Overview of Configuration Tools https://www.upguard.com/articles/the-7-configuration-management-tools-you-need-to-know July 2017
18 - Bryon Root - The Difference Between CI and CD http://blog.nwcadence.com/continuousintegration-continuousdelivery/ August 2014
19 - Docker - What is a Container? https://www.docker.com/what-container 2017
20 - WeaveWorks Comparing Container Orchestrators https://www.weave.works/blog/comparing-container-orchestration/
November 2016
21 - Provost, Foster, and Tom Fawcett. Data Science for Business: What you need to know about data mining and data-analytic thinking. " O'Reilly Media, Inc.", 2013.
22 - Humble, Jez, and David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader). Pearson Education, 2010.
23 - Friedman, Ellen, and Kostas Tzoumas. Introduction to Apache Flink: Stream Processing for Real Time and Beyond. " O'Reilly Media,
Inc.", 2016.
24 - Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." arXiv preprint arXiv:1412.6572 (2014).
25 - https://github.com/spotify/luigi
26 - http://jupyter.org/

Photos by Artificial Photography, rawpixel.com, Brooke Lark, Daniel Jensen, Markus Spiske.

Container Solutions

The

CLOUD NATIVE ATTITUDE

PART 3

Cloud Native Case Studies

DOWNLOAD E-BOOK

www.container-solutions.com