Container Solutions

The CLOUD NATIVE ATTITUDE

Move Fast Without Breaking Everything

Anne Currie

PART 1

Introduction to the Major Concepts of 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

INTRO

WHAT ON EARTH IS CLOUD NATIVE?

According to the Cloud Native Computing Foundation (CNCF) Cloud Native is about scale and resilience or **"distributed systems capable of scaling to tens of thousands of self healing multi-tenant nodes"** (1).

That sounds great for folk like Uber or Netflix who want to hyperscale an existing product and control their operating costs. But is a Cloud Native approach just about power and scale? Is it of any use to enterprises of more normal dimensions? What about folk that just want to get new products and services to market faster, like the UK's Financial Times newspaper. Five years ago, they were looking for an architectural approach that would let them innovate more rapidly. Did Cloud Native deliver speed for them?

Others, like my own startup Microscaling Systems, wanted to create and test new business ideas without large capital expenditure, starting small with minimal costs. Was Cloud Native a way to reduce bills for us?

Why Does This Book Even Exist?

The Container Solutions team and I wanted to understand what Cloud Native was actually being used for, what it could deliver in reality and what the tradeoffs and downsides were.

We interviewed a range of companies who adopted a Cloud Native approach because we wanted to understand what they learned. Enterprises like the flight booking unicorn Skyscanner, the international ecommerce retailer ASOS and the global newspaper The Financial Times. We've also built and operated systems ourselves for well over 20 years and many of the brand new ideas coming out of Cloud Native seem oddly familiar.

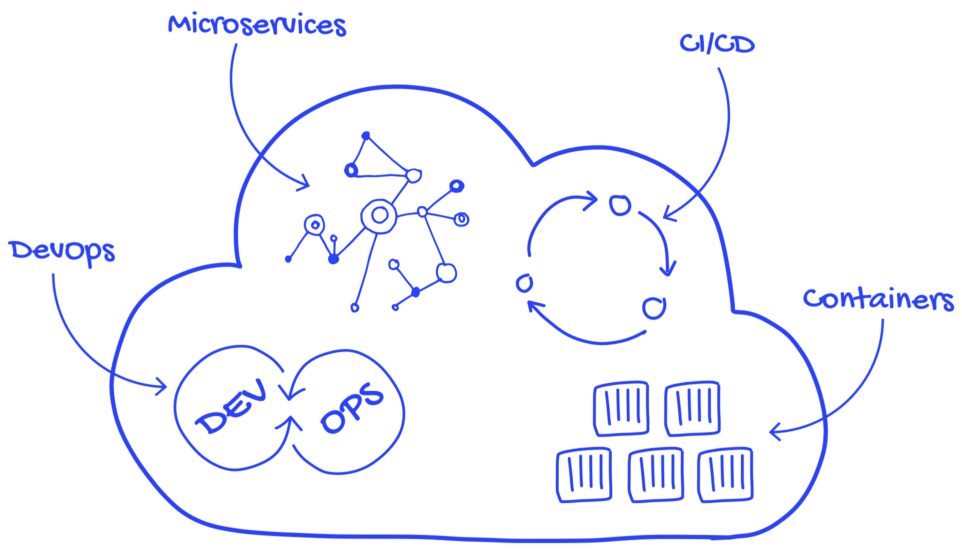
This book is a distillation of what we gleaned from our conversations with users, vendors, hosting providers, journalists and researchers.

It made us ask ourselves,
"What the heck is Cloud Native? Is it a way to move faster? A powerful way to scale? A way to reduce operational costs or capital expenditure?"

How can these different aims be achieved with in one paradigm? Finally, is it good that Cloud Native can potentially do so much or is that a risk?

With everyone from the ordinary developer to the CTO in mind, this book explores Cloud Native's multiple meanings and tries to cut through the waffle to identify the right Cloud Native strategy for specific needs. We argue that moving fast, being scalable and reducing costs are all achievable with a Cloud Native approach but they need careful thought. Cloud Native has huge potential, but it also has dangers.

Finally, we reflect on what Cloud Native really means. Is it a system of rules or more of a frame of mind? Is it the opposite of Waterfall or the opposite of Agile? Or are those both utterly meaningless questions?



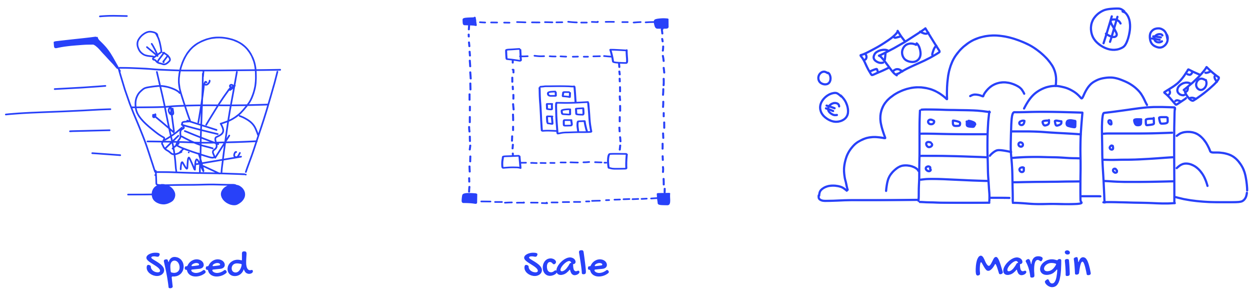
What is Cloud Native? Sounds Like Buzzwords

"Cloud Native" is the name of a particular approach to designing, building and running applications based on cloud (infrastructure-as-a-service or platform-as-a-service) combined with microservice architectures and the new operational tools of continuous integration, containers and orchestrators. The overall objective is to improve speed, scalability and, finally, margin.

Speed:
Companies of all sizes now see strategic advantage in being able to move quickly and get ideas to market fast. By this, we mean moving from months to get an idea into production to days or even hours. Part of achieving this is a cultural shift within a business, transitioning from big bang projects to more incremental improvements. Part of it is about managing risk. At its best, a Cloud Native approach is about de-risking as well as accelerating change, allowing companies to delegate more aggressively and thus become more responsive.

Scale:
As businesses grow, it becomes strategically necessary to support more users, in more locations, with a broader range of devices, while maintaining responsiveness, managing costs and not falling over.

Margin:
In the new world of cloud infrastructure, a strategic goal may be to pay for additional resources only as they're needed - as new customers come online. Spending moves from up-front CAPEX (buying new machines in anticipation of success) to OPEX (paying for additional servers on-demand). But this is not all. Just because machines can be bought just in time does not mean that they're being used efficiently [14]. Another stage in Cloud Native is usually to spend less on hosting.
At its heart, a Cloud Native strategy is about handling technical risk. In the past, our standard approach to avoiding danger was to move slowly and carefully. The Cloud Native approach is about moving quickly by taking small, reversible and low-risk steps. This can be extremely powerful but it isn't free and it isn't easy. It's a huge philosophical and cultural shift as well as a technical challenge.



How Does Cloud Native Work?

The fundamentals of Cloud Native have been described as container packaging, dynamic management and a microservices-oriented architecture, which all sounds like a lot of work. What does it actually mean and is it worth the effort?

We believe Cloud Native is actually all about five architectural principles.

Use infrastructure or platform-as-a-service: run on compute resources that can be flexibly provisioned on demand like those provided by AWS, Google Cloud, Rackspace or Microsoft Azure.

Design systems using, or evolve them towards, a microservices architecture: individual components are small and decoupled.

Automate and encode: replace manual tasks with scripts or code. For example, using automated test suites, configuration tools and CI/CD.

Containerize: package processes together with their dependencies, making them easy to test, move and deploy.

Orchestrate: abstract away individual servers in production using off-the-shelf dynamic management and orchestration tools.

These steps have many benefits, but ultimately they are about the reduction of risk. Over a decade ago in a small enterprise, I lay awake at night wondering what was actually running on the production servers, whether we could reproduce them and how reliant we were on individuals and their ability to cross a busy street. Then, I'd worry about whether we'd bought enough hardware for the current big project. We saw these as our most unrecoverable risks. Finally, I worried about new deployments breaking the existing services, which were tied together like a tin of spaghetti. That didn't leave much time for imaginative ideas about the future (or sleep).

In that world before cloud, infrastructure-as-code (scripted environment creation), automated testing, containerization and microservices, we had no choice but to move slowly, spending lots of time on planning, on testing and on documentation. That was absolutely the right thing to do then to control technical risk. However, the question now is "is moving slowly our only option?" In fact, is it even the safest option any more?

We're not considering the Cloud Native approach because it's fashionable – although it is. We have a pragmatic motivation: the approach appears to work well with continuous delivery, provide faster time to value, scale well and be efficient to operate. Most importantly, it seems to help reduce risk in a new way – by going fast, but small. It's that practical reasoning we'll be evaluating in the rest of this book.

01

THE CLOUD NATIVE QUEST

In our introduction we defined Cloud Native as a set of tools for helping with three potential objectives:

- Speed: faster delivery for products and features (aka feature velocity or “Time To Value”).
- Scale: maintaining performance while serving more users.
- Margin: minimizing infrastructure and people bills.

We also implied that Cloud Native strategies have a focus on infrastructure.

- Start with a cloud (IaaS or PaaS) infrastructure.
- Leverage new architectural concepts that have infrastructural impact (microservices).
- Use open source infrastructure tools (orchestrators and containers).

We believe Cloud Native is a technique that marries application architecture and operational architecture, and that makes it particularly interesting.

In this chapter, we’re going to talk about the goals we’re trying to achieve with CN: going faster, bigger and cheaper.

The Goals of Speed, Scale & Margin

First of all, let's define what we mean by these objectives in this context. Right now, the most common desire we're seeing from businesses is for speed. So that's where we'll start.

Speed

In the Cloud Native world we're defining speed as "Time to Value" or TTV - the elapsed clock time between a valid idea being generated and becoming a product or feature that users can see, use and, hopefully, pay for. But value doesn't only mean revenue. For some start-ups, value may be user numbers or votes. It's whatever the business chooses to care about.

We've used the phrase "clock time" to differentiate between a feature that takes 3 person days to deliver but launches tomorrow and a feature that takes 1 person day but launches in 2 months time. **The goal we're talking about here is how to launch sooner rather than how to minimize engineer hours.**

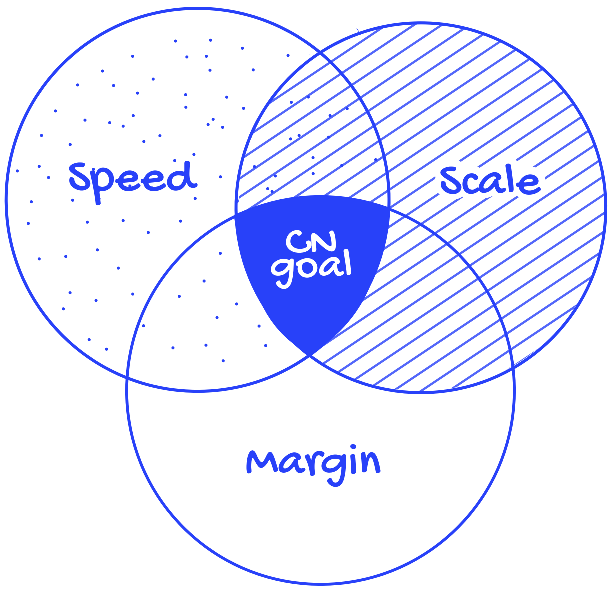
Scale

We all know you can deliver a prototype that supports 100 users far more quickly, easily and cheaply than a fully resilient product supporting 100,000. Launching prototypes that don't scale well is a sensible approach when you don't yet know if a product or feature has appeal. There's no point in over-engineering it. However, the point of launching prototypes is to find a product that will eventually need to support those 100,000 users and many more. When this happens your problem becomes **scale - how to support more customers in more locations whilst providing the same or a better level of service.** Ideally, we don't want to have to expensively and time-consumingly rewrite products from scratch to handle success (although, in some cases that's the right call).

Margin

It's very easy to spend money in the cloud. That's not always a bad thing. Many start-ups and scale-ups rely on the fact that it's fast and straightforward to acquire more compute resources just by getting out a credit card. That wasn't an option a decade ago.

However, the time eventually comes when folk want to stop giving AWS, Microsoft or Google a big chunk of their profits. At that point their problem becomes how to maintain existing speed and service levels whilst significantly cutting operational costs.



What Type of Business Are You?

But before we jump into choosing an objective, let's consider that a goal is no use unless it's addressing a problem you actually have and that different companies in different stages of their development usually have different problems.

Throughout this book we'll be talking about the kinds of business that choose a Cloud Native strategy. Every business is different, but to keep things simple we're going to generalize to three company types that each represent a different set of problems: the start-up, the scale-up and the enterprise.

The start-up

A "start-up" in this context is any company that's experimenting with a business model and trying to find the right combination of product, license, customers and channels. A start-up is a business in an exploratory phase – trying and discarding new features and hopefully growing its user base.

Avoiding risky up-front capital expenditure is the first issue, but that's fairly easily resolved by building in the cloud. Next, speed of iteration becomes their problem, trying various models as rapidly as possible to see what works. Scale and margin are not critical problems yet for a start-up.

A start-up doesn't have to be new. Groups within a larger enterprises may act like start-ups when they're investigating new products and want to learn quickly.

There's an implication here that the business is able to experiment with their business model. That's easy for internet products and much harder for hardware or on-premise products. For the "speed" aspect of Cloud Native we are primarily describing benefits only available to companies selling software they can update at will. If you can't update your end product, continuous integration or delivery doesn't buy you as much, although it can still be of use.

The scale-up

A scale-up is a business that needs to grow fast and have its systems grow alongside it. They have to support more users in more geographic regions on more devices. Suddenly their problem is scale. They want size, resilience and response times. Scale is not just about how many users you can support. You might be able to handle 100X users if you accept falling over a lot but I wouldn't call that proper scaling. Similarly, if you handle the users but your system becomes terribly slow, that isn't successful scaling either. A scale-up wants more users, with the same or better SLA and response times and doesn't want to massively increase the size of their operations and support teams to achieve it.

The Enterprise

Finally, we have the grown-up business – the enterprise. This company may have one or many mature products at scale. They will still be wrestling with speed and scale but margin is also now a concern: how to grow their customer base for existing products while remaining profitable. They no longer want to move quickly or scale by just throwing money at the problem.

They are worried about their overall hosting bills and their cost per user. Being big, resilient and fast is no longer enough. They also want to be cost effective.

Where to Start?

It's a good idea to pursue any wide-ranging objective like speed, scale or margin in small steps with clear wins.

For example, pursue faster feature delivery for one product first. Then, when you are happy with your progress and delivery, apply what you've learned to other products.

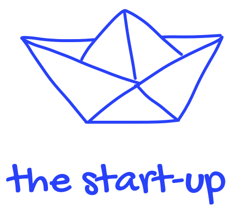
It's a dangerous idea to pursue multiple objectives of Cloud Native simultaneously. It's too hard. Every Cloud Native project is challenging and, as we'll read in our case studies, it requires focus and commitment. Don't fight a war on more than one front.

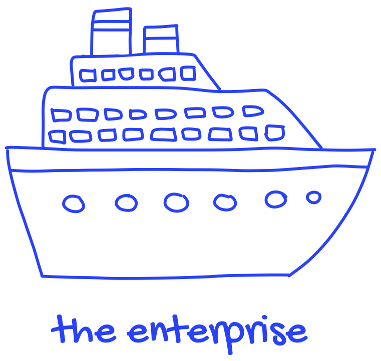
Your objectives don't have to be extreme. Company A might be happy to decrease their deployment time from 3 months to 3 days. For Company B, their objective will only be achieved when the deployment time is 3 hours or even 3 minutes. Neither Company A or Company B is wrong - as long as they've chosen the right target for their own business.

When it comes to "define your goal" the operative word is "your".

So, if you're searching for product fit you are in "start-up" mode and are probably most interested in speed of iteration and feature velocity. If you have a product that needs to support many more users you may be in "scale-up" mode and you're interested in handling more requests from new locations whilst maintaining availability and response times. Finally, if you are now looking to maximize your profitability you are in "enterprise" mode and you're interested in cutting your hosting and operational costs without losing any of the speed and scalability benefits you've already accrued.

OK, that all sounds reasonable! In the next chapter we are going to start looking at the tools we can use to get there.





02

DO CONTAINERS HAVE IT ALL WRAPPED UP?

In the last chapter we described the Cloud Native goals of speed, scale and margin, or going faster, bigger and cheaper. Next we're going to look at some of the tools that Cloud Native uses to tackle these goals, including container packaging, dynamic management and a microservices-oriented architecture.

In this chapter we'll consider container packaging – what it is and the effect it has. But first, let's take a big step back. What are we running on?

IaaS, PaaS or Own Data Centre?

Before we start talking about software and tools, a good question is where is all this stuff running? Does Cloud Native have to be in the cloud?

Crucially, does a Cloud Native strategy have to use infrastructure-as-a-service (IaaS) or platform-as-a-service (PaaS) with the physical machines owned and managed by a supplier like Microsoft, Google or AWS? Or could we build our own servers and infrastructure?

We'd argue that **Cloud Native strategies fundamentally exploit the risk-reduction advantages of IaaS or PaaS:**

- Very fast access to flexible, virtual resources (expand or contract your estate at will). This changes infrastructure planning from high to low risk.
- Lower cost of entry and exit for projects. The transition from CAPEX (buying a lot of machines up front) to OPEX (hiring them short term as needed) de-risks project strategy by minimizing sunk costs and making course corrections or full strategy shifts easier.
- Access to cloud-hosted, managed services like databases-as-a-service, load balancers and firewalls as well as specialist services like data analytics or machine learning makes it faster and easier to develop more sophisticated new products. This can help identify opportunities more quickly and reduce the risk of experimentation.

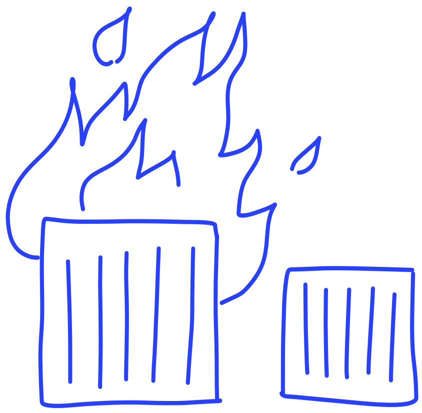
These advantages can potentially be duplicated by a large organization in their own data centers - Google, Facebook and others have done so. However, it is difficult, distracting, time-consuming and costly. Therefore, it's a risky process. For many enterprises it's more efficient to buy these IaaS/PaaS advantages off-the-shelf from a cloud provider. If you have a tech team who are smart enough to build a private cloud as well as Google or AWS then is that the best way for your business to use them?

So, Cloud Native systems don't have to run in the cloud but Cloud Native does have tough prerequisites that are already met by many cloud providers, increasingly commoditized, and difficult to build internally. To be honest, I'd probably use the cloud unless I was Facebook.

Containers! They're so Hot!

In the Cloud Native vision, applications are supplied, deployed and run in something called a "container". A container is just the word we use to describe cleverly wrapping up all the processes and libraries we need to run a particular application into a single package and putting an interface on it to help us move it about. The original and most popular tool for creating these containerized applications was Docker.

Containers are so hot because containerization accomplished three incredibly sensible things.



A Standard Packaging Format - Docker invented a simple and popular packaging format that wrapped an application and all its dependencies into a single blob and was consistent across all operating systems. This common format encouraged other companies and tons of startups to develop new tools for creating, scanning and manipulating containerized applications. Docker's format is now the de-facto standard for containerized application packaging. Docker's containerized application packages or "images" are used on most operating systems with a wide set of build, deployment and operational tools from a variety of vendors. The image format and its implementation are both open source. In addition, Docker's container images are "immutable" - once they are running you cannot change or patch them. That also turns out to be a very handy feature from a security perspective.

Lightweight Application Isolation Without a VM - A "container engine" like Docker's Engine or CoreOS's rkt is required to run a containerized application package (aka an "image") on a machine. However, an engine does more than just unpack and execute packaged processes. When a container engine runs an application image, it limits what the running app can see and do on the machine. A container engine can ensure that applications don't interfere with one another by overwriting vital libraries or by competing for resources. The engine also allows different versions of the same library to be used by different containers on the host.
A running containerized application behaves a bit like an app running in a very simple virtual machine but it is not - the isolation is applied by the container engine process but enforced directly by the host kernel. A container image once running is referred to as just a "container" and it is transient - unlike a VM, a container only exists while it is executing (after all it's just a process with some additional limitations being enforced by the kernel). Also, unlike a heavyweight VM a container can start and stop very quickly - in seconds. We call this potential for quick creation and destruction of containers "fast instantiation" and it is fundamental to dynamic management.

A Standard Application Control Interface
Just as importantly, a container engine also provides a standard interface for controlling running containers. This means third-party tools can start and stop containerized applications or change the resources assigned to them. The concept of a common control interface for any application running on any operating system is surprisingly radical and is, again, vital to dynamic management.

Together, these 3 revolutionary innovations have changed our assumptions about how data centers can be operated and about how rapidly new applications can be deployed.

Alternatives to Containers

Now that these concepts of standardized application packaging, isolation and control are out there, we're already seeing alternative approaches being developed that provide some of the same functionality. For example:

- Serverless or function-as-a-service products like AWS Lambda (cloud services that execute user-defined code snippets on request).
- Unikernels and their ilk (potentially self-sufficient application packages that also include the minimum required host operating system).
- Applications inside new lighter-weight VMs.

In addition, other container types to Docker exist and even more ways to achieve the benefits of containers will undoubtedly be developed. However, **what's important is understanding the advantages of common packaging, control interfaces, and application isolation** even if in 5 years we end up using something other than containers to provide these features.

ASIDE - To avoid confusion, although the interface for managing Docker images is consistent across all operating systems, the contents of the image are not necessarily portable. The contents of a container image are a set of executables. A Linux container image will only include executables compiled to run on Linux. A Windows image will only include exes and dlls compiled to run on Windows. You therefore cannot run a Linux container image on Windows or a Windows container image on Linux any more than you can run an executable compiled for one on the other. However, once the containers are running on the right host OS you can control them all with the same format of API calls. Remember - the container engine is not a runtime environment like Java. Containers run natively on the host so the executables must be compiled for that OS.

Is a Container As Good As a VM?

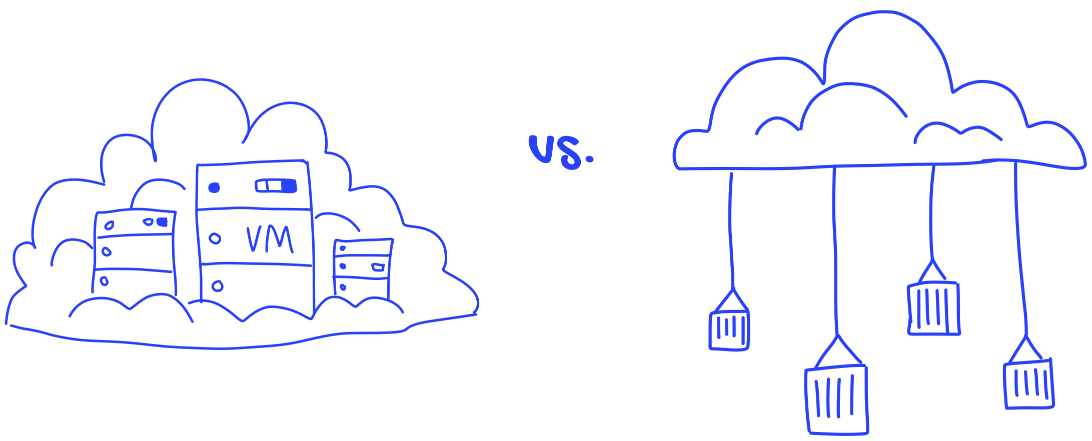
Before we get too carried away, there are still ways a VM is better than a container. On the downside:

- a VM is more massive than a container
- a VM consumes more host machine resources to run than a container
- VMs take much longer to start and stop (minutes vs seconds).

In the VM's favour, however, it is a much more mature technology with years of tooling behind it. Also, containers isolate processes, but they don't do it perfectly yet - especially for antagonistic applications. The VM's heavyweight approach is currently more secure.

Why is Everyone Mad About Containers Anyway?

The reason everyone's going crazy about containers is not just because they are a nice packaging format that plays well with automated deployments. Containers also provide us with lightweight application isolation and a standard application control API. Paired with dynamic management that can give us **automation, resilience and much better resource utilization, making containers potentially greener and cheaper.** But more on that in the next chapter.



03

IS DYNAMIC MANAGEMENT THE PRIME MOVER?

Dynamic infrastructure management is sometimes described as programmable infrastructure and its purpose is to automate data centre tasks currently done by ops folk. This potentially has multiple benefits.

- Improved ops team productivity.
- Systems that can react faster and more consistently to failure or attack and are therefore more resilient.
- Systems that can have more component parts (e.g. be bigger)
- Systems that can manage their resources more efficiently and therefore be cheaper to operate.

Dynamic management relies on a brand new kind of operational tool called a container orchestrator.

What is an Orchestrator?

According to Wikipedia, "Orchestration is the automated arrangement, coordination, and management of computer systems" [2].

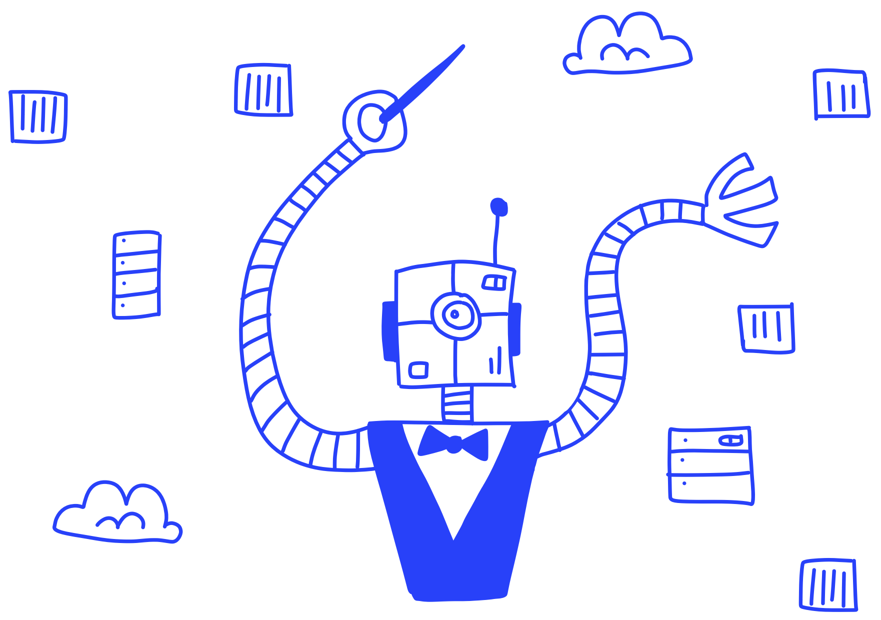
Orchestration tools have been around a long time for controlling virtual machines (VMs) running on physical servers. VM orchestrators underpin the modern cloud - they allow cloud providers to pack many VMs efficiently onto huge servers and manage them there. Without that, operating the cloud would cost too much. However, container orchestrators can do even more than VM orchestrators.

Container Orchestrators

New container orchestrators like Kubernetes, DC/OS, Nomad or Swarm remotely control containers running on any machine within a defined set called a cluster. Amongst other things, these orchestrators dynamically manage the cluster to automatically spot and restart failed applications (aka fault tolerance) and ensure the resources of the cluster are being used efficiently (aka bin packing).

The basic idea of any orchestrator (VM or container) is that we puny humans don't need to control individual machines, we can just set high level directives and let the orchestrator worry about what's happening on any particular server.

We mentioned in the last chapter that containers are lightweight compared to VMs and highly transient (they may only exist for seconds or minutes). We are already dependent on VM orchestrators to operate virtualized data centres because there are so many VMs. Within a containerized data centre there will be orders of magnitude more containers. Google, one of the earliest users of container technology in production, start over two billion containers every week [3]. Most of us are not going to do that (!), but if we don't operate way more containers than we currently do VMs then we're missing out. Container orchestrators will almost certainly be required to manage these greater numbers effectively.



Is Dynamic Management Just Orchestration?

Right now, dynamic management is mostly what we can do out-of-the box with orchestrators (better resource utilization and automated resilience) although even that entry-level functionality is extremely useful.

However, orchestrators also let third parties write tools to control the containers under the orchestrator's management. In future, these tools will do even more useful things like improve security and energy consumption. We know of at least one small company who has cut some hosting bills by 70% using a container orchestrator and their own custom tools in production [4].

Automation

The purpose of dynamic management is to automate data centres. We can do that with container orchestrators because of our 3 revolutionary features of containers:

- a standard application packaging format
- a lightweight application isolation mechanism
- a standard application control interface.

We have never had these features before in a commonly adopted form (Docker-compatible containers in this case) but with them we can quickly, safely and programmatically move applications from place to place and co-locate them. Data centres can be operated:

- at greater scale
- more efficiently (in terms of resources)
- more productively (in terms of manpower)
- more securely

Orchestrators play a key role in delivering the Cloud Native goals of scale and margin, but can also be useful in helping to automate deployment, which can improve feature velocity or speed.

Sounds Marvellous. Is There a Catch?

As we've discussed, dynamic management relies on container features like very fast instantiation speeds - seconds or sub-seconds compared to minutes for VMs. The problem is lots of tools designed for working with applications running in VMs do not yet respond quickly enough to handle dynamically managed containers. Many firewalls and load balancers cannot handle applications that appear and disappear in seconds. The same is true of service discovery, logging and monitoring services. I/O operations can also be a problem for extremely short-lived processes.

These issues are being addressed by new products that are much more container-friendly, but companies may have to move away from some old familiar tools to newer ones to be able to use dynamic management. It also might make sense in a container world to hold state in managed stateful services like Databases-as-a-Service rather than to battle the requirements of fast I/O.

Which Came First, The Container or the Orchestrator?

Companies that start by running containers in production often then move on to using orchestrators because they can save so much hosting money. Many early container adopters like The Financial Times or the cloud hosting provider Cloud66 (who you'll hear more about later) initially wrote their own orchestrators but are now adopting off-the-shelf versions like Kubernetes as those commercial products become more mature.

So is the first step in a Cloud Native strategy always to adopt containers, quickly followed by orchestrators? Actually not necessarily. Many companies start first with microservices, as we'll see in our next chapter.

04

MICROSERVICES - THE KILLER HORDE?

In the last chapters, we talked about two of the architectural and operational weapons of Cloud Native: containers & dynamic management. However, when I go out and speak to experienced Cloud Native users I find that containers and orchestrators aren't always where they started. Many companies begin with microservices and don't adopt containers until later.

In this chapter we are going to look at "microservices-oriented architectures" and think about how they fit in with the other Cloud Native tools.

Microservices Architectures

The microservice concept is a deceptively simple one. **Complex, multi-purpose applications (aka monoliths) are broken down into small, single-purpose and self-contained services that are decoupled and communicate with one another via well-defined messages.**

In theory, the motivation is threefold - microservices are potentially:

- Easier to develop and update.
- More robust and scalable.
- Cheaper to operate and support.

However, these benefits are not trivial to deliver. How to architect microservices is a difficult thing to get your head around. Microservices can achieve several competing objectives and it's very important that you think carefully about what your initial goal is or you could end up with a mess.

Let's Talk About State

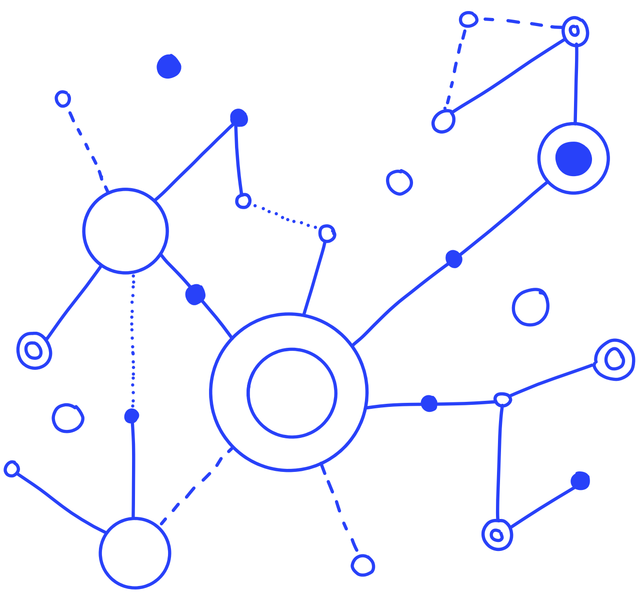
Let's briefly step back and discuss something that often comes up when we're talking about microservices. State.
There are broadly two types of microservice: "stateless" and "stateful".

Stateful microservices possess saved data in a database that they read from and write to directly. Note that well-behaved stateful microservices don't tend to share databases with other microservices because that makes it hard to maintain decoupling and well-defined interfaces. When a stateful service terminates it has to save its state.

Stateless microservices don't save anything. They handle requests and return responses. Everything they need to know is supplied on the request and once the request is complete they forget it. They don't keep any permanent notes to remind them where they got to. When a stateless service terminates it has nothing to save. It may not complete a request but - c'est la vie - that's the caller's problem.

The Point of Microservices

In an earlier chapter we discussed how Cloud Native has three potential goals: speed (i.e. feature velocity or time to value), scale and margin. To optimize for each of these you might design your microservice architecture differently.



Microservices for Speed (Feature Velocity)

A very common motivation for moving to a microservices architecture is to make life easier for your tech teams. If you have a large team all working on the same big codebase then that can cause clashes and merge conflicts and there's a lot of code for everyone to grok. So it would instantly seem easier if every service was smaller and separated by a clear interface. That way each microservice can be owned by a small team who'll all work together happily so long as they like the same two pizza toppings. Teams can then safely deploy changes at will without having to even talk to those four cheeses down the hall - as long as no fool changes the API....

Microservices for Scale

In the very olden days you would spend $5M on a mainframe and it would run for 10 years with no downtime (in fact, IBM see the market for mainframes lasting another 30 years for some users!) Mainframes are the classic example of vertical scaling with all its strengths and weaknesses. I don't want a mainframe for many reasons, but three particularly leap to mind:

- any one machine will eventually run out of capacity
- a single machine can only be in one place - it can't provide fast response times for users all over the world
- I have better things to do with my spare bedroom.

If I want to scale forever or I have geographically dispersed users, I may prefer to architect for horizontal scaling, i.e. lots of distributed small machines rather than one big one.
Basically, for horizontal scaling I want to be able to start more copies of my application to support more users. The self-contained nature of microservices works well with this. An individual instance of a microservice is generally decoupled not only from other microservices but also from other instances of itself, so you can safely start lots and lots of copies. That effectively gives you instant horizontal scaling. How cool is that?
Actually it gets cooler. If you have lots of copies of your application running for scale that can also provide resilience - if one falls over you just start up another. You can even automate this if you put your application in a container and then use an orchestrator to provide fault tolerance. Automating resilience is a good example of where microservices, containers and dynamic management work particularly well together.

Microservices for Margin

Switching to a more modern example, if my monolithic application is running out of memory on my giant cloud instance then I have to buy a bigger instance, even if I'm hardly using any CPU. However, if my memory-intensive function was split out into its own microservice, I could scale that independently and possibly use a more specialized machine type for hosting it. A flexible microservices architecture can give you more hosting options, which generally cuts your costs.

What's The Catch?

If this all sounds too good to be true, it kind of is. **Microservices architectures can be really, really complex to manage. Distributed systems have ways of failing that you've never thought of before.**

The dilemma is if you want your system to be easy for your developers, you can architect your microservices for that. Your architecture will probably involve a lot of asynchronous external queues (stateful services) to minimize unpredictable data loss and it will be expensive to host and relatively slow to run, but it will be robust and easier to develop on and support. Don't knock that!

However, if you want your system to be hyperscale, hyperfast and cheap then you will have to handle more complex distributed failure modes, which we'll talk about in a later chapter. In the short term, it will be more difficult for your developers and they'll have lots to learn.

So you have an initial decision to make. Do you start with feature velocity and ease or with scale and margin? It's absolutely sensible to start easy and add incremental complexity as you gain familiarity and expertise.

In our experience, folk tend to use more than one approach but everyone starts with something relatively straightforward if they want to be successful. In the longer term, some services will need to be hyperfast and some just won't.

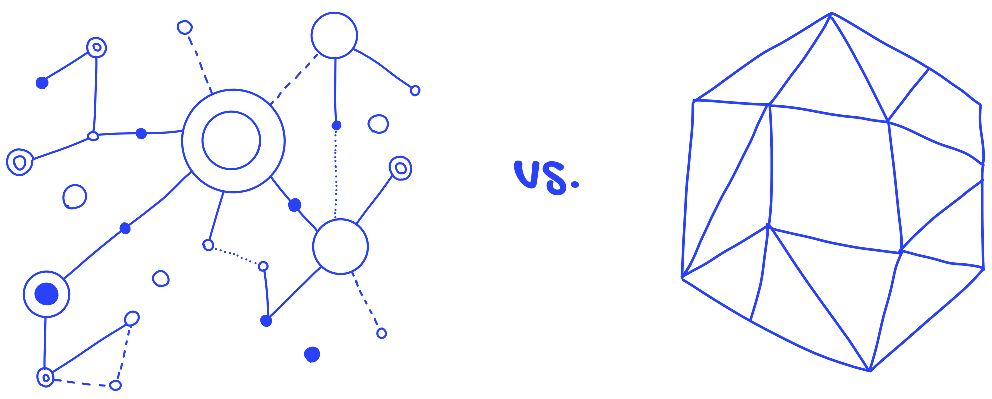
Microservice vs Monolith

Not all application architectures fully benefit from a Cloud Native approach. For example, stopping a container fast, which is important to dynamic management, only works if the application inside the container is happy to be stopped quickly. This may not be true if the app is maintaining lots of information about its internal state that needs to be saved when the process terminates. Saving state is slow.

Lots of older applications maintain state because that was how we used to architect things – as big, multi-purpose "monoliths" that were slow to stop and start. We often still architect that way because it has many benefits. It just happens not to work so well with some aspects of dynamic management.

If you have a monolith there are still advantages to a Cloud Native approach, but Cloud Native works optimally for scalability and resilience with a system of small, independent microservices that are quick to stop and start and that communicate with one another via clear interfaces. The scaling and resilience advantages of containers and orchestrators exist whether you have gone for an easy-but-expensive microservice architecture or a hyperscale-and-hyperfast one or even somewhere in between, which is where most folks are.

So there are clear speed, scale, productivity, resilience and cost advantages to using microservices, containers and dynamic management. And they all work even better together! Great! But what about continuous delivery? Is that required too? We seem to have forgotten about that.



05

THE DREAM OF CONTINUOUS DELIVERY

In the CNCF description of Cloud Native as “container packaging, dynamic management and a microservices-oriented architecture” there is no mention of continuous integration (CI) or continuous delivery (CD). However, they are both present in every successful Cloud Native setup that we’ve seen. Present and vital.

So this chapter is devoted to the philosophy behind CI/CD. Why do we want it, why is it so hard to achieve and how did we tackle it in the past? Finally, we’ll ponder how we tackle it in a Cloud Native environment.

Is Faster Really Better?

There's huge variation between companies in the elapsed time taken for a new product idea to appear in front of users. For some folk it's months. For others it's hours or minutes. Some marketing teams can have a wild thought at 9 a.m. and see it in production that afternoon. Others have given up having wild thoughts.

The route followed by the speedier companies to achieve this velocity has not been easy. It has usually taken several years and they've progressed gradually from cloud to continuous delivery to microservices, containers and orchestration. But, before we look into all that let's step back. What's so good about fast?

The Need for Speed

It's still not unusual for a tech team to have a planned feature roadmap of 18+ months. Hot concepts from dev, marketing or the executives go to a slow heat-death at the end of the roadmap. By the time these experiments are implemented the business has gone completely cold on the whole thing.

Why is it all so slow? Do all changes take months to implement? Often no, some might be a few day's work. Are dev and ops deliberately obstructive? Usually not, an 18 month roadmap is as frustrating to techies as it is to everyone else. Contrary to popular belief, we are humans too.
In fact, there are several things that cause slowness.

Mega-Projects

Big mega-projects like ERP implementations can take up all the time, brain cycles and will to live of a tech team for months or years. They have few natural break points or early ROI milestones and they have a very long time to value.

Unfortunately, teams involved in mega-projects with a high-risk, single delivery milestone in 12 months are unlikely to benefit from a Cloud Native approach to speed. To go fully Cloud Native, we need to be able to deploy small, discrete units of value. Sometimes mega-projects are unavoidable, but they are not what this book is about. I wish you the best of luck.

Manual Tasks and Handover

If even small tasks within your tech organization require manual processes or, even worse, high-friction handovers between multiple parties then considerable cost and elapsed time is added to every project. For example, the developer who writes the code may have to wait days for their comrade on the ops team to provide a test environment.

Multiple handovers can easily delay deployment by weeks. This is an area where a Cloud Native strategy could help by automating or simplifying some of the handover processes to reduce friction.

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

Container Solutions

The

CLOUD NATIVE ATTITUDE

PART 2

Next Steps With Cloud Native

DOWNLOAD E-BOOK

www.container-solutions.com