Microplatforms: Product platforms as code

Stuart Harris[*] and Viktor Charypar[†]
Red Badger Consulting Ltd.
(Dated: 22nd August 2018)

Microplatforms are an automation-based approach to provisioning and operating platforms running digital products and services, which emphasises autonomy of cross-functional product teams. We discuss the traditional approach of building a large-scale, shared platform and explore the resulting cost to the organisation, chief of which is the cost of coordination between teams. Instead, we offer the concept of each cross-functional team owning and operating a separate, self-contained platform. A key enabler for this approach is full automation - capturing all aspects of the product and the platform from provisioning to service orchestration and policy as source code. This approach results in significantly increased team autonomy and enables agile architecture, continuous deployment to production and ongoing innovation. Microplatforms allow organisations to move engineering effort higher up the value chain and focus on using technology to solve customer problems.

INTRODUCTION

When building and delivering a real-world client-server application or service, it is important to consider all the parts that form the digital product, and without which the product would not operate. Typically we think of the product as two major components - the application itself and the "platform" it is hosted on. However, there are many other tools, processes, activities and concerns involved in delivering a Software as a Service (SaaS) product to its users than is covered by the functional features and the runtime environment alone (See figure 1).

Service name, Programming language(s), Programming paradigm(s), Architectural choices, Integration pattern(s), Transport protocols, Authentication, Authorization, Reporting, ETLs, Databases, Caching, Platform libraries, Service dependencies, CI Pipeline dependencies, 3rd party library dependencies, 3rd party service dependencies, Security threat model, License audit, Compliance audit, Capacity plan, Provisioning plan, Cost reporting plan, Monitoring plan, Maintenance process, Backup and Restore process, Secret management, Secret rotation, On-Call schedule, Configuration management, Workflow management, Alerts, Log aggregation, Unhandled failure aggregation, Operations and incident response runbooks, API documentation, Source Code Repository, Humane Service Registry, Service Discovery Registry, Distributed Tracing Registry, Monitoring Dashboard Registry, Build Artifact Repository, CI pipeline(s): Build, Test, Publish, Integration tests, Contract tests, Canary, Deploy, Post deploy tests.

Figure 1: 48 concerns for a single microservice [36]

[*] stuart.harris@red-badger.com
[†] viktor.charypar@red-badger.com

Starting with the software itself, it is crucial that it is thoroughly tested before any part of it can be considered done. Testing is now typically largely automated [23], although manual exploratory testing is also strongly advised [28]. Testing activities fit on a scale of the level of integration of all the components involved, from testing each in isolation (e.g. unit testing) to fully integrated testing of the run-time behaviour of the entire application stack.

Typically, this is supported by a number of different environments [19] starting from the lower ones, where external dependencies of services are stubbed [32] and moving all the way up to the production environment which serves customers. Traditionally, these environments are named, long-lived, and are often reused for the purposes of various different testing activities (e.g. functional, performance, and penetration testing).

In order to get applications deployed, configured and started in these environments a process of deployment activities is followed, ranging from completely manual to fully automated. There are several different methodologies addressing the change-related risk of deploying to the production environment, ranging from large regular deployments done every few months all the way to Continuous Delivery and Continuous Deployment [26].

A digital product which is available to customers any time of day, seven days a week also has to be built with the ability to tolerate failure of infrastructure or individual components without failure of the product itself. This is commonly referred to as high-availability [33] and is generally achieved by running several instances of the same services spread across geographical locations, so that the application can keep working even when an entire data center is lost. This horizontal replication is also used for scaling the application with customer demand [17].

Given the scope and complexity of all the activities we just described, it makes economic sense to design a set of processes and tools used to perform and support them and bundle them into a reusable, commodity solution we will refer to as a product platform. The effort involved should not be under-estimated [35] and is the chief reason why most large organisations choose the approach of hosting their digital product portfolio on a single centrally managed shared platform. Most organisations also opt to run the product platform on cloud infrastructure, which has proven to be more flexible and significantly cheaper to operate than the alternative of using on-premise infrastructure. (We will discuss the reasons for this cost reduction in more detail in sections II and III.)

There has been a clear trend over the past two decades of automating as many of the above activities as possible and treating them as another part of the product itself, managing them within the same teams as the product engineering. This is the core proposition of cloud services and a key concept of the DevOps movement [25]. In this paper we will investigate ways of leveraging automation to design, build and manage everything involved in building and deploying digital product to the level where the entire product platform can be fully owned by a small, cross-functional product delivery team, enabling significantly increased capability and velocity of product delivery.

In section I we outline the goals a good product platform should aim to support. In section II the current approach of using a shared platform is discussed and an alternative (Microplatforms) is suggested in section V, supported by the concept of managing all aspects of it as source code in section IV. Finally, we describe a reference architecture of an example Microplatforms system in Appendix A.

I. BENEFITS PROVIDED BY A PRODUCT PLATFORM

A product platform should ideally support as many goals of the organisation as possible in order to realise return on the investment of designing and building it in the first place. The main benefit of a product platform should be reducing the time spent on manual operational tasks, deployments and testing, ultimately translating to increased reliability and greater velocity of the teams. A platform should support the cross-functional team in practicing DevOps: building the product features and delivering them to customers quickly and with minimum risk of causing defects or outages.

Deployment of code should be automated and the platform should support practices that reduce or eliminate the risks involved (e.g. techniques like rolling updates [29], blue-green deployments [22], canary releases [34], etc.). Regression testing should also be automated and be reliable enough to make it virtually impossible to deploy a broken application. This automation ultimately enables continuous delivery of customer value as an overarching goal to gain the ability to rapidly improve the product based on real customer behaviour [24].

The platform also needs to provide good support for the chosen architecture of the application itself. The currently accepted best practice for medium to large scale systems is Microservices Architecture [30], which provides numerous benefits for modularity and scalability of the overall solution, but comes with its own set of difficulties in managing the orchestration of a large number of microservices that need to work together as a system. The platform should support this approach with service discovery and configuration management features and reduce the overhead that managing microservice orchestration generally involves.

In the long term, the platform should be flexible enough to support safe technical innovation in all directions and enable practices like evolutionary architecture [21] and continuous improvement [27] even for the infrastructure. We will refer to these features using the term *Agile Infrastructure*. Having the ability to evolve the infrastructure design reduces the need for up-front planning and enables ongoing innovation in a way that is similar to Agile software delivery.

Finally, a product platform needs to include features supporting the constraints of running digital products in safety-critical or regulated environments, e.g. medicine or finance. This means supporting information security features (e.g. secure secrets management for credentials), role based access control with good identity and access management [20] and full audit trail and other controls necessary to reduce the risks involved in all of the activities described above.

Finally, the platform should have built-in features to support information security - prevent intrusions and minimise their impact if they cannot be prevented - and compliance with internal and external regulation that can be easily demonstrated. At a bare minimum, the platform should have access control features and full auditability support. Ideally, it should also be agnostic of the underlying infrastructure provider in order to avoid vendor lock-in and allow the platform to be portable if needed.

It is important to keep in mind that the overall goal of using a product platform is to reduce the work not directly related to delivering customer value to a minimum. It is easy for a product platform to turn from an enabler into a source of impediments or even blockers and great care must be taken to address this. As examples, consider platform downtime, working around (missing) platform features, repeatedly performing manual tasks or coordinating with other teams to perform tasks (we will discuss the coordination costs in the next section in more detail).

II. THE COST OF A SHARED PRODUCT PLATFORM

Given the goals and required features listed in the previous section, it should be no surprise that designing and building a product platform for a medium to large organisation with many products and product teams involves a large investment of resources and time. The obvious, and thus far the only way of offsetting this cost is to build a single large product platform shared by all the product teams [refs - google, facebook...]. The cost of building a product platform generally breaks down into two parts - the one-time upfront cost and the ongoing cost.

The initial investment involves the design, building, stabilising and scaling of the platform, typically done manually in the past, i.e. provisioning servers on premise or in the cloud, configuring them, linking them into highly-available clusters, setting up networking, etc. This requires specialist skills typically leading to the formation of a platform engineering team []. The cost of this initial phase always ends up significantly more than expected, understood or attributed.

Sharing the platform implicitly requires multi-tenancy features, e.g. the concept of organisations and projects, granular role-based access control, resource limits and isolation, etc. Moreover, in order for the platform to be shared by all products, the feature set needs to be a union of all the features required by each product. In the theoretical extreme cases, this means supporting n disjoint sets of features for n products, or forcing all the products to use the single set of features prescribed by the platform.

The former extreme is very costly to scale and requires a lot of engineering effort to support, while the latter limits the product team's ability to deliver exactly what is needed by the customer, as the team has to work around decisions made by someone else (and for someone else). This is an example of the 80/20 rule where a shared platform will get you 80% of the way very easily and the rest becomes a significant challenge, which just ends up frustrating the efforts of the team and the business.

Overall the feature set of a shared platform ends up significantly larger than the set of features required by a single team, increasing the cost and complexity of the platform. The pressure to reduce this cost to a manageable level often leads to limitations for individual product teams requiring niche features that were dropped out of scope.

After the initial build is completed, more investment needs to be made on an ongoing basis to run operations, i.e. provision extra computing resources, patch operating systems with fixes, clean up older instances of services, etc. In order to serve the business for several years, new features need to be added as they become available in the industry in order to stay within the recognised best practice (often referred to as evergreening).

By far the largest ongoing cost to the business, which is often not explicitly recognised, is the cost of coordination between the product teams and the platform engineering team. Unless every common task from provisioning and getting access to regular deployments and elastic scaling is self-service, the product teams need to ask for help from the platform team. The platform team quickly forms a large backlog of day to day tasks to perform, product teams end up queueing behind others, and the platform team have much less time to invest into improvements of the platform itself and automation of the daily tasks. As a result, the platform quickly falls behind compared to capabilities generally available in the industry.

In large organisations, the team structure is typically much more complex, forcing the platform team to request help from other teams, like, for example, IT infrastructure, IT security, networking, etc. amplifying the slowdown even further. This results in a situation sketched in figure 2, where each product team has several dependencies on solution delivery teams, each of which have multiple different customers and often depend on each other as well.

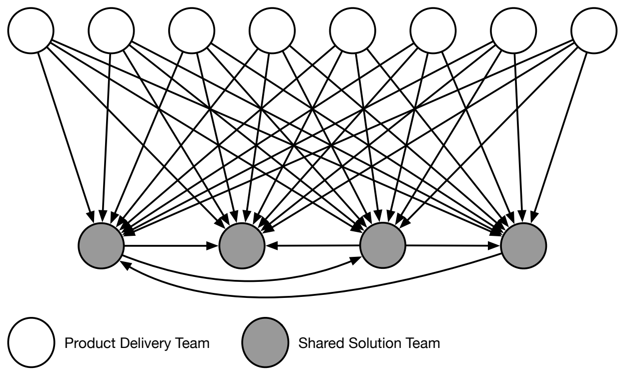


Figure 2: Team dependency structure with shared solution teams

The impact of this effect, especially in the early stages of building a shared platform is so significant, it can cause the entire project to fail completely (since it is no longer helping the product teams deliver value to customers, it is now taking away from that ability), unless the platform team has significant resources to withstand this pressure.

The overall cost of building and operating a shared product platform is large enough to pose a serious risk to the business. Typically, when things do not go well, this leads to resistance to cut the losses - the sunk cost fallacy [31]. Instead of trying again, or exploring different solutions, the attitude turns into "we will make it work" and results in products being built on a mediocre platform with reliability and scaling issues, increasing risk of preventable outages and slowing down product delivery. This is the final, often unrecognised, cost of a shared product platform - the missed opportunity cost of not delivering as much customer value as possible.

Even when things go well, the size of the investment means that when, eventually, the time comes to abandon the current platform in order to innovate using the latest available technology, nobody is willing to make the investment again, even though it could bring an order of magnitude improvement in efficiency of software delivery. This could be described as technology lock-in or a solution lock-in (similar to vendor lock-in []) - with a large-scale platform build, the required investment is so large that a single organisation is not likely to get a return on investment before the need to iterate on technology choices appears because the pace of technical innovation in the industry is ever accelerating.

It may seem like the only option left is to not spend the effort of building a shared platform in the first place and instead reach for a solution available in one of the cloud vendor offerings. This reduces the cost of building and scaling the platform, but increases the cost of sharing and the risk of lock-in stays. Furthermore, the organisation is no longer fully in control of the available tools or the way they are being used by the teams and as a result, most organisations opt for some form of governance around the cloud offering as well, which effectively leads to the same situation as the custom build in terms of coordination cost, and the limited, centrally managed set of available features. Moving to a cloud native solution seems to treat some of the symptoms of the problem, but it does not cure the underlying cause, nor does it guarantee the end result will be any more fit for purpose.

The common theme and, in our opinion, the underlying source of the shared platform costs is the fact they are *shared* among a relatively large number of different product teams. (Notice that the problems listed above seem much less significant if you imagine the platform is only supporting three product teams). The shared nature is what leads to the need for the extra features that enable the sharing, it is also the source of scaling and reliability challenges and, most significantly leads to a sharp rise of coordination costs.

Given the above, our goal should be to enable as much product team autonomy as possible, removing all of the incidental dependencies that are not part the business or customer problem being solved, but rather are created by the solution that has been chosen (often without involving the product delivery team). With a better informed choice of solution, instead of solving the incidental problems and making incidental dependencies more efficient, we can simply make the problems not occur in the first place and remove the dependencies entirely.

The ideal situation would therefore be for each product team to be fully in control of their entire product - the application, the platform, the entire life-cycle and everything described in section I. The only shared dependencies left are the essential ones that have to be shared, i.e. sharing them brings clear benefits (e.g. central systems of record, single sign-on identity providers, source control repositories, artefact registry, etc.). In order for this approach to be practical in a small agile team and safe in the context of a large organisation, we need to leverage as much automation as possible.

III. THE RISE OF AUTOMATION

In the previous section, we covered the level of effort and the number of people necessary to build a product platform. This, however, is largely a result of building and configuring infrastructure manually. Provisioning and setting up servers, clusters, networks and systems by hand requires a large amount of effort and the complexity and difference of individual tasks involved requires specialisation, requiring yet more people.

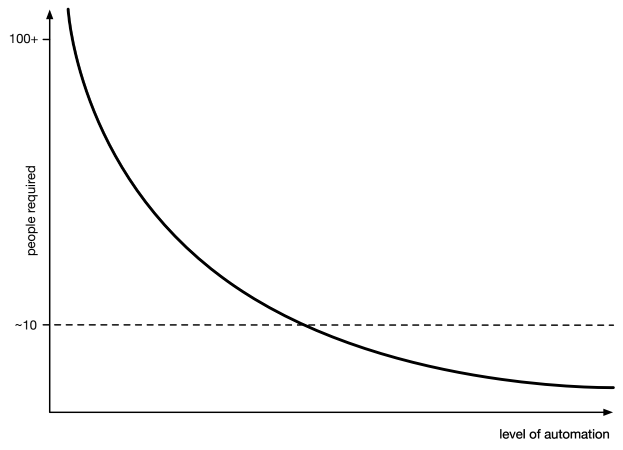


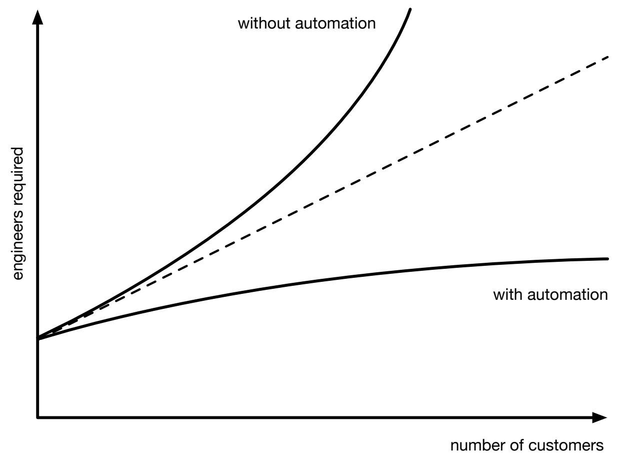
Figure 3: Effect of automation

Fortunately, as software engineers, we know the way out of this problem: encoding the repetitive tasks (even the ones that require a specialist) in automation. Simplifying a little bit, as more and more tasks are automated, fewer and fewer people and less specialisation is required (figure 3).

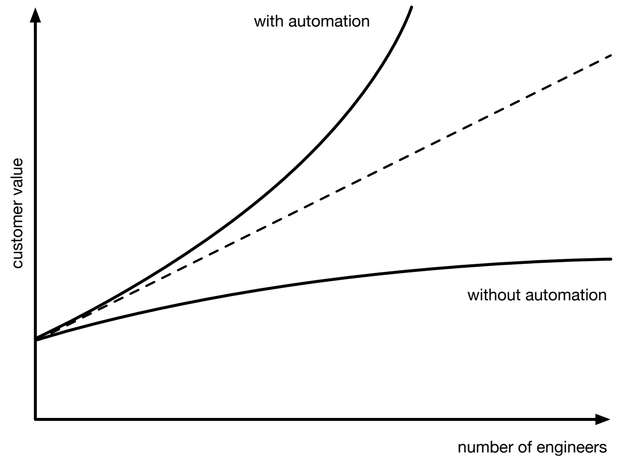
This is not a new finding, automating certain tasks is an accepted practice in multiple industries including software engineering. An obvious example is regression testing using automated unit tests, integration tests and end-to-end tests. Testing used to be a manual task performed by humans following test scripts. Automating it significantly decreased the time between deployments allowing us to get from deploying every few months to several times a day, enabling a completely new working practice known as Continuous Deployment to Production (CDP). We will focus on the consequences of CDP at the end of section V.

Similarly, the success of cloud services is in a huge part down to enabling automation via infrastructure provisioning APIs. Instead of spending weeks manually provisioning servers and network infrastructure, the same infrastructure can be provisioned automatically within minutes. Leveraging cloud services, a small team of engineers can set up a web or mobile application infrastructure that will vastly outperform one set up manually (over weeks or months) by a traditional IT organisation in every measure. [ref?]

With increasing level of automation, the number of engineers necessary to support a product grows significantly slower with growth in the number of customers using that product, compared to more manual solutions [18] (figure 4a). As a corollary, with automation, the customer value delivered grows faster with the number of engineers (as more engineers add even more automation, boosting



(a) engineers per customer



(b) customer value per engineer

Figure 4: Scaling customer service

productivity even more), while without it, adding more engineers relatively quickly brings diminishing returns (figure 4b). Automation enables relatively small teams of engineers to run products relying on a complex infrastructure whilst supporting millions of customers [ref WhatsApp]This is one of the reasons start-up companies outperform large corporations: their resource constraints mean they are forced to invest in automation and smart solutions over adding more people, which in turn results in a much more efficient system. When we employ automation, we move the engineering effort higher up the value chain towards customer facing features.

Recent advances in technology, and commoditization of tools and infrastructure, have now closed the gap in automation to the point where the entire product platform from infrastructure provisioning to deploying complex orchestrations of containerised microservices onto clusters of servers can be easily fully automated and managed, evolved, tested and reused in the same way (and by the same team) as application source code.

However, automation does not only bring speed and efficiency. Every time an engineer makes a manual change to the configuration of a platform, the knowledge encapsulated in that change is lost. The reason for the change, the time, and the previous state of the infrastructure are all lost. By storing the change as code in a version control system, we have a permanent, irrefutable record of when, who, why etc. This not only provides an audit trail of everything the platform has been since it was conceived, but, more importantly, also accelerates the evolution of the platform because each change can build on top of the captured knowledge of previous changes.

IV. PRODUCT PLATFORMS AS CODE

Starting from the industry accepted practices and moving to more novel, recently introduced concepts, we can cover the types of automation and configuration that will together form a fully-featured product platform entirely defined as source code.

First, **pipelines as code** is a common approach in the industry. The Continuous Integration and Continuous Deployment automation is managed as code and stored next to the application code it is building, testing and deploying. This automation has obvious efficiency benefits and storing the configuration as part of the codebase means application and builds do not get out of sync and no extra authorisation process is needed to change the automation configuration. The CI/CD service becomes a simple runtime environment for build and testing tasks.

Second, **immutable infrastructure as code** enables automated, repeatable, reliable and fast infrastructure provisioning. All instances of infrastructure can be identical to the production environment. Since provisioning is automated, the infrastructure can be torn down and recreated within minutes. This lets us treat infrastructure as ephemeral, and allows us to update configuration by destroying and recreating, rather than modifying infrastructure in place, making it effectively immutable. Because no in-place modification is allowed, no human access to servers is required, which simplifies configuration and increases security significantly.

With Docker, containerised **application builds** can also be defined **as code**. Running applications in containers has several benefits. First, containers are lightweight and provide resource isolation guarantees while running on the same underlying operating system. More importantly, with Docker, containers are stored and transferred as images of a standard shape.

A container is a unit of deployment agnostic of the underlying infrastructure and the technology stack used by the application inside it. This draws a very strong boundary between the infrastructure and the application. The images are described with Dockerfiles, recipes for creating images step by step, stored as code []. Docker images are formed of layers, which are immutable and content addressed (by a cryptographic hash of their content) []. These properties mean that Docker images can be built once and deployed on any infrastructure supporting Docker.

In order to support high availability and reliability guarantees, it is necessary to run applications across multiple computers, generally called a cluster. Operating a cluster is a complex task. Among other tasks, we need to manage nodes joining and leaving, make scheduling decisions, i.e. which instance of a particular service to put on which of the available nodes, supporting affinity and anti-affinity constraints while maximising the efficiency of resource usage, and manage automated scaling and failover. Cluster orchestration should also provide built-in service discovery, so that instances do not need to be aware of the layout of the cluster.

In recent years, cluster orchestration has been distilled into easy to use tools, which provide all of the features above out of the box, e.g. Mesos, Docker in Swarm Mode, Kubernetes, etc. These are generally based on a concept of control loops reconciling the current state of the cluster with the desired state, declared in the cluster configuration [ref]. This means the cluster can automatically respond to changes in the underlying infrastructure (e.g. a machine failing, load increasing) as well as changes in the desired state, e.g. a new version of a service being available. Constraints can be put on the reconciliation strategies (e.g. only one instance of a service can change at a time) resulting in gradual, zero-downtime upgrades.

An important consequence of the way modern cluster management software is controlled is the ability to provide **service orchestration as code**, i.e. the state of the cluster is configured down to the desired microservices that should run on it and their names, number of replicas and deployment strategies. Coupled with service discovery, this makes it significantly easier to operate applications built with a microservices architecture.

Microservices provide numerous architectural benefits - they can be built, tested and deployed in isolation, independently evolved, scaled and replaced and generally provide great design flexibility. The downside is the complexity of building, testing and operating complete systems composed of many microservices. Testing and operating a monolithic application is very easy. It is built as a whole, deployed as a whole and tested as a whole and there is no risk of a part of it suddenly changing, breaking the whole. This property does not hold for microservices - by definition they come and go, their API contracts change and even knowing what exactly is running in a given runtime environment is often a difficult task.

With service orchestration captured as code, some benefits of the monolith come back. In essence we reify the service orchestration and gain a single entry point (a "handle") into what the application consists of when running. There can be multiple sets of configuration for different environments, potentially adding or omitting certain services, configuring certain services differently (via environment variables), etc. All this information would normally be either lost or scattered across configuration files and deployment scripts.

With most cluster orchestration tools (e.g. Docker in Swarm mode, Kubernetes), sets of configuration - we will refer to them as "stacks" (a name used by Docker) - can be deployed under an alias, which also forms a namespace for the services. Within the namespace, services can refer to each other by local names, making the entire stack portable, without any change of configuration. This gives a really light-weight mechanism for creating logical environments in a cluster.

As an added benefit, the service orchestration is immutable. In order to change the configuration or topology of the microservices, the code describing it is changed and the orchestration redeployed as a logical unit (as if the application was monolithic). The cluster orchestration system then reconciles the differences so that only the parts that are different actually change. Repeated deployments and updates are idempotent - if there is no difference, nothing changes. This declarative approach means we can logically treat the whole service orchestration as a single unit for deployment and let the automation handle the details. We show the specifics of making sure this property holds in Appendix A.

Service orchestration as code also enables a practice that has become known as GitOps [ref], which advocates performing operations in a runtime environment through changes to a version control repository. In practice, it can mean a cluster has an associated git repository holding all the service orchestration configuration for every single application running in it. In order to deploy an application, a CI service pushes a new revision into this repository and the cluster (more precisely a service running within it) responds to the event by pulling the configuration and attempting to deploy it. This serves as an inversion of control mechanism, removing the need for a CI service (typically in a non-production environment) to have direct access to and control over clusters in a production environment, and also allows for final checks to be performed by the cluster itself to decide whether to accept the deployment. As a side-effect, the state of the cluster is fully audited over time and can be restored in a different cluster even months later. Optionally, this also means no outside access to deployment APIs is needed for the production clusters, increasing security.

In figure 1 we show 48 cross-cutting concerns, identified by Sean Treadway [36], that need addressing for each microservice. None of them are directly related to adding customer value. With microservices architecture, we have made a rod for our own back: developers should be able to write code that directly adds customer value and they should not need to worry about any of the concerns listed. These can be addressed by employing a service mesh (such as Istio []). It allows us to declaratively specify **policy as code** providing services such as intelligent routing and load balancing, traffic shaping and shifting, security policies, ingress and egress rules, circuit breaking [] and upstream call retries, rate limits, call tracing, telemetry and reporting. It is even possible to use a service mesh to inject faults into the system to test reliability features that may have been overlooked (similar concept to Netflix's Chaos Monkey []). All of this configuration is declarative and stored as code alongside the orchestration configuration.

With microservice orchestration and policy being captured as code, the question arises as to whether or not to share microservice instances between application stacks. In other words, can each application have its own replica of the same microservice or should there still be a shared instance? Notice this choice is only valid for stateless microservices, as when replicating microservices which hold state, data replication and distributed transaction concerns arise, which are typically hard enough to solve to warrant being solved once and only once. This difficulty should not be underestimated and it is often a good approach to try to split the microservice into its business logic and persistent data storage concerns and separate out as much of the stateless part of as possible, e.g. by using a Database as a Service (DbaaS). The business logic then becomes a separate, now stateless, microservice.

For stateless microservices, it is simpler if each of the applications has its own copy of a microservice treating it simply as a module of the system, re-using and sharing only its source code, not the running instance(s). In order to share a microservice instance, it needs to be fully treated as a service, as it is no longer under the control of the team managing the application - it essentially becomes a 3rd party dependency. In that case, it should have an agreed service level, support model, documentation, versioning and migration strategy, etc.

In a sense, there are two levels of reuse: microservices, reusing the code, where operations fall under the same team as the rest of the application, and services, reusing the instance as well, where the ownership is more complex and coordination is necessary.

V. MICROPLATFORMS

Building on everything from the previous sections, we can now give some clarity to the term Microplatforms, that we use to refer to the model of runtime environment management we describe in this paper.

The key feature of Microplatforms is not technical, it is organisational - the concept of product team autonomy. This is important, because all of Engineering can (and should) be answerable to Product (ultimately customers) rather than Technology. Technology is a means to an end, an enabler, a multiplier. Autonomous cross-functional teams can leverage technology to meet the product demands of the customer. Infrastructure automation is what enables them to do so.

Consider the diagram from figure 3 again, this time with infrastructure approaches and technologies superimposed on the spectrum (figure 5). With the currently available level of automation, it is now entirely possible for a cross-functional team of ten to operate a fully-featured product platform for their application. This level of vertical integration has numerous benefits.

It removes the coordination cost of all product teams depending on a single platform team. If a team needs to change some aspect of their platform or add a missing feature, they can do so for themselves. This is a fundamental shift in the scope of developers' responsibility for their application - they are not only responsible for the product features but also for reliably delivering the service to their customers (i.e. true DevOps).

A platform team will most likely still exist, but the product it delivers is the design and automation of a platform, not the infrastructure itself. This team can become an owner and steward of the platform design and automation code, which can and should be internally open source, so that everyone can contribute and the platform engineers can transition into expert support and innovation roles, rather than day to day operations. This is a tried and tested approach which works well for shared, reusable code libraries and services []. With the shared dependency removed, the organisation becomes more horizontally scalable, just like software systems do. Vertical integration leads to horizontal scalability.

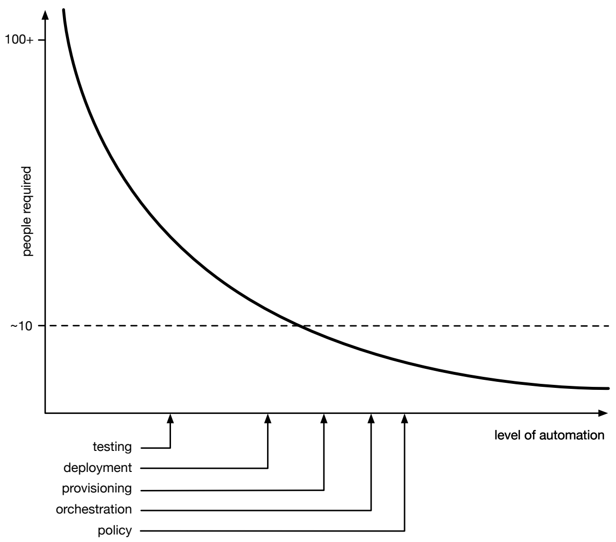


Figure 5: Automation techniques

Small separate platforms also provide a much smaller blast radius in case things go wrong. On a shared platform, a lot of effort needs to go into preventing cascading failures caused by one application overloading a shared resource and in turn causing a different bottleneck to cause more failure. Isolation features like this are a case of multi-tenancy support, which needs to be built into a shared product platform. This can be completely omitted from a Microplatforms based design.

Shifting all the design, management and reuse towards code enables Agile Architecture and a degree of diversification to fit individual product's requirements without ending up with either a superset of all the features or the least common denominator that works for nobody. This also opens the platform up to a free competition of ideas leading to continuous innovation without big, dedicated innovation budget.

The fully automated provisioning, operations and deployments enable provisioning separate, ephemeral platforms for various purposes (e.g. performance testing or penetration testing). In an extreme case, all environments could be ephemeral and only exist for hours or days, with the exception of the live production one, but even the production environment can be regularly rebuilt in a blue-green fashion []. As a byproduct, this facilitates natural slipstreaming of software upgrades and security patches (by using the latest stable version of the software) that would otherwise have to be specially considered.

Targeting a cluster orchestration technology, applications become fully portable between any infrastructure supporting the technology from a developer's laptop through any public cloud provider to on-premise solutions. In the past year, Kubernetes [] has rapidly become a de-facto standard abstraction in cluster orchestration: it is possible to run a local cluster using a tool like Minikube [13], clusters can be provisioned using simple compute resources in different clouds with Kops [], all three major cloud providers are offering hosted services (GKE [], EKS [] and AKS []) and managed on-premise solutions also exist, e.g. OpenShift by RedHat []or Pivotal Kubernetes Service [].

Finally, the level of automation required for a small platform to be managed by an average sized product team enables a capability crucial to agile software delivery: continuous deployment to production.

With continuous deployment to production, every change to the application *and* infrastructure goes through thorough automated testing and validation. Once the validation passes without issues, the change is immediately deployed to production. This can happen several times a day, even multiple times an hour.

This sounds really risky at first, but consider that as the time between deployments tends towards zero, the size of the change also tends towards zero and the risk that change carries also tends towards zero. If things go wrong after a particular deployment, it is trivial to understand the cause of the problem, because the set of causes to consider is minimal. The age of the change deployed tends towards zero as well, and since it was made very recently, it is fresh in everyone's heads which makes it much easier for them to understand the possible causes.

Continuous deployment to production also enables the practice of "fixing forward" rather than rolling back to a likely safe state. Both techniques are still relevant, but the ability to quickly and reliably fix forward becomes important when a problem is discovered after some time has passed. We consider a special process for deploying a hotfix an anti-pattern to be avoided: hotfixes should be deployed through the same process as any other change. Because hotfixes are relatively rare, the special hotfix process is not executed very often and is more likely to fail, which is the worst problem to face when trying to fix an urgent production issue.

With continuous deployment to production, it is necessary to decouple deployments from customer releases, via a mechanism like feature flags, which bring a benefit of decoupling the engineering process of deployment from the business process of customer release. The control of feature release can even be directly given to the product owner.

Altogether, the reduction in coordination cost, the automation of provisioning, continuous integration and deployment, increase in reliability and availability and mean time to recovery from incidents adds up to a significant amount of time which can be invested into innovation and, most crucially, delivering value to customers.

VI. SIMPLICITY AS THE GUIDING PRINCIPLE

We have shown how several main principles contribute to overall efficiency of the Microplatforms concept. We will now attempt to tie these together with a single guiding principle of design we believe leads to the most robust, efficient and future proof solutions.

The first overarching principle is automation, in other words committing decisions and processes to code. This means designs and processes can evolve over time, are repeatable, easy to adopt, specifications are precise and complete and with version control systems, history does not get lost. This enables us to keep all instances of infrastructure and automation perfectly aligned, simplifying the overall result. Automation also allows as many things as possible to become immutable.

Immutability, i.e. preventing things from being updated in place, implicitly creates a version history of all changes and makes it impossible for anything to change unexpectedly or get into a certain state in any other way than being recreated. Removing previous versions of things is a case of garbage collection to reclaim resources they consume. Immutability decouples values, i.e. the entities (infrastructure, build artifacts, deployments, etc.) from the identities, the way we refer to them over time, e.g. "the customer service". [Rich Hickey]

Immutability is typically quite easily attained by content addressing - referring to any data by a cryptographic hash of the content itself. This concept appears a few times in our approach: code stored in git is referred to by the `sha1` hash of the contents of a given revision [], Docker images and layers can be immutably referred to by a `sha256` hash of their content [].

With immutability and with content addressing especially, it is much easier and efficient to compare two items of the same type (i.e. just by their names), enabling idempotence: the concept of operations which can be repeated many times producing no further action beyond the first application. Enabling easy idempotence leads to huge simplification, as we can typically stop focusing on detail and let idempotence handle the unwanted repetition. In a sense, idempotence unifies creation and update and makes both operations the same from outside, moving solutions from imperative to declarative. We can say that idempotence decouples intent from current state. This also appears multiple times: infrastructure provisioning with Terraform, Docker image builds and pushes to registry, deployment to a cluster or response to failure with the reconciliation loop. Everything tries to reach a declared state.

The third major principle is autonomy. In organisations, autonomy removes coordination costs, confusion, bottlenecks and queueing which leads to politics. From a systems perspective, lack of autonomy is a form of coupling. In software engineering, unnecessary coupling is understood to be a problem []. An interesting case of coupling is sharing and reuse, typically seen as purely beneficial without considering the cost.

All of the above - automation, immutability, idempotence and autonomy are cases of simplicity, *simple* being distinctly different from *easy* [simple made easy]. By simple, we mean having a minimum of structure, little coupling and entanglement, avoiding N:M relationships between entities. Immutability is also a case of decoupling - mutable state is one of the hardest couplings to understand - a coupling through time.

The chief guiding principle in every decision made when designing a product platform should be to try to minimise complexity above everything else. This, in our opinion, extends to all software engineering, as the human capacity for understanding complexity is the biggest limiting factor of the systems we design and build.

SUMMARY

The overall goal of a product platform should be to minimise engineering effort spent on anything but delivering value to customers. This covers a vast array of capabilities and as a consequence, designing and building a product platform requires a large up-front and ongoing investment. The traditional way of offseting this cost and ensuring consistency is to build a single, shared, centrally managed product platform to host the entire product portfolio and serve all product delivery teams. We discussed how this approach leads to a growth in scope of the platform features and large coordination costs. The overall effort required can very easily grow to a point where the platform project fails entirely to deliver, or does not bring any return on investment in time, before it is necessary to iterate on the underlying technology choices, requiring yet more investment.

As an alternative, we suggest a different approach focused on enabling autonomy of product delivery teams. It is based on high levels of automation, managing the platform design in its entirety as an internal open source software project, which can be contributed to by everyone, adapted for individual requirements and can evolve with the needs of the organisation and its customers.

The central concept of the approach is to manage the application, build, testing and deployment pipelines, service orchestration, runtime policies and infrastructure provisionig as code. The full automation leads to repeatability, consistency of environments and enables safe Continuous Deployment to Production. Adopting solutions based on principles of simplicity, immutability and idempotence reduces the complexity of understanding and effort required to the point of allowing full ownership of their platform by every product delivery team. With the shared platform team dependency removed, the engineering organisation becomes more horizontally scalable and agile and a larger portion of the engineering effort can be focused on delivering value to customers.

Appendix A: Microplatform Reference Architecture using Kubernetes

A microplatform reference architecture should uphold all of the principles discussed above: full automation, immutability, idempotency, autonomy, and most of all, simplicity. It should use techniques such as Everything-as-Code, with declarative (rather than imperative) code where possible, which describes the desired state and is stored in a version controlled repository for audit and, most importantly, the ability to evolve. This results in Agile applications, built using an Agile architecture, running on Agile infrastructure.

This section describes a reference architecture that attempts to meet these criteria, but does not represent a fully evolved or production-ready state, although it may well be used to model an implementation that does.

We use a "forward-only" paradigm throughout. This means that every change is first written as code, committed to a git repository, a pull request is raised and peer reviewed, ultimately triggering a re-creation of the relevant resources. This applies equally to the infrastructure, architecture and the applications themselves.

We'll first look at the flow of application build, test, and deploy, from an engineer's laptop through to production. Then we'll look at the cluster configuration and cloud architecture.

1. Build, test and deployment flow

Figure 6 shows the forward-only flow of changes, all the way to production. It shows how Continuous Integration (CI) can trigger Continuous Deployment to Production (CDP) (or, if there are human gates, Continuous Delivery (CD)).

2. It starts with a change to a component

A developer makes a change to one or more of the application's microservices (or its architecture, or infrastructure), builds and tests the change locally (not shown in the figure, but using an identical cluster configured in Minikube [13]) and then pushes the change to a monorepo. It forms part of a Pull Request, which is tested and peer reviewed before merging to master. While it is still on a branch, though, the push is communicated to a bot in the tooling cluster (CI) via a webhook. The bot triggers a build and test workflow, which in this diagram is shown as being implemented in Argo [1] because, in our opinion, this is a well implemented workflow engine that is well suited for running CI Docker builds inside containers running in a Kubernetes cluster. Because it is Kubernetes-native, Argo gives us the ability to leverage the declarative nature of Kubernetes resources, including Custom Resource Definitions (CRDs). This in turn allows us to store the workflow configuration, and the full CI/CD pipeline, as code in the monorepo alongside the application microservices. (Argo is much lighter than Jenkins [6] for example and gives us full control of every component's build and test - each step in the workflow is entirely declared as code in the monorepo [10] and is run in its own Kubernetes pod).

This is Pipeline-as-Code and keeps the build and deploy in step with the application, its architecture and infrastructure.

3. A note about the monorepo

Although not required, there are significant advantages in using a monorepo so that we can do atomic commits across the whole application (including the pipeline). Figure 7 shows how we can use the git commit SHA (e.g. db77f39) as a version number. This version represents a single handle on our application which is consistent across all microservices and associated artefacts. It's a pointer to a configuration where every component is designed, built and tested to work together with every other component. It eliminates the need for SemVer [15] and massively reduces the complexity of dependency management within large microservice orchestrations, ultimately reducing risk and improving reliability of our application.

4. Dynamically configured workflow

The triggered build and test workflow first performs analysis of the push to the monorepo in order to identify which components have changed. A useful open source tool for this (written by the authors) is Monobuild [2] which can produce a build schedule based on the dependencies of changed components. This minimises the amount of work that is even attempted by the CI workflow and can be used to easily configure a dynamic Argo workflow template. However, because each component build is idempotent, building an unchanged component will result in the exact same artefacts, so that subsequent idempotent steps will also not result in anything being updated (or deployed).

In general, we don't want to do any unnecessary work (for example, by repeating work already done). One way to achieve this would be for each build step to be considered a pure function of the input file tree, where running the same function on the same tree would always result in the same output (which is also a file tree). By creating a hash of the input tree, we have a very useful cache key which could easily be used to retrieve a cached output tree from a previous execution of the function, preventing the need to run the function a second time. Furthermore, if the artefacts have already been stored (for example, in a registry) the cache could instead just return a hash of the output tree, ready to pass to the next step directly. A CI capability built this way would be very fast, most of the time.

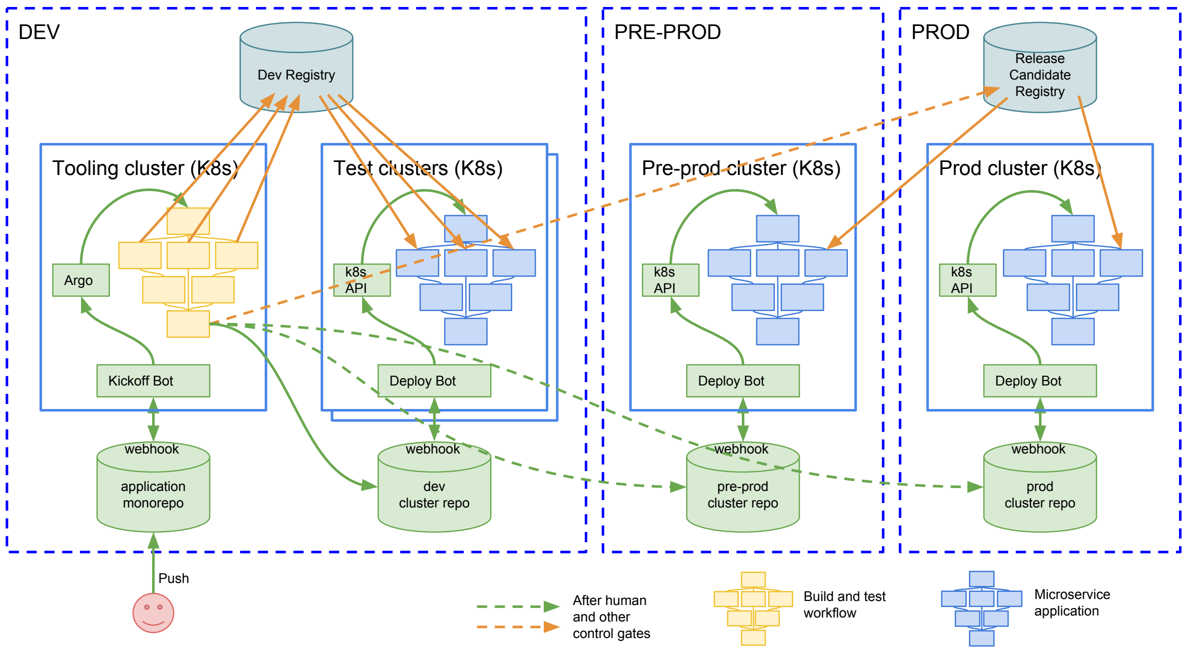


Figure 6: Build, test, and deployment workflow

5. Content based addressing

Let's pause for a moment to understand how content based addressing is important here. Git uses a Merkle tree [9] where every commit is identified by a SHA [14] (a hash) of its full file tree and those of every commit before it. We referred to using this as the version number in the section, above, on the monorepo. This SHA is a content-based address derived from the source code itself, where the same source code, with the same history, will produce the same SHA. This uniquely identifies every aspect of the source code itself and allows us to introduce idempotency into the build, where building the same commit will produce the same artefacts, which, importantly, are also identified using content based addressing.

If the artefact is a docker image, for example, then each layer is identified by a SHA (which can be thought of as a hash of its bytes), and an image can be identified by a SHA (which can be thought of as a hash of all of its layers' hashes). These are used by the registry, to store the layers, and can be used by our pipeline during the deployment phases to ensure idempotency (more on this below).

In summary, content based addressing gives us idempotency:

∴ Builds become idempotent (i.e. steps not repeated unnecessarily) because of the source code hash (Git SHA).

∴ Registry pushes become idempotent (i.e. layers not pushed unnecessarily) because of the layer digest.

∴ Service deployments become idempotent (services not redeployed if they have not changed) because of the image digest.

All this means that we can actually submit a description of the entire service orchestration to the CI workflow and only the changed components would be rebuilt (only attempted if the source hash changed, and even then only the top layers of the container are likely to be rebuilt due to docker layer caching), pushed to the registry (also due to layer hashes), and deployed to the environment (due to the image hash).

Being able to submit an entire orchestration, declaratively, brings reliability, repeatability and efficiency, with benefits arising from not having to worry about service connectivity, versioning and compatibility.

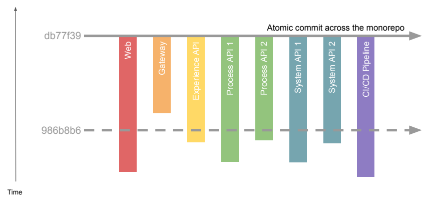


Figure 7: Atomic commits across a monorepo

6. Branch build and test

The workflow template is populated for the changed components, and builds in series and in parallel based on the build schedule. Unit tests are run as part of the Docker build, so an image is not created if any of the unit tests fail. Code quality analysis may also happen during the build. Docker multi-stage builds are used to ensure that the final image does not contain any of the build/test dependencies (i.e. the release artefacts are copied into a vanilla, lightweight container image that is usually based on alpine linux [5]). This allows our release images to be much smaller, facilitating easier deployment and start up time, whilst also reducing our attack surface, as only software that is required at run time is included.

Each component is built and tested in its own Kubernetes pod, using a Docker-in-Docker [8] sidecar container running a Docker daemon. This means that layer caches from previous builds are not available, potentially meaning that work has to be re-done in order to successfully build a full image. To remediate this, the authors have written an open source drop-in replacement for "docker build", called build-with-cache [11]. This small utility analyses the multi-stage Dockerfile to pull (and push back) images from previous builds of each target, allowing the current build to re-use unchanged layers. We have found that this can save a lot of unnecessary rework and potentially reduce the build time (we incur an overhead of pulling the previously built image and pushing the newly built image, but this usually pays off).

Successfully built images are pushed to a local registry, tagged with the commit SHA, ready for deployment to the dev cluster. Because the tag is the hash of the whole monorepo, an unchanged component will receive a new tag. This is not a problem, as we'll see in the next section, because we resolve this to a hash of the image itself before deployment.

7. Branch deploy

Outside-in testing (such as BDD style functional tests and integration tests) need a running instance to test against. So, for every branch with an open pull request (PR), an instance of the whole application is deployed to the dev environment.

In order to ensure idempotency during deployment, the Kubernetes manifests (yaml files) that reference Docker images (e.g. deployment manifests) have their image names resolved before deployment. This is done by making a HEAD request to the registry and getting the image content digest from the response headers. This is important, because of the monorepo. Remember that there is only one commit SHA across all the components, so even if a component has not been changed, its SHA (and hence tag in the registry) will still be updated (this is one of the drawbacks of using a monorepo). However, the resolution from commit SHA to content digest SHA brings this back in line, meaning that only manifests of changed components will end up being different from those of a currently deployed instance.

A URL (that includes the branch name) is posted back to the PR, so that developers, testers and stakeholders have easy access to review the new feature. The pipeline can now execute these tests against the instance, and report a successful (or failed) build back to the PR comments. This could be adapted to perform on-demand deployments for manual testing, in order not to waste resources with a long-lived preview environment that nobody needs.

8. GitOps

In the reference architecture deployment to each environment is done through an associated git repository. This is called GitOps [4]. We have written an open source tool (simply called deploy [12]) that can both raise pull requests against a cluster specific repo, and act as an agent (or bot) inside the associated target cluster that listens on a webhook, validates the deployment against the cluster, and deploys the updated applications to the cluster.

One of the most important security features of this mechanism is that the CI workflow does not need elevated permissions in order to deploy to higher environments. GitOps allows for your configuration to be pulled by the cluster, rather than pushed by the CI system, and provides a point for controls to be implemented. For additional security, the bot runs under the specified RBAC service account within the cluster.

The master branch of the cluster repo provides a non-repudiable history of the cluster's configuration, which is great for audit. Furthermore, historical configurations can be deployed to another cluster for fault finding. GitOps fits nicely with the DevSecOps [3] philosophy.

9. Master build, test and deploy

When a pull request is merged to master, the exact same steps as described above are performed. The only difference is that these builds have the potential to progress further, through higher environments, and into production. Therefore, on successful completion of the build and test phases, the resulting images are pushed into a release candidate registry.

There may be human, or other control gates, in this part of the pipeline, but what's important is that it is still 100% automated, and each environment deployment is identical to the one before it (with the exception that the images come from a release candidate registry). These deployments are also performed according to the GitOps philosophy, through the git repo that is associated with that environment's cluster.

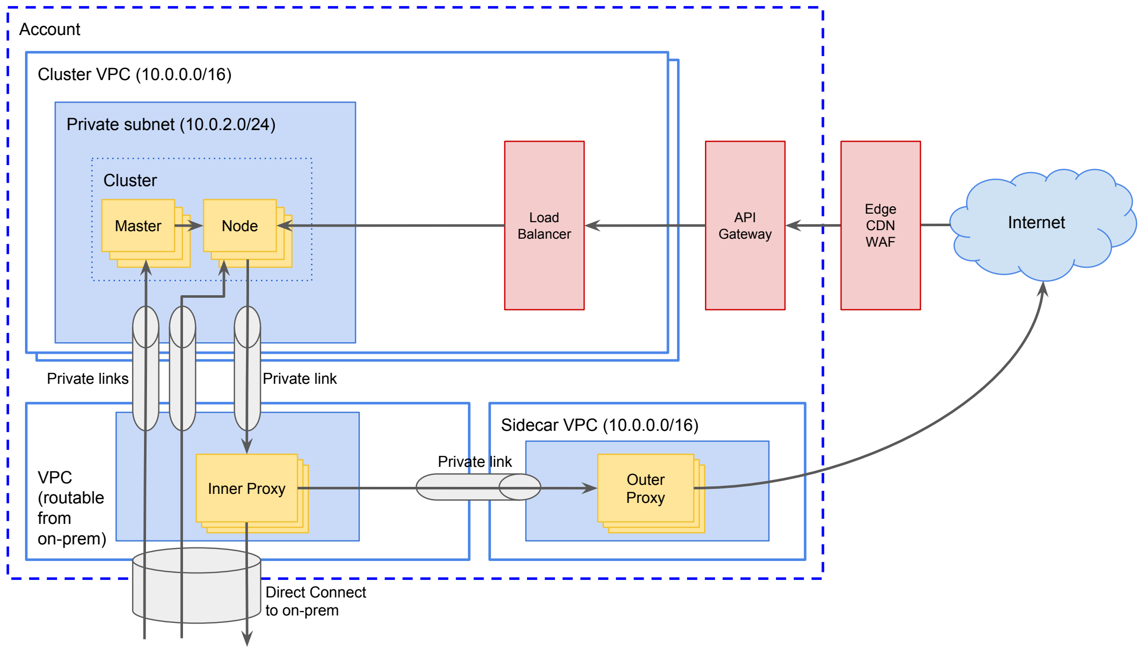

Account
Cluster VPC (10.0.0.0/16)
Private subnet (10.0.2.0/24)
Cluster
Master
Node
Load Balancer
API Gateway
Edge CDN WAF
Internet
Private links
Private link
VPC (routable from on-prem)
Inner Proxy
Private link
Sidecar VPC (10.0.0.0/16)
Outer Proxy
Direct Connect to on-prem

Figure 8: Cloud architecture with connection to on-prem backends

As the application is deployed to each higher environment, confidence in the correctness of the software is increased. Smoke tests in each environment are triggered by the workflow to ensure that the application has bedded into the environment correctly (e.g. it has discovered and connected to the right upstream services).

If CDP (rather than CD) is being practised, this final part of the workflow ensures that every commit to master will end up in production. If it passes all its tests and controls and has successfully been through all the lower environments, it will get to production. New features should be behind feature flags to allow separation of deployment from release.

10. Infrastructure

Infrastructure and cluster provisioning are fully automated, using tools such as Kops [7] and Terraform [16]. Figure 8 shows a typical AWS architecture where connection to on-prem backends is required.

The Kubernetes clusters are provisioned in non-routable VPCs. There may be several of these VPCs, each with one or more clusters (with each cluster in its own private subnet). Each non-routable VPC has its own private address range, and these ranges can each be the same (e.g. 10.0.0.0/16), which means they don't have to be assigned/managed and it gets us closer to our goal of having every environment be identical to every other. The fact that it is non-routable means that the private links (layer 4 routers with associated L4 load balancers) are the only way into and out of the VPC. These well-defined ingress and egress points provide control and mean that even if the VPC is compromised, the attacker would only be able to route traffic to known locations.

For high availability, masters, nodes, and proxies are provisioned across multiple availability zones. So, for instance, there would be 3 inner and 3 outer proxies, and at least 3 masters and and least 3 cluster nodes, spread across 3 availability zones.

Internet traffic comes in through an edge Web Application Firewall and Content Delivery Network, an API Gateway and a L7 load balancer so there are well defined opportunities to regulate incoming HTTP traffic.

All outbound traffic from the cluster must exit through a private link to an "inner" L7 forward proxy (e.g. squid), which routes Internet traffic through another private link to an "outer" L7 forward proxy in another non-routable (sidecar) VPC, and on-prem traffic through firewalled direct connect to internal destinations.

Only non-routable VPCs have Internet gateways attached, providing ingress-only to the cluster VPCs and egress-only from the sidecar VPC.

Ideally humans should not require access to any machines in this infrastructure because the infrastructure is immutable (can only be created and destroyed, not modified), and so it should be possible to deny access for ssh throughout (not only is port 22 not configured in any security groups, but the virtual machines do not even need to run an ssh daemon). If there is a problem somewhere, it is picked up by extensive monitoring and logging, the problem is fixed in the source code and the infrastructure reprovisioned.

The infrastructure provisioning is 100% automated using Terraform scripts. The cluster configuration can be generated by Kops. By using Kops to output Terraform scripts (and overriding resource definitions as required), we can also use Terraform to provision the clusters, committing the code to github as we go. All infrastructure can be provisioned this way, and any infrastructure changes must be made by modifying, and re-applying, the scripts.

[1] argoproj/argo: Get stuff done with container-native workflows for kubernetes. https://github.com/argoproj/argo. (Accessed on 06/20/2018).

[2] charypar/monobuild: A build orchestration tool for continuous integration in a monorepo. https://github.com/charypar/monobuild. (Accessed on 06/20/2018).

[3] devsecops. http://www.devsecops.org/. (Accessed on 06/20/2018).

[4] Gitops part 4 - application delivery, compliance, and secure ci/cd - dzone security. https://dzone.com/articles/gitops-part-4-application-delivery-compliance-and. (Accessed on 06/20/2018).

[5] index — alpine linux. https://alpinelinux.org/. (Accessed on 06/20/2018).

[6] Jenkins. https://jenkins.io/. (Accessed on 06/20/2018).

[7] kubernetes/kops: Kubernetes operations (kops) - production grade k8s installation, upgrades, and management. https://github.com/kubernetes/kops. (Accessed on 06/20/2018).

[8] library/docker - docker hub. https://hub.docker.com/_/docker/. (Accessed on 06/20/2018).

[9] Merkle tree - wikipedia. https://en.wikipedia.org/wiki/Merkle_tree. (Accessed on 06/20/2018).

[10] Monorepos - trunk based development. https://trunkbaseddevelopment.com/monorepos/. (Accessed on 06/20/2018).

[11] redbadger/build-with-cache: Docker multi-stage build with layer caching. https://github.com/redbadger/build-with-cache. (Accessed on 06/20/2018).

[12] redbadger/deploy: Deploy to kubernetes through a github repo. https://github.com/redbadger/deploy. (Accessed on 06/20/2018).

[13] Running kubernetes locally via minikube - kubernetes. https://kubernetes.io/docs/setup/minikube/. (Accessed on 06/20/2018).

[14] Secure hash algorithms - wikipedia. https://en.wikipedia.org/wiki/Secure_Hash_Algorithms. (Accessed on 06/20/2018).

[15] Semantic versioning 2.0.0 — semantic versioning. https://semver.org/. (Accessed on 06/20/2018).

[16] Terraform by hashicorp. https://www.terraform.io/. (Accessed on 06/20/2018).

[17] AWS. What is amazon ec2 auto scaling? https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html. (Accessed on 03/14/2018).

[18] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. Google - site reliability engineering. https://landing.google.com/sre/book.html. (Accessed on 03/14/2018).

[19] Cal Evans. Professional programming: Dtap - part 1: What is dtap? — php[architect]. https://www.phparch.com/2009/07/professional-programming-dtap-%E2%80%93-part-1-what-is-dtap/, 2009. (Accessed on 03/14/2018).

[20] David Ferraiolo. Role based access control — csrc. https://csrc.nist.gov/projects/role-based-access-control, 2012. (Accessed on 03/14/2018).

[21] Neil Ford and Rebecca Parsons. Microservices as an evolutionary architecture — thoughtworks. https://www.thoughtworks.com/insights/blog/microservices-evolutionary-architecture, 2016. (Accessed on 03/14/2018).

[22] Martin Fowler. Bluegreendeployment. https://martinfowler.com/bliki/BlueGreenDeployment.html, 2010. (Accessed on 03/14/2018).

[23] Martin Fowler. Testpyramid. https://martinfowler.com/bliki/TestPyramid.html, 2012. (Accessed on 03/14/2018).

[24] Martin Fowler. Continuousdelivery. https://martinfowler.com/bliki/ContinuousDelivery.html, 2013. (Accessed on 03/14/2018).

[25] Martin Fowler. Devopsculture. https://martinfowler.com/bliki/DevOpsCulture.html, 2015. (Accessed on 03/14/2018).

[26] Jez Humble. What is continuous delivery? - continuous delivery. https://continuousdelivery.com/, 2010. (Accessed on 03/14/2018).

[27] Masaaki Imai. Kaizen : The key to japan's competitive success: Mcgraw-hill, new york, isbn 978-0075543329. https://www.amazon.co.uk/Kaizen-key-Japans-competitive-success/dp/007554332X, 1986. (Accessed on 03/14/2018).

[28] Jonathan Kohl. Exploratory software testing: Finding the music of software investigation. http://www.methodsandtools.com/archive/archive.php?id=65, 2007. (Accessed on 03/14/2018).

[29] Kubernetes. Performing a rolling update. https://kubernetes.io/docs/tutorials/kubernetes-basics/update-intro/, 2017. (Accessed on 03/14/2018).

[30] James Lewis and Martin Fowler. Microservices. https://martinfowler.com/articles/microservices.html, 2014. (Accessed on 03/14/2018).

[31] David McRaney. The sunk cost fallacy – you are not so smart. https://youarenotsosmart.com/2011/03/25/the-sunk-cost-fallacy/, 2011. (Accessed on 03/14/2018).

[32] Gerard Meszaros. Test stub at xunitpatterns.com. http://xunitpatterns.com/Test%20Stub.html, 2008. (Accessed on 03/14/2018).

[33] Floyd Piedad and Michael Hawkins. High availability: Design, techniques, and processes - floyd piedad, michael hawkins - google books. https://books.google.co.uk/books?id=kHBOHdQ98qYC&dq=high+availability+floyd+piedad+book&printsec=frontcover&source=bn&hl=en&ei=gsOLSrLvBKjm6gOT3ISPCA&sa=X&oi=book_result&ct=result&redir_esc=y#v=onepage&q=high%

20availability%20floyd%20piedad%20book&f=false, 2001. (Accessed on 03/14/2018).

[34] Danilo Sato. Canaryrelease. https://martinfowler.com/bliki/CanaryRelease.html, 2014. (Accessed on 03/14/2018).

[35] Jan Sysmans. saas.pdf. http://www.winnou.com/saas.pdf, 2006. (Accessed on 03/14/2018).

[36] Sean Treadway. 48 concerns for a single microservice. https://youtu.be/NX0sHF8ZZgw?t=17m48s. (Accessed on 03/14/2018).