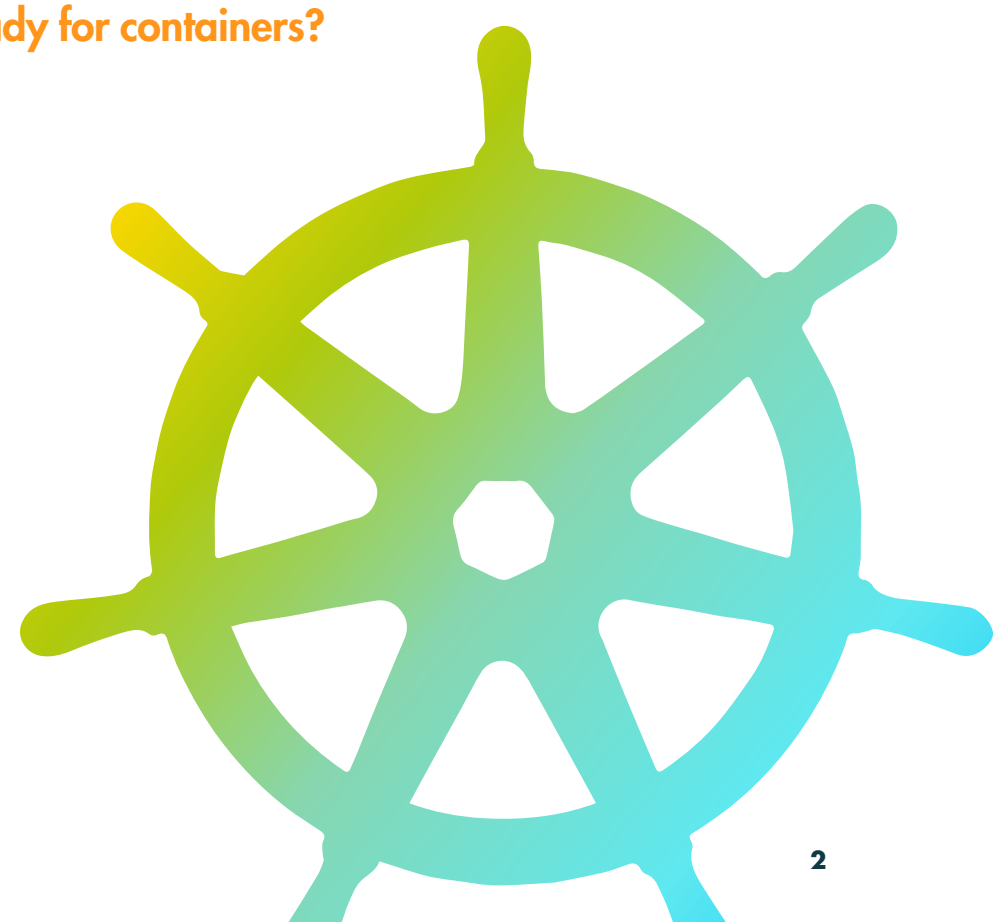
devopscon.io

DevOpsMagazine
RE-THINK IT!

Kubernetes is Eating the World

Index



Editorial

Sarah Schlothauer,
Editor

Kubernetes is Eating the World: A Technological Revolution

In modern computing, Kubernetes stands tall as the ultimate game-changer. Its influence is a mythical kraken with a voracious appetite, devouring outdated paradigms and ushering in a new era of digital transformation. Kubernetes has become the linchpin for businesses aiming to thrive in an increasingly competitive landscape. By simplifying complex tasks, the open source marvel allows developers to focus on innovation, fostering continuous improvement. Embracing Kubernetes is no longer a choice; it's a strategic imperative for smooth sailing.

In this whitepaper, you'll explore the high seas of Kubernetes with our experts. Follow Kobi Biton as he guides you through the most important Kubernetes decisions to help organizations stay relevant and resilient in the face of an ever-changing technological landscape.

@devops_con

Kubernetes, Decisions — Part 1

Multi-Tenancy, Purpose-Built Operating System

Kubernetes is a container orchestrator that has rapidly gained popularity among developers and IT operations teams alike. While it started out as a tool to manage containerized applications, it has evolved into much more than that — Kubernetes is a microcosmos and it is "eating the world."

Kobi Biton

A few months ago, I was honored to receive an invitation to speak at DevOpsCon, Munich [1]. I delivered a session [2] on architectural decisions for companies building a Kubernetes platform. Having spent time with strategic companies adopting Kubernetes, I've been able to spot trends, and I am keen to share these learnings with anyone building a Kubernetes platform. I had many interesting 'corridor' discussions after my session (seemed like I touched a raw nerve), and this led me to write this article series.

The challenge(s)

So, you decided to run your containerised applications on Kubernetes, because everyone seems to be doing it these days. Day 0 looks fantastic. You deploy, scale, and observe. Then Day 1 and Day 2 arrive. Day 3 is around the corner. You suddenly notice that Kubernetes is a microcosmos. You are challenged to make decisions. How many clusters should I create? Which tenancy model should I use? How should I encrypt service-to-service communication?

Decision #1, multi-tenancy

Security should always be job-0, if you do not have a secure system, you might not have a system.

Whether you are a Software-as-a-Service (SaaS) provider or simply looking to host multiple in-house tenants, i.e., multiple developer teams in a single cluster, you will have to make a multi-tenancy decision. There's a great deal of difference in how you should approach multi-tenancy given the use cases.

No matter how you look at it, Kubernetes (to date) was designed as a single-tenant orchestrator from the ground up. The temptation to 'enforce' a multi-tenant setup using (for example) namespace isolation is considered a logical separation which is far from a secure approach.

Let's look at the following diagram to understand the challenge (figure 1).

The control plane

The k8s control plane is a cluster-wide shared entity; you enforce access to the API and its objects by enforcing RBAC permissions, but that's (almost) all you got, my friend. In addition, RBAC controls are far from easy to write and audit, increasing the likelihood of the operator making mistakes.

You may ask yourself what's the 'big deal,' however, imagine the scenario where a developer needs to write an app that needs to enumerate the k8s API, and unintentionally, the app has a k8s identity which allows it to list all the k8s namespaces. This means that you just potentially exposed the entire set of customer names that runs on your cluster.

Another example could be a simple exposed app configuration which allows the tenant to enrich log entries with k8s constructs(pod) fields. This has the potential to saturate the k8s API, rendering it inoperable. While capabilities such as API Priority and Fairness [3] can help in mitigation (enabled by default), it is still considered

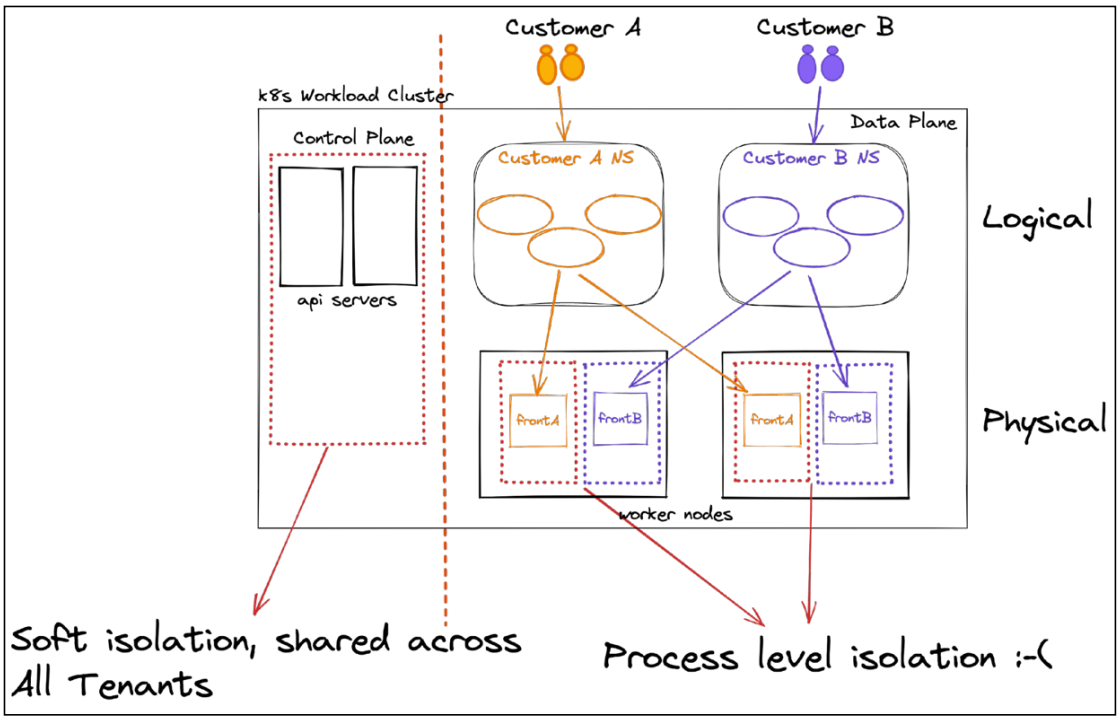


Figure 1

runtime preventative protection rather than strong isolation.

The data plane

Containers provide process-level isolation running on the same host, sharing the same single kernel, that's all [4]. So, with only k8s namespace isolation, tenants will share the same compute environments pretty much as running multiple processes on the same virtual machine (well, almost). When extending the k8s namespace isolation with tenant affinity, resulting (tenant=namespace=dedicate compute), your tenants still share the same networking space.

Kubernetes offers a flat networking model (out of the box); a pod can communicate with any pod/node regardless of its data-plane location. Very often, network policies [5] are used to limit this access to specific approved flows, i.e., all pods in namespace-a can access all pods in namespace-b but not namespace-c. This practice is still considered a soft-isolation tenancy as the tenants will share the same networking namespace. However, it will mitigate the noisy neighbor effect as tenants are collocated (affinity) onto specific dedicated compute resources.

So how do we make a decision? There seem to be so many vectors to consider. The answer, like anything in architecture, stems from business requirements, risk analysis, constraints, and the extraction of tools and technologies which will mitigate gaps.

In the next section, we start by first defining tenant profiles and then showcase a few tenancy use cases and their corresponding patterns.

Trusted/untrusted tenant/code

It is crucial to first determine whether your tenants can be trusted or if their corresponding applications run untrusted code.

A **trusted tenant** could be applications, services, and developers aligned to a certain division, i.e., a developer from team prod-a. This tenant is part of the organization's security policy. It is up to the organization's leadership to decide what and who is considered a trusted tenant (it even might be developers, contractors, or partners). It is up to the cluster operators to translate

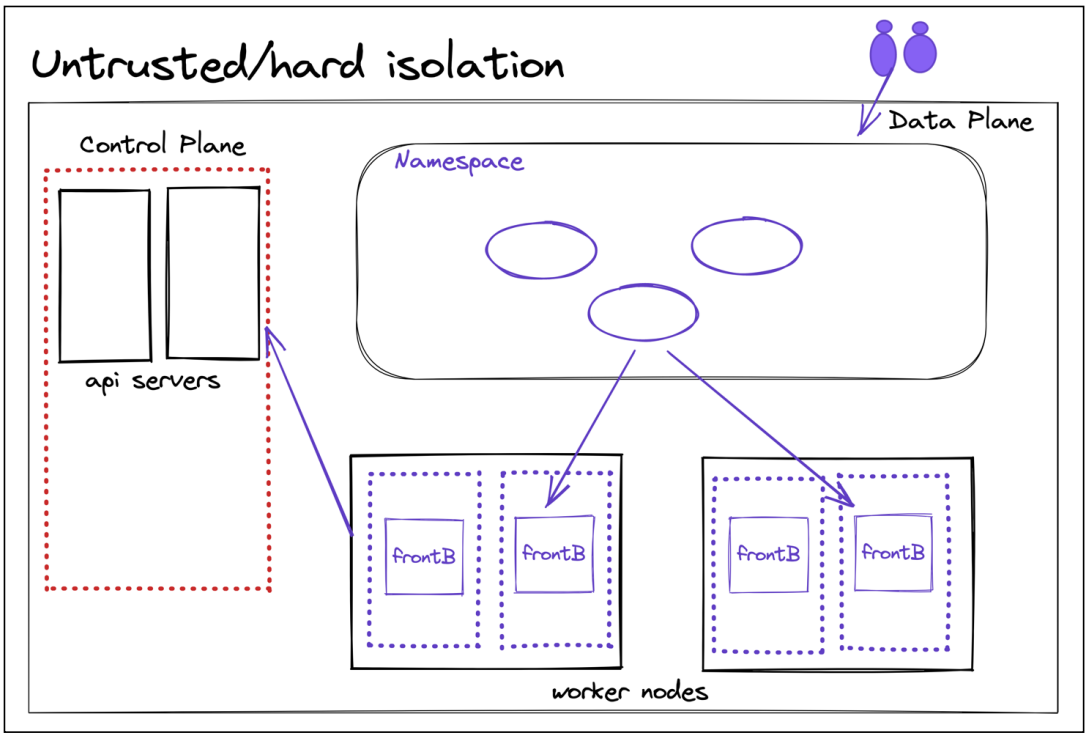


Figure 2

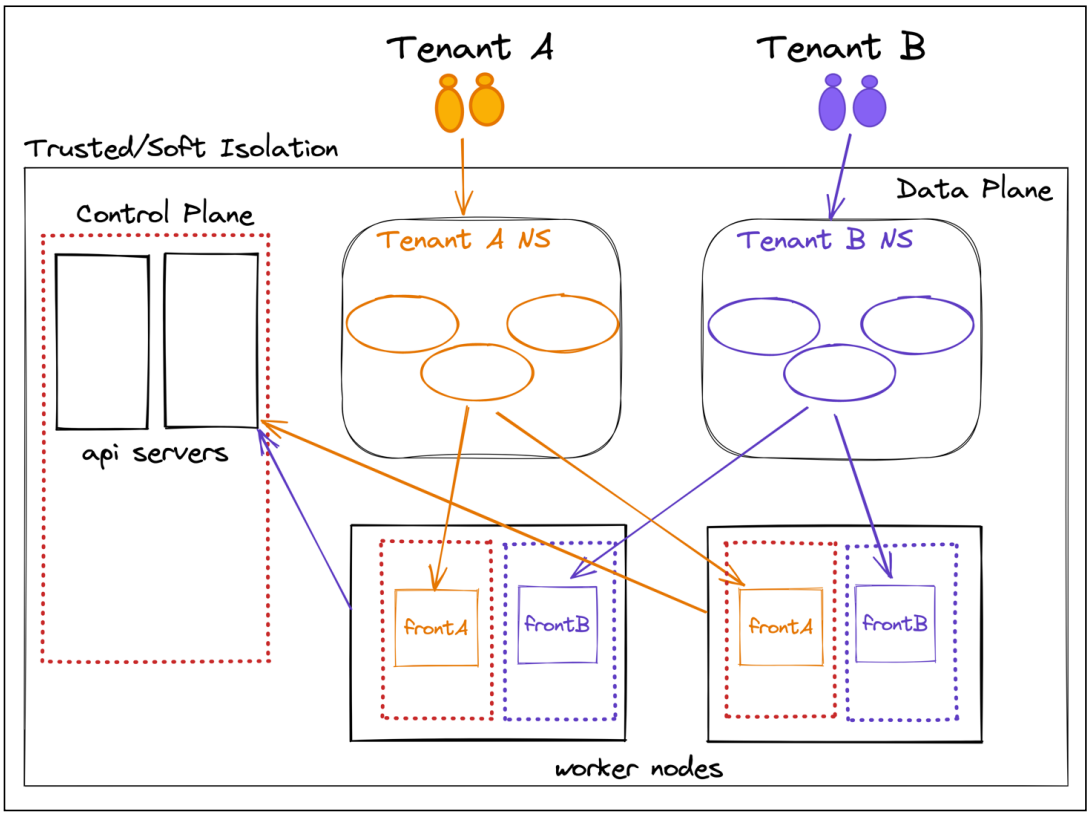


Figure 3

the trusted entities into the corresponding operational tenancy model, reflecting the threats and risks to the organization's leadership.

An **untrusted tenant** normally refers to entities which reside outside the organization's boundaries, hindering the ability of the organization to enforce its security policy. In large multi-hierarchy organizations, it is not unlikely to declare certain departments as untrusted tenants. Services and developers running untrusted code (see untrusted code explained in the next section) are to be considered untrusted tenants.

Untrusted code means that your code (or code you did not write/audit) may include low-level OS system calls or require escalated privileges which in the event of a container escape [6] may lead to catastrophic consequences that could result in other tenants' breaches. If

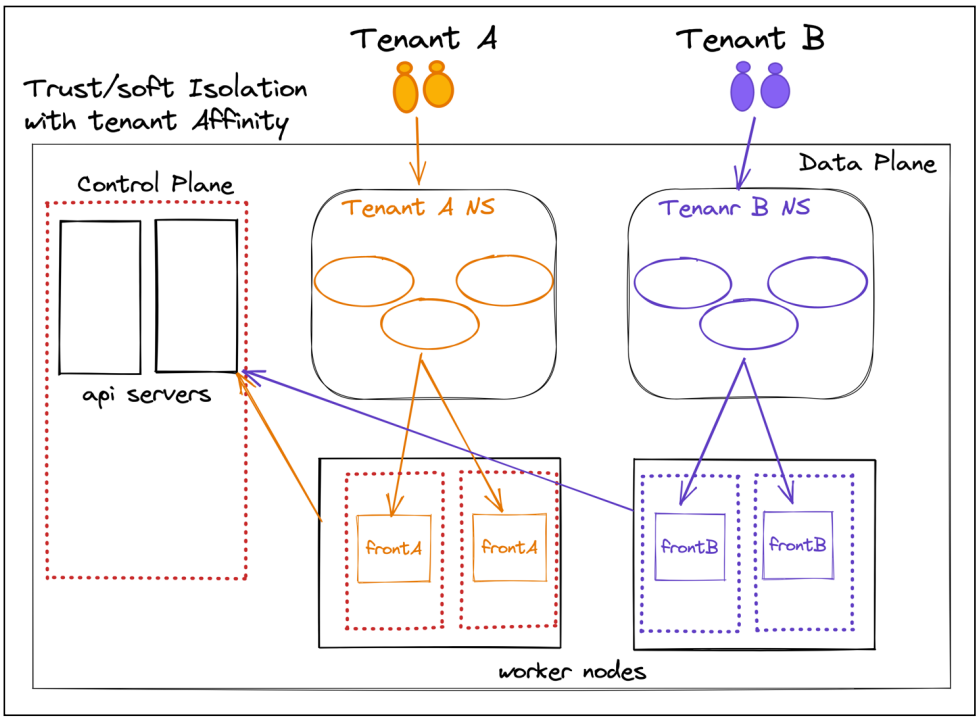


Figure 4

your applications must provide your tenants the possibility to run untrusted code (especially if they are considered untrusted tenants), I would recommend isolating this workload into a dedicated cluster/boundary. This pattern is often referred to as untrusted/hard isolation.

Use cases for the untrusted/hard isolation pattern range from k8s as Platform-as-a-Service (PaaS), apps which mandate privilege access (host level), apps which can execute arbitrary code in a container, to apps with unfettered access to the k8s API.

The following HLD depicts the untrusted/hard isolation pattern (figure 2).

Trusted/soft isolation, on the other hand, builds on the Kubernetes namespace concept. It isolates tenants logically within a logical boundary called a namespace. The operator can define per-namespace resource quotas [7] to enforce a fair share of resources within a namespace. As well as set Limit Ranges [8] on the pod level.

Logical isolation means that multiple tenants' pods will share the same compute nodes and cluster-wide resources. Due to those reasons, this soft isolation pattern use cases fit trusted tenants. Use cases can range from developer teams sharing the same cluster to very mature SaaS applications which achieve strong tenant isolation on the application level.

The following HLD depicts the trusted/soft isolation pattern (figure 3).

I am also able to spot trends which extend the trusted/soft isolation pattern, adding tenant affinity to dedicated nodes.

The following HLD depicts the trusted/soft isolation/affinity pattern (figure 4).

This pattern will solve the noisy neighbor effect and will provide a better day-2 tenant upgrade/update experience (tenants= NodeGroup). On the other hand, the k8s flat networking model still means soft isolation.

Is it a zero-one endgame?

So now that we have those three patterns, what happens when we are in the process of SaaSifying our applications/services? Should we wait until we finish the process? How should we tackle 'old' application(s), and unsecure design patterns?

An example based on previous experience included a CMS system hosting a particular microservice which allowed users to execute untrusted code. The architect team ended up building a software-based scenario engine that the customer would use to upload the code, controls, and scenario. In return, the platform team would spin up a (short-lived) dedicated k8s cluster which executed the scenario and using the async bus, reported the result to the main application APIs (hosted on a multi-tenant cluster).

You get the idea: isolate any apps or services that don't meet the multi-tenant paradigm and collaborate with the software architect teams to find modern solutions. Do not compromise on Job-0 (security).

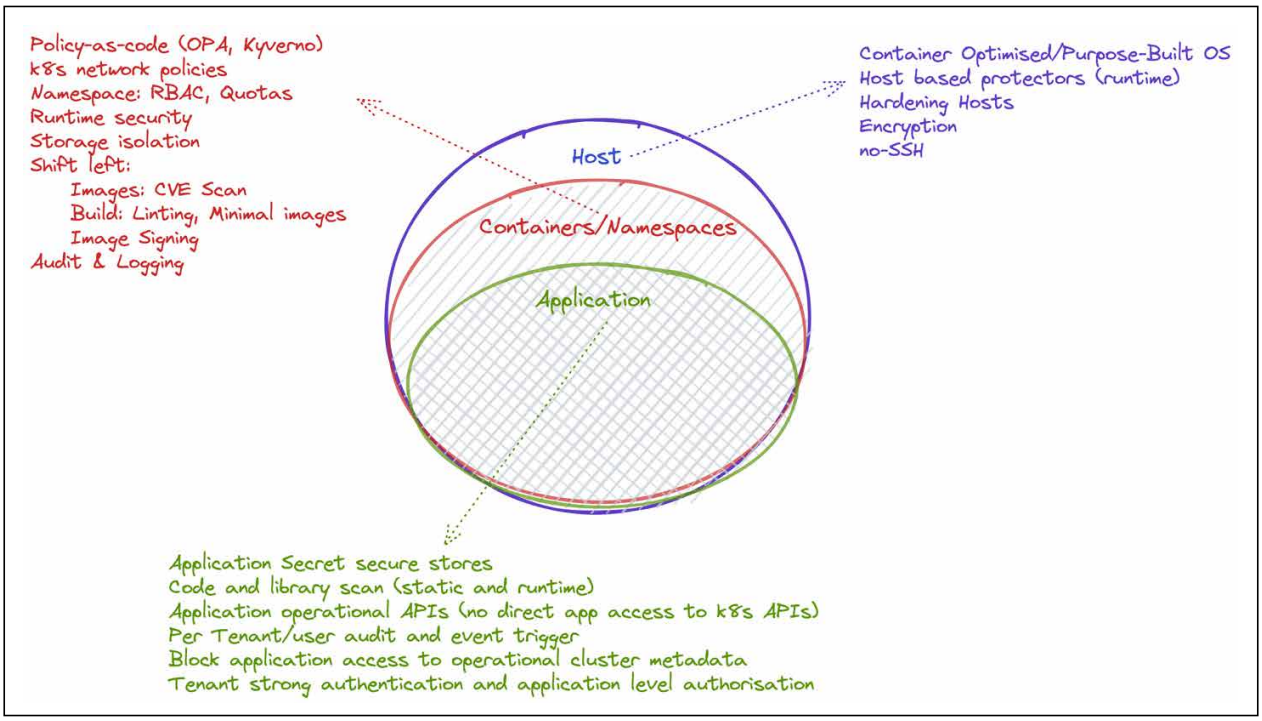


Figure 5

Regardless of the pattern(s) you choose, you are encouraged to implement common defense-in-depth [9] techniques, so let's explore those.

The common: defense-in-depth across all tenancy patterns

The following diagram depicts the controls that (at min) should be implemented (figure 5).

It is beyond this blog to dive into the controls mentioned above. They are to be implemented across all patterns. The objective is to mitigate the vertical attack vector (i.e., Pod to host), and horizontal attack vector (i.e., Pod to cluster data plane nodes).

Conclusions

- Multi-tenancy is everyone's responsibility. It's the organization's leadership which defines the business profiles (eventually translated into tenants). The CTO is in charge of ensuring that the developer teams write multi-tenant-aware applications. It is up to the platform teams to convert the business profile into a secure architecture. The CISO is the one who shapes and approves the tenant threat model.
- Do not bring the 'old' into the new, modernize and SaaSify your application while carefully considering the tenancy patterns. An old pattern could be an untrusted tenant which uses VPN to connect to designated pods (running privileged VPN server containers) in order to execute code or send data.
- Mixing multiple tenancy patterns can and should be used to bridge the gap for a true application-level multi-tenancy. At the same time, do not obsess about true application-level multi-tenancy. It often takes years to get there, especially if the application(s) is not modern. It's perfectly OK to use multiple tenancy patterns to bridge this gap.
- My advice is not to fall into deterministic definitions; trusted and untrusted are used to bridge the gaps between the organizational leadership and the operational teams. I have seen internal developers which were declared untrusted tenants due to permission to run untrusted code inside pods, posing cluster-wide risks.

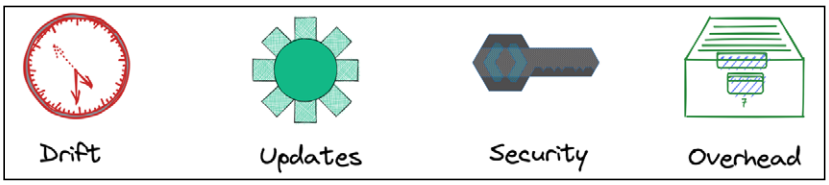


Figure 6

Where to go from here?

You get the idea; there is no "Enable multi-tenant cluster mode" tick box which will split the cluster into VM-level isolated spaces. It is your responsibility to work backwards from the business requirements, choose the proper pattern, and enforce controls to implement.

New technologies are emerging, providing VM-level-like workload isolation for containers. To learn more on those check Firecracker [10], Kata Containers [11], and gVisor [12].

Decision #2, purpose-built container operating system?

Most of the containers (to date) are still hosted on virtual machines installed with mainstream Linux operating systems: Debian-based; Vendor-based Linux OS. So, you may ask yourself: And? Is that a problem? Well, it's not a 'problem' in terms of functionality per se. Traditional operating systems were not designed to be "Just enough OS to run containers."

Challenges observed (figure 6):

- **Security** - Installing and updating extra packages simply to satisfy dependencies can increase the attack surface, and most of the time those packages are not needed.
- **Updates** - Traditional package-based update systems and mechanisms are complex and error-prone and can have issues with dependencies, leaving a fleet of virtual machines in an inconsistent state, increasing the attack surface and vulnerability.
- **Overhead** - Extra, unnecessary packages consume disk space and compute cycles and also increase startup time.
- **Drift** - Inconsistent packages and configurations can damage the integrity of a cluster over time.

Most cloud providers will use those traditional OS as a starting point to vent their own version of a container-optimized OS. At the core, those include kubelet (the k8s agent), container runtime (i.e., containerd), Network CNI plug-in, and bootstrap scripts, allowing the nodes to successfully register and connect to the k8s API servers. However, those are still based on fully blown Linux OS distros, containing package managers, and a large number of installed packages which are simply not relevant for running container workloads.

Purpose-built operating systems are a paradigm shift in the operating system domain. They are built from the ground up for one sole purpose: Running container workloads. Hence, they will be fast, secure (limited attack vectors), and adhere to the defense-in-depth principle. At the core, they will still provide kubelet, container runtime, Network CNI plug-in, and bootstrap scripts, often modified to meet the secure surface.

I am a big fan of container purpose-built OS; CoreOS spearheaded this effort releasing what is known as CoreOS Container Linux [13]. This led to multiple vendor innovations on that namespace: CoS(GCP), Bottlerocket(AWS), and FlatCar(Azure).

So why should you consider a purpose-built OS? Here's a list of features that a few of them showcase:

- No package managers
- SELinux/AppArmor (enforced)
- Kernel lockdown
- Read-only root filesystem
- API-driven configuration

Some features might be supported/enabled on purpose-built distro-a and not on distro-b.

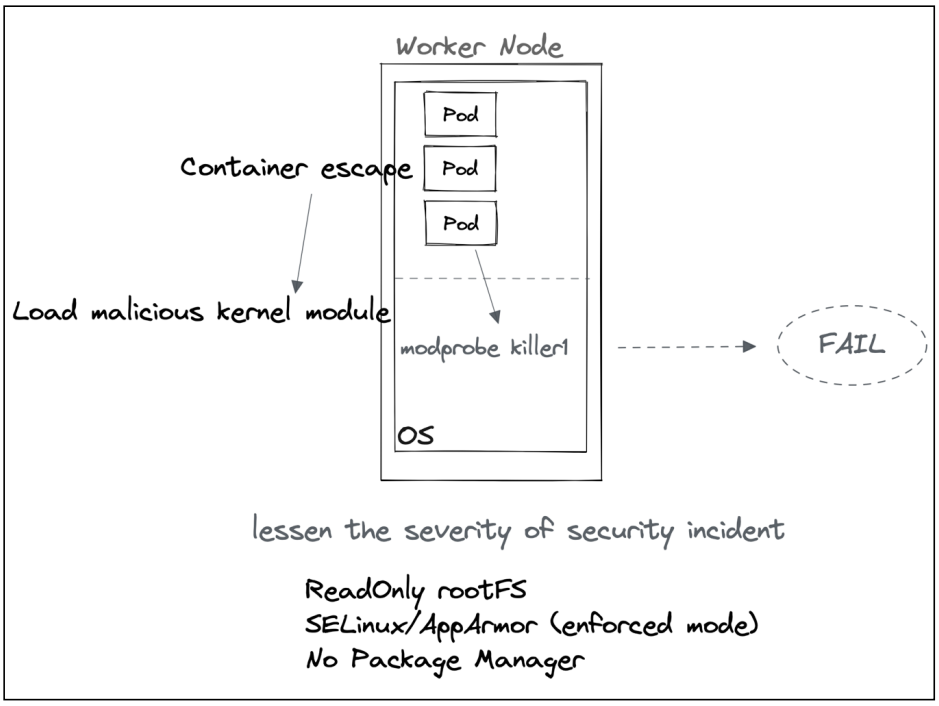


Figure 7

This is not a curated list of the features/characteristics, but you get the idea, it could fit very well into your defense-in-depth security paradigm. See this diagram that depicts it (figure 7).

In this example, an attacker escapes a container and is able to gain access to the host namespace. The attacker (as root) is trying to load a malicious kernel module. This will fail due to a feature called kernel lockdown (given the feature is available and enabled). Not to get confused, all the runtime controls you know well should be implemented in the first place in order to prevent a container escape.

Considering the above, often your workloads might not meet the purpose-built OS paradigm. Those operating systems very often require that all software, including operational (log agents and runtime protectors), runs as containers and not as binary processes in the OS host namespace. In other cases, actions such as modifying configuration files will fail or will not persist due to the read-only or tmpfs OS immutability principle.

Conclusions

- Purpose-built container OS should be your first choice OS.
- Prefer purpose-built container OS which can be deployed anywhere (cloud, on-premises), the reason being the amount of overhead you might spend on customization.
- OS customization is very limited with purpose-built container OS
- In use cases where your OS customization is wide and deep, purpose-built container OS might mean significant management overhead to a larger attack surface. In those cases, consider not to opt in.

To sum up

In this article, I have highlighted a few choices that enterprises will need to make if they decide to implement a Kubernetes platform. I hope that you find this decision analysis useful. I am curious to know if I was successful in sparking in-depth discussions within/across your teams. So what's next? Stay tuned for the next part of this article series.

Opinions expressed in this article are solely my own and do not express the views or opinions of my employer.

Kobi Biton is a Specialist Solutions Architect at AWS. With 22 years of industry experience, Kobi is specialized in solution architecture, distributed systems, and container networking. He also led architect teams for a few years. In recent years, he has been working closely with strategic Tech C-Level individuals (CTOs, Chief Architects), assisting them in their journey to application modernisation.

Links & References

[1] https://devopscon.io/munich/

[2] https://www.linkedin.com/in/kobib/overlay/1635518268530/single-media-viewer/?profileId=ACoAAABV-HgBOd4zKSFWcUiuIE4D5j9TCAFMvyk

[3] https://kubernetes.io/docs/concepts/cluster-administration/flow-control/

[4] https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/

[5] https://kubernetes.io/docs/concepts/services-networking/network-policies/

[6] https://www.cybereason.com/blog/container-escape-all-you-need-is-cap-capabilities#:~:text=Breaking%20out%20from%20the%20container,container%20to%20the%20underlying%20host

[7] https://kubernetes.io/docs/concepts/policy/resource-quotas/

[8] https://kubernetes.io/docs/concepts/policy/limit-range/

[9] https://en.wikipedia.org/wiki/Defense_in_depth_(computing)

[10] https://firecracker-microvm.github.io/

[11] https://katacontainers.io/

[12] https://gvisor.dev/

[13] https://www.redhat.com/en/technologies/cloud-computing/openshift/what-was-coreos

Kubernetes Decisions – Part 2

Number of Clusters, Authentication, Shared Responsibility

Kubernetes is a container orchestrator that has rapidly gained popularity among developers and IT operations teams alike. While it started out as a tool to manage containerized applications, it has evolved into much more than that — Kubernetes is a microcosmos and it is "eating the world."

Kobi Biton

Decision #3, how many clusters?

Should you have a large number of small-scale clusters or a small number of large-scale clusters? Without getting too deep into the Kubernetes scalability envelope [1], the following diagram is a good start (fig.1).

Possible decision model to follow:

- Step 1 - Analyze constraints: untrusted tenants, organizational boundaries, areas of impact concerns
- Step 2 - Optimize within constraints: Fewer cluster leads to more efficient bin packing and use of resources but will increase the blast radius

Conclusions

- Any production workload should be isolated from any non-production workload. This should be hard isolation (separated network boundaries, separated clusters).
- For each cluster, try to define the business deployment unit while you analyze the constraints (i.e., cluster A will host 5 customers, 10 node group, 10 VMs per node group)
- Tooling for creating multiple clusters is the new norm (Terraform, CrossPlane, cloud provider-specific IaC tools). If you opt in for multiple clusters, I recommend defining a clear automation/deployment methodology (tooling, organizational policy)

Decision #4, authentication & encryption

In sensitive zero-trust environments [2], Pod-to-Pod encryption is very often a mandatory requirement, largely implemented using mTLS [3]. Implementing your own mTLS infrastructure is complex, very often you will implement a service mesh such as Istio [4] or linkerd [5] to achieve this capability.

Implementing a service mesh will have a steep learning curve and additional operational overhead for you and

Figure 1

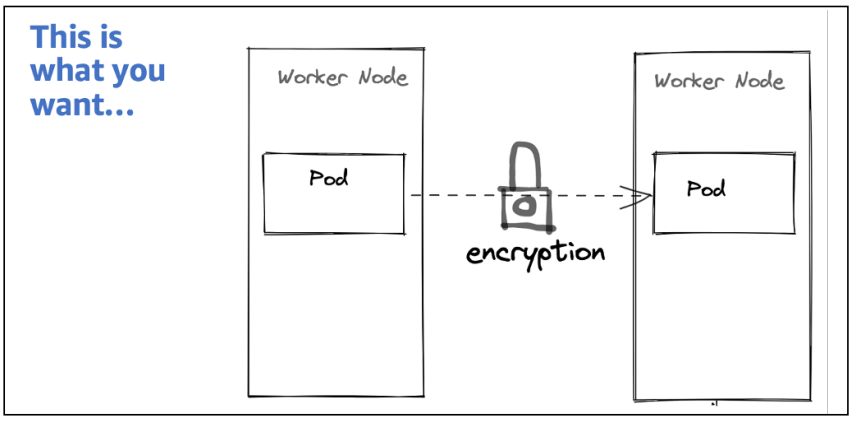


Figure 2

your teams, this needs to be taken into consideration. Going the service mesh route, if successful, will allow you to democratize the application networking by offloading communication construct to the mesh data plane, essentially your developers will only care about business logic. mTLS will then be a foundational feature inside a set of rich features: circuit breakers, smart routing, automatic retries, and deep Observability(o11y) capabilities.

The following diagrams depict the dilemma (fig.2, fig.3).

So, you end up with a full-blown service mesh control plane and data plane. To be clear, it is not an anti-pattern to implement a service mesh just for mTLS. Truth be told, I have yet to observe a proper non-service mesh mTLS solution that will make your life easier. By that, I mean that both Istio and Linkerd will include mTLS as a flag, taking care of all the heavy-lifting parts. mTLS is hard, from key generation/auto-rotation and data-plane proxies' integration to creating and associating identities. You get the idea that proper mTLS has many moving parts to take care of at installation and mainly at runtime.

However, if mTLS is not a hard requirement in your case but rather encryption [6], I encourage you to look into alternative options. Fig. 4 shows one of them.

Cilium [7] and WireGuard [8] can be integrated to create a pod-to-pod transparent in-transit encryption (network-layer). This is by far one of the fastest routes I have ever experienced to set up pod-to-pod encryption.

WireGuard was described [9] as a "work of art" by Linus(2018), and it is considered a performant and lightweight secure Virtual Private Network (VPN) solution built into the Linux kernel. One of the main advantages is that the operator does not need to manage a Public Key Infrastructure (PKI). Each node in this 'overlay' will generate its own keys, encrypting and decrypting traffic transparently.

Another transparent option that I am familiar with is using a cloud provider that offers VMs that are protected by hardware encryption primitives. Those normally transparently encrypt all 'on-wire' node-to-node traffic; this is by far the most performant and frictionless option I've seen.

The above being said, network-layer encryption (by itself) is not considered zero-trust. Why? Network-layer encryption does not provide application/workload-level identities. WireGuard identity sits on the network layer (Node), and we all know that, in our new world, nodes are an ephemeral multi-app construct. mTLS (when backed with a PKI and identity management such as

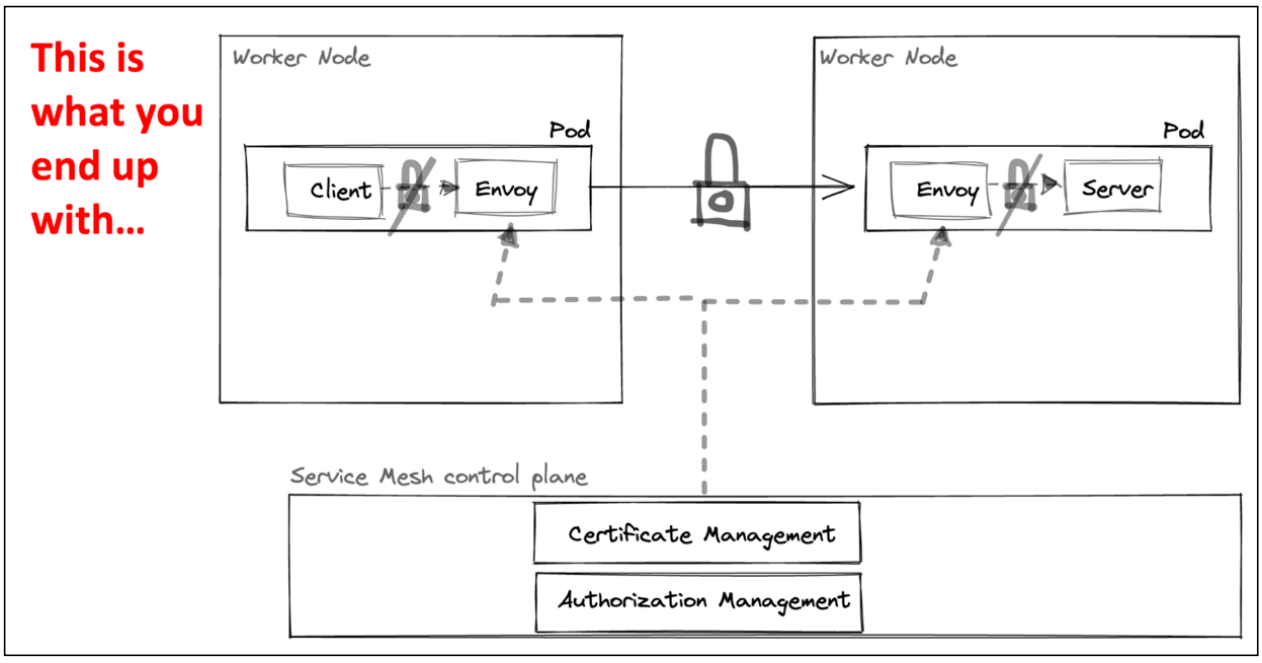


Figure 3

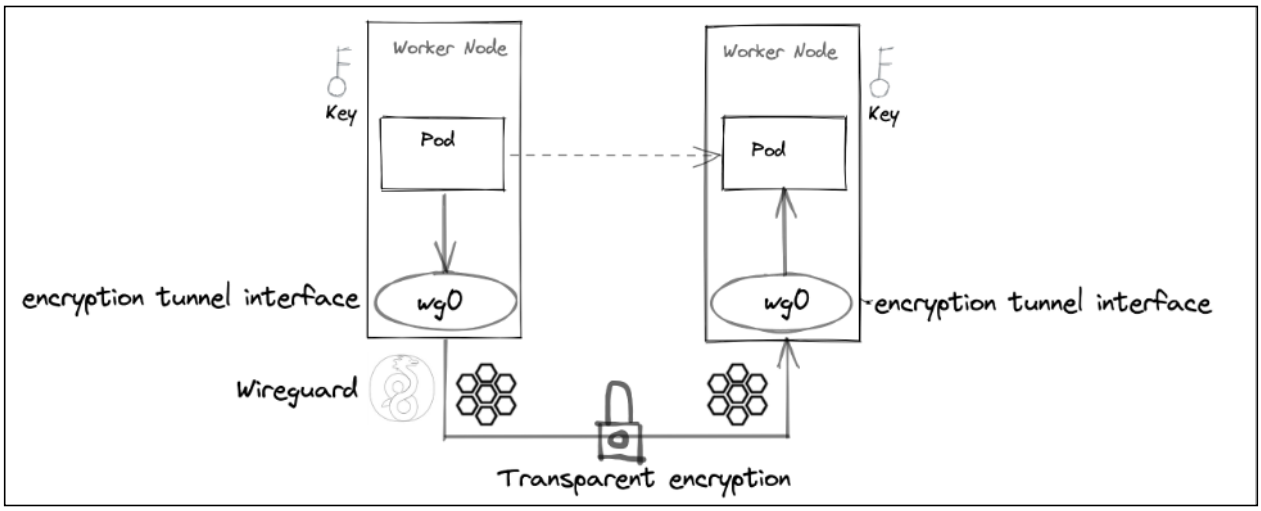


Figure 4

SPIFFE or Linkerd k8s service account integration) brings the identity into the workload level. Hence, the identity is application aware.

I have also seen companies that use both practices, network-layer level encryption and mTLS service-to-service encryption, as a defense-in-depth strategy. mTLS will do a very good job for microservice-to-microservice communication patterns. However, if you are operating k8s clusters, you are aware that a significant burden comes from operational add-ons, those add-ons (take CoreDNS as an example) traffic will not be part of your mTLS 'overlay'. While you can strive to add TLS support for those operators (DNS over TLS in the case of CoreDNS), managing the lifecycle for those operators (PKI) will be outside the scope of the service mesh. This will result in higher operational costs.

Conclusions

- If zero-trust and mTLS are strict requirements, then a service mesh with an mTLS infrastructure might seem overkill. On the other hand, it will be cumbersome to manage mTLS outside the scope of a service mesh. Some cloud providers are working to introduce technologies/services which aim to (eventually) abstract those operational complexities.
- There is no such thing as a "silver bullet," and mTLS will not be your ultimate solution for achieving zero-trust requirements. Your plan should start with a defense-in-depth model and then work backwards to identify the technologies that will help you in implementing the model, one of which is mTLS.
- It is an anti-pattern to weaken zero-trust hard requirements in order to meet operational constraints; it is a policy that drives architecture and tools, not the other way around.
- As transparent encryption (network-layer) is not workload aware, often it might not support encryption for all pod-to-pod communication patterns. Validate that the encryption patterns you expect are supported.

To sum up

Number of clusters, Zero-Trust, mTLS are decisions which are not easily reversible therefore should not be scope limited to the Architect/SRE/Platform Teams. I

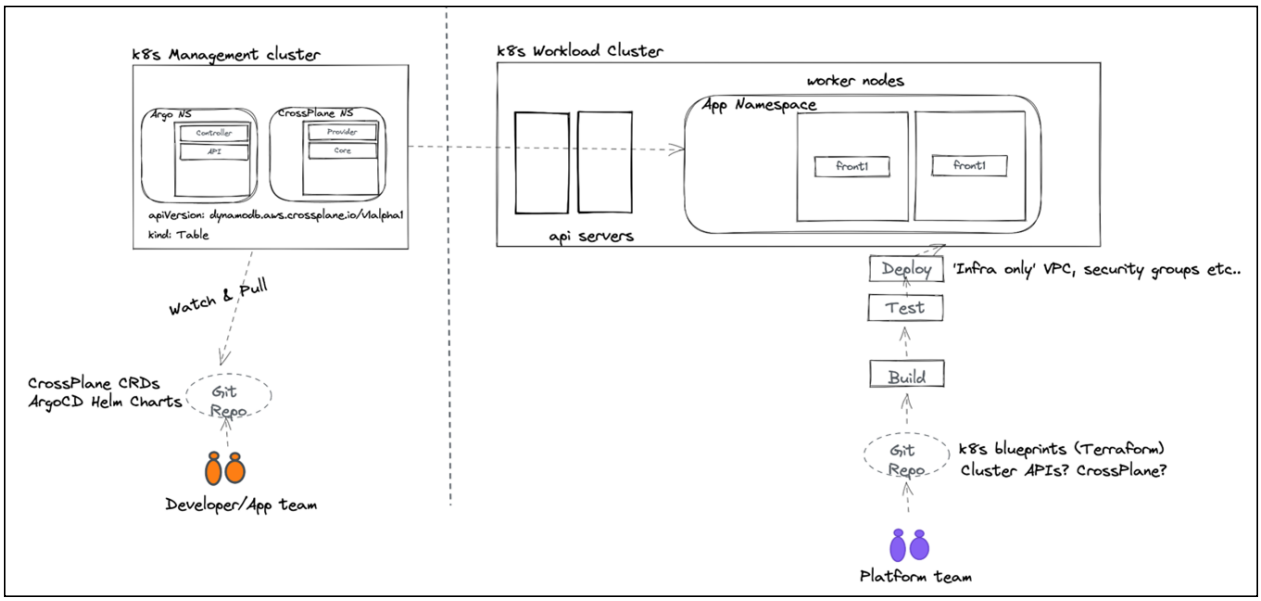


Figure 5

would go up to product/sales, which will help in better defining the roadmap/customer on- boarding, sales-cycles and the CISO which will define the Zero-Trust security policy.

Decision #5, The shared responsibility conflict

When your operational footprint grows with your business you normally need to meet those requirements. Customers I work with use relatively small dedicated teams which own the lifecycle of the infrastructure & applications "From code to a running workload". These groups are often called, Platform teams and Developer/Application teams. Without getting into the nuances of who does what. See the following diagram (**Fig. 5**) that depicts this typical architecture and the team responsibility boundaries, what I often refer to as **shared responsibility**.

So, the platform team is, well, creating the platform(s). Platforms are a set of foundational workload environments which support the internal teams (internal customers) in shared operational aspects. This team also sets the de-facto standard of how the platform will be architected, deployed, maintained, secured and scaled. So, networking, security, connectivity, technologies(k8s), APIs, IdP, etc...

The application/developer teams are the ones which are responsible to define the application requirements, i.e., how the platform should be built to support all the application aspects.

This soft separation, if played well, works well in creating a separation of duties yet fostering strong collaboration points to drive the business forward, so kind of a constant healthy level of conflict. (One hopes! 😊)

So, you may ask, what's the challenge here?

Every application needs a database/store and perhaps few other external services.

So, who owns and deploys/maintains those app dependencies the platform team or the app/dev team? If the platform team owns, it means that very often the dev/app team will need to wait for those to be deployed as part of the platform. This can take time, hindering the ability for the app/dev team to iterate fast. Trust me, I have seen many cases such as those, creating the following depicted challenge (**Fig. 6**).

When the platform team owns, the application team needs to wait for those application dependencies which are far more dynamic than platform-based dependencies.

So, what's the alternative? Recent innovation on the k8s application awareness area led to projects which

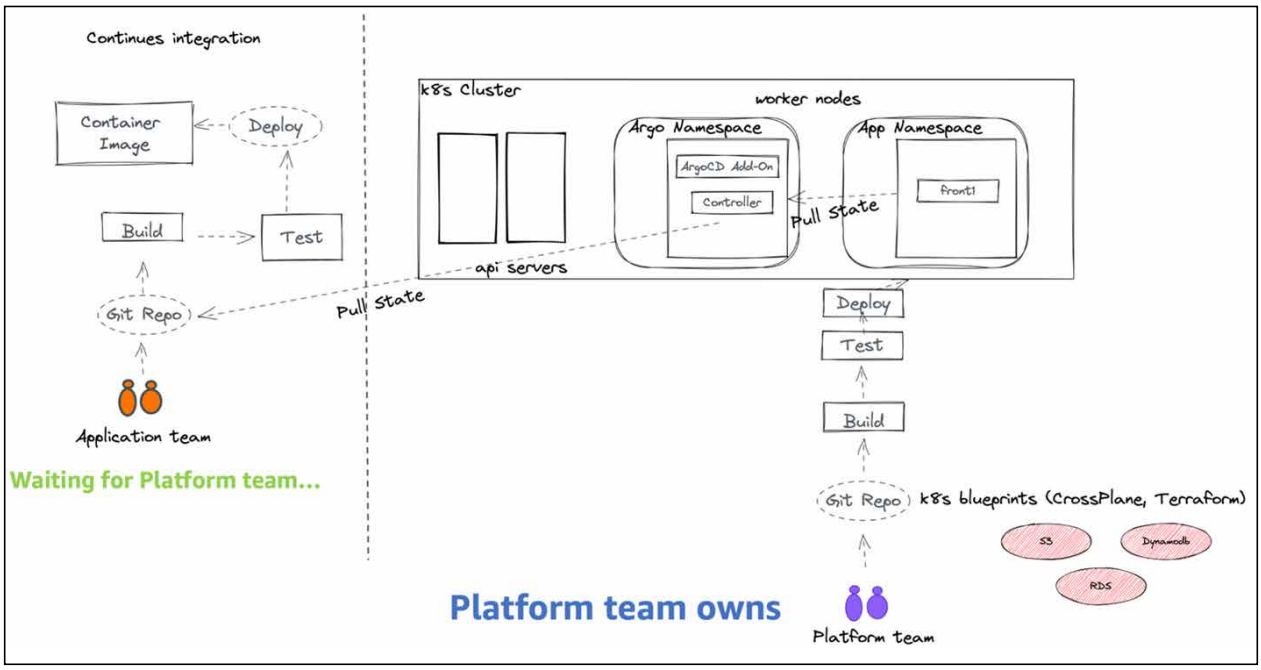


Figure 6

are able to provide a declarative cloud services operator framework (CrossPlane [10], ACK [11], ASO).

Essentially, allowing the developer to use declarative k8s constructs which deploy the k8s application and its dependencies in a full k8s declarative manner. In simple words, your helm chart contains the app baked into the container/envs/configs but also CRDs which will create a cloud key/value store table, object store, sql databases, cache stores.

This agility comes with few concerns:

- **Security & Governance** – Resources are created and managed outside the platforms team tooling and policies.
- **Cost control** – Resources are deployed with no platform team gates (DB compute size, cleanup of old resources, resources are not reported on platform team cost tools)
- **Environment drifts** – Mismatch between current vs desired, which can lead to service interruptions.

Figure 7 depicts the application team's own approach.

Conclusions

- Not necessarily should you choose one pattern over the other, have seen customers who mix-and-match the 2 approaches depending on the teams at question.
- The 'demarcation point' does not necessarily need to be drawn and kept at point-A, it is advised to keep it dynamic per the team maturity, appetite for control and agility level.
- In any pattern you choose, besides the human factor of collaboration mechanisms, I highly recommend to implement an event-driven programmatic points-of-integrations. Platform teams get notified with a new managed database service is provisioned by app/dev team. This will allow the platform team to govern and mitigate the concerns mentioned.

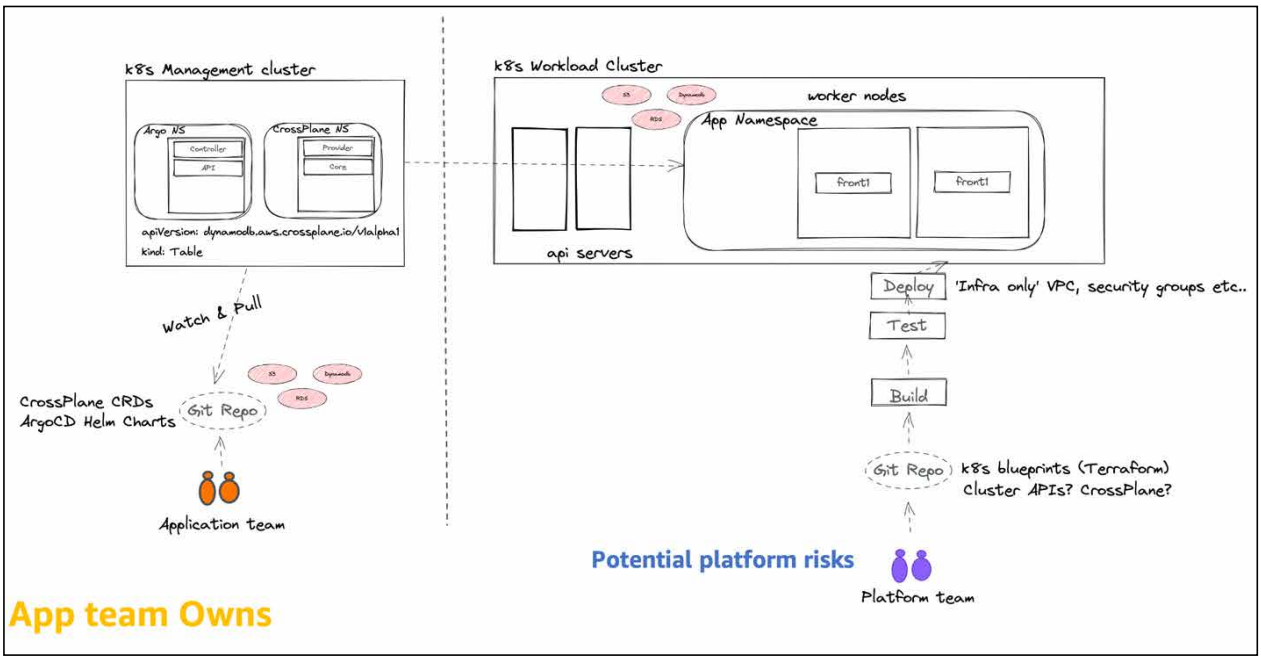


Figure 7

- Successful platform teams act as product teams. They gather requirements from app teams, prioritize feature requests in their backlog, consult their internal customers in architecture questions & provide enablement for users of the platform.

To sum up

In this article series, I have showcased few key decisions which I believe organizations will face given they chose to build a Kubernetes platform. I hope that you find those decisions analysis useful. I am eager to learn if I succeeded to generate deep discussions in/across your teams.

I am sure we will meet again, thank you very much for spending the time to read I will be VERY interested in hearing your feedback.

Opinions expressed in this article are solely my own and do not express the views or opinions of my employer.

Kobi Biton is a Specialist Solutions Architect at AWS. With 22 years of industry experience, Kobi is specialized in solution architecture, distributed systems, and container networking. He also led architect teams for a few years. In recent years, he has been working closely with strategic Tech C-Level individuals (CTOs, Chief Architects), assisting them in their journey to application modernisation.

Links & References

[1] https://github.com/kubernetes/community/blob/master/sig-scalability/configs-and-limits/thresholds.md

[2] https://www.cncf.io/blog/2022/11/04/seven-zero-trust-rules-for-kubernetes/

[3] https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/

[4] https://istio.io/latest/about/service-mesh/

[5] https://linkerd.io/2.12/features/automatic-mtls/

[6] https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/

[7] https://cilium.io/

[8] https://www.wireguard.com/

[9] https://lists.openwall.net/netdev/2018/08/02/124

[10] https://github.com/crossplane-contrib/provider-aws

[11] https://aws-controllers-k8s.github.io/community/

Is your organization ready for containers?

Are You Getting All You Can Out of Kubernetes?

If you feel underwhelmed by your Kubernetes implementation, it might be because you're not finished yet. Organizational limitations can stifle your K8s deployment.

Jeff Smith

I want to talk about Kubernetes for a little bit. But before we get into Kubernetes, I want to talk about containers. And not the containers that run inside of pods on Kubernetes. I want to talk about the old boring intermodal shipping containers that have become a staple of the shipping industry.

These dull, 20 feet long, 8 feet 6 inch tall titans of industry don't look like the type of invention that would help revolutionize commerce, yet that's exactly what it did. Back in the early 1900s, the shipping industry looked very different. The norm for shipping back then was called "Breakbulk Cargo". (It's still in use today, but for specific cases) Breakbulk cargo or "general cargo" was cargo that was shipped in individually counted units. That might sound innocuous enough, but when you're dealing with a ship that has 20,000 goods on it, you can imagine how inconvenient it must be to keep track of an individual PS5, for example. But in the early 1900s, dealing with cargo in this fashion was pretty commonplace.

There are a few issues with breakbulk cargo. One of the biggest issues was the speed at which you could load and unload the cargo. Dealing with breakbulk cargo is a labor-intensive process. Then along came a guy named Malcom Mclean.

Malcom Mclean was an American businessman and a former truck driver who was compelled to find a better

solution for the shipping industry. Mclean is credited with inventing the intermodal shipping container which changed the shipping industry forever. The interesting thing about Mclean's story was that he wasn't the first person to try using shipping containers. Shipping containers were in use as early as 1926, almost 9 years before Mclean even finished high school. So why did Mclean's idea for containers take off while other implementations surfed in mediocrity?

Mclean's biggest insight into the process wasn't just the container. It was the need for the entire industry to change in order to align practices and standards around the container. Having a container show up at a port, only to be manually unloaded a single item at a time would limit the impact that the container would have. But if ports were all standardized to expect these 20'x8'x8'6" containers, the loading, offloading and storage process could bring an advantage that just wouldn't be possible if ports used the same breakbulk infrastructure as they did with containers. Everything needed to change to maximize the success of the container.

Is your organization ready for containers?

Fast forward to today and to a different type of container. Kubernetes is on fire across the tech industry as companies rush to adopt it in hopes of turbo-charging their development organization's productivity. And why shouldn't they? Kubernetes has been touted as the tool to help teams feel empowered, to move faster, and deliver without interference or bottlenecks from other teams. Well, at least this is what the marketing information for Kubernetes says. But as the killer of hype trains, I have to tell you I've got bad news. Kubernetes can't help you with any of that because the technology was never the thing that was stopping you in the first place.

Before Kubernetes, there was the cloud, before the cloud there were data centers, and in the data centers, we had virtual machines. These virtual machines could have been automated and orchestrated in such a way to empower developers to do all the magical things we talk about today. VMWare's vCenter product was introduced in 2003. Even then vCenter provided some basic deployment and provisioning technology that, if properly exposed, could have revolutionized the speed at which developers operated. But old word patterns continue to creep in. Now, engineers still made a request to IT, but instead of that request turning into a hardware requisition, it was a VM requisition. There was definitely an improvement in speed and turnaround time, since we weren't talking about purchase orders and delivery dates, but it still shows how the organization itself limited the opportunity that the technology provided.

This is a very common scenario. Organizational mindsets and practices are usually what gets in our way. The fear of "separation of duties", operations teams that felt they were the only ones who cared about the safety of production, and of course the specter of change management, forced us all to continue to live in a world of gates, approvals, and go/no-go meetings. A lot of these processes have eased up today but the ideas and frameworks of old world thinking still lurk in our minds and our processes.

When a technology like Kubernetes comes around, we have to ask ourselves, "How is our current way of doing things holding back the true potential of this tool?" If you implement Kubernetes but still intend on concentrating all of the operational responsibility on to a centralized OPS team, you'll never achieve the full potential and capability of the tool. The marketing material on Kubernetes talks about empowering development teams, but empowerment isn't just a tool. It's a mindset and an attitude that can only exist if the organization breathes life into it. Empowerment is the ability to make a decision. Kubernetes can never give that to an engineer. It can only give them the capability to act on that decision.

If a developer finds a bug in their software, empowerment is giving them the discretion to decide that it's safe to release a bug fix on a Friday at 9pm. That decision will obviously have multiple competing factors before it's finalized. But if that same engineer needs an approval from their manager, an approval from a five person change board and a VP level sign off, then the engineer isn't empowered to do anything and running Kubernetes won't help you.

When you look to introduce a tool like Kubernetes, you must look and see how the entire ecosystem of your organization has to change. Responsibilities might be

enhanced, diluted and generalized. Software engineering teams might be thrust into the role of both software engineer and operations engineer, since it doesn't make a lot of sense for a team to be able to brainstorm, develop and launch a service, only to turn it over to another group to manage it in production.

The operations team that previously had intimate knowledge of the various applications and services, along with their corresponding idiosyncrasies, now might need to meld into the background as they become more concerned with the underlying platform (Kubernetes) and become relatively ignorant to the applications that are running on top of them.

We also have to consider the touch points that exist outside of Kubernetes and how those will be automated in a way that lets a developer continue to move quickly. Take any common service launch as an example. Chances are the service is going to have some state that it needs to store, typically in a relational database. State isn't something you necessarily want to manage inside Kubernetes (although admittedly, it's getting better), so you might look to leverage something like Amazon RDS. Well now we're in an ecosystem outside of Kubernetes. Are teams capable of also launching AWS resources to meet their needs or do they need to talk to an operations team? Does the operations team or someone else need to approve the spend for the AWS resources?

Tools like Crossplane in the Kubernetes ecosystem allow us to give engineers the capabilities to perform those actions but is your organization giving them the empowerment to make those choices? And this is something simple like a relational database! It gets way more complex if we're talking about things like API Gateways, ElastiCache or even Cognito.

Your thinking now might be "but I can't let spend run out of control" and that's a fair reaction. But the reason you might be making that argument is because you are responsible for the spend and not the creator of the spend. If the creator of the spend became responsible for it, that becomes a self-managing process as the spend starts to violate company norms. Again, this is where there has to be organizational buy-in to get the true effects of Kubernetes. Without that transfer of responsibility, the responsible party for the spend will continue to push back and create unnecessary hurdles and gates in order to relieve pressure on the spend side. But that relief comes at the cost of developer productivity. These types of battles are probably going on all throughout your organization and without a careful and specific process to address these things, they will forever limit your ability to truly get the most out of Kubernetes.

"But what about that time we had a massive outage because we let user X do action Y?" That's a common refrain I hear when talking to people about doing something potentially new and scary. The truth however is yes, those things can happen. But they probably still happen even with draconic organization practices and policies. Jason Fried once wrote that "policies are organizational scar tissue" and he's never been more accurate.

Change is uncomfortable for everyone, but sometimes change is necessary to reap all the benefits and rewards out of a choice. If you're choosing to implement Kubernetes, I urge you to also examine how your organization is going to change as a result. Do you have buy-in for the vision you have planned? How high does that buy-in go? This isn't to say that your Kubernetes deployment can't be successful, because it can be. But you might notice that you've got all of the same problems but just with a ton more complexity. And I'm sure that's not what you were thinking of when you started your Kubernetes journey.

Jeff Smith has been in the technology industry for over 20 years, oscillating between management and individual contributor. Jeff currently serves as the Senior Director of Production Operations for Basis Technologies (formerly Centro), an advertising software company headquartered in Chicago, Illinois. Before that he served as the Manager of Site Reliability Engineering at Grubhub. Jeff is passionate about DevOps transformations in organizations large and small, with a particular interest in the psychological aspects of problems in companies. He lives in Chicago with his wife Stephanie and their two kids Ella and Xander. Jeff is also the author of Operations Anti-Patterns, DevOps Solutions with Manning publishing.