foreword by Bart FARRELL

KUBERNETES STORIES

FROM THE TRENCHES

GULCAN TOPCU

Table of contents

Foreword

Humans love stories. They guide, warn, and teach. Through them, we learn what to repeat and what to avoid.

Kubernetes, even after 10 years, remains an untamed frontier. During my 3 years leading the Data on Kubernetes Community, I saw one topic resurface constantly—stateful workloads. It’s still unfinished business, with no clear standard practices.

KubeFM creates a space where engineers share the raw truth of their work—successes, struggles, and deep dives, from network policies and secrets management to debugging webhooks all the way down to the Linux kernel.

The podcast thrives because it mirrors the reality of engineers stepping into uncharted territory, navigating challenges, and returning with hard-earned lessons. Our guests are those pioneers, and I’m grateful for their willingness to share.

Let’s keep telling these stories—making our community stronger, more open, and ready to face what’s next.

Why This Book Matters

Kubernetes isn’t just about code or architecture diagrams. It’s about the people who wrestle with, and ultimately master, this technology. Our goal is to bring you into their world. By the time you reach the last page, you'll not only understand Kubernetes better, but you'll also appreciate the incredible challenges and triumphs that come with it.

What You’ll Learn & Who This Book is For

From scaling hundreds of clusters to the secrets of multi-tenancy, from breaking down complex service meshes to innovating at cloud-native companies like Adidas, each chapter unpacks the real-world applications of Kubernetes. You'll gain insights into automation at scale, security pitfalls, optimization hacks, and even hear stories of how things went wrong—and what was learned from those failures.

Whether you’re a seasoned Kubernetes operator, a developer curious about cloud-native patterns, or someone exploring tech career paths, there’s something here for you. And if you’re wondering why stories are so powerful, remember this: Concepts are easier to grasp when we see them play out in real scenarios. Here, you’ll learn from the firsthand experiences of engineers and leaders who have been in the trenches.

Contrasting Opinions & Tips for Reading

One thing you'll notice as you read is that these experts don't always agree. And that's okay. In fact, it's one of the strengths of the Kubernetes community. As you move from chapter to chapter, you'll see how different strategies can lead to success—or sometimes, to surprising challenges. You're invited to think critically and form your own views.

Each chapter stands alone, so feel free to jump to the stories that catch your eye. If you're interested in scaling, you might love Pierre Mavro's account of automating cluster upgrades. If security is your passion, dive into the saga of breaking into cloud environments with Hillai and Ronen.

Chapter 1

Building at Scale

Upgrading Hundreds of Kubernetes Clusters, Pierre Mavro

Automating the upgrade process for hundreds of Kubernetes clusters is a formidable task, but it's one that Pierre Mavro, the co-founder and CTO at Qovery, is well-equipped to handle. With his extensive experience and a dedicated team of engineers, they have successfully automated the upgrade process for both public and private clouds.

Bart Farell sat with Pierre to understand how he did it without breaking the bank.

You can watch (or listen) to this interview here.

Bart: If you installed three tools on a new Kubernetes cluster, which tools would they be and why?

Pierre: The first tool I recommend is K9s. It's not just a time-saver but a productivity booster. With its intuitive interface, you can speed up all the usual kubectl commands, access logs, edit resources and configurations, and more. It's like having a personal assistant for your cluster management tasks.

The second one is a combination of tools: External DNS, cert-manager, and NGINX ingress. Using these as a stack, you can quickly deploy an application, making it

available through a DNS with a TLS without much effort via simple annotations. When I first discovered External DNS, I was amazed at its quality.

The last one is mostly an observability stack with Prometheus, Metric server, and Prometheus adapter to have excellent insights into what is happening on the cluster. You can reuse the same stack for autoscaling by repurposing all the data collected for monitoring.

Bart: Tell us more about your background and how you progressed through your career.

Pierre: My journey in the tech industry has been diverse and enriching. I've had the privilege of working for renowned companies like Red Hat and Criteo, where I honed my skills in cloud deployment. Today, as the co-founder and CTO of Qovery, I bring a wealth of experience in distributed systems, particularly for NoSQL databases, and a deep understanding of Kubernetes, which I began exploring in 2016 with version 1.2.

To provide some context to Qovery's services, we offer a self-service developer platform that allows code deployment on Kubernetes without requiring expertise in infrastructure. We keep our platform cloud-agnostic and place Kubernetes at the core to ensure our deployments are portable across different cloud providers.

Bart: How was your journey into Kubernetes and the

cloud-native world, given the changes since 2016?

Pierre: Actually, learning Kubernetes was quite a journey. You had a less developed landscape with most Kubernetes components in alpha at these times. In 2016, I was also juggling between my job at Criteo and my own company.

When it came to deployment, I had several options, and I chose the hard way: deploying Kubernetes on bare metal nodes using KubeSpray. Troubleshooting bare metal Kubernetes deployments honed my skills in pinpointing issues. This hands-on experience provided a deep understanding of how each component, like the Control Plane, kubelet, Container Runtime, and scheduler, interacts to orchestrate containers.

Another resource that I found pretty helpful was "Kubernetes the Hard Way" by Kelsey Hightower despite its complexity.

Lastly, I got help from the official Kubernetes docs.

Bart: Looking back, is there anything you would do differently or advice you would give to your past self?

Pierre: Not really. Looking back, KubeSpray was the best option at the time, and there were no significant changes I would make to the decision.

Bart: You've worked on various projects involving bare metal and private clouds. Can you share more about your Kubernetes experience, such as the scale of clusters and

nodes?

Pierre: At Criteo, I led a NoSQL team supporting several million requests per second on a massive 4,500-node bare-metal cluster. Managing this infrastructure particularly node failures and data consistency across stateful databases like Cassandra, Couchbase, and Elasticsearch was a constant challenge.

While at Criteo, I also had a personal project where I built a smaller 10-node bare-metal cluster.

This experience with bare metal management solidified my belief in the benefits of Kubernetes, which I later implemented at Criteo.

When we adopted Kubernetes at Criteo, we encountered initial hurdles. In 2018, Kubernetes operators were still new, and there was internal competition from Mesos. We addressed these challenges by validating Kubernetes performance for our specific needs and building custom Chef recipes, StatefulSet hooks, and startup scripts.

Migrating to Kubernetes took eight months of dedicated effort. It was a complex process, but it was worth it.

Bart: As you've mentioned, Kubernetes had competitors in 2018 and continues to do so today. Despite the tooling's immaturity, you led a team to adopt Kubernetes for stateful workloads, which was unconventional. How did you guide your team through this transition?

Pierre: We had large instances — all between 50 and

100 CPUs each and 256 gigabytes of RAM up to 500 gigabytes of RAM.

We had multiple Cassandra clusters on a single Kubernetes cluster, and each Kubernetes node was dedicated to a single Cassandra node. We chose this bare metal setup to optimize disk access with SSD or NVMe.

Running these stateful workloads wasn't just a matter of starting them up. We had to handle them carefully because stateful sets like Elasticsearch and Cassandra must keep their data safe even if the machine they're running on fails.

Kubernetes helped us detect issues with these apps using features like Pod Disruption Budgets (PDBs) that limit how often pods can be disrupted, StatefulSets that have consistent ordering of execution and stable storage, and automated probes that trigger actions and alerts when something goes wrong.

Bart: Your experiences helped me better understand your blog post, The Cost of Upgrading Hundreds of Kubernetes Clusters. After managing large infrastructures, you founded Qovery. What drove you to take this step as an engineer?

Pierre: Kubernetes has become a standard, but managing it can be a headache for developers. Cloud providers offer a basic Kubernetes setup, but it often needs more features developers need to get started and

deploy applications quickly. Managing the cluster and nodes and keeping them up-to-date is time-consuming. Developers must spend a lot of time adding extra tools and configurations on top of the basic setup and then updating everything, which can be time-consuming.

To tackle these challenges, I founded Qovery.

Qovery provides two critical solutions. First, it offers a unified, user-friendly stack across cloud providers, simplifying Kubernetes deployment and management complexity. Second, it enables developers to deploy code without hassle.

Bart: Managing clusters can have various interpretations. The term can be broad. How do you define cluster management at Qovery in the context of upgrading and recovery?

Pierre: Yes, that's right. At Qovery, we understand the complexity of managing Kubernetes for customers. That's why we automate and simplify the entire process.

We automatically notify you about upcoming Kubernetes updates and handle the upgrade process on schedule, eliminating the need for manual intervention.

We deploy and manage various essential charts for your environment, including tools for logging, metrics collection, and certificate management. You don't need to worry about these intricacies.

We deploy all the necessary infrastructure elements to

create a fully functional Kubernetes environment for production within 30 minutes. We provide a complete solution that's ready to go.

We build your container images, push them to a registry, and deploy them based on your preferences. We also handle the lifecycle of the applications deployed.

We use Cluster Autoscaler to automatically adjust the number of nodes (cluster size) based on your actual usage to ensure efficiency. Additionally, we deploy Vertical and Horizontal Pod Autoscalers to scale your applications' resources as their needs change automatically.

By taking care of these complexities, Qovery frees your developers to focus solely on what matters most: building incredible applications.

Bart: How large is your team of engineers?

Pierre: We have ten engineers working on the project.

Bart: How do you manage hundreds of clusters with such a small team?

Pierre: We run various tests on each code change, including unit tests for individual components and end-to-end tests that simulate real-world usage. These tests cover configurations and deployment scenarios to catch potential issues early on.

Before deploying a new cluster for a customer, we put it through its paces on our internal systems for weeks.

Then, we deploy it to a separate non-production environment where we closely monitor its performance and address any problems before it reaches your applications.

We closely monitor Kubernetes and cloud providers' updates by following official changelogs and using RSS feeds, allowing us to anticipate potential issues and adapt our infrastructure proactively.

We also leverage tools like Kubent, popeye, kdave, and Pluto to help us manage API deprecations (when Kubernetes deprecates features in updates) and ensure the overall health of our infrastructure.

Our multi-layered approach has proven successful. We haven't encountered any significant problems when deploying clusters to production environments.

Bart: Managing new releases in the Kubernetes ecosystem can be daunting, especially with the extensive changelog. How do you navigate this complexity and spot potential difficulties when a new release is on the horizon?

Pierre: While reading the official update changelogs from Kubernetes and cloud providers is our first step, there are other paths to smooth sailing. Furthermore, understanding these detailed technical documents can be challenging, especially for newer team members who don’t have prior on-premise Kubernetes experience.

Cloud providers typically offer well-defined upgrade processes and document significant changes like removed functionalities, changes in API behavior, or security updates in their changelogs. However, many elements are interconnected in a Kubernetes cluster, especially when you deploy multiple charts for components like logging, observability, and ingress. Even with automated tools, we still need extensive testing and a manual process to ensure everything functions smoothly after an update.

Bart: So, what is your upgrading plan for helm charts?

Pierre: Upgrading Helm charts can be tricky because they bundle both the deployment and the software; for example, upgrading the Loki chart also upgrades Loki itself. To better understand what's changing, you need to review two changelogs: one for the chart itself and another for the software it includes.

We keep a close eye on all the charts we use by storing them in a central repository. This way, we have a clear history of every version we've used. We use a tool called `helm-freeze` to lock down the specific version of each chart we want to use. We can also track changes between chart and software versions using the `git diff` command.

If needed, we can also adjust specific settings within the chart using values override.

Like any other code change, we thoroughly test the

upgraded charts with unit and functional tests to ensure everything works as expected.

Once testing is complete, we route the updated charts to our test cluster for a final round of real-world testing. After a few days of monitoring, if everything looks good, we confidently release the updates to our customers.

Bart: How do you handle unexpected situations? Do you have a specific strategy or write more automation in the Helm charts?

Pierre: We're excited to see more community Helm charts, including built-in tests! This practice will make it easier for everyone to trust and use these charts in the future.

At Qovery, we enable specific Helm options by default, like 'atomic' and 'wait,' which help prevent upgrade failures during the process. However, there can still be issues that only show up in the logs, so we run additional tests specifically designed to catch these hidden problems.

Upgrading charts that deploy Custom Resource Definitions (CRDs) requires special attention. We've automated this process to upgrade the CRDs first (to the required version) and then upgrade the chart itself. Additionally, for critical upgrades like cert-manager (which manages certificates), we back up and restore

resources before applying the upgrade to avoid losing any critical certificates.

If you’re running an older version of a non-critical tool like a logging system, upgrading through each minor version one by one can be time-consuming. We have a better way! Our system allows you to skip to the desired newer version, bypassing all those intermediate updates.

We've also built safeguards into our system to handle potential problems before they occur during cluster upgrades. For example, the system checks for issues like failed jobs, incorrect Pod Disruption Budgets configuration, or ongoing processes that might block the upgrade. If it detects any problems, our engine automatically attempts to fix or clean up the issue. It will also warn you if any manual intervention is needed.

Our ultimate goal is to automate the upgrade process as much as possible.

Bart: Would you say CRDs are your favorite feature in Kubernetes, or do you have another one?

Pierre: CRDs are a powerful tool for customizing Kubernetes, offering a high degree of flexibility. However, the current support and tooling around them leave room for improvement. For example, enhancing Helm with better CRD management capabilities would significantly improve the user experience.

Despite these limitations, the potential of CRDs for

customizing Kubernetes is undeniable, making them a genuinely standout feature.

Bart: With your vast Kubernetes experience since 2016, how does your current process scale beyond 100 clusters? What do you need for such scalability?

Pierre: While basic application metrics can provide a general sense of health, managing hundreds of clusters requires more in-depth testing. Here at Qovery, with our experience handling nearly 300 clusters, we've found that:

More than basic metrics are needed. We need comprehensive testing that leverages application-specific metrics to ensure everything functions as expected.

Scaling requires more granular control over deployments, such as halting failures and providing detailed information to our users. For instance, quota issues from the cloud provider might necessitate user intervention.

Drawing from my experience at Criteo, where robust tooling was essential for managing complex tasks, powerful tools are the key to effectively scaling beyond 100 clusters.

Bart: Looking ahead at Qovery's roadmap, what's next for your team?

Pierre: Qovery will add Google Cloud Platform (GCP) by year-end, joining AWS and Scaleway! This expansion

gives you more choices for your cloud needs.

We're extracting reusable code sections, like those related to Helm integration, and transforming them into dedicated libraries. By making these functionalities available as open-source libraries, we empower the developer community to leverage them in their projects.

We strongly believe in Rust as a powerful language for building production-grade software, especially for systems like ours that run alongside Kubernetes.

We're also developing a service catalog feature that offers a user-friendly interface and streamlines complex deployments. This feature will allow users to focus on their applications, not the intricacies of the underlying technology.

Bart: Do you have any plans to include Azure?

Pierre: Yes, we have, but integrating a new cloud provider, given our current team size, is challenging. While we are a team of seniors, each cloud provider has nuances; some are more mature or resource-extensive than others.

Today, our focus is on AWS and GCP, as our customers most request. However, we're also working on a more modular approach that will allow Qovery to be deployed on any Kubernetes cluster, irrespective of the cloud provider, although this is still in progress.

Bart: We're looking forward to hearing more about that.

So, with your black belt in karate, how does that experience influence how you approach challenges, breaking them down into manageable steps?

Pierre: Karate has taught me the importance of discipline, focus, and breaking down complex tasks into manageable steps. Like in karate, where each move is deliberate and precise, I apply the same approach to challenges in my work, breaking them down into smaller, achievable goals.

Karate has also instilled in me a sense of perseverance and resilience, which are invaluable when facing difficult situations.

Bart: I'm a huge martial arts fan. How do you see martial arts' influence on managing stress in challenging situations?

Pierre: It varies from person to person. My experience in the banking industry has shown me that while some can handle stressful situations, others struggle. Martial arts can help manage stress somewhat, depending on the person.

Bart: How has your 25-year journey in karate shaped your perspective?

Pierre: Karate has become a part of me, and I plan to continue as long as possible.

Bart: What's the best way to reach out to you?

Pierre: You can reach me on LinkedIn or via email. I'm

always happy to help.

Chapter 2

The Magic of Ingress

Clusters are Cattle Until you Deploy Ingress, Dan Garfield

Managing repeatable infrastructure is the bedrock of efficient Kubernetes operations. While the ideal is to have easily replaceable clusters, reality often dictates a more nuanced approach. Dan Garfield, Co-founder of Codefresh, briefly captures this with the analogy: "A Kubernetes cluster is treated as disposable until you deploy ingress, and then it becomes a pet."

Dan Garfield joined Bart Farrell to understand how he managed Kubernetes clusters, transforming them from "cattle" to "pets" weaving in fascinating anecdotes about fairy tales, crypto, and snowboarding.

You can watch (or listen) to this interview here.

Bart: What are your top three must-have tools starting with a fresh Kubernetes cluster?

Dan: Argo CD is the first tool I install. For AWS, I will add Karpenter to manage costs. I will also use Longhorn for on-prem storage solutions, though I'd need ingress. Depending on the situation, I will install Argo CD first and then one of those other two.

Bart: Many of our recent podcast guests have highlighted Argo or Flux, emphasizing their significance

in the GitOps domain. Why do you think these tools are considered indispensable?

Dan: The entire deployment workflow for Kubernetes revolves around Argo CD. When I set up a cluster, some might default to using `kubectl apply`, or if they're using Terraform, they might opt for the Helm provider to install various Helm charts. However, with Argo CD, I have precise control over deployment processes.

Typically, the bootstrap pattern involves using Terraform to set up the cluster and Helm provider to install Argo CD and predefined repositories. From there, Argo CD takes care of the rest.

I have my Kubernetes cluster displayed on the screen behind me, running Argo CD for those who can't see. I utilize Argo CD autopilot, which streamlines repository setup. Last year, when my system was compromised, Argo CD autopilot swiftly restored everything. It's incredibly convenient. Moreover, when debugging, the ability to quickly toggle sync, reset applications, and access logs through the UI is invaluable. Argo CD is, without a doubt, my go-to tool for Kubernetes. Admittedly, I'm biased as an Argo maintainer, but it's hard to argue with its effectiveness.

Bart: Our numerous podcast discussions with seasoned professionals show that GitOps has been a recurring theme in about 90% of our conversations. Almost every

guest we've interviewed has emphasized its importance, often mentioning it as their primary tool alongside other essentials like cert manager, Kyverno, or OPA, depending on their preferences.

Could you introduce yourself to those unfamiliar with you? Tell us your background, work, and where you're currently employed.

Dan: I'm Dan Garfield, the co-founder and chief open-source officer at CodeFresh. As Argo maintainers, we're deeply involved in shaping the GitOps landscape. I've played a key role in creating the GitOps standard, establishing the GitOps working group, and spearheading the OpenGitOps project.

Our journey began seven years ago when we launched CodeFresh to enhance software delivery in the cloud-native ecosystem, primarily focusing on Kubernetes. Alongside my responsibilities at CodeFresh, I actively contribute to SIG security within the Kubernetes community and oversee community-driven events like ArgoCon. Outside of work, I reside in Salt Lake City, where I indulge in my passion for snowboarding. Oh, and I'm a proud father of four, eagerly awaiting the arrival of our fifth child.

Bart: It’s a fantastic journey. We'll have to catch up during KubeCon in Salt Lake City later this year. Before delving into your entrepreneurial venture, could you

share how you entered Cloud Native?

Dan: My journey into the tech world began early on as a programmer. However, I found myself gravitating more towards the business side, where I discovered my knack for marketing. My pivotal experience was leading enterprise marketing at Atlassian during the release of Data Center, Atlassian's clustered tool version. Initially, it didn't garner much attention internally, but it soon became a game-changer, driving significant revenue for the company. Witnessing this transformation, including Atlassian's public offering, was exhilarating, although my direct contribution was modest as I spent less than two years there.

I noticed a significant change in containerization, which sparked my interest in taking on a new challenge. Conversations with friends starting container-focused experiences captivated me. Then, Raziel, the founder of Codefresh, reached out, sharing his vision for container-driven software development. His perspective resonated deeply, prompting me to join the venture.

Codefresh initially prioritized building robust CI tools, recognizing that effective CD hinges on solid CI practices and needed to be improved in many organizations at the time (and possibly still is). As we expanded, we delved into CD and explored ways to leverage Kubernetes insights.

Kubernetes had yet to emerge as the dominant force when we launched this journey. We evaluated competitors like Rancher, OpenShift, Mesosphere, and Docker Swarm. However, after thorough analysis, Kubernetes emerged as the frontrunner, boldly cueing us to bet on its potential.

Our decision proved visionary as other platforms gradually transitioned towards Kubernetes. Amazon's launch of EKS validated our foresight. This strategic alignment with Kubernetes paved the way for our deep dive into GitOps and Argo CD, driving the project's growth within the CNCF and its eventual graduation.

Bart: It's impressive how much you've accomplished in such a short timeframe, especially while balancing family life. With the industry evolving rapidly, How do you keep up with the cloud-native scene as a maintainer and a co-founder?

Dan: Indeed, staying updated involves reading blogs, scrolling through Twitter, and tuning into podcasts. However, I've found that my most insightful learnings come from direct conversations with individuals. For instance, I've assisted the community with Argo implementations, not as a sales pitch but to help gather insights genuinely. Interacting with Codefresh users and engaging with the broader community provides invaluable perspectives on adoption challenges and user

needs.

Oddly enough, sometimes, the best way to learn is by putting forth incorrect opinions or questions. Recently, while wrestling with AI project complexities, I pondered aloud whether all Docker images with AI models would inevitably be bulky due to PyTorch dependencies. To my surprise, this sparked many helpful responses, offering insights into optimizing image sizes. Being willing to be wrong opens up avenues for rapid learning.

Bart: That vulnerability can indeed produce rich learning experiences. It's a valuable practice. Shifting gears slightly, if you could offer one piece of career advice to your younger self, what would it be?

Dan: Firstly, embrace a mindset of rapid learning and humility. Be more open to being wrong and detach ego from ideas. While standing firm on important matters is essential, recognize that failure and adaptation are part of the journey. Like a stone rolling down a mountain, each collision smooths out the sharp edges, leading to growth.

Secondly, prioritize hiring decisions. The people you bring into your business shape its trajectory more than any other factor. A wrong hire can have far-reaching consequences beyond their salary. Despite some missteps, I've been fortunate to work with exceptional individuals who contribute immensely to our success. When considering a job opportunity, I always emphasize

the people's quality, the mission's significance, and fair compensation. Prioritizing in this order ensures fulfillment and satisfaction in your career journey.

Bart: That's insightful advice, especially about hiring. Surrounding yourself with talented individuals can make all the difference in navigating business challenges. Now, shifting gears to your recent tweet about Kubernetes and Ingress, who was the intended audience for that tweet?

Dan: Honestly, it was more of a reflection for myself, perhaps shouted into the void. I was weighing the significance of deploying Ingress within Kubernetes. In engineering, a saying that "the problem is always DNS" suggests that your cluster becomes more tangible once you configure DNS settings. Similarly, setting up Ingress signifies a shift in how you perceive and manage your cluster. Without Ingress, it might be considered disposable, like a development environment. However, once Ingress is in place, your cluster hosts services that require more attention and care.

Bart: For those unfamiliar with the "cattle versus pets" analogy in Kubernetes, could you elaborate on its relevance, particularly in the context of Ingress?

Dan: While potentially controversial, the "cattle versus pets" analogy illustrates a fundamental concept in managing infrastructure. In this analogy, cattle represent

interchangeable and disposable resources, much like livestock in a ranching operation. Conversely, pets are unique, loved entities requiring personalized care.

In Kubernetes, deploying resources as "cattle" means treating them as replaceable, identical units. However, Ingress introduces a shift towards a "pet" model, where individual services become distinct and valuable entities. Just as you wouldn't name every cow on a farm, you typically wouldn't concern yourself with the specific details of each interchangeable resource. But once you start deploying services accessible via Ingress, each service becomes unique and worthy of individual attention, akin to caring for a pet.

Bart: It seems the "cattle versus pets" analogy is stirring some controversy among vegans, which is understandable given its context. How does this analogy relate to Kubernetes and Ingress?

Dan: In software, the analogy helps distinguish between disposable, interchangeable components (cattle) and unique, loved entities (pets). For instance, in my Kubernetes cluster, the individual nodes are like cattle—replaceable and without specific significance. If one node malfunctions, I can easily swap it out without concern.

However, once I deploy Ingress and start hosting services, the cluster takes on a different role. While the

individual nodes remain disposable, the cluster becomes more akin to a pet. I care about its state, its configuration, and its uptime. Suddenly, I'm monitoring metrics and ensuring its well-being, similar to caring for a pet's health.

So, the analogy underscores the shift in perception and care that occurs when transitioning from managing generic infrastructure to hosting meaningful services accessible via Ingress.

Bart: That's a fascinating perspective. How do Kubernetes and Ingress relate to all of this?

Dan: The ingress in Kubernetes is a central resource for managing incoming traffic to the cluster and routing it to different services. However, unlike other resources in Kubernetes, such as those managed by Argo CD, the ingress is often shared among multiple applications. Each application may have its own deployment rules, allowing for granular control over updates and configurations. For example, one application might only update when manually triggered, while another automatically updates when changes are detected.

The challenge arises because updating Ingress impacts multiple applications simultaneously. Through this centralized routing mechanism, you're essentially juggling the needs of various applications. This complexity underscores the importance of managing the

cluster effectively, as each change to Ingress affects the entire ecosystem of applications.

The Argo CD community is discussing introducing delegated server-side field permissions. This feature would allow one application to modify components of another, easing the burden of managing shared resources like Ingress. However, it's still under debate, and alternative solutions may emerge. Other tools, like Contour, offer a different approach by treating each route as a separate custom resource, allowing applications to manage their routing independently.

Ultimately, deploying the ingress marks a shift in the cluster's dynamics, requiring considerations such as DNS settings and centralized routing configurations. As a result, the cluster becomes more specialized and less disposable as its configuration becomes bespoke to accommodate the routing needs of various applications.

Bart: Any recommendations for those who aim to keep their infrastructure reproducible while needing Ingress?

Dan: One approach is abstraction and leveraging wildcards. While technically, you can deploy an Ingress without external pointing; I prefer the concept of self-updating components. Tools like Crossplane or Google Cloud's Config Connector allow you to represent non-Kubernetes resources as Kubernetes objects. Incorporating such tools into your cluster bootstrap

process ensures the dynamic creation of necessary components.

However, there's a caveat. Despite reproducible clusters, external components like DNS settings may not be. Updating name servers remains a manual task. It's a tricky aspect of operations that needs a perfect solution.

Bart: How do GitOps and Argo CD fit into solving this challenge?

Dan: GitOps and Argo CD play a crucial role in managing complex infrastructure, especially with sensitive data. The key lies in representing all infrastructure resources, including secrets and certificates, as Kubernetes objects. This approach enables Argo CD to track and reconcile them, ensuring that the desired state defined in Git reflects accurately in your cluster.

Tools like Crossplane, vCluster (for managing multiple clusters), or Cluster API (for provisioning additional clusters) can extend this approach to handle various infrastructure resources beyond Kubernetes. Essentially, Git serves as the single source of truth for your entire infrastructure, with Argo CD functioning as the engine to enforce that truth.

A common issue with Terraform is that its state can get corrupted easily because it must constantly monitor changes. Crossplane often uses Terraform under the

hood. The problem is not with Terraform's primitives but with the data store and its maintenance. Crossplane ensures the data store remains uncorrupted, accurately reflecting the current state. If changes occur, they appear as out of sync in Argo CD.

You can then define policies for reconciliation and updates, guiding the controller on the next steps. This approach is crucial for managing infrastructure effectively. Using etcd as your data store is an excellent pattern and likely the future of infrastructure management.

Bart: What would happen if the challenges of managing Kubernetes infrastructure extend beyond handling ingress traffic to managing sensitive information like state secrets and certificates? This added complexity could lead to a "pet" cluster scenario. Would you think backup and recovery tools like Velero would be easier to use without these additional challenges?

Dan: I need to familiarize myself with Velero. Can you tell me about it?

Bart: Velero is a tool focused on backing up and restoring Kubernetes resources. Since you mentioned Argo CD and custom resources earlier, I'm curious about your approach to backing up persistent volumes. How did you manage disaster recovery in your home lab when everything went haywire?

Dan: I've used Longhorn for volume restoration, and clear protocols were in place. I'm currently exploring Velero, which looks like a promising tool for data migration.

Managing data involves complexities like caring for a pet, requiring careful handling and migration. Many people need help managing stateful workloads in Kubernetes. Fortunately, most of my stateful workloads in Kubernetes can rebuild their databases if data is lost. Therefore, data loss is manageable for me. Most of the elements I work with are replicable. Any items needing persistence between sessions are stored in Git or a versioned, immutable secret repository.

Bart: It's worth noting, especially considering what happened with your home lab. Should small startups prioritize treating their clusters like cattle, or is ClickOps sufficient?

Dan: It depends on the use cases. vCluster, a project I'm fond of, is particularly well-suited for creating disposable development clusters, providing developers with isolated sandboxes for testing and experimentation. It allows deploying a virtualized cluster on an existing Kubernetes setup, which saves significantly on ingress costs, especially on platforms like AWS, where you can consolidate ingress into one.

Another example is using Argo CD's application sets to

create full-stack environments for each pull request in a Git repository. These environments, which include a virtual cluster, are unique to each pull request but remain completely disposable and easily recreated, much like cattle.

However, managing ingress for disposable clusters can be challenging. When deployed and applied to vClusters, ingress needs custom configurations, requiring separate tracking and maintenance. Despite this, it's still beneficial to prioritize treating infrastructure as disposable. For example, while my on-site Kubernetes cluster is a "pet" that requires careful maintenance, its nodes are considered "cattle" that can be replaced or reconfigured without disrupting overall operations. This abstraction is a core principle of Kubernetes and allows for greater flexibility and resilience.

By abstracting clusters away from custom configurations and focusing on reproducibility, you can treat them more like cattle, even if they have some pet-like qualities due to ingress deployment and DNS configurations. This commoditization of clusters simplifies management and enables greater scalability. The more you abstract and standardize your infrastructure, the smoother your operations will become. And to be clear, this analogy has nothing to do with dietary choices.

Bart: If you could rewind time and change anything,

what scenario would you create to avoid writing that tweet?

Dan: We've been discussing a feature in Argo CD that allows for delegated field permissions to happen server-side. It addresses a problem inherent in Kubernetes architecture, particularly regarding ingress. The current setup doesn't allow for external delegation of its components, even though many users operate it that way. If I could make changes, I might have split ingress into an additional resource, including routes as a separate definition that users could manage independently.

Exploring other scenarios where delegated field permissions would be helpful is crucial. Ingress is the most obvious example, highlighting an area for potential improvement. Creating separate routes and resources could solve this issue without altering Argo CD. This approach, similar to Contour's, could be a promising solution. Contour's separate resource strategy demonstrates learning from Ingress and making improvements. We should consider adopting tools like Contour or other service mesh ingress providers, as several compelling options are available.

Bart: If you had to build a cluster from scratch today, how would you address these issues whenever possible?

Dan: Sometimes you just have to accept the challenge and not try to work around it. Setting up ingress and

configuring DNS for a single cluster might not be a big deal, but it's worth considering a re-architecture if you're doing it on a large scale, like 250,000 times. For instance, with Codefresh, many users opt for our hybrid setup. They deploy our GitOps agent, based on Argo CD, on their cluster, which then connects to our control plane.

One of the perks we offer is a hosted ingress. Instead of setting up ingresses for each of their 5000 Argo CD instances, users can leverage our hosted ingress, saving money and configuration headaches. Consider alternatives like a tunneling system instead of custom ingress setups, depending on your use case. A hosted ingress can be a game-changer for large-scale distributed setups like multiple Argo CD instances, saving costs and simplifying configurations. Ultimately, re-architecting is always an option tailored to what works best for you.

Bart: We're nearing the end of the podcast and want to touch on a closing question, which we are looking at from a few different angles. How do you deal with the anxiety of adopting a new tool or practice, only to find out later that it might be wrong?

Dan: I've seen this dynamic play out. Sometimes, organizations invest heavily in a tool, only to realize a few years later that there are better fits. Take the example of a company transitioning to Argo workflows for CICD

and deployment, only to discover that Argo CD would have been a better fit for most of their use cases. However, these transitions are well-spent efforts. In their case, the journey through Argo workflows paved the way for a smoother transition to Argo CD. Sometimes, detaching the wrong direction is necessary to reach the correct destination faster.

You can only sometimes foresee the ideal solution from where you are, and experimenting with different tools is part of the learning process. It's essential not to dwell on mistakes but to learn from them and move forward. After all, even if a tool ultimately proves to be the wrong choice, it often still brings value. The key is recognizing when a change is needed and adapting accordingly. Mistakes only become fatal if we fail to acknowledge and learn from them.

Bart: We stumbled upon your blog, Today Was Awesome, which hasn't seen an update in a while. You wrote a post about Bitcoin, priced at around $450 in 2015. Are you a crypto millionaire now?

Dan: Not quite! Crypto is a fascinating topic, often sparking wild debates. While there's no shortage of scams in the crypto world, there's also genuine innovation happening. I dabbled in Bitcoin early on and even mined a bit to understand its potential use cases better. One notable experience was mentoring at Hack

the North, a massive hackathon where numerous projects leveraged Ethereum. I strategically sold my Bitcoin for Ethereum, which turned out well. However, I'm still waiting on those Lambos—I'm not quite at millionaire status yet!

Bart: Your blog covers many topics, including one post titled "What are we really supposed to learn from fairy tales." How did you decide on such diverse content?

Dan: I can't recall the exact inspiration, but my wife and I often joke about how outdated the moral lessons in fairy tales feel. Exploring their relevance in today's world is an interesting angle to explore.

Bart: What's next for you? More fairy tales, moon-bound Lamborghinis, or snowboarding adventures? Also, let's discuss your recent tweet about making your bacon. How did that start?

Dan: Ah, yes, making bacon! It's surprisingly simple. First, you get pork belly and cure it in the fridge for seven to ten days. Then, you smoke it for a couple of hours.

My primary motivation was to avoid the nitrates found in store-bought bacon linked to health issues. Homemade bacon tastes better, is of higher quality, and is cheaper. My freezer now overflows with homemade bacon, which makes for a unique and well-received gift. People love the taste; overall, it's been a rewarding and delicious

effort!

Bart: Regardless of dietary choices, considering where your food comes from and being involved in the process—whether by growing your food or making it yourself and turning it into a gift for others—creates a different, enriching experience. What's next for you?

Dan: This year, my focus is on environment management and promotion. In the Kubernetes world, we often think about applications, clusters, and instances of Argo CD to manage everything. We're working on a paradigm shift: we think about products instead of applications. In our context, a product is an application in every environment in which it exists. Hence, if you deploy a development application, move it to stage, and finally to production, you're deploying the same application with variations three times. That's what we call a product. We're shifting from thinking about where an application lives to considering its entire life cycle. Instead of focusing on clusters, we think about environments because an environment might have many clusters.

For instance, retail companies like Starbucks, Chick-fil-A, and Pizza Hut often have Kubernetes clusters on-site. Deploying to US West might mean deploying to 1,300 different clusters and 1,300 different Argo CD instances. We abstract all that complexity by grouping them into

the environments bucket. We focus on helping people scale up and build their workflow using environments and establishing these relationships. The feedback has been incredible; people are amazed by what we’re demonstrating.

We're showcasing this at ArgoCon next month in Paris. After that, I plan to do some snowboarding and then make it back in time for the birth of my fifth child.

Bart: That's a big plan. 2024 is packed for you. If people want to contact you, what's the best way to do it?

Dan: Twitter is probably the best. You can find me at @todaywasawesome. If you visit my blog and leave comments, I won't see them, as it's more of an archive now. I keep it around because I worked on it ten years ago and occasionally reference something I wrote.

You can also reach out on LinkedIn, GitHub, or Slack. I respond slower on Slack, but I do get to it eventually.

Chapter 3

eBPF and the Future of Service Meshes

eBPF, Sidecars, and the Future of the Service Mesh, William Morgan

Kubernetes and service meshes may seem complex, but not for William Morgan, an engineer-turned-CEO who excels at simplifying the intricacies. In this enlightening podcast, he shares his journey from AI to the cloud-native world with Bart Farrell.

Discover William's cost-saving strategies for service meshes, gain insights into the ongoing debate between sidecars, Ambient Mesh, and Cilium Cluster Mesh, his surprising connection to Twitter's early days and unique perspective on balancing tech expertise with the humility of being a piano student.

You can watch (or listen) to this interview here.

Bart: Imagine you've just set up a fresh Kubernetes cluster. What's your go-to trio for the first tools to install?

William: My first pick would be Linkerd. It's a must-have for any Kubernetes cluster. I then lean towards tools that complement Linkerd, like Argo and cert-manager. You're off to a solid start with these three.

Bart: Cert Manager and Argo are popular choices, especially in the GitOps domain. What about Flux?

William: Flux would work just fine. I don't have a strong preference between the two. Flux and Argo are great options, especially for tasks like progressive delivery. When paired with Linkerd, they provide a robust safety net for rolling out new code.

Bart: As the CEO, who are you accountable to? Could you elaborate on your role and responsibilities?

William: Being a CEO is an exciting shift from my previous role as an engineer. I work for myself, and I must say, I’m a demanding boss. As a CEO, I focus on the big picture and align everyone toward a common goal. These are the two skills I’ve had to develop rapidly since transitioning from an engineer, where my primary concern was writing and maintaining code.

Bart: From a technical perspective, how did you transition into the cloud-native space? What were you doing before it became mainstream?

William: My early career was primarily focused on AI, NLP, and machine learning long before they became trendy. I thought I’d enter academia but realized I enjoyed coding more than research.

I worked at several Bay Area startups, mainly in NLP and machine learning roles. I was part of a company called PowerSet, which was building a natural language processing engine and was acquired by Microsoft. I then joined Twitter in its early days, around 2010, when it had

about 200 employees. I started on the AI side but transitioned to infrastructure because I found it more satisfying and challenging. We were doing what I now describe at Twitter as cloud-native, even though the terminology differed. We didn’t have Kubernetes or Docker, but we had Mesos, the JVM for isolation, and cgroups for a basic form of containerization. We transitioned from a monolithic Ruby on Rails service to a massive microservices deployment. When I left Twitter, we tried to apply those same ideas to the emerging world of Kubernetes and Docker.

Bart: How do you keep up with the rapid changes in the Kubernetes and cloud-native ecosystems, especially transitioning from infrastructure and AI/NLP?

William: My current role primarily shapes my strategy. I learn a lot from the engineers and users of Linkerd, who are at the forefront of these technologies. I also keep myself updated by reading discussions on Reddit platforms like r/kubernetes and r/Linkerd. Occasionally, I contribute to or follow discussions on Hacker News. Overall, my primary source of knowledge comes from the experts I work with daily, giving me valuable insights into the latest developments.

Bart: If you could return to your time at Twitter or even before that, what one tip would you give yourself?

William: I'd tell myself to prioritize impact. As an

engineer, I was obsessed with building and exploring new technologies, which was rewarding. However, I later understood the value of stepping back to see where I could make a real difference in the company. Transitioning my focus to high-impact areas, such as infrastructure at Twitter, was a turning point. Despite my passion for NLP, I realized that infrastructure was where I could truly shine. Always look for opportunities where you can make the most significant impact.

Bart: Let’s focus on "Sidecarless eBPF Service Mesh Sparks Debate," which follows up on your previous article “eBPF, sidecars, and the future of the service mesh.” You're one of the creators of Linkerd. For those unfamiliar, what exactly is a service mesh? Why would someone need it, and what value does it add?

William: There are two ways to describe service mesh: what it does and how it works. Service mesh is an additional layer for Kubernetes that enhances key areas Kubernetes doesn't fully address.

The first area is security. It ensures all connections in your cluster are encrypted, authorized, and authenticated. You can set policies based on services, gRPC methods, or HTTP routes, like allowing Service A to talk to /foo but not /bar.

The second area is reliability. It enables graceful failovers, transparent traffic shifting between clusters,

and progressive delivery. For example, deploying new code and gradually increasing traffic to it to avoid immediate production traffic. It also includes mechanisms like load balancing, circuit breaking, retries, and timeouts.

The last area is observability. It provides uniform metrics for all workloads across all services, such as success rates, latency distribution, and traffic volume. Importantly, it does this without requiring changes to your application code.

The most prevalent method today involves using many proxies. This approach has become feasible thanks to technological advancements like Kubernetes and containers, which simplify the deployment and management of many proxies as a unified fleet. A decade ago, deploying 10,000 proxies would have been absurd, but it is feasible and practical today. The specifics of deploying these proxies, their locations, programming languages, and practices are subject to debate. However, at a high level, service meshes work by running these layer seven proxies that understand HTTP, HTTP2, and gRPC traffic and enable various functionalities without requiring changes to your application code.

Bart: Can you briefly explain how the data and control planes work in service meshes, especially compared to the older sidecar model with an extra container?

William: A service mesh architecture consists of two main components: a control plane and a data plane. The control plane allows you to manage and configure the data plane, which directs network traffic within the service mesh. In Kubernetes, the control plane operates as a collection of standard Kubernetes services, typically running within a dedicated namespace or across the entire cluster.

The data plane is the operational core of a service mesh, where proxies manage network traffic. The sidecar model, employed by service meshes like Linkerd, deploys a dedicated proxy alongside each application pod. Therefore, a service mesh with 20 pods would have 20 corresponding proxies. The overall efficiency and scalability of the service mesh rely heavily on the size and performance of these individual proxies.

In the sidecar model, service A and service B communication flows through service A's and service B's proxy. Service A sends its message to its sidecar proxy, and then the service A proxy forwards it to service B's sidecar proxy. Finally, service B's proxy delivers the message to service B itself. This indirect communication path adds extra hops, leading to a slight increase in latency. You must carefully consider the potential performance impacts to ensure that service mesh benefits outweigh the trade-offs.

Bart: We've been discussing the benefits of service meshes, but running an extra container for each pod sounds expensive. Does cost become a significant issue?

William: Service meshes have a compute cost, just like adding any component to a system. You pay for CPU and memory, but memory tends to be the more significant concern, as it can force you to scale up instances or nodes.

However, Linkerd has minimized this issue with a "micro proxy" written in Rust. Rust's strict memory management allows fast, lightweight proxies and avoids memory vulnerabilities like buffer overflows, which are common in C and C++. Studies from both Google and Microsoft have shown that roughly 70% of security bugs in C and C++ code are due to memory management errors.

Our choice of Rust as the programming language in 2018 was a calculated risk. Rust offers the best of both worlds: the speed and control of languages like C/C++ and the safety and ease of use of languages with runtime environments like Go. Rust and its network library ecosystem were still relatively young at that time. We invested significantly in underlying libraries like Tokio, Tower, and H2 to build the necessary infrastructure.

The critical role of the data plane in handling sensitive application data drove this decision. We ensured its

reliability and security. Rust enables us to build small, fast, and secure proxies that scale with traffic, typically using minimal memory, directly translating to the user experience. Instead of facing long response times (like 5-second tail latencies), users experience faster interactions (closer to 30 milliseconds). A service mesh can optimize these tail latencies, improving user experience and customer retention. Choosing Rust has proven to be instrumental in achieving these goals.

While cost is a factor, the actual cost often stems from operational complexity. Do you need dedicated engineers to maintain complex proxies, or does the system primarily work independently? That human cost usually dwarfs the computational one.

Our design choices have made managing Linkerd's costs relatively straightforward. However, for other service meshes, costs can escalate if the proxies are large and resource-intensive. Even so, the more significant cost is often not the resources but the operational overhead and complexity. This complexity can demand considerable time and expertise, increasing the overall cost.

Bart: You raise a crucial point about the human aspect. While we address technical challenges, the time spent resolving errors detracts from other tasks. The community has developed products and projects to tackle these concerns and costs. One such example is

Istio with Ambient Mesh. Another approach is sidecarless service meshes like Cilium Cluster Mesh. Can you explain what Ambient Mesh is and how it enhances the classic sidecar model of service meshes?

William: We've delved deep into both of these options in Linkerd. While there might come a time when adopting these projects makes sense for us, we're not there yet.

Every decision involves trade-offs regarding distributed systems, especially in production environments within companies where the platform is a tool to support applications. At Linkerd, our priority is constantly reducing the operational workload.

Ambient Mesh and eBPF aren't primarily reactions to complexity but responses to the practical annoyances of sidecars. Their key selling point is eliminating the need for sidecars. However, the real question is: What's the cost of this shift? That's where the analysis becomes crucial.

In Ambient Mesh, rather than having sidecar containers, you utilize connective components, such as tunnels, within the namespace. These tunnels communicate with proxies located elsewhere in the cluster. So essentially, you have multiple proxies running outside of the pod, and the pods use these tunnels to communicate with the proxies, which then handle specific tasks.

This setup is indeed intriguing. As mentioned earlier,

running sidecars can be challenging due to specific implications. One such implication is the cost factor, which we discussed earlier. In Linkerd’s case, this is a minor concern. However, a more significant implication is the need to restart the pod to upgrade the proxy to the latest version, given the immutability of pods in Kubernetes.

This situation necessitates managing two separate updates: one to keep the applications up-to-date and another to upgrade the service mesh. Therefore, while the setup has advantages, it also requires careful management to ensure smooth operation and optimal performance.

We operate the proxy as the first container for various reasons, which can lead to friction points, such as when using `kubectl logs`. Typically, when you request logs, you're interested in your application's logs, not the proxy's. This friction, combined with a desire for networking to operate seamlessly in the background, drives the development of solutions like Ambient and eBPF, which aim to eliminate the need for explicit sidecars.

Both Ambient and eBPF solutions, which are closely related, are reactions to this sentiment of not wanting to deal with sidecars directly. The aim is to make sidecars disappear. Take Istio and most service meshes built on

Envoy, for instance. Envoy is complex and memory-intensive and requires constant attention and tuning based on traffic specifics.

Challenges with sidecars are more of a cloud-native trend to market solutions, like writing a blog post proclaiming the death of sidecars rather than being specific to Linkerd. They can sometimes be an inaccurate reflection of the reality of engineering.

In Ambient, eliminating sidecars by running the proxy elsewhere and using tunnel components allows for separate proxy maintenance without needing to reboot applications for upgrades. However, in a Kubernetes environment, the idea is that pods should be rebootable anytime. Kubernetes can reschedule pods as needed, which aligns with the principles of building applications as distributed systems. Yet, there are legacy applications or specific scenarios where rebooting could be more convenient, making the sidecar approach less appealing.

Historically, running cron jobs with sidecar proxies in Kubernetes posed a significant challenge. Kubernetes lacked a built-in mechanism to signal the sidecar proxy when the main job was complete, necessitating manual intervention to prevent the proxy from running indefinitely. This manual process went against the core principle of service mesh, which aims to decouple services from their proxies for easier management and

scalability.

Thankfully, one significant development is the Sidecar Container Kubernetes Enhancement Proposal. With this enhancement, you can designate your proxy as a sidecar container, leading to several benefits, like jobs terminating the proxy once finished and eliminating unnecessary resource consumption.

For Linkerd, adopting Ambient mesh architecture introduces more complexity than benefits. The additional components, like the tunnel and separate proxies, add unnecessary layers to the system. Unlike Istio, which has encountered issues due to its architecture, Linkerd's existing design hasn't faced similar challenges. Therefore, the trade-offs associated with Ambient aren't justified for Linkerd.

In contrast, the sidecar model offers distinct advantages. It creates clear operational and security boundaries at the pod level. Each pod becomes a self-contained unit, making independent decisions regarding security and operations, aligning with Kubernetes principles, and simplifying management in a cloud-native environment.

This sidecar approach is crucial for implementing zero-trust security. The critical principle of zero trust is to enforce security policies at the most granular level possible. Traditional approaches relying on a perimeter firewall and implicitly trusting internal components are

no longer sufficient. Instead, each security decision must be made independently at every system layer. This granular enforcement is achieved by deploying a sidecar proxy within each application pod, acting as a security boundary and enabling fine-grained control over network traffic, authentication, and authorization.

In Linkerd, every request undergoes a rigorous security check within the pod. This check includes verifying the validity of the TLS encryption, confirming the client's identity through cryptographic algorithms, and ensuring the request comes from a trusted source. Additionally, Linkerd checks whether the request can access the specific resource or method it's trying to reach. This multi-layered scrutiny happens directly inside the pod, providing the highest possible level of security within the Kubernetes framework. Maintaining this tight security model is crucial, as any deviation, like separating the proxy and TLS certificate, weakens the model and introduces potential vulnerabilities.

Bart: The next point I'd like to discuss has garnered significant attention in recent years through Cilium Service Mesh and various domains. What is eBPF?

William: eBPF is a kernel technology that enables the execution of specific code within the kernel, offering significant advantages. Firstly, operations within the kernel are high-speed, eliminating the overhead of

context switching between kernel and user space. Secondly, the kernel has unrestricted access to all system resources, requiring robust security measures to ensure eBPF programs are safe. This powerful technology empowers developers to create highly efficient and secure solutions for various system tasks, particularly networking, security, and observability.

Traditionally, user-space programs lacked direct access to kernel resources, relying on system calls to communicate with the kernel. While providing security, this syscall boundary introduced cost overhead, especially with frequent requests like network packet processing.

eBPF revolutionized this by enabling user-defined code to run within the kernel with stringent safety measures. The number of instructions an eBPF program can execute is limited, and infinite loops are prohibited to prevent resource monopolization. The bytecode verifier meticulously ensures every possible execution path can be explored to avoid unexpected behavior or malicious activity. The bytecode is also verified for GPL compliance by checking for specific strings in its initial bytes.

These security measures make eBPF a powerful but restrictive mechanism, enabling previously unattainable capabilities. Understanding what eBPF can and cannot

do is crucial, despite marketing claims that might blur these lines. While many promote eBPF as a groundbreaking solution that could eliminate the need for sidecars, the reality is more nuanced. It's crucial to understand its limitations and not be swayed by marketing hype.

Bart: There appears to be some confusion regarding the extent of limitations associated with eBPF. If eBPF has limitations, does that imply that these limitations constrain all service meshes using eBPF?

William: The idea of an eBPF-based service mesh can sometimes need clarification. In reality, the Envoy proxy still handles the heavy lifting, even in these eBPF-powered meshes. eBPF has limitations, especially in the network space, and can't fully replace the functionality of a traditional proxy.

While eBPF has many applications, including security and performance monitoring, its most interesting potential lies in instrumenting applications. The kernel can directly measure CPU usage, function calls, and other performance metrics by residing in the kernel.

However, when it comes to networking, eBPF faces significant challenges. Maintaining large amounts of state, essential for many network operations, is difficult, bordering on impossible. This challenge highlights the limitations of eBPF in entirely replacing traditional

networking components like proxies.

The role of eBPF in networking, particularly within service meshes, is often overstated. While it excels in certain areas, like efficient TCP packet processing and simple metrics collection, other options exist beyond traditional proxies. Complex tasks like HTTP2 parsing, TLS handshakes, or layer seven routings are challenging, if possible, to implement purely with eBPF.

Some projects attempt complex eBPF implementations for these tasks but often involve convoluted workarounds that sacrifice performance and practicality. In practice, eBPF is typically used for layer 4 (transport layer) tasks, while user-space proxies like Envoy handle more complex layer 7 (application layer) operations.

Service meshes like Cilium, despite their claims of being sidecar-less, often rely on daemonset proxies to handle these complex tasks. While eliminating sidecars, this approach introduces its own set of problems. Security is compromised as TLS certificates are mixed in the proxy's memory, and operational challenges arise when the daemonset goes down, affecting seemingly random pods scheduled on that machine.

Linkerd, having experienced similar issues with its first version (Linkerd1.x) running as a daemonset, opted for the sidecar model in subsequent versions. Sidecars provide clear operational and security boundaries,

making management and troubleshooting easier.

Looking ahead, eBPF can still be a valuable tool for service meshes. Linkerd, for instance, could significantly speed up raw TCP proxying by offloading tasks to the kernel. However, for complex layer seven operations, a user-space proxy remains essential.

The decision to use eBPF and the choice between sidecars and daemonsets are distinct considerations, each with advantages and drawbacks. While eBPF offers powerful capabilities, it doesn't inherently dictate a specific proxy architecture. Choosing the most suitable approach requires careful evaluation of the system's requirements and trade-offs.

Bart: Can you share your predictions about conflict or uncertainty concerning service meshes and sidecars for the next few years? Is there a possibility of resolving this? Should we anticipate the emergence of new groups? What are your expectations for the near and distant future?

William: While innovation in this field is valuable, relying solely on marketing over technical analysis needs more appeal, especially for those prioritizing tangible customer benefits.

Regarding the future of service meshes, their value proposition is now well-established. The initial hype has given way to a practical understanding of their necessity,

with users selecting and implementing solutions without extensive deliberation. This maturity is a positive development, shifting the focus from explaining the need for a service mesh to optimizing its usage.

Functionally, service meshes converge on core features like MTLS, load balancing, and circuit breaking. However, a significant area of development and our primary focus is mesh expansion, which involves integrating non-Kubernetes components into the mesh. We have a big announcement regarding this in mid-February.

Bart: That sounds intriguing. Please give us a sneak peek into what this announcement is about.

William: It is about Linkerd 2.15! The release of Linkerd 2.15 is a significant step forward. It introduces the ability to run the data plane outside Kubernetes, enabling seamless TLS communication for both VM and pod workloads.

The industry mirrors this direction, as evidenced by developments like the Gateway API, which converges to handle both ingress and service mesh configuration within Kubernetes. This unified approach allows consistent configuration primitives for traffic entering, transiting, and exiting the cluster.

The industry will likely focus on refining details like eBPF integration or the advantages of Ambient Mesh

while the fundamental value proposition of service meshes remains consistent. I'm particularly excited about how these advancements can be applied across entire organizations, starting with securing and optimizing Kubernetes environments and extending these benefits to the broader infrastructure.

Bart: Shifting away from the professional side, we heard you have an interesting tattoo. Is it of Linkerd, or what is it about?

William: It’s just a temporary one. We handed them out at KubeCon last year as part of our swag. While everyone else gave out stickers, we thought we'd do something more extraordinary. So, we made temporary tattoos of Linky the Lobster, our Linkerd mascot.

When Linkerd graduated within the CNCF, reaching the top tier of project maturity, we needed a mascot. Most mascots are cute and cuddly, like the Go Gopher. We wanted something different, so we chose a blue lobster—an unusual and rare creature reflecting Linkerd's unique position in the CNCF universe.

The tattoo featured Linky the Lobster crushing some sailboats, which is part of our logo. It was a fun little easter egg. If you were at KubeCon, you might have seen them. That event was in Amsterdam.

Bart: What's next for you? Are there any side projects or new ventures you're excited about?

William: I'm devoting all my energy to Linkerd and Buoyant. That takes up most of my focus. Outside of work, I'm a dad. My kids are learning the piano, so I decided to start learning, too. It's humbling to see how fast they pick it up compared to me. As an adult learner, it's a slow process. It's interesting to be in a role where I'm the student, taking lessons from a teacher who's probably a third my age and incredibly talented. It’s an excellent reminder to stay humble, especially since much of my day involves being the authority on something. It’s a nice change of pace and a bit of a reality check.

Bart: That's a good balance. It's important to remind people that doing something you could be better at is okay. As a kid, you're used to it—no expectations, no judgment.

William: Exactly. However, it can be a struggle as an adult, especially as a CEO. I've taught Linkerd to hundreds of people without any panic, but playing a piano recital in front of 20 people is terrifying. It's the complete opposite.

Bart: If people want to contact you, what's the best way?

William: You can email me at william@buoyant.io, find me on Linkerd Slack at slack.linkerd.io, or DM me at @WM on Twitter. I'd love to hear about your challenges and how I can help.

Chapter 4

The Long-Term Support Debate

Kubernetes Needs a Long Term Support (LTS) Release Plan, Matthew Duggan

Kubernetes updates can be complex. However, Principal Security Engineer Mathew Duggan proposes a solution: a Long-Term Support (LTS) release plan.

In a podcast with Bart Farrell, Duggan explains how LTS could revolutionize cloud-native operations. They discuss essential strategies for open-source beginners and achieving stable, reliable deployments.

You can watch (or listen) to this interview here.

Bart: The cloud-native world moves fast. How do you stay updated with blogs, podcasts, or other methods?

Mat: I have a few go-to sources. I prefer Lobsters over Hacker News for its focus on programming topics. KubeWeekly is essential for me to discover the latest developments. I also rely heavily on RSS feeds, especially those from cloud providers announcing new features and services.

Attending online conferences is also crucial for me to stay informed. Conferences are the best way to stay updated on the latest industry trends. If I could advise my younger self, it would be to be more involved in open source. As a young programmer, I hesitated to

contribute, feeling I didn't know enough. Looking back, I realize that was a missed opportunity.

Bart: What advice would you give those unsure how to start contributing to open source?

Mat: If you're new to open source, I recommend starting with sites like ‘My First Issue,’ which offer beginner-friendly ways to contribute. It's a good idea to begin by improving a project's documentation. This path could involve clarifying concepts, adding detailed explanations, or creating diagrams. These valuable contributions help you learn about the project's workflow, contributors, and expectations.

Another option is to try Linux packaging. It may sound complicated, but it involves preparing someone else's code, assigning a version number, signing it with a GPG key, and distributing it. It’s a detailed but structured process; however, you can find mentors who will guide you. Since you're not writing the code yourself, it's a less intimidating way to get involved in open source.

Bart: In our previous episode with you, we discussed startups' concerns about potential application disruptions and the difficulty of updating their Kubernetes clusters. Your article Why Kubernetes Needs an LTS addresses this concern. Is Kubernetes as unstable as some people claim?

Mat: Due to its inherent complexity, finding people

experienced in managing Kubernetes is challenging. This complexity arises from the numerous layers, including the physical infrastructure, virtual machines, cloud services, Kubernetes, and other elements like network overlays and service meshes. Each layer adds complexity, and to truly master Kubernetes, you need to understand how all these pieces interact.

While the core Kubernetes engine has become more reliable and accessible to upgrade, the explosion of third-party integrations and applications built on top of it has added further complexity. Managing dependencies with third-party integrations and figuring out safe upgrade paths can be difficult.

Bart: Kubernetes has a continuous release cycle. Can you explain what a Kubernetes release cycle involves?

Mat: Kubernetes follows a rapid release cadence, with new versions typically released every four months. Each version, such as 1.2.X, has a 14-month lifecycle. For the first 12 months, the Kubernetes team provides bug fixes and security updates to maintain the platform's stability. The remaining two months serve as a grace period for users to upgrade to the next version before the current one is retired.

Compared to traditional Linux distributions, which may have release cycles spanning years, Kubernetes' faster pace can be challenging for less experienced teams.

Dependencies further complicate the process. For example, projects like

Istio usually only supports the latest three Kubernetes versions. This practice is becoming more common, putting pressure on teams to update their systems.

Cloud providers may offer extended support for specific Kubernetes versions, lasting anywhere from two to 12 months. This gives teams more time to plan and execute upgrades, easing the burden of frequent changes.

Bart: Upgrading Kubernetes seems more complicated than updating other software. Why is that, and how can we make the process easier?

Mat: While Kubernetes manages application compatibility well through API versioning, a successful upgrade involves deeply understanding the entire cluster ecosystem, including third-party integrations, CNI plugins, and storage configurations.

Cloud providers like GCP implement API version checks to prevent upgrades if deprecated calls are detected. Tools like Pluto offer similar checks.

Features like cron jobs for scheduled tasks and stateful deployments for persistent data add further complexity, especially when moving databases into Kubernetes. Managing persistent volumes and handling disk operations, such as detaching and reattaching disks, require careful attention.

A smooth upgrade requires a meticulous plan. This process includes reviewing release notes, understanding your software and its dependencies, and having in-depth knowledge of crucial Kubernetes components like etcd, the control plane, and the API server. Smaller teams might find this overwhelming.

Bart: Are teams expected to manage Kubernetes upgrades independently, or is community support available?

Mat: Upgrading Kubernetes is a significant challenge, especially for smaller or less experienced teams. However, there's a strong community and various tools available to help.

Understanding the different upgrade strategies, each with pros and cons, is crucial. A team's risk tolerance will influence their choice.

For example, blue-green deployments involve creating a new cluster with the updated Kubernetes version and gradually shifting traffic. This method is safe but expensive, as you have to run two clusters simultaneously.

In contrast, in-place upgrades involve updating the existing cluster. This upgrade approach is faster but potentially riskier. Tools like Velero can help mitigate this risk by allowing you to roll back to a previous state if something goes wrong.

Managing the control plane involves ensuring different components within the cluster, such as the kube-apiserver, kubelet, and kube-proxy, are running compatible versions, known as version skew.

There's no one-size-fits-all approach to Kubernetes upgrades. However, tools like Kubespray can automate the setup and maintenance of Kubernetes clusters, including configuring the underlying OS. Community projects like Flatcar Linux offer a container-optimized OS that can be managed remotely and updated automatically. These tools help to reduce the complexity and manual effort involved in managing the operating systems on which Kubernetes nodes run.

Bart: How do cloud providers help with the upgrade process?

Mat: Amazon EKS has improved the control plane management, and tools like Karpenter automate worker node provisioning, making EKS upgrades smoother. Microsoft Azure has also made upgrading easier by offering automatic upgrades for Kubernetes clusters.

Google Cloud Platform (GCP) is leading the way with a comprehensive set of upgrade tools. Autopilot, for example, automates upgrades, reducing the manual effort GCP users require.

While cloud providers offer many advantages, challenges remain, particularly in hybrid cloud

environments. Integrating on-premises data centers with cloud services through Kubernetes and service meshes adds another layer of complexity to upgrades. Datacenter limitations and budget constraints can also impact the feasibility of strategies like blue-green deployments.

Upgrades can be particularly daunting for startups and smaller teams. While cloud providers offer support, it's ultimately the users' responsibility to implement robust backup strategies.

Additionally, while Kubernetes works well for standard web applications, using it for GPU-intensive tasks, such as machine learning, makes upgrades even more complex. These use cases require special care to ensure compatibility with specific hardware and software configurations to avoid disruptions or issues related to hardware compatibility, software dependencies, resource allocation, scheduling, and scaling of GPU-accelerated workloads.

Bart: Is it fair for companies to charge a premium for support for older versions of EKS?

Mat: This is a controversial issue in the Linux community because of the significant resources required to backport security updates. Historically, approaches have varied, with some Linux distributions like Red Hat charging licensing fees, while others like Debian rely on Long-Term Support funded by businesses.

The debate centers on the fairness of a one-time cost model, particularly for organizations that heavily rely on Kubernetes but lack in-house expertise. With their deep understanding of each customer's setup, cloud providers sometimes update older software versions for key clients to ensure security and smooth operation. They may charge this service hourly per cluster, potentially generating substantial revenue. However, the fairness of this pricing model remains a subject of discussion.

Bart: What are the best practices for scheduling updates for services? Can automation help with this process?

Mat: There's a wide range of opinions on the ideal level of automation. While some enthusiasts on platforms like Hacker News advocate for full automation, envisioning systems that can effortlessly spin up new clusters with a single command, the reality for many organizations is more complex.

Especially in sectors with strict operational frameworks, like FinOps and government contracting, adherence to regular and critical update schedules and strict compliance protocols necessitates a different approach to automation. These organizations must balance the efficiency of automation with the rigidity of their regulatory environments.

Kubernetes has indeed evolved to include features that ensure safer upgrade paths. These include secure control

node updates through TLS encryption, Role-Based Access Control (RBAC), and comprehensive audit logging. Additionally, Kubernetes can roll back to previous versions if a failure occurs, such as an unexpected shutdown, network partitions, or hardware failures. These improvements have made it increasingly feasible for companies to align their setups with the standard Kubernetes configuration, benefiting from the built-in safety features. This standardization also allows for integrating new node worker groups, which is crucial for testing kubelet version compatibility and maintaining system stability.

However, a consistent industry standard for automation scripts and their management remains a significant challenge, especially for medium-sized organizations looking to automate their Kubernetes upgrades. The difficulty lies in navigating the complexities of Kubernetes upgrades while developing custom automation solutions without a universally accepted framework.

Bart: Can we stop updating Kubernetes to avoid potential problems that might arise with updates?

Mat: Some people have explored the idea of making Kubernetes "read-only," meaning it wouldn't change and would remain static indefinitely. However, Kubernetes is designed to evolve, receive new features from its

developers, and store data, especially with etcd. It's impractical to turn Kubernetes into a static platform.

Nevertheless, some find the concept of a Kubernetes version that you set up once and forget about appealing. While not updating isn't widely supported, it's worth considering how Google, which regularly updates products like Chrome OS, influences Kubernetes.

Bart: Would a stable long-term support (LTS) Kubernetes version benefit sectors like FinOps and the government? Would this require a consensus from a Special Interest Group (SIG)?

Mat: A workgroup is already exploring this idea. There likely wouldn't be much opposition to creating an LTS version of Kubernetes. The main question is how long the LTS version would be supported. It would receive security updates for several years but wouldn't get any new features or major upgrades after that. Once the LTS period ends, users will need to create a new Kubernetes cluster and migrate their applications and data to it.

Support for transitioning away from LTS versions would likely come from the community and potentially third-party consultants.

The challenge for the groups working on Kubernetes is determining the best way to assist users when migrating from an LTS version to a new one. This situation is similar to what happens with Linux systems, where it's

common for people to set up new virtual machines and transfer their data when the old system reaches the end of its support lifecycle.

Bart: Are there any unexpected features in the upcoming Kubernetes release?

Mat: The Working Group Long-Term Support (WG LTS) is working on simplifying the naming scheme for Kubernetes versions. They aim to use a straightforward identifier that is intuitive for those familiar with Kubernetes but might be confusing for those outside the community.

Another focus of the WG LTS is figuring out how to handle feature flags for specific parts of the Kubernetes API. Feature flags allow certain features to be toggled on or off, and managing them for specific API paths can be tricky. If they can solve this effectively, it would help cloud service providers better manage Kubernetes upgrades for their customers, ensuring everyone is using the latest and most secure version.

However, organizations that manage their own Kubernetes clusters (as opposed to using a cloud provider's managed service) will have a different level of support and tools for managing upgrades than cloud providers offer their customers. They'll need to invest in developing their tools to handle these upgrades.

While the WG LTS's efforts could make upgrades more

accessible for those using managed services, managing Kubernetes on your own remains challenging.

Bart: Could the extended support for an LTS version lead to teams favoring it over other versions, potentially neglecting those with shorter support cycles?

Mat: While an LTS version could become more popular than other versions, most organizations would likely continue to use the non-LTS versions. Organizations heavily invested in Kubernetes and cloud infrastructure find it relatively easy to upgrade and manage multiple accounts and clusters, so they wouldn't necessarily need to rely solely on an LTS version.

Businesses already using Kubernetes, such as local grocery chains and larger retailers like Chick-fil-A and IKEA, are equipped to handle it and would welcome the stability and extended transition period that an LTS version offers.

For most online companies, upgrading Kubernetes is becoming more manageable as they either improve their technical capabilities or rely on their cloud providers for upgrade prompts. An LTS version would attract those seeking consistency across their data centers and cloud operations, enabling them to run Kubernetes more smoothly, transfer workloads quickly, and reduce configuration management challenges and hardware maintenance.

Bart: We spoke with a Chick-fil-A chief architect who mentioned managing Kubernetes across their 3,000 locations, which could be daunting with constant upgrades. Are upgrades more of a mental or technical challenge, or both?

Mat: It's a complex challenge that involves both mindset and technical aspects. New releases are exciting opportunities for those deeply involved in the Kubernetes ecosystem. For example, the Gateway API excites me because it allows for decoupling HTTP routes from load balancers.

However, many teams I've worked with view upgrades differently. These smaller teams prioritize delivering features and shipping software. For them, upgrades are risks, not opportunities, similar to our hesitation with end-of-life programming languages.

Many smaller companies use outdated software versions to manage their applications because they struggle with fixing dependency issues. They might say, "We can't upgrade because of a package dependency," reflecting a broader resistance to change. This mindset also affects their approach to cluster management, where the perceived risks of upgrading often overshadow the potential benefits. The reluctance to upgrade clusters usually comes from an "if it ain't broke, don't fix it" mentality, which is both an encouragement and a

challenge.

Disaster recovery plans are a recommended best practice for upgrades. However, many teams find it challenging to create these plans. The standard advice is to manage infrastructure as code, launch a new cluster, switch deployments, update DNS entries, and hope for a smooth transition. This plan acts as a fallback if the upgrade goes differently than planned.

Applications are sensitive to their environment, and any overlooked details can cause failures, so relying solely on backups is risky. For instance, I faced an unexpected issue with a service mesh during load testing. If Linkerd were to fail, causing the proxy on every pod to reject traffic, the consequences would be severe. Troubleshooting such issues can be daunting.

Developing a recovery plan for critical and deeply embedded Kubernetes systems in companies' operations is challenging. The complexity of Kubernetes leads some people to attempt partial upgrades, but difficulties in implementing changes and fear or resistance to transformation also drive this avoidance.

Kubernetes is known for its complexity, which can intimidate teams and influence their upgrade approach. The solution lies in finding a balance between innovation and caution: a middle ground where teams can take advantage of new features while carefully managing the

potential risks of modifying their systems.

Bart: Does the tendency to avoid risk change throughout one's career, or does it remain constant?

Mat: Early in one's career, the fear of making mistakes can lead to risk aversion. New roles often come with imposter syndrome, causing self-doubt and a reluctance to change. Leaders need to foster growth and confidence in these individuals.

As people gain experience, they sometimes become overly cautious and hesitant to make quick decisions, which can lead to missed opportunities or mistakes. Making and correcting a significant error is a learning experience that can make a person more responsible and careful in the future.

However, attitudes towards risk can evolve. There is a shift from earlier in my career when we meticulously executed software releases and followed detailed instructions. Younger professionals today are less concerned with absolute stability, trust more cloud services, and are more comfortable with frequent changes.

Bart: Imposter syndrome can be pretty stressful, especially in higher roles. The pressure to be the expert increases as you climb the ladder. Would you agree?

Mat: I recall starting at a startup as the senior infrastructure expert. In my first week, I realized I wasn't

just a senior member; I was the most senior. That was a moment of panic because I had always been in junior or mid-level positions with senior colleagues to rely on. Suddenly, I was the one everyone turned to, which was a significant psychological challenge.

Working in this field, especially among skilled and passionate colleagues, can be awe-inspiring. Stepping into the role of the most senior person can be intimidating and challenging. The responsibility of being the point person for critical issues is impactful.

There has been a positive cultural shift. It's now more acceptable to embrace uncertainty, particularly in DevOps and infrastructure roles. Despite this progress, the anxiety associated with leadership remains.

Leading a tech team, especially during crises, is an overwhelming experience with a steep learning curve. While one may grow and improve, the stress associated with such roles always remains.

Bart: What are you working on currently? Are there any surprises on the horizon?

Matt: I'm deeply involved in several projects to simplify Kubernetes for everyone. Because of its complexity, Kubernetes can be intimidating, so I'm working on defining a minimal viable Kubernetes setup for a small team. This setup would include only the bare essentials: a load balancer, virtual machines, and

containers—nothing more.

My team and I are developing a Command Line Interface (CLI) tool to streamline the initial Kubernetes setup. This project was inspired by startups that needed the flexibility to switch between cloud platforms like Amazon ECS without being tied to any single one.

We're testing an open-source CLI that simplifies the Kubernetes setup process across different cloud providers. This tool handles the essential components needed to run applications, such as installing a load balancer, assigning an IP address, managing SSL certificates, and orchestrating containers. I'm focused on refining this tool to make it robust and user-friendly for widespread adoption.

Ultimately, I want to demystify Kubernetes and make it more accessible, especially for smaller teams or startups that need a more straightforward solution.

Bart: Many people are interested in simplifying Kubernetes, and your work could be a game-changer for smaller organizations with limited resources. We're excited to see how it develops. How can people contact you for more information or to follow your work?

Matt: Anyone interested can check out my website or connect with me on Mastodon, where my username is MattDevDoug. I'm active on that platform, and while not every post may be groundbreaking, I'm always happy to

engage with people and discuss my work and related topics.

Chapter 5

Lessons in Multi-Tenancy

Surviving Multi-Tenancy in Kubernetes: Lessons Learned, Artem Lajko

Artem Lajko, an innovative engineer at the Hamburg Port Authority, turned the complexities of multi-tenancy in Kubernetes into an opportunity. His deep Kubernetes knowledge and hands-on expertise made multi-tenant cluster sharing seamless and efficient.

In this insightful podcast, Bart Farrell explores how Artem's approach revolutionized developer experience and platform performance. What key strategies made this transformation possible? How does he ensure efficiency and reliability in a multi-tenant setup?

You can watch (or listen to) this interview here.

Bart: If you have a brand-new Kubernetes cluster, which three tools would you install first?

Artem: First, I would install cert-manager because it automates TLS certificate management, which is essential for secure communication, especially when building MTLS certificates. Next, I would deploy Argo CD for its declarative GitOps approach, as it's crucial for platform engineers developing the platform and customer software. The third tool would be Kyverno, as it manages and enforces policies across the cluster without needing complex admission control webhooks.

Bart: Have you tried Flux for GitOps or OPA (Open Policy Agent) for policy management?

Artem: Yes, I have experience with both Flux and Argo CD for GitOps. I initially started with Flux but found Argo CD a better fit for my workflows.

I've used Open Policy Agent (OPA) extensively for policy management. OPA is powerful and can handle policies for Kubernetes, virtual machines, and other environments. You write policies in a language called Rego, which offers flexibility like custom functions and modules. However, using OPA with Kubernetes could be cumbersome due to the reliance on webhooks.

While OPA continues to evolve, I've recently switched to Kyverno for policy management in Kubernetes. Kyverno is more approachable for developers as it works directly with Kubernetes manifests. This acquaintance simplifies defining and enforcing policies within the Kubernetes ecosystem.

I'm a platform engineer with several years of experience building and maintaining Kubernetes infrastructure. I'm passionate about automation, security, and developer experience. My goal is to create robust, secure, and easy-to-use platforms that enable developers to focus on building great applications.

Bart: Can you tell us more about yourself and your work?

Artem: My name is Artem Lajko, and I'm from

Dortmund, Germany. Dortmund is known for its football team, Borussia Dortmund, and companies like Adesso. It also has the renowned TU Dortmund University, where I earned my master's degree in computer science, focusing on software development, infrastructure, and hardware engineering.

My journey into cloud computing started in 2016 during my undergraduate studies. While working on a computationally intensive deep learning project, we encountered difficulties effectively assigning GPU resources to specific processes. Initially, we attempted to use ChangeRoute as a temporary solution to isolate processes, but it proved cumbersome. However, we soon discovered Docker, a containerization platform that transformed our workflow. With Docker, we could easily package applications and their dependencies into portable containers, streamlining GPU resource allocation and enhancing scalability.

I soon discovered Docker's limitations, particularly with port management and the lack of horizontal scaling capabilities. I explored Docker Swarm as a potential solution but found that it didn't fully meet our needs, which led me to Kubernetes. I learned the fundamentals of Kubernetes through Kelsey Hightower's 'Kubernetes the Hard Way' tutorial.

Over time, I gained hands-on experience working with OpenShift 3.11 and OpenShift 4. By 2019 or 2020, I had

transitioned into the public cloud. I managed Kubernetes clusters, focusing on building internal developer platforms—tools that streamline development workflows and improve productivity for development teams.

I have always championed a hands-on approach to learning and implementing new technologies. While this approach is valuable, it has led to numerous failures, even in production environments. Initially, some customers doubted my methods and questioned my experience with these emerging technologies. However, over time, they became pioneers in their own right and embraced the innovative solutions we developed together.

In those early days, proper documentation, formal courses, or extensive market experience were often unavailable. I viewed these failures not as setbacks but as invaluable learning opportunities. There was no other option but to experiment, learn from our mistakes, and adapt our approach.

In addition to hands-on experience, learning platforms like Udemy, KodeKloud, books, and blogs helped me expand my understanding and troubleshoot specific deployment issues, such as improper resource allocation, security misconfigurations, and scalability challenges.

While I hold certifications like CKA, CKAD, and Azure Solution Architect Expert, I view them as snapshots of my skills at a particular moment. I am not an "expert"

but a lifelong learner. These certifications motivate me to challenge myself and think outside the box continuously.

Bart: Given your extensive experience, if you could return to 2017 or 2018, what advice would you give your younger self?

Artem: I would start using managed cloud services like Azure Key Vault earlier to focus more on productive work rather than maintaining services.

Initially, I worked with on-premise data centers, deploying everything self-hosted. Shifting to managed services sooner would have been more efficient.

Bart: You recently wrote an article named Surviving 2400 Hours of Multi-Tenancy: What I Know Now. Can you explain what multi-tenancy is in Kubernetes and why it's necessary?

Artem: Multi-tenancy in Kubernetes refers to running applications for multiple tenants, such as different teams, departments, clients, or customers, within the same Kubernetes cluster. However, this definition was more relevant two or three years ago.

Today, multi-tenancy has evolved into an approach that often involves providing dedicated clusters through managed clusters. The control plane runs on the managed cluster as a tenant, using platforms like Kamaji or Kubernetes Platform (KKP), where you can bring your node pool.

Multi-tenancy is now essential for teams with limited

resources or sustainability goals. It can also be an effective solution for companies like ours that must provide new team members with a quick start.

We operate on a ticket-based system, which can be cumbersome and slow. Sometimes, it takes one to two weeks to get a new Kubernetes cluster on our on-premise infrastructure. Using this multi-tenancy approach, we can offer new employees the same experience as they would have on a dedicated cluster, allowing them to start working early and experiment freely.

Bart: What tools did you use to share your cluster with several tenants while working with the Hamburg Port Authority?

Artem: We use Argo CD HA (the high-availability version) to manage the platform's configuration and critical services like external DNS and cert-manager. We have a hybrid infrastructure setup, with one part running on-premise using various Kubernetes distributions and the other in the cloud, Azure Kubernetes Service (AKS). Argo CD allows us to create and organize Kubernetes service catalogs across these environments.

External DNS functions differently in the cloud, enabling us to deploy our configurations and services to various endpoints like Azure. We use Argo CD core instances for every team, allowing them to deploy their applications. This method is effective for both shared and dedicated clusters.

We leverage external DNS, cert-manager, and an ingress controller to enhance our infrastructure and guarantee web services are accessible outside the cluster. For Role-

Bart: Can you describe the architecture of your shared cluster in more detail?

Artem: Our shared cluster runs on vSphere with Tanzu, enabling us to utilize node pools similar to other Kubernetes distributions. We deploy node pools with specific labels and then vCluster onto these designated nodes using a node selector.

We maintain a default node pool for essential platform tools such as the ingress controller, cert-manager, and external DNS. This method ensures that these core components are always available and running smoothly.

To cater to individual developers' or customers' needs, we create dedicated node pools where each developer has an isolated environment. This approach differs from our project-specific dedicated clusters.

Each team has an Argo CD core instance, a URL, and access credentials to kickstart their work. We strive to make the Kubernetes cluster experience similar to a managed cloud service. Developers have access to the resources they need without the burden of managing the underlying infrastructure.

We've streamlined the stack on our shared cluster by removing parts like monitoring because maintaining multi-tenancy at scale requires a dedicated team. This simplified setup enables faster onboarding and is an excellent learning platform for developers, especially

those new to Kubernetes. Strict tenant isolation is a low priority in this environment, as it's primarily used for learning and non-critical workloads like documentation.

Bart: How long did it take to create this multi-tenant setup from start to finish?

Artem: It took us approximately four to five weeks to develop the initial production-ready setups for our learning platform. This platform was designed for non-critical workloads and was our first foray into multi-tenancy.

Our initial implementation of multi-tenancy utilized Argo CD natively in our Q3 cluster. However, we encountered some challenges, particularly with project separation. We were constrained to deploying applications within the Argo namespace, and although Argo was working on enabling deployments in other namespaces, that functionality was only available later.

The primary issue we faced was that deployed applications could inadvertently modify project settings to default or other configurations, causing unintended consequences. While using Argo CD to deploy applications proved successful in our dedicated clusters (where teams could utilize a Git repository to create applications), it wasn't ideal for shared clusters.

To address these challenges, we transitioned to Capsule, a tool from Clastix similar to Kamaji. Capsule seamlessly integrated with Flux but wasn't compatible

with Argo CD. We also experimented with vCluster as another potential solution.

Bart: Could you share any tips based on your experience for understanding the process and the rollout of a multi-tenant cluster?

Artem: When building a multi-tenant Kubernetes cluster, it's crucial to approach every aspect with multi-tenancy in mind, considering how each tool and component will be used and secured across different tenants. For instance, if you're implementing monitoring with Grafana or the kube-prometheus stack, you need to determine how to isolate metrics and access based on tenant groups, such as Active Directory (AD) groups for on-premise environments and Azure Active Directory (AAD) groups for cloud environments.

This meticulous approach is essential for every tool in your stack, especially when strong isolation is a priority. Adding too many tools without careful consideration can lead to increased resource consumption and maintenance overhead. It balances the desired isolation level and the practical constraints of your team's size and skill set.

Building a thriving multi-tenant cluster is not just about technical implementation; it's about aligning your strategy with your team's capabilities and resources. In some cases, achieving robust isolation might be best served using dedicated clusters for each tenant. However, this can be resource-intensive. Combining

multiple approaches, like namespaces and network policies, can be more efficient but often requires more engineering effort. The optimal solution depends on your specific requirements, team resources, and the trade-offs you're willing to make.

Bart: Considering the significant effort invested in planning, implementing, and coordinating with various stakeholders, was adopting a multi-tenancy approach ultimately beneficial?

Artem: Experimenting with new approaches is rewarding, making the multi-tenancy endeavor a valuable learning experience. However, applying this approach to workloads requiring strict isolation, as opposed to a softer multi-tenancy model, demands significantly more effort.

While we achieved a minor reduction of 200-300 gigabytes in storage usage, this is a relatively insignificant saving for most organizations. Furthermore, maintaining two separate approaches and incorporating new tools like vCluster led to increased engineering overhead, introducing additional complexities and potential security vulnerabilities.

Developers benefited from the seamless experience of using the shared cluster, as it felt indistinguishable from a dedicated cluster. However, maintaining dedicated and shared clusters with different approaches became overwhelming for our small team. The workload of

managing over 20 projects, 300 virtual machines, and 20+ clusters with just two people proved unsustainable in the long run.

Bart: As a two-person team, how did you keep up with their requests to improve the platform after delivering it?

Artem: The developers appreciated the multi-tenant setup because it provided a consistent experience across dedicated and shared clusters. The only noticeable difference was the login process, which required an additional step to access the virtual cluster, although this wasn't strictly mandatory.

We streamlined the request management process by providing templates that developers could easily fork. They would then submit a pull request, which we would approve, and Argo CD would automatically handle the deployment based on our Kubernetes manifests.

For example, if a team needed a new virtual cluster for a project, they would create a ticket, provide an AD group for us to map, and then submit a pull request. Argo CD would care for the rest, seamlessly provisioning and configuring the new cluster.

Developers were responsible for deploying specific tools like RabbitMQ, as we didn't provide these services directly. However, requesting and obtaining a new cluster was straightforward: fork the template, create a pull request, and approve it. Thanks to GitOps and our declarative approach, Argo CD handled the heavy lifting,

ensuring everything remained in sync.

Bart: Would it have been easier to have separate clusters instead of a shared single cluster?

Artem: Maintaining two distinct setups for dedicated and shared clusters was unsustainable for our small team. Focusing on a single, dedicated cluster approach simplifies management and reduces overhead.

Bart: What's the future direction for this setup? Are there any plans to evolve the multi-tenant cluster?

Artem: Based on the advice of our enterprise architect, we've decided to abandon the multi-tenant model and exclusively use dedicated clusters. While this might lead to some underutilized resources, it significantly improves stability and simplifies our management processes in the long term.

Bart: As a team of two, how do you keep up with the constant learning required for your roles?

Artem: Blogs are my primary resource for staying current with market trends and new developments. They offer a quick and accessible way to gain insights into other companies' activities. Conferences like KubeCon also provide valuable opportunities for networking and knowledge exchange with industry peers. Learning from the successes and failures of others helps us avoid common pitfalls.

Bart: How do you structure your learning time? Do you have a specific schedule for reading blogs or watching

educational content?

Artem: It's pretty simple for me. My company generously allocates time for learning and certification during work hours. If there are no urgent tasks, I can dedicate a week and a half to focus on certifications. My typical workday starts at 7 am and ends around 3 or 4 pm. After a short break, exercise, and a power nap, I spend an hour or two in the evening, from 7 pm to 9 pm, learning new things. I treat this like a daily workout for my brain. While it impacts my time, staying ahead in this field is essential.

Bart: You mentioned sports earlier. What kind of sports do you do?

Artem: I focus on general fitness and exercises like Freeletics to stay healthy and active. Nothing is too extreme, just enough to maintain a good fitness level.

Bart: What's next for you? Can we look forward to more insights and knowledge sharing from your work at the Hamburg Port Authority?

Artem: I'm collaborating with a colleague on a book about GitOps deployments for Kubernetes. It combines our perspectives as developers and platform engineers. We received numerous requests for detailed insights, which prompted us to write this book. It's my next big project.

I also plan to attend KubeCon in Paris next year to continue learning and networking. Additionally, we're

exploring managed Kubernetes solutions, looking into providers like Giant Swarm, which offers Kubernetes clusters to companies without platform engineers. This exploration involves understanding how Cluster API providers like Kamaji, KKP, and Giant Swarm operate.

Bart: When can we expect the book to be released?

Artem: We're aiming for mid-next year, around May or June.

Bart: What's the best way for people to contact you?

Artem: Feel free to send me a direct message on LinkedIn; that's the quickest and easiest way to reach me.

Chapter 6

Hacking Alibaba

Hacking Alibaba Cloud's Kubernetes Cluster, Hillai Ben-Sasson & Ronen Shustin

Securing Kubernetes clusters is one of the toughest challenges in cloud security, but for Ronen Shustin and Hillai Ben-Sasson at Wiz, it's just another day at work. These top-tier researchers are fearless in diving into the deep end. Their latest exploit? Cracking Alibaba Cloud's Kubernetes clusters through clever PostgreSQL vulnerabilities.

Join Bart Farell as he dives into how their innovative approach identifies vulnerabilities and enhances the overall security of cloud ecosystems.
You can watch (or listen to) this interview here.

Bart: What are three emerging Kubernetes or other tools that you're keeping an eye on?

Hillai: Ronen and I have extensive knowledge of Kubernetes, but our expertise only originates from working directly with Kubernetes. We're hackers who transitioned into Kubernetes hacking, not Kubernetes experts who started hacking. So, we need to familiarize ourselves with many Kubernetes tools. Most of the tools we know are those we've encountered and exploited during our engagements. Therefore, we might not be the

best sources for the latest Kubernetes tools, but we are excited about ongoing Kubernetes research.

Bart: Are there any specific tools or infrastructure that you particularly like?

Ronen: Instead of specific tools, we're more interested in infrastructure elements like service meshes. From an attacker's perspective, engaging with these is quite fascinating. Currently, we need to mention standout tools.

Bart: For those unfamiliar, can you tell us more about your roles and what you do at Wiz?

Hillai: Ronen and I work at Wiz, a cloud security company, as part of the vulnerability research team. We focus on researching primary cloud services and providers like Azure, GCP, AWS, and more. We utilize their open bug bounty programs to find and report vulnerabilities. By sharing our findings, we aim to enhance the security of the cloud community, not just for our clients but for everyone.

Bart: Is hacking cloud environments your primary focus, or is this a specialized area within security research?

Hillai: It's unique. We didn't start with cloud environments. We began as general security researchers, focusing on hacking techniques. Over time, we transitioned into specializing in cloud security. Our research aims to discover innovative ways attackers

might exploit cloud systems, ultimately leading to more secure cloud environments for everyone.

Bart: How has your hacking experience influenced your approach to Kubernetes security? Did you discover any exciting findings during this research?

Hillai: Many cloud providers rely on Kubernetes and container technology to manage their services efficiently. Traditionally, setting up individual virtual or physical machines for each customer would only be scalable for some companies. Containers offer a more efficient way to manage large infrastructures. Focusing on cloud environments, we discovered Kubernetes as the go-to tool for Alibaba Cloud and companies like IBM. Our journey started with cloud security research and ultimately led us to specialize in Kubernetes security within that domain.

Ronen: Our initial focus was on container security. We researched container escapes and other vulnerabilities that might impact containers. This research naturally led us to Kubernetes, as many infrastructures we encountered used it. We had to learn Kubernetes and develop specific techniques to achieve our goals.

Bart: If you could go back in time and share one career tip with your younger self, what would it be?

Hillai: Always follow your curiosity. Research is all about pursuing leads and hunches. We were curious

about cloud security, even though we didn't start in that field. It became popular, and we wanted to explore this new area.

Bart: What resources do you use to stay updated on Kubernetes?

Ronen: I rely on technical documents the most. I also follow blogs from cloud providers, mainly the CNCF blog, because they have valuable information. I use The Kubernetes community on Twitter to learn about new features and technologies; they are highly active there.

Hillai: Additionally, I recommend Reddit. Many communities focused on security, Kubernetes, and cloud computing offer great content.

Bart: We came across an article about how you hacked Alibaba Cloud's Kubernetes cluster and a talk you gave at KubeCon. What motivated you to do this research, and did your company support you?

Hillai: Our company supports security research. At Wiz, we focus on cloud security research, often utilizing offensive security methodologies. We act like attackers to find vulnerabilities and then report them to the vendors. By identifying vulnerabilities, we can report them to the cloud providers and prevent actual attacks. Alibaba Cloud is just one example of this engagement.

Ronen: Our research often leads us to discover new hacking techniques we need to learn about. We share

these discoveries with everyone so they can protect themselves.

Bart: One of our previous guests talked about Kubernetes secrets management and threat modelling. How do you approach exploiting vulnerabilities from a hacker's perspective?

Ronen: Our best security insights come from working with different applications, frameworks, and cloud systems. When we engage with one, our primary goal is to find critical security mistakes in its setup. To do this, we must fully understand how the system works and where attackers might discover weaknesses.

Hillai: There's an interesting difference between traditional and cloud security research. In traditional research, the goal is often to achieve "Remote Code Execution" (RCE) on a specific application, which means taking control of a machine and running unauthorized code. However, in the cloud, things are different. Since you often have access to a virtual machine yourself, RCE becomes less attractive.

The real challenge in cloud security lies in breaching the barriers between different customers. Unlike traditional environments, the cloud is a shared space with hundreds of thousands of users. Our focus is to demonstrate the possibility of attackers moving between these customers, even without data access. This risk highlights a unique

cloud security risk the potential for attackers to "jump" from one user to another and compromise their information. This type of research, proving a breach of trust without actually stealing data, is a crucial aspect of cloud security and something rarely seen in traditional security research.

Bart: When starting this research, why did you choose Alibaba Cloud?

Ronen: Our initial study focused on PostgreSQL. Since many cloud providers offer managed PostgreSQL instances, we were interested in how they handle the infrastructure. We discovered vulnerabilities that allowed us to execute code on these instances. We tested several providers, including Alibaba, and presented our findings at the Black Hat talk.

Hillai: We began with PostgreSQL and expanded to Alibaba and other cloud providers. Our blog post provides more details about PostgreSQL and our Black Hat talk.

Bart: Why did you choose to focus on PostgreSQL for your research?

Ronen: PostgreSQL is a robust database with many features, including the ability to execute code within the database. While this capability can benefit certain users, it poses a potential security risk in cloud environments. Cloud providers typically modify PostgreSQL to prevent

users from executing code on their managed instances. However, our research identified vulnerabilities in these modifications, not in the core PostgreSQL code itself. We were able to exploit these vulnerabilities to bypass the restrictions and still execute code on the managed databases.

Bart: How does PostgreSQL relate to Kubernetes in this context? Did you find a way to access a Kubernetes cluster by exploiting the PostgreSQL vulnerabilities?

Hillai: Cloud providers often use containers and orchestration tools like Kubernetes to manage large-scale services, including PostgreSQL. This approach allows them to offer these services to many customers efficiently. While exploiting the PostgreSQL vulnerabilities, we discovered that we were actually in a Kubernetes environment. The user interface typically abstracts away the underlying infrastructure from the user, but our research methods disclosed it.

Ronen: We've seen various infrastructures, but Alibaba and IBM used Kubernetes for their managed services. Other providers might use different implementations.

Bart: Security experts often talk about avoiding vulnerabilities caused by misconfigurations, which can be human errors. What were the biggest misconfigurations you found that created security risks?

Hillai: The biggest misconfiguration we found is treating

containers as the only security barrier. It's important to remember that containers can be a security layer within a more extensive security system, but they should be relied on only partially. Containers alone wouldn't be strong enough to isolate each company's data from each other entirely because any security flaw in the core Linux system (the kernel) could bypass container security. We were able to exploit such misconfigurations during our research.

Another problem is poorly managed secrets within the Kubernetes environment. These secrets could read information across the system and write and change it, which meant we could overwrite software packages used by many cloud services and customer accounts within Alibaba. Essentially, these powerful secrets allowed someone to access different environments, services, and customer data—all with a single key. That's a significant security risk we wouldn't recommend taking.

Ronen: The specific secret we found was the image pull secret. In Kubernetes, when you want to download images from a private registry, you need this secret to configure network access. If you misconfigure it, you might accidentally include a secret key with push permissions instead of pull permissions. This key should only allow downloading images, not uploading them. If an attacker gains access to a key with push permissions (like what we achieved in Alibaba), it could have

devastating consequences for your entire environment.

Bart: To those without a strong background in security, it may seem that security experts click a button, scan your system, and find vulnerabilities. However, security research, like many other fields, is a blend of art and science. Can you elaborate on this further?

Hillai: Security research requires a lot of creativity. When you hear about a new attack vector, it boils down to creative thinking coming up with something no one else has considered. In this research, we started by looking for patterns we already knew were risky, like overly permissive settings and shared volumes. We had to think outside the box. Returning to the Alibaba Cloud control panel, we began experimenting. This exploration led us to a breakthrough when we discovered a button enabling SSL encryption for the PostgreSQL instance. Clicking it triggered new activity in the container, which we followed to escape the container.

Bart: To help our audience understand, could you explain SCP, its role in the attack, and how you exploited it?

Hillai: SCP stands for Secure Copy. It's a standard tool on Linux systems that transfers files between machines using secure SSH connections. In our case, the SSL encryption feature we triggered used a new Alibaba management container. This container ran the SCP

command on our container to move the SSL certificate.

SCP reads its configuration from a directory we control within our container by default. We placed a malicious SSH configuration file there. When the SCP command loaded this configuration, it ran a command we placed within the file. This trick let us escape our limited container and jump to the Alibaba Management Container because it unknowingly executed our command.

Ronen: A crucial factor in this exploit was the shared volume. This volume acted like a shared home directory for our container and the management container since the same user existed in both containers. We could exploit this shared space because SCP reads its configuration from the user's home directory by default. By replacing the default configuration with ours containing a malicious command, we tricked the management container into running it when it used SCP.

Bart: What does successfully creating a privileged container using the Docker API tell us about cloud security in general?

Ronen: Many cloud environments rely on Docker to manage their containers. You can create a new container through an HTTP request if you gain access to the Docker API socket. This container could be privileged, meaning it shares resources like namespaces and

possibly even volumes with the underlying host machine, the Kubernetes node. Spawning a privileged container grants you access to almost everything the node has access to.

Hillai: You transition from being a guest in the container to gaining complete control of the host machine.

Bart: Gainin access to the node would only give you control of some of the Kubernetes clusters, would it?

Ronen: With code execution on the node, we could use Kubelet credentials to explore further, looking for commands, codes, secrets, and other information. In our case, Alibaba had misconfigured its Kubelet credentials: it was too powerful. We could list all pods, see all the code in the cluster, potentially containing customer data, and even retrieve all the secrets using the "kubectl get secret" command. This misconfiguration was the key that unlocked broader access for us.

Bart: Did you achieve the entire exploit on a single node within the cluster?

Ronen: Yes, we were on a single node. Using the compromised Kubelet credentials, we could see all the other nodes and resources in the cluster.

Hillai: While the specific node we compromised was isolated and didn't contain data from other customers, the service account associated with Kubelet had excessive permissions. Even though the node itself was secure, this

service account allowed us to access sensitive information across the entire cluster, including pods, nodes, and secrets belonging to other customers.

Bart: What was the next step after taking over Alibaba's managed PostgreSQL offering? Did you contact Alibaba to report your findings?

Hillai: Once we discovered the ability to access data belonging to other customers, our research stopped immediately. We wouldn't risk even accidentally accessing someone else's data. At that point, we documented everything we found and sent a detailed report to Alibaba Cloud, and they responded quickly and professionally. They kept us updated on the fixes they deployed throughout the research process. We immediately report any critical issues to prevent others from exploiting them.

Bart: Can you tell us about any specific fixes they implemented based on your findings?

Ronen: The first issue was a misconfiguration that falsely indicated increased resource consumption. We exploited it to execute unauthorized code on the operating system. We collaborated with Alibaba Cloud to fix this problem. They also resolved the SCP vulnerability problem that allowed unauthorized access to their management container. Finally, they restricted the Kubelet permissions to a narrower scope, granting

only specific permissions.

Hillai: Following our research, Alibaba took several steps to address the vulnerabilities we discovered. They limited image pull secret permissions to read-only access, preventing unauthorized uploads. Additionally, they implemented a secure container technology similar to Google's gVisor project. This technology hardens containers and makes them more difficult to escape from, adding another layer of security.

Bart: Throughout this process, what key lessons did you learn?

Hillai: There are two main lessons learned. First, containers shouldn't be relied on as the sole security barrier. While they can be a layer of security, they can be bypassed in various ways. Additional precautions are crucial to ensure proper isolation between customers. We recommend building a layered defense so that a single vulnerability doesn't allow unauthorized access to a competitor company's data.

Second, strong credentials require careful management. As Ronen mentioned, Alibaba originally had a powerful secret that could be read and written across the cluster. This secret also had push access to the central Docker image registry. Following our report, they limited the scope of these credentials. It's essential to be very cautious with such powerful secrets. Ideally, you should

scope the secrets to specific actions and minimize them whenever possible. A powerful secret can allow attackers to move across different environments, including production, development, testing, and even development workstations.

Another lesson learned relates to the container itself. The SCP vulnerability we exploited highlights the risk of shared namespaces between containers. In the Alibaba incident, the shared namespace and home directory allowed us to exploit the SCP vulnerability. Always be very careful when sharing namespaces between trusted and untrusted containers. The lesson learned is to minimize what you share and never grant unnecessary permissions. Attackers may exploit even seemingly minor misconfigurations.

Bart: Can you recommend any specific tools that people might need to be aware of if they want to discuss implementing some of these mitigation tactics with their managers?

Hillai: There's one framework I highly recommend: Peach. It's an open-source project developed by our research team and contributions from fantastic people at many companies.

Peach is a framework that outlines how to build secure and isolated environments, whether in the cloud or not. Like a white paper, it's a valuable resource that guides

you on properly isolating tenants or customers in a multi-tenant environment. It covers common mistakes to avoid, what to look out for, and how to implement the necessary precautions.

If you manage a multi-tenant environment or need to isolate resources within your environment, Peach is a valuable resource worth exploring. It covers the common mistakes to avoid and offers best practices for implementing protection. It's completely open-source and available on GitHub. We also welcome contributions from anyone with additional tips or tricks we might need to know.

Ronen: I also recommend using secret scanning tools. These tools are essential in our research; we use them to identify potential secrets-related vulnerabilities.

Bart: Do you have any recommendations for securing multi-tenant Kubernetes clusters?

Ronen: Securing multi-tenant Kubernetes clusters involves a few key areas. First, prioritize network security. By default, Kubernetes doesn't restrict node communication, so strong network isolation is essential.

Second, separating namespaces between customers is a good practice when dealing with multi-tenancy.

Additionally, consider implementing container security technologies like gVisor or Kata Containers. Don't solely rely on Docker's security features to prevent container

escapes.

Bart: What advice would you give for hardening containers to make them more secure?

Ronen: Our case study with Alibaba revealed they were using shared Linux namespaces between containers, such as their management container and our container. Sharing Linux namespaces can be dangerous. When designing a system that shares namespaces or resources between management and regular user containers, constantly carefully assess and be aware of the risks involved. Container technologies like GVisor and Kata Containers can mitigate the risk of attackers exploiting Linux kernel vulnerabilities in your environment to achieve kernel-level code execution and jump to the Kubernetes node.

Bart: What advice would you give to Kubernetes engineers needing more security experience?

Hillai: Security is crucial. Companies of all sizes, from startups to large corporations, are constantly targeted by malicious actors, not just ethical hackers like us. Anyone managing a service on the internet must understand that they are a potential target for cyberattacks. These attacks range from data breaches to ransomware attacks that turn off your entire operation. Even small projects need to pay more attention to security.

The good news is that many tools can help you achieve

security without being a security expert. Tools like gVisor are relatively easy to implement because you don't need to write them from scratch. By using security hardening tools, you gain significant protection benefits.

Ronen: Besides the tools, many online resources are available to learn about security. These resources can help you understand security risks and how to mitigate them. Kubernetes itself has built-in security features, including default security policies. Be security-conscious and take steps to secure your environment.

Bart: You discover a vulnerability and report it to the vendor. What prevents you from exploiting the vulnerability for malicious purposes instead? Wouldn't Alibaba eventually find the problem on its own?

Ronen: We started seeing signs that Alibaba was taking steps to address the issue while we were still in the research phase. They were transparent with us about their efforts. Cloud providers all have security teams that constantly monitor their environments. They likely knew we were there.

Hillai: Cloud providers are doing a great job with security. We're ethical hackers; our goal is to improve security for the cloud community. Penetration testing, or offensive research, is a tool to achieve that goal. We want to fix the vulnerabilities, and it's rewarding to hear that our reports lead to security updates that benefit

many customers. We do this to make cloud products more secure and help users learn how to secure their deployments.
We publish blogs and give talks so that security professionals and developers can learn from our research and identify potential problems in their environments.

Bart: What's next on the agenda for you both?

Hillai: We're always working on new research projects. Sagi from our team recently published a blog about a vulnerability in Hugging Face, an AI provider. We have several ongoing projects under disclosure, meaning we can only reveal them once we fix the vulnerabilities.
Follow our blog; it's the first place we announce new findings.

Ronen: Our research will benefit the Kubernetes security community as well.

Bart: How can people contact you if they have questions?

Hillai: We're both on Twitter. My handle is @hillai, and Ronen's is @RonenSHH. You can also email us at https://research@wiz.io, but Twitter is the best way. Make sure to spell the names correctly.

Chapter 7

Scaling Jenkins to 10,000 Builds

From 0 to 10k Builds a Week With Self-Hosted Jenkins on Kubernetes

10,000 builds a week with self-hosted Jenkins on Kubernetes sounds impossible, but Stéphane Goetz, a Senior Developer at Swissquote Bank, achieved it. With his extensive experience in web development and cloud-native technologies, Stéphane and his team have implemented a robust CI/CD pipeline, managing the entire process with Kubernetes and Jenkins, ensuring massive scale and efficiency.

Bart Farell sat down with Stéphane to discuss the journey, the initial challenges, the transition to Kubernetes, and how they used a unique 'sandbox' environment and KubeVirt to overcome scaling challenges and revolutionize their CI/CD pipeline.

You can watch (or listen to) this insightful interview here.

Bart: If you have a new Kubernetes cluster, which three tools would you install?

Stéphane: Firstly, I highly recommend ArgoCD. It has been recommended many times on your podcast, and I agree that it is an excellent tool for managing the deployment of applications within a Kubernetes

environment.

Secondly, we switched from using the basic NGINX ingress controller to Traefik because it better suited our specific needs. Traefik has proven to be highly effective at managing incoming traffic to our custom-built Kubernetes cluster, and we are delighted with its performance.

Finally, I recommend Backstage for overall management. While it is not designed explicitly for Kubernetes, it is an excellent platform for consolidating and displaying information about the various tools we use, the operations within our cluster, and the health and performance of our system.

Bart: With observability becoming more complex, do you find the number of tools overwhelming? Is there a trend toward consolidation?

Stéphane: Observability always feels incomplete because it encompasses so many different aspects. While tools like Prometheus and Grafana effectively handle metrics, and the ELK stack effectively aggregates logs, challenges arise in areas like alerting and error tracking.

Tools like Sentry can help track errors logged by applications, and more comprehensive application performance monitoring (APM) tools provide full application traceability. However, integrating all of these different aspects is still a challenge. Some tools cover

most or some of these areas, but the ecosystem has yet to mature fully.

Bart: Can you tell us a little about yourself and your work?

Stéphane: I'm Stéphane Goetz. I started my career as a web developer with PHP, HTML, and CSS. I'm still a web developer but mainly work with React and TypeScript and use Java. I've been working at Swissquote Bank for many years. It's an online bank based in Switzerland that serves several European countries. My team and I have worked on various projects, but lately, we've been focusing on architecture and developing libraries that other developers across the company can use.

Bart: With your web developer background, how did you first get into cloud-native technology?

Stéphane: Many years ago, I ran a small company where I had to do some sysadmin work, like managing a Linux server. At one point, we decided to set up our own Jenkins cluster. That was our first experience with cloud-native technology, and it introduced us to Kubernetes and containerizing our workload. That's how we started down that path.

Bart: What differences do you see between web and cloud-native environments?

Stéphane: One of the things I appreciate about

deploying applications in Kubernetes is the use of configuration files. Whether you like YAML or not, the ability to define your environment and automate deployments with tools like ArgoCD is a game-changer. Scaling your applications and reliably deploying them using templates is straightforward. Compared to how we used to deploy applications, it's like night and day.

Rather than hands-on Kubernetes work, I focus on understanding the platform's core concepts and benefits. My expertise lies primarily in JavaScript developer tools, and I rely on colleagues, particularly our SRE team, to manage our production clusters and stay informed about the latest Kubernetes developments. We regularly discuss new ideas and implementations.

Bart: Let's discuss your article "From Zero to 10,000 Jenkins Builds a Week." What were the initial challenges and motivations that flared this journey?

Stéphane: This story dates back to around 2015. When I started at Swissquote, we weren't using Jenkins at all. But soon, we noticed that teams were running their own Jenkins instances on their personal machines. These setups were often unreliable, with machines breaking down, teams skipping updates, and builds failing due to missing configuration files. Plus, these machines weren't powerful enough to be used for both development and releases. It was clear that something needed to change.

Bart: What were your next steps to fix the pain points?

That's when we officially formed our team to support the development teams. Before that, we were a loosely organized group. Our first big project was to facilitate the process and create a solution to ensure builds were always successful and automated. At first, we tried getting a bigger, more powerful server from IT and centralizing Jenkins on it. But this didn't work because it didn't fix the underlying problems—missing configurations and inconsistent build results still happened across different projects, even though we tried to consolidate everything.

Then, one of my colleagues introduced us to Kubernetes, a relatively new concept. He suggested that Kubernetes could help us solve our problems with scaling and consistency by letting us manage applications in a cluster. We were interested, so we looked into how other companies started using Kubernetes in 2015, even though it wasn't widely adopted.

We created a small MVP (Minimum Viable Product) and set up a small Kubernetes cluster with four machines. Each development team has its own Jenkins controller within the cluster to manage its builds. We containerized each build using Docker, which we already used for development, to ensure that all builds were isolated and consistent. Automating configuration management was a

key focus, ensuring that every build in our system had a basic configuration applied automatically.

Bart: How did your initial MVP's architecture work? How did a code change progress through your setup's CI/CD pipeline?

Stéphane: At that time, we were using Mercurial instead of GitHub. The process was similar to Git: you'd push a single commit for a change. We had a daemon that watched for these changes on the central server. The daemon would then determine which repository and commit ID were affected and send that information to a central server.

Stéphane: The central server would determine which maintainer and team were responsible for the code change. It would communicate with the Jenkins controller assigned to that team, create a new job or update an existing one if necessary, and then trigger a build. After the build finishes, the server sends an email to notify people of the build's success or failure.

Bart: Was the MVP successful?

Stéphane: It was very successful, mainly because of its automation capabilities. Automating job creation was a big breakthrough. We used metadata from our repositories, especially Maven's POM XML files, to assign jobs to the right teams automatically. Teams didn't have to configure their CI processes anymore

manually—it was all automatic. This was very well-received by our teams. We had about 20 teams back then, and each team eventually had its own Jenkins instance configured and running smoothly.

Bart: Did any challenges arise during the MVP's development or implementation?

Stéphane: One ongoing challenge, even today, is reliably downloading and managing assets like Docker images, NPM packages, and Maven libraries. Caching helps but can introduce conflicts with concurrent writes during builds. Docker images, in particular, can be large and slow to download. We've used caching and automated cleanup scripts to mitigate these issues.

Another challenge was the configuration process. In Jenkins 1, manual configuration through the user interface was complex and error-prone when replicating configurations across builds. The introduction of Groovy scripting in Jenkins 2.0 allowed us to define configurations as code, making tasks like checking out repositories, building Maven projects, and sending email notifications much more painless to automate and manage.

Bart: How did you scale the MVP to handle 10,000 weekly builds after setting it up? Were there any significant changes or challenges you faced during that scaling process?

Stéphane: Scaling to 10,000 builds a week was a gradual process with several improvements. An early enhancement was integrating SonarQube for code quality. We set up dedicated SonarQube instances for each Jenkins controller, allowing teams to configure and define quality gates. Every build in Jenkins automatically underwent SonarQube checks, a process simplified by the new configuration scripting language in Jenkins. We wrote shared functions to simplify adding SonarQube steps to builds, making it easily accessible to most teams. This addition was well-received and remains a crucial part of our workflow today.

Another change was creating a new "Productivity" team that worked alongside us. The team focused on improving and standardizing our pipeline and developer tools. They enhanced our standard pipeline by adding automated reports like Allure and tailoring tools to specific team needs. The Productivity team enhancements enabled our CI/CD pipeline to support a broader range of projects, including mobile app builds and Python projects, beyond our initial focus on Java.

Around 2019, we migrated from Mercurial to GitHub Enterprise. This complex process required ensuring compatibility between both systems during the transition. After completion, we phased out Mercurial entirely and fully adopted GitHub Enterprise.

As our company grew, so did the number of teams using our CI/CD infrastructure. We started with about 30 teams and now manage around 15 Jenkins instances. Managing these instances became challenging as teams customized their setups, sometimes leading to issues we had to investigate and resolve.

Bart: How did the team address the issues arising from teams breaking things in the Jenkins instances?

Stéphane: Addressing the issues with Jenkins became a significant undertaking for our team, primarily composed of engineers rather than system administrators. We initially managed updates with basic 'kubectl apply' deployments, but this method proved unsustainable as we scaled to 20-30 Jenkins controllers.

The discovery of Helm charts led to a transition to a Helm chart framework. Simultaneously, we implemented partially immutable configurations for Jenkins, meaning that essential configurations like pod startup parameters and user authentication settings would automatically reset upon container restart. This standardization ensured a consistent baseline for our Jenkins instances.

To maintain team flexibility, we ensured that our adjustments didn't override team-specific settings. This approach fostered experimentation and encouraged teams to contribute improvements that we could incorporate into the standard pipelines.

Another challenge stemmed from Jenkins' file-based configuration system. Newly launched Docker images would copy plugins onto the disk, sometimes causing conflicts. To resolve these conflicts, we standardized our Docker images always to include the latest versions of essential plugins. This standardization ensured consistency across deployments.

Bart: You've essentially built a Jenkins system within your existing Jenkins setup. The only thing you still need to include is ArgoCD.

Stéphane: We've created a self-contained Jenkins environment that automates its processes. However, our SRE team, which is responsible for managing all production clusters, was already using ArgoCD for their infrastructure. They offered to integrate it with our initially independent cluster, and we were happy to do so. On the first integration day, we set up ArgoCD to manage our Helm charts, enabling it to update our Jenkins deployments automatically after each build.

Bart: Your setup is unique in scale, with multiple teams, including Jenkins within Jenkins and now ArgoCD. What challenges have you faced that others might not typically encounter?

Stéphane: Our main challenge is the comprehensive management of resources within the cluster, which includes personnel, network bandwidth, CPU,

input/output operations, disk space, and memory. We've exceeded the limits of each of these resources at different times and had to find solutions.

Initially, we underestimated the resource requirements and assumed we would stay within the limits. However, we quickly reached those limits. For example, Jenkins logs grew to over a gigabyte per build, so we implemented a strict limit of 10 megabytes per log to conserve storage. Kubernetes handles CPU and memory efficiently by default, but we needed more precise control over pod lifecycles during builds. To address this, we introduced an automated mechanism to shut down containers after 90 minutes to guarantee efficiency.

A persistent challenge has been the setup of integration test environments, which has required ongoing innovation and problem-solving over time.

Bart: Did you run the integration tests against a test or staging environment?

Stéphane: For integration testing, we used a rather creative approach. We built a system called the sandbox for our development environment – we like to be inventive with names. Essentially, it's a tool that analyzes Maven dependencies. If a dependency indicates a sandbox, meaning a development environment in our context, the tool adds it to a Docker Compose YAML file and starts it up.

Each sandbox has a corresponding Docker image listed in the Docker Compose file for easy reference. For example, if your application needs a database, we'll start that database. If it depends on another application, we'll start that one, too. The system works recursively, meaning if one application starts another, and so on, up to 60 applications deep, it automatically handles all dependencies.

This approach makes adding dependencies and starting applications accessible on developer machines, provided they have enough CPU and RAM. However, it does come with challenges. Even though it's technically possible, developer machines can struggle to run many containers simultaneously. We don't recommend pushing the limits because it strains the cluster resources.

Bart: Could you distribute this load across the cluster using Kubernetes?

Stéphane: Kubernetes is excellent at managing resources for containers it directly oversees. However, our setup involved Docker Compose, which meant our pods were utilizing Docker sockets and launching containers outside Kubernetes' direct control. We had to employ some clever networking tricks to connect these environments.

The challenge was that Kubernetes wasn't aware of all the containers running in these integration environments.

While Kubernetes might report everything as working, nodes could be struggling with the load—especially when we had 200 containers running simultaneously and consuming a significant amount of memory.

Additionally, Kubernetes did not manage operations like Docker builds, which use the Docker socket to start containers. To address these issues, we improved our Sentinel system. This new version intelligently monitored container usage, enforced memory quotas, and terminated containers after 90 minutes to prevent lingering instances from consuming resources unnecessarily, which helped stabilize our environment and ensure smoother builds for our users.

Bart: Did you learn any other valuable lessons from this experience?

Stéphane: We realized a significant oversight in our cluster design. Initially, we hadn't anticipated the complexities of running Docker outside of Docker—a technique known as Docker-in-Docker(DinD). While technically possible, it presented unforeseen challenges. Docker's local image caching system, designed for single daemon use, required extensive image downloads when starting new daemons. For example, downloading 60 images for an application with multiple sandboxes took up to 45 minutes, highlighting network limitations and scalability issues.

We attempted an alternative approach using Kubernetes to manage short-lived namespaces and pods for Docker operations. However, this approach introduced significant complexity, especially in aligning with developers' existing Docker Compose configurations for local development. Features like local file mounts and more significant configuration limits posed additional hurdles we needed help to overcome. Even though using Kubernetes for Docker operations seemed theoretically sound and aligned well with Docker Compose, the challenges in implementation outweighed the potential benefits.

Bart: As we approach 2023, how large is the team responsible for managing the CI/CD pipeline?

Stéphane: At the beginning of 2023, our team dedicated to Jenkins infrastructure grew to four members. The adjacent productivity team also had four members, and the SRE team, which supports us, also had four members. During this time, we decided to move away from the cluster we originally built due to its limitations and fragility. It became clear that our sister team, the productivity team, which already managed build content, best practices, and pipelines—including SonarQube—was better suited to oversee the Jenkins controllers' management.

Bart: With such a significant transfer of code and

knowledge between teams over almost a decade, how did you manage that process?

Stéphane: It was a monumental challenge. One of the initial hurdles was adapting our workflows to fit within the stricter operational constraints of the SRE-managed, production-grade cluster. For instance, running Docker within Docker wasn't allowed due to security and stability concerns for a finance company's infrastructure. We had extensive discussions with the SRE team to address these challenges. They proposed using KubeVirt—a virtualization system allowing containers to run on Kubernetes with similar primitives—which met our needs for Docker operations like builds and test containers.

Another challenge was efficiently managing Docker image downloads.

SRE developed a clever solution involving a pre-populated image containing essential Docker images, significantly speeding up our build processes.

A critical technical constraint was the shift from using local storage for Jenkins configurations to a more resilient and scalable solution. SRE implemented Ceph to address this, ensuring high availability and flexibility in node rescheduling.

Once we resolved these technical blockers, we carefully planned the migration from our existing Jenkins

instances to the new cluster without disrupting user workflows. We collaborated closely with the productivity team throughout this process, ensuring they were onboarded and equipped with the necessary knowledge to manage the new environment effectively.

Sharing our expertise and planning ensured a smooth transition and phased out legacy practices and deprecated systems from the old cluster. This approach guaranteed that everything running on the new cluster was current, operational, and well-understood by all stakeholders.

Bart: How does the new setup with Jenkins on KubeVirt differ from the previous generation?

Stéphane: From a user's perspective, the core development process remains the same: code commits, build triggers, and pipeline management all function as before. However, the switch to KubeVirt has enhanced the underlying infrastructure for these processes. Each build now operates within its dedicated virtual machine (VM), like a separate computer within the whole system. This isolation prevents resource-heavy builds from impacting the performance or stability of other builds simultaneously. As a result, the overall system becomes more reliable, with each build having guaranteed access to its allocated resources and consistent performance throughout

We also focused on enhancing observability across all resources. With KubeVirt and our properly managed Kubernetes setup, we gained better insights into resource usage beyond memory and CPU. This new setup eliminated the need for our previous Sentinel system, as Kubernetes now comprehensively handles resource management within each Jenkins controller's dedicated namespace. It also allowed us to allocate resources dynamically based on each team's needs, ensuring optimal performance without wastage.

We retained and improved our caching techniques for tools like npm and Maven, significantly enhancing build speeds. However, integrating KubeVirt did require us to adapt the Jenkins Kubernetes plugin. Since KubeVirt operates VMs rather than pods, we devised a workaround where Jenkins starts a pod that, in turn, launches a VM through a bash script. While this approach introduces a slight delay compared to pods, our optimizations have kept VM boot times under 30 seconds, which is manageable given our build durations.

Bart: Given these improvements, why not consider using microVMs?

Stéphane: MicroVMs, such as those supported by technologies like Kata Containers, are intriguing. However, during our transition, some limitations with kernel calls didn't align with our requirements. We're

watching this for future enhancements, especially considering the potential for microVMs to offer faster startup times, which remains a challenge with larger VMs.

Bart: Has the new Jenkins cluster provided additional opportunities for optimization?

Stéphane: Moving to a well-managed cluster has unlocked several optimization avenues. One of the immediate benefits is our access to Prometheus metrics, which provide extensive insights into KubeVirt's performance and overall cluster health. We've created dashboards that monitor VM usage across the cluster, including team-specific resource utilization in real-time and historically.

At the end of each build, we include a link to a specific dashboard that details resource consumption (network, IO, memory, CPU) for that build. This level of granularity allows teams to fine-tune their resource allocations using predefined profiles—small, medium, large, and extra-large. Teams can adjust resource allocations based on their needs, optimizing for either a few large builds or numerous smaller ones. This flexibility enables teams to manage their resources efficiently and aligns with our goal of providing robust, scalable CI/CD capabilities out of the box.

Bart: Transitioning to Jenkins on KubeVirt was a

significant change. How did you ensure team buy-in and maintain commitment throughout the process?

Stéphane: Encouraging teams to embrace the change wasn't difficult from a business standpoint because we had allocated a budget for such technical advancements. Transparency was vital; we focused on making developers' lives easier and communicated the benefits and steps of the transition. We worked around teams' schedules and concerns, ensuring minimal disruption. There wasn't much resistance; management understood the risks of maintaining an outdated cluster versus modernizing our infrastructure for stability and uptime guarantees.

Bart: Were there any obstacles or resistance you faced during the migration?

Stéphane: Management supported the migration due to the risks posed by our old cluster. The main challenge was migrating builds from Docker out of Docker to KubeVirt. We conducted thorough testing and phased rollouts to minimize impact. Our productivity team played a crucial role in ensuring success by monitoring and addressing issues promptly.

Bart: Looking back, is there anything you would have done differently?

Stéphane: We underestimated the importance of meticulous resource management early on. Initially, we

assumed Kubernetes would handle everything seamlessly, but we learned the importance of proactive resource planning to prevent build failures from affecting the entire cluster.

Additionally, managing our cluster was a distraction; relying on SRE-managed infrastructure has proven more efficient and reliable.

Bart: What's next for your team after completing this migration?

Stéphane: My team will phase out our old cluster, focus on architectural projects, and update internal libraries for future Java versions. Meanwhile, SRE is exploring advanced technologies like eBPF for automation and observability enhancements, while the productivity team focuses on optimizing Docker image caching.

This transition has prepared us well for future innovations and efficiency gains across our CI/CD pipeline.

Bart: How can people get in touch with you?

Stéphane: You can contact me through LinkedIn, Twitter, or email. I actively respond on those platforms.

Chapter 8

From Docker Compose to Kubernetes

Migrating 24 services from Docker compose to Kubernetes, Ronald Ramazanov & Vasily Kolosov

Managing the intricate dance of 25 microservices across diverse environments is no small feat, but it's a challenge that Ronald Ramazanov, the DevOps lead at Loovatech, and Vasily Kolosov, Loovatech's CTO and co-founder, have mastered.

With a wealth of experience and a passion for cloud-native technologies, they have successfully navigated the transition from Docker Compose to Kubernetes, ensuring seamless scalability and peak performance for their client's applications.

In a conversation with Bart Farell, Ronald and Vasily shared their secrets of success, the lessons they had learned along the way, and whether every project should start with Kubernetes.

You can watch (or listen) to this interview here.

Bart: Ronald, if you were starting with a brand-new Kubernetes cluster, which three tools would you install first?

Ronald: My first choice would be ArgoCD, a robust GitOps tool known for its intuitive web interface.

ArgoCD simplifies the deployment of other components within the cluster, making it a valuable starting point. Next, I would install Prometheus to monitor both the cluster's activities and the performance of the applications running on it. When combined with visualization tools like Grafana and other management tools like Alertmanager or kube-state-metrics, Prometheus provides a complete and detailed understanding of how the Kubernetes cluster performs and its overall health.

Finally, I would add KEDA to the mix. It extends Kubernetes' built-in scaling capabilities by allowing the cluster to automatically scale based on custom metrics, leading to more efficient resource utilization.

Bart: Vasily, do you agree with Ronald's choices, or would you adjust this list of tools?

Vasily: These tools, like Swiss army knives, are incredibly versatile and essential for our projects. They prepare us for all sorts of concerns that can come up when we deploy software, like scaling resources up or down to meet demand, updating software without interrupting service, and recovering quickly from any failures.

Bart: Ronald, please tell us about your professional background and current role.

Ronald: As the DevOps lead at Loovatech, I've spent the

past four years working with a team of four engineers, primarily focusing on cloud infrastructure. We mainly utilize AWS and prefer Kubernetes as our go-to tool for container orchestration. Our responsibilities are twofold: we manage Loovatech's internally developed applications and provide DevOps services to external clients and their development teams.

Bart: Vasily, can you tell us about your role at Loovatech?

Vasily: As Loovatech's CTO, I'm responsible for ensuring the successful delivery of our diverse services to our clients. These services encompass developing custom software solutions tailored to their unique needs, creating and maintaining cloud-based applications accessible online (Software as a Service, or SaaS), and designing, building, and managing the cloud infrastructure that supports our clients' applications.

Ronald and I have been working closely together since 2018, when we first met during a search for a systems administrator. Over the years, we've both grown professionally alongside Loovatech as it has embraced cloud-native technologies, and it's been an incredible journey for us.

Bart: Thinking back about six years ago, could you describe how your company shifted towards cloud-native technologies, particularly Kubernetes?

Vasily: Initially, our company focused on more straightforward projects, but as we tackled more complex challenges, we realized we needed to strengthen our technical expertise. In 2018, we decided to embrace containerization, even though it was yet to be the industry standard.

The difficulty of maintaining high availability and reliable service with our limited engineering resources pushed us toward this decision. While automation tools like Ansible and Terraform have been helpful, they can no longer handle the increasing complexity of our applications.

We saw that Kubernetes could offer solutions to these challenges. We knew that investing in training our team was essential to manage the increasingly complex applications we were developing effectively.

Bart: Ronald, what specific resources or strategies did you utilize to learn Kubernetes?

Ronald: My initial approach to learning Kubernetes involved immersing myself in theoretical resources such as articles, Medium posts, and conference videos. However, I soon realized that true platform mastery required hands-on experience and practical application. The official Kubernetes documentation and the Stack Overflow community proved indispensable resources, guiding me through real-world implementation and

troubleshooting complexities.

Bart: Vasily, with technologies like Kubernetes constantly changing, how have you seen the approach to learning and staying up-to-date with these advancements evolve?

Vasily: In the past, people traditionally learned new technologies by reading books. However, with the rapid evolution of technologies like Kubernetes, information in books can become outdated within a year. Today, we rely on platforms like Google and Stack Overflow to access current information. This shift in learning methods underscores the accelerating pace of technological advancement and emphasizes the need to adapt our approaches to stay current.

Bart: Vasily, what advice would you give your younger self about learning Kubernetes?

Vasily: Reflecting on our journey, it's clear that our company made the right technological choices, but we could have benefited from embracing them sooner. We could have spent more time debating traditional and containerized approaches, slowing our progress. As engineers, we must embrace experimentation with emerging technologies, even if many of those experiments don't immediately bear fruit. The knowledge and experience gained from this trial-and-error process are invaluable for staying ahead in our

rapidly changing field.

Furthermore, I now understand the importance of actively engaging with the broader community early on. By working in isolation initially, we missed out on the wealth of knowledge and insights that come from collaborating and sharing experiences with others. Actively participating in the community would have undoubtedly accelerated our team's learning and growth.

Bart: Your recent article delved into migrating from Docker Compose to Kubernetes, exploring the reasons, methods, and outcomes of such a transition. Given your diverse clientele, could you provide specific examples of challenges you've encountered while guiding different companies through these technology shifts?

Vasily: Our clients range from emerging startups to established enterprises, each with unique needs and technical maturity. For instance, we recently partnered with a food manufacturer who sought to optimize their supplier interactions. Their system had modest performance requirements and needed to be hosted on-premises. When we introduced Kubernetes, they prioritized simplicity over advanced features, opting for a minimal setup without failover or high availability. They felt potential downtime wouldn't significantly impact their operations, making Kubernetes' overhead unnecessary.

This preference for simplicity is typical among clients with limited scalability needs. Docker Compose remains popular due to its user-friendly nature and rapid deployment capabilities. However, we consistently containerize all applications, ensuring a seamless future transition to Kubernetes if needed.

On the other end of the spectrum, we also work on projects that demand robust infrastructure solutions. These complex initiatives require substantial investment from our team. We've developed standardized Docker Compose templates to facilitate deployment for more straightforward applications that efficiently set up standard services.

Bart: One of the clients that stood out in your article is Picvario. Could you walk us through their journey, from their initial stages to where they are now regarding their app and infrastructure?

Vasily: Picvario began as a brand-new project, born from the founder's vision for a digital asset management system tailored to industries with vast media content needs, such as sports teams, news media, and museums. Their core challenge was facilitating the organization and retrieval of media assets such as historical sports photos or museum artifacts.

With limited funding, the focus was initially on the rapid development of a minimum viable product (MVP) to

secure their first enterprise client. During this phase, we prioritized adding features over building complex infrastructure. The system needed to handle essential functions like content upload, processing, and simple search. With an initial content size of around 50 gigabytes, the demands on the infrastructure were manageable.

This experience taught us a valuable lesson: infrastructure complexity should align with the product's development stage. Early on, simplicity was vital for rapid development and conserving resources. As Picvario evolved and added more features, including AI-driven image analysis and complex search capabilities across millions of assets, our infrastructure grew organically to meet these new demands.

Starting with a simple Docker Compose file managing a few services, Picvario's infrastructure rapidly expanded to 25 services within a year. This growth prompted us to reassess its scalability and performance under increased load. The diverse workloads, which now included file uploads, media processing, AI workflows, and extensive search functionalities, presented significant challenges.

Ronald and I implemented strategic enhancements to the infrastructure architecture to tackle these challenges and better support Picvario's continued growth. These changes focused on improving scalability and optimizing

resource allocation, ensuring the system remained robust and efficient to meet the platform's ever-increasing demands.

Bart: Ronald, you mentioned having 25 different services running on a single host. Please provide us with more details about the size and capabilities of that virtual machine.

Ronald: We had a robust setup, but it had its flaws. While the infrastructure was cost-effective and manageable, it lacked fault tolerance. For instance, if the virtual machine or network in a specific availability zone encountered issues, our application would fail. This single point of failure meant that problems in one component could quickly escalate and affect others. This became a critical issue when resource-intensive problems like memory leaks or high CPU usage impacted essential elements like the frontend or backend, potentially leading to downtime.

Bart: Considering these challenges, what ultimately led you to choose Kubernetes over other alternatives like Ansible and Docker Swarm? When did you start the migration process?

Ronald: Given our circumstances, it was clear that our existing infrastructure needed a complete rebuild. We carefully evaluated various options, and Kubernetes emerged as the top choice due to its widespread adoption

and robust capabilities in managing containerized applications. However, our team had internal discussions and differing opinions about whether Kubernetes might be overly complex for our specific needs, especially compared to more straightforward and potentially more cost-effective alternatives like Ansible with Docker or Docker Swarm.

After conducting thorough research and evaluation, we realized that while these alternatives could address some of our immediate challenges, they couldn't match Kubernetes regarding long-term effectiveness and scalability. Therefore, we decided to move forward with Kubernetes. However, we took our time with the migration process. We took the time to carefully plan and prepare before initiating the transition to ensure a smooth and successful implementation.

Vasily: The primary challenge during the migration stemmed from the fundamental principle of statelessness in Kubernetes. Applications within Kubernetes should not store persistent data on individual nodes or within containers. While conceptually straightforward, implementing this principle posed a significant challenge.

We encountered two specific issues. First, our application's file upload process was complex. Large files, often several gigabytes, cannot be uploaded in a

single request due to web service and browser timeout limitations. We had previously used a chunked upload approach to split files into smaller pieces, sending them individually to the backend for reassembly. This approach was practical when all files were stored on a single machine, utilizing a temporary folder. However, the transient nature of container storage in Kubernetes made relying on specific folder locations problematic.

We transitioned to Amazon S3, a reliable and scalable cloud storage solution to resolve this. We leveraged S3's multipart upload feature, which allows uploading file chunks and instructing S3 to reassemble them, eliminating the need for custom code.

The second challenge revolved around processing uploaded assets. This involved tasks such as transcoding, applying AI algorithms, and extracting metadata – all handled asynchronously by Celery workers. However, many tools used in these tasks, like FFmpeg, require file system paths. Ensuring temporary file availability on every node in a distributed system posed difficulties.

Initially, we uploaded files to S3 and had workers pull them to a temporary folder for processing, optimizing this by avoiding duplicate downloads. Eventually, we adopted a shared scratch file system, reducing delays and ensuring consistent file access.

I recommend solutions like Amazon Elastic File System

(EFS) for scenarios involving large or numerous temporary files. EFS automatically scales, integrates seamlessly with existing infrastructure, and efficiently handles large files, providing reliable access. This approach simplifies the complexity and overhead of managing temporary storage in a distributed environment. These adjustments enabled us to maintain a stateless architecture while managing our file-processing requirements effectively.

Bart: Regarding the deployment and planning of your infrastructure, how did you organize your Kubernetes clusters? Were you working with a specific cloud provider at that stage, or were you agnostic?

Ronald: Our infrastructure was already running on a cloud provider, and we intended to keep it that way. Our focus was on creating a Kubernetes cluster.

We faced two options: either deploy a Kubernetes cluster on-premises, running it on virtual machines with tools like KubeSpray or leverage a managed Kubernetes service provided by a cloud provider, like AWS EKS or Azure AKS.

A managed Kubernetes service offers a significant advantage: the cloud provider configures and maintains the control plane, the core of the Kubernetes system. Using a managed Kubernetes service frees our engineers to focus on deploying and managing applications within

the cluster and setting up essential Kubernetes components like auto-scaling, ingress controllers, and monitoring. Additionally, the cloud provider handles control plane updates and troubleshooting, significantly reducing our workload. Some research and preparation are still necessary for Kubernetes version upgrades (for instance, reviewing AWS documentation for EKS). The cloud provider substantially reduces the overall effort required.

We chose to use a cloud-managed service because it requires less time from our engineers, is easier to maintain, and is more cost-effective. For example, EKS costs around $75 per month for the high availability of the control plane.

Bart: With 25 services deployed in both test and production environments, did you use any templating engine for managing configurations?

Ronald: We used Helm charts to streamline our configuration management. These charts significantly reduced the number of configuration files we needed by allowing us to create several universal, custom-built charts that could be used across all environments and application components. This approach simplified our workflow and aligned with the "don't repeat yourself" (DRY) principle, promoting efficiency and maintainability.

During the migration, we also took the opportunity to update our CI/CD tools and practices. Before, we relied on TeamCity to build and deploy Docker images. After the migration, we continued using TeamCity for image building. We switched to ArgoCD for deployments, streamlining the process by having ArgoCD manage the deployment of our Helm charts to the Kubernetes cluster.

Bart: Your article highlighted Argo CD as an essential tool for a new Kubernetes cluster and discussed the GitOps push and pull models. How did you use these models in your migration process?

Ronald: There are two distinct deployment models. The push model involves the CI/CD tool directly executing deployment steps. It has access to the environment and applies changes directly, for example, using commands like `helm install` or `kubectl apply`. This model was our approach before the migration, where TeamCity executed Docker Compose commands on virtual machines via SSH.

In contrast, the pull model, central to GitOps, involves an operator like Argo CD or Flux running within the Kubernetes cluster. This operator continuously monitors the state of your Git repository and automatically synchronizes and updates your application whenever it detects changes.

Each model has pros and cons, so we opted for a hybrid

approach: we continue to use TeamCity for building images, but it now triggers Argo CD to synchronize applications by passing the image ID to it. This mixed model combines the strengths of both models.

Bart: What were the results of all this work? What feedback did you receive from the client, and what were the key takeaways from your discussions?

Vasily: The outcome was very positive—our primary challenge before the migration was handling high volumes of content uploads. The previous system would become overwhelmed when multiple clients uploaded large amounts of data, leading to overloaded queues and significant processing delays. This lag understandably frustrated clients, especially those in time-sensitive fields like sports photography, who needed their uploads processed immediately.

However, the migration to Kubernetes and cloud services effectively addressed these issues. The cloud's inherent scalability allowed us to dynamically adjust resources based on demand, showcasing the ideal use case for cloud technology to efficiently manage costs and resource allocation. Kubernetes further streamlined resource management by enabling us to define scaling policies and triggers.

As a result, the quality of service improved dramatically. We no longer received complaints about delays, and the

system efficiently handled even peak demands. This success validated our investment and effort into the migration, proving that the solution met and exceeded our initial goals.

Bart: How did you approach communicating these technical decisions to the client, especially given their initial hesitation about Kubernetes?

Vasily: We believe in transparency and open communication with our clients. However, we also understand that explaining the benefits of complex technologies like Kubernetes to non-technical stakeholders requires a different approach.

Clients are primarily concerned with two things: cost and value. To address these concerns, we focus on building trust and clearly articulating the value proposition of our proposed solutions. Building trust means demonstrating our expertise and ability to make informed decisions that lead to efficient project management and cost control.

Our value proposition needed to be more compelling initially. We highlighted benefits like improved availability and reliability, but these weren't immediate concerns for the client then. They had a manageable number of users, and their system could handle downtime.

However, as their user base grew and the problems with the existing system became more apparent, the value

proposition of Kubernetes became clearer. We showed how migrating to Kubernetes would directly address the issues impacting their business and user satisfaction. This tangible value made the investment in Kubernetes much more justifiable.

The key is to translate technical benefits into concrete business value. If you can demonstrate how a solution will solve specific business problems and improve the client's bottom line, they are much more likely to see the value and approve the investment.

Bart: What were your main goals in writing the article, and how were the reactions?

Ronald: My primary goal was to give back to the community by sharing my experience and insights. During the migration, I heavily relied on community resources, so I wanted to contribute my lessons learned. Writing the article was also a personal challenge for me to explore technical writing. It was my first attempt, and I plan to continue writing.

Additionally, the article serves as documentation for new team members, providing an overview of the project, our infrastructure decisions, and the reasoning behind those choices.

Bart: Vasily, do you have plans to continue writing blogs or start your podcast to share your experiences?

Vasily: We plan to write more about our ongoing

projects. Since our last update, we've implemented custom metric scaling and addressed complex multi-tenant architecture issues. We also have exciting developments at Picvario. Ronald is working on large-scale infrastructure projects with a global reach, and we aim to share these experiences with the community.

Ronald, please elaborate a bit on these upcoming projects.

Ronald: To add to Vasily's earlier mention, we're currently utilizing CloudFormation, but we're planning a strategic shift to Terraform. Let me clarify the reasons behind this decision.

Besides my involvement in Picvario, I'm also deeply engaged in another project that involves AWS infrastructures and Kubernetes. In the context of Picvario, our responsibility lies in maintaining the infrastructure for a single, large-scale application across various environments. However, my other project presents a different challenge: managing many small to medium-sized applications with dynamic lifecycles. These distinct applications operate in separate EKS clusters based on specific customer requirements. This challenge necessitates swiftly launching, updating, migrating between accounts, or removing applications and their associated infrastructures.

We currently manage 15 applications, and we anticipate

this number will grow. Consequently, automation becomes paramount in this scenario. While we currently leverage CloudFormation—AWS's native infrastructure as code tool—there are compelling advantages to transitioning to Terraform and Terragrunt.

Despite CloudFormation's strengths, the combination of Terraform and Terragrunt offers greater efficiency and a more user-friendly experience. Terragrunt empowers us to define our Terraform code once and then customize parameters like variables for the application, environment, region, or AWS account level. This approach allows us to adhere to the DRY (Don't Repeat Yourself) principle in our automation configuration, a feature currently absent in CloudFormation.

Although we're still in the initial phases of this transition, the benefits of this approach are already becoming evident.

Bart: Besides writing articles and working on technical projects, you're also a guitarist. Please share a bit about your journey with the guitar, Ronald.

Ronald: I play electric guitar and have been playing for about two or three years. I take weekly lessons, learning new techniques and songs I enjoy. I mainly play metal and some indie rock, recording these pieces for myself.

Bart: As a drummer, I can appreciate the intensity and precision required in metal drumming. Vasily, can you

tell us more about your involvement with music?

Vasily: I'm essentially a one-person band—I produce music, write lyrics, sing, and handle all aspects of my music creation. It's fascinating how technology has revolutionized music production. Modern tools like plugins and sound design techniques have opened up many creative possibilities, much like how technologies like Kubernetes have expanded our capabilities in the tech world. Staying current with advancements in both fields is essential for continuous improvement and innovation.

Bart: Ronald, do you remember how your guitar playing sounded when you first started?

Ronald: I recorded my early practice sessions and uploaded them privately to YouTube. Looking back, it’s clear how much I've improved since then, though I might not want to share those early recordings publicly!

Bart: If people want to connect with you or learn more, what’s the best way to reach you?

Ronald: The best way to connect is through LinkedIn. You can find us there easily by searching for our names.

Chapter 9

Outpacing the Optimization Challenges in Kubernetes

Tortoise: Outpacing the Optimization Challenges in Kubernetes, Kensei Nakada

Optimizing resource management for hundreds of Kubernetes microservices is no small feat, but Kensei, a seasoned Software Engineer, and Tortoise's creator, took on the challenge headfirst. Drawing on deep technical expertise and an inventive approach, he and his team developed Tortoise to streamline autoscaling, offload complexity from developers, and even explore cutting-edge Wasm extensibility within Kubernetes.

Bart Farrell sat down with Kensei to unpack how he’s transforming Kubernetes optimization—without overwhelming service developers.

You can watch (or listen) to this interview here.

Bart: What are the top three up-and-coming Kubernetes tools you're excited about?

Kensei: My top choice is the Gateway API, which originates from SIG Network and stands out for its unique framework-based structure. This versatile API framework offers various controllers, such as Envoy Gateway, Kong Gateway, and Istio Gateway, each implementing the interface, allowing users to choose the

implementation that best suits their needs. For example, users who prefer Envoy can opt for Envoy Gateway, while others may select Kong or Istio Gateways based on their specific requirements.

In second place, in-place pod resizing is a long-awaited feature in Kubernetes. Traditionally, resource requests and limits for Kubernetes pods have been immutable once a pod is created, which limits flexibility. This new feature introduces a significant improvement by enabling modifications to these resource parameters after a pod starts, which enhances Kubernetes' auto-scaling capabilities. The Vertical Pod Autoscaler (VPA) will integrate with this feature, dynamically adjusting pod requests based on real-time resource consumption—a promising advancement that has been anticipated for years.

The third project that excites me is SpinKube, a serverless Kubernetes solution built on WebAssembly (Wasm). While Wasm was initially developed for web browsers, its potential within Kubernetes is now being actively explored. SpinKube is particularly promising because Wasm modules start and stop much faster than traditional containers, allowing more efficient resource usage. Given its rapid startup and shutdown times, SpinKube could offer resource savings and operational efficiency.

Bart: For those unfamiliar with you, could you tell us a bit about yourself, your work, and where you're based?

Kensei: I'm Kensei Nakada, a software engineer based in Japan.

Bart: How did you transition to Cloud Native technologies? What was your background before that?

Kensei: My journey began with Google Summer of Code in 2021, a mentoring program for open-source projects. I was selected for the Kubernetes project, where I developed a Kubernetes scheduler simulator. This experience connected me with the Kubernetes open-source community.

Following that, I contributed to the Kubernetes scheduler, not just the simulator but the scheduler itself. At the time, my professional work barely involved Kubernetes, but these contributions sparked my interest in Kubernetes and Cloud Native technologies. This experience inspired me to explore other job opportunities.

Before Cloud Native, I was a backend engineer primarily working with Golang. Mercari offered me to join as a backend engineer on their search team. However, my interest in Kubernetes and Cloud Native grew, so I requested a transfer to the platform team, which led to my transition to Cloud Native.

Bart: With the fast pace of change in Kubernetes and the

broader Cloud Native ecosystem, how do you stay up-to-date?

Kensei: My primary focus is staying updated with upstream Kubernetes developments. I don’t actively follow other Kubernetes tools until they gain significant attention like SpinKube recently did. I stay abreast of new features and updates mainly through my contributions and peer reviews, which often involve examining the core of Kubernetes itself.

Additionally, I follow LWKD (Last Week in Kubernetes Development), a popular resource that is published weekly and summarizes ongoing developments in Kubernetes.

Bart: If you could go back in time, what would be the most important career advice you'd give to your younger self?

Kensei: I would advise my younger self to start learning English earlier to avoid some of the language challenges I face today. For instance, speaking at conferences can be daunting, especially when I worry about not understanding questions due to language limitations. This language barrier is something I still encounter, even recently, when working with Tetrate, an American company. Learning English sooner would have eased many of these situations.

Bart: As part of our monthly content review, we came

across your article titled *Tortoise: Outpacing the Optimization Challenges in Kubernetes at Mercari*. First, can you explain your role at Mercari and what responsibilities your team handled as platform engineers?

Kensei: At Mercari, the platform team is divided into four sub-teams: network, CI/CD, infrastructure, and platform. I work within the platform infrastructure team, which is responsible for the foundational elements of our platform, such as managing Kubernetes clusters and providing cloud infrastructure. The architecture we manage is predominantly cloud-based and heavily relies on Kubernetes. Additionally, we are accountable for observability, which, alongside managing our infrastructure, forms our team's core responsibilities.

Mercari has also embraced a company-wide initiative to spin off applications onto independent platforms. Since almost every Mercari application runs on our platform, any infrastructure change we implement can significantly impact all applications. Our team's work on the platform is crucial to this initiative, as we play a key role in supporting Mercari's scaling needs.

Bart: Kubernetes can be expensive to run at scale. How do you approach overall cost optimization at Mercari, and what strategies have been most effective for you?

Kensei: Our approach to Kubernetes cost optimization

involves node-level and pod-level strategies, with each platform sub-team focusing on optimizing the components under their purview.

The platform infrastructure team, for example, is tasked with optimizing computing resources and observability tools. My primary focus has been computing resource optimization, specifically at the node level, where we select the most efficient instance types and configure the Cluster Autoscaler. For instance, we prioritize using spot instances, which are significantly more cost-effective than standard on-demand nodes. We also implemented T2D instance types on GCP, which has led to substantial cost savings. Because managing infrastructure is our core responsibility, implementing these infrastructure-level changes is more streamlined for us.

On the other hand, pod-level optimizations involve adjusting resource requests, limits, and auto-scaling configurations specific to each service. Service developers, not the platform team, are typically responsible for monitoring and fine-tuning resource consumption for their respective services. This division of responsibilities allows us to address cost optimization at infrastructure and service levels effectively.

Bart: Optimizing applications indeed requires an in-depth understanding of the applications themselves.

Kensei: Ideally, developers are responsible for

optimizing their services, but effective optimization requires deep Kubernetes expertise, which not all service developers possess. This is why we have a dedicated platform team to manage Kubernetes, building tools, and an abstraction layer so developers can work with Kubernetes without needing extensive knowledge of its complexities.

As Kubernetes specialists, we do need to be involved in pod-level optimization. However, given the scale of Mercari's infrastructure—with over 1,000 microservices and numerous departments—it's unrealistic for the platform team to work individually with each team on optimization. Instead of directly involving each team's efforts, we focused on reducing the engineering burden associated with optimization and encouraged each team to handle their optimization tasks.

To support this, we began by documenting best practices for optimization, although we found that more than documentation was needed, as developers still had to follow all necessary steps manually. To further streamline the process, we developed the Slack Resource Recommender, a Slack bot that recommends optimal resource requests based on the service's resource consumption history.

Bart: Could you elaborate on how the resource recommender Slackbot works? Is it based on the

VerticalPod Autoscaler?

Kensei: Slackbot is built around the VerticalPod AutoScaler (VPA), a subproject from SIG Scheduling. VPA monitors each service's resource usage and calculates the optimal resource requirements based on past consumption. It then adjusts each pod's resource requests accordingly. While VPA is an effective tool, we found it can disrupt services. Instead of using it directly, we created a Slackbot that recommends resource adjustments based on its logic.

Here's how VPA works: when it needs to change the resources for a pod, it evicts and deletes the pod so that the ReplicaSet can create a new one. During this process, the VPA's mutating webhook changes the resource request for the new pod. While this method effectively applies VPA's recommendations, it also results in service interruptions because of the pod replacements. Another issue is that frequent pod restarts can occur if the recommendations change often. These drawbacks led us to refrain from using VPA directly.

So, we built the resource recommender bot based on VPA. Instead of acting automatically, it collects a month's worth of VPA recommendations and sends them monthly as suggestions via Slack. This way, people can consider the recommendations without worrying about sudden changes or disruptions. A bot that provides such

advice on resource requests and limits would appeal to many.

Bart: Are there any potential downsides or challenges people should consider regarding the Slackbot?

Kensei: We encountered several challenges with the resource recommender. The first major issue was the accuracy of the recommendations. The suggested resource values can quickly become outdated since they are based on the VPA's one-month history. Mercari's applications evolve constantly, with implementation changes or shifts in traffic patterns, which can affect the relevance of these recommendations. The recommendations are accurate when generated, but if someone applies a week-old recommendation, there's a risk of running out of memory.

The second issue was engagement. Users may ignore the recommendation messages we send, as deciding whether to apply the suggestion is up to them. Due to the potential for outdated information, the recommendations aren't foolproof; they act more like hints, leaving users unsure about their reliability. Each team must verify if a recommendation still aligns with current resource usage and ensures safe application, which requires engineering time. Over time, interest in the recommender messages has declined, and some users ignore them.

The final challenge was a fundamental one: optimization

is a continuous process. It’s not just the recommender; optimal resource values must be re-evaluated with every application change. Developers are, therefore, in a cycle of constant tuning, as optimization never truly ends. Given Mercari’s large number of microservices, this ongoing effort accumulates significant engineering time, making it a burden to sustain manual resource optimizations.

Bart: Despite the challenges, do you find that using the bot ultimately results in worthwhile cost savings?

Kensei: Having the bot is undoubtedly better than not having it at all, but, in practice, it didn’t have as much impact as we’d hoped. Mercari’s extensive use of Horizontal Pod Autoscaler is a big reason. HPA is used to manage scaling based on CPU utilization, and we typically set a target of 80%. When the service’s CPU usage exceeds this threshold, HPA automatically increases the number of pods.

The platform team has documented best practices for Kubernetes, and we recommend using HPA for CPU scaling. Most large services rely on HPA, with hundreds of HPAs active in the cluster. However, when HPA manages the CPU, the recommender bot has a limited impact on CPU optimization.

For example, if an HPA is set to target 80% CPU utilization and the service consistently uses only 8 out of

10 cores, the recommender might suggest reducing the CPU request to 8 cores. However, this doesn’t optimize resource utilization because HPA will target 80% of the new, reduced eight cores. In other words, changing the CPU request alone doesn’t alter how HPA manages CPU usage. This is why the recommender cannot effectively optimize the CPU when HPA is in control. While HPA automates CPU scaling, it doesn’t ensure optimal CPU utilization. Although Mercari relies heavily on HPA, its CPU parameters—and sometimes the resource requests—must be fine-tuned to achieve the best performance.

Bart: Given the recommender bot's limited impact on CPU optimization when HPA is involved, how did you tackle these challenges?

Kensei: Initially, we considered adding HPA recommendations to the resource recommender bot, but we realized it would face the same issue as before—users tend to ignore these messages. This led us to explore a different approach to streamline the optimization process. The core challenge is the sheer number of parameters that require tuning for each service: HPA parameters, resource requests, and resource limits. With so many microservices and configurations, it’s impractical to expect service owners to optimize every service manually, as each adjustment demands expertise and significant effort.

We then considered automating these optimizations as much as possible to reduce the manual workload. While VPA offers automated tuning, we encountered issues and limitations, leading us to refrain from using it directly. We considered creating a wrapper or alternative to VPA to address these challenges. For HPA optimization, we identified multiple scenarios that can lead to inefficiencies and devised ways to handle these situations. We began envisioning an automated solution to detect and address these inefficiencies in HPA settings.

These ideas culminated in the creation of Tortoise, an open-source project designed to replace both HPA and VPA by handling optimizations automatically. Tortoise is structured to require minimal user input; it only exposes a few necessary parameters, such as the target deployment name, so users aren’t burdened with additional tuning. Once implemented, Tortoise eliminates manual optimization adjustments, creating a self-managing system. This was the vision we set out to achieve with Tortoise—an environment where optimization happens seamlessly, without ongoing intervention.

Bart: You mentioned that Tortoise functions as a wrapper. Can you explain this in more detail? How does it operate in terms of CRDs and data storage?

Kensei: Tortoise introduces a Custom Resource Definition (CRD) called "Tortoise," which manages HPA and VPA behind the scenes. When a user creates a Tortoise instance, HPA and VPA are automatically set up for the specified targets. However, Tortoise is more than just a setup shortcut; it manages and optimizes these autoscalers to ensure efficient resource utilization that HPA and VPA alone can’t achieve.

For example, while HPA often requires manual tuning for optimal CPU utilization, Tortoise is designed to handle this optimization automatically once created. This is the crucial distinction between Tortoise and the underlying autoscalers—it abstracts away the complexities of HPA and VPA management. With Tortoise, users can focus solely on the parameters Tortoise exposes without worrying about the underlying optimization, as Tortoise handles it all.

Bart: Regarding standard tools for collecting and recommending metrics, we often think of Prometheus, Metrics Server, and HPA for monitoring and observability. Does Tortoise aim to replace all of these?

Kensei: Tortoise replaces both HPA and VPA but doesn’t replace tools like Prometheus or Metrics Server. VPA’s recommendations are based on P90 or P95 resource consumption, reflecting typical 90th or 95th percentile usage. Tortoise relies on this historical consumption data

to make its recommendations but doesn’t store any of it directly. Instead, it refers to VPA as a data source for past consumption.

While Tortoise takes over HPA and VPA functionality, it occasionally relies on VPA to generate specific recommendations. VPA needs a Metrics Server for real-time consumption data and Prometheus to store its recommendation history. So, Metrics Server and Prometheus are indirect dependencies for Tortoise, but it doesn’t aim to replace them—only to use the data they provide.

Tortoise allows users to provide input via its Custom Resource Definition (CRD), "Tortoise." This CRD lets users define specific parameters like the target service name or department and configure the scaling method for each resource. For example, you can set Tortoise to scale a container’s CPU horizontally and its memory vertically, with a separate scaling strategy per container and resource.

Tortoise continuously pulls recommendations from VPA for vertical scaling, then calculates and applies a stable recommendation to the pods. While it operates similarly to VPA, Tortoise reduces the frequency of recommendation changes to minimize disruptions, which was one reason we opted not to rely on VPA alone. Another significant difference is that Tortoise performs a

rolling upgrade instead of evicting pods. This upgrade process is similar to using the Kubernetes CLI command `kubectl rollout restart`, which restarts a deployment while adhering to a rolling update strategy. For instance, if a deployment has three replicas, VPA might delete one or two pods based on the Pod Disruption Budget (PDB) to replace them, leaving fewer replicas to handle the traffic temporarily. To avoid this, Tortoise adds a new replica with updated resources before removing the old one, ensuring traffic stability throughout the update.

Tortoise also supports certain Golang environment variables. Since Mercari relies heavily on Golang, specific environment variables must be updated alongside resource requests, a feature VPA doesn't support. Tortoise addresses these variables in its vertical scaling. It continuously optimizes HPA parameters for horizontal scaling, allowing HPA to scale pods as needed. Though calculating HPA's target utilization is complex, documentation is available in the Tortoise repository for anyone interested.

In addition to HPA parameters, Tortoise optimizes cases where resource requests affect HPA's efficiency. Achieving optimal CPU usage requires aligning CPU requests and HPA parameters, a task often difficult to manage manually. Tortoise handles these complexities automatically, reducing the need for manual intervention.

Tortoise manages horizontal and vertical scaling, and in Mercari's fast-paced environment, frequent migrations are common. From its design phase, Tortoise was built to simplify migration. Since many services already have HPA configured, a new HPA from Tortoise could create conflicts. To avoid a complicated process of deleting and recreating HPAs, Tortoise includes an option to attach existing HPAs. This approach significantly streamlined migration, allowing developers to migrate their HPAs to Tortoise by merging a pre-configured pull request.

Despite these efforts, we faced challenges with adoption. While the migration process was smooth, many developers hesitated to adopt Tortoise because it was a new tool. Teams often wanted to see other services successfully using Tortoise before migrating, creating a "wait-and-see" effect.

To address this hesitation, we held open sessions to introduce Tortoise, explain why it was necessary, and share how it addressed challenges with resource recommendation and optimization. These sessions helped secure early adopters, and as more teams began using Tortoise, we showcased their successes to encourage wider adoption. Ultimately, these migration efforts aim to achieve a complete transition from HPA to Tortoise across Mercari's infrastructure.

Bart: Are there any other reasons you think made

Tortoise so successful at Mercari?

Kensei: A key advantage of Tortoise is the shift in responsibility it enables. By exposing only a few parameters to users and managing underlying autoscaling automatically, Tortoise transfers the responsibility for pod optimization from individual service developers to the platform team. This means optimizing autoscaling and resources for each pod is now the platform's responsibility. If Tortoise doesn't fully optimize a microservice, it's on us, not the service developers. In that case, we work to improve Tortoise to meet their requirements better. This shift relieves service developers from managing per-service resource optimization and places it within the platform team's scope.

Technically, many of Mercari's microservices are gRPC or HTTP services, primarily written in Golang. Most are based on a standardized internal template for microservice implementation. This consistency allows us to create a unified solution that optimizes all microservices.

Bart: Looking back, is there anything you would approach differently in the design or implementation of Tortoise?

Kensei: Looking back, I would have focused on creating a more streamlined internal design to make Tortoise

easier to maintain and contribute to. While we made the interface user-friendly by exposing only essential parameters, like the microservice name, the underlying processes for generating recommendations and adjusting pod resources are intricate. I recognized that maintaining this complexity might be challenging for the team, so I worked to pass on as much knowledge as possible before leaving Mercari. Simplifying these internal components would have paved the way for smoother collaboration and long-term development.

Bart: Was there something specific about tortoises that inspired you?

Kensei: I have two pet tortoises at home, Azuki and Okada. They're Japanese names. During Mercari's internal hackathon, I created Tortoise as an experimental project and got to choose the name. There wasn't any deeper meaning behind it—I like turtles!

Bart: You're also the author of the KubeScheduler Wasm extension. How did it start?

Kensei: Currently, the scheduler has two main extensibility options: a Webhook-based approach and the Go-Plugin SDK. While both provide flexibility, they also come with certain limitations. To offer a more user-friendly way for developers to extend their schedulers based on custom use cases, we began exploring WebAssembly (Wasm) as a new option. This is the first

time the Kubernetes community is attempting Wasm-based extensibility in the official ecosystem. Envoy has implemented a similar Wasm runtime concept, but this approach is new for Kubernetes.

Our scheduling team has gained valuable insights on Wasm, and we're considering sharing these findings with other teams with similar interests.

Bart: What are your upcoming goals and priorities?

Kensei: I've just started a new role at Tetrate, so my main focus is getting onboarded. We're also working on some significant internal improvements to the scheduler on the open-source side. If you notice any issues with the new scheduler binaries, there's a chance I might be responsible for those!

Bart: For people who want to reach out, what's the best way to connect?

Kensei: You can find me on X (formerly Twitter) and LinkedIn—feel free to reach out.

Chapter 10

GitOps at Scale

How We are Managing a Container Platform With Kubernetes at Adidas, Ángel Barrera

Managing a container platform for thousands of developers is no small task, but it's an exciting challenge for Ángel Barrera, Senior Platform Engineer at Adidas. From pioneering tools like Virtual Cluster and Crossplane to transforming the company's infrastructure with GitOps, Ángel is at the forefront of Adidas's cloud-native evolution. He has streamlined cluster management, enhanced deployment processes, and boosted transparency and accountability—all while navigating the complexities of a global retail giant.

Bart Farrell catches up with Ángel to unpack the strategies, tools, and insights powering Adidas's cloud-native journey.

You can watch (or listen) to this interview here.

Bart: Could you tell us about three Kubernetes tools that have grabbed your attention lately?

Ángel: Virtual Cluster is a game-changer. Multi-tenancy in Kubernetes has always fascinated me. Typically, we use namespaces to separate various parts of a cluster, but with Virtual Cluster, you can create isolated control

planes on top of a shared console plane. This separation makes it ideal for scenarios like A/B testing different Kubernetes versions or letting teams try out newer releases without impacting the central cluster. It's also perfect for users with older dependencies since they can spin up a virtual cluster just for their needs.

Next is Crossplane, a tool making significant strides in the infrastructure space. Crossplane bridges platform users and owners by allowing users to request infrastructure—like databases or network components—fully declaratively.

Meanwhile, platform owners can build reusable and flexible customizable modules. I first tested Crossplane a couple of years ago when it was still rough around the edges, especially with etcd and CRDs, but it's come a long way since then, thanks in part to some friends working at Upbound, the company behind it.

The third tool I'm excited about is Service Mesh, especially those using eBPF for packet interaction. This technology allows precision in use cases like observability and tracing by working directly at the kernel level, bypassing container or application-level integration.

Bart: Please tell us more about your background, role, and where you're based.

Ángel: I'm Ángel Barrera, based in Madrid, Spain, and

work at Adidas as a Senior Platform Engineer. Adidas is a global retail giant, and my role is to support and enable our development teams. With about a thousand developers worldwide, managing our container platform is a huge responsibility, and my work centers on making that process as seamless as possible for our teams.

Bart: And how did you come to be a Platform Engineer? What were you doing before?

Ángel: I came on board at Adidas as a Platform Engineer and eventually stepped into a senior role. Before that, I was at a startup where I wore many hats, handling a bit of everything. At Adidas, I now focus on managing and improving our platform while supporting teams across the organization.

Bart: What first drew you into the Cloud Native world?

Ángel: I can't recall the exact year, but it was around 2016 or earlier. At that time, I worked as a Java architect for one of Spain's largest banks, heading backend architecture. We were transitioning from SOAP to RESTful APIs and JSON, shifting the whole architecture to be compatible with containers. That's when I stumbled upon Kubernetes, OpenShift, and the cloud. AWS was my first introduction to cloud services, and I was part of the team that decided on the bank's platform, evaluating options like OpenShift for Kubernetes or DC/OS from Mesosphere. The cloud's capabilities and the power of

container orchestration opened my eyes.

As a developer, I was already familiar with Docker since it was part of our daily toolkit. However, this was the first time I saw the bigger picture—continuous deployment, cloud-native patterns, and Kubernetes—all while working in a massive banking environment.

Bart: Would you say that experience boosted your cloud-native career?

Ángel: Around that time, I also teamed up with my friend Paul Rosselló, now at Giant Swarm, to build a SaaS project. Those days were full of passion—we developed something groundbreaking. It was essentially a "namespace-as-a-service" model, where we set up hard multi-tenancy on Kubernetes at a time when no such solution existed. We connected with folks from Google, AWS, IBM, and Docker. Other companies became interested in our solution, but sharing namespaces within a Kubernetes cluster didn't hold up for multi-tenancy. The real win came from creating multi-tenancy by managing separate clusters. It felt like we were taking on giants back then, and it was an unforgettable journey.

Bart: The Kubernetes ecosystem moves fast. How do you stay current with all the changes? Do specific resources like blogs and YouTube channels work best for you?

Ángel: Keeping pace with the CNCF ecosystem,

especially Kubernetes, is challenging. I follow key industry accounts on X (formerly Twitter). I also monitor updates from the Learnk8s community and CTOs like Darren Shepherd from Rancher and Henning Jacobs from Zalando.

Sysdig is another go-to; they regularly publish concise blogs on new Kubernetes versions, highlighting the essential changes without diving too deep into release notes. Kubernetes release notes are full of information, but they can be overwhelming. Sysdig distills this into what matters. Besides that, I occasionally check LinkedIn for professional updates, though it's more job-oriented. Startups like Uber and Lyft also have fantastic tech blogs that are worth reading.

Bart: If you could give your younger self one career advice, what would it be?

Ángel: Push as hard as possible. In your 20s and early 30s, you have the energy to pursue big goals. Later, with more job and family responsibilities, you don't have the same freedom or time. Your brain also feels different in your 20s than in your 30s! It's rewarding to see your work's impact, like improved performance in your company's products, but keeping up with the pace of change is a real challenge as time goes on.

Bart: In your article "How We Are Managing a Container Platform," you mention a specific date, May

10th, 2022. What happened on this date?

Ángel: On May 10th, 2022, we made our first commit in the GitOps repository. This date marked the moment we fully committed to the GitOps approach. Before that, we tested various configurations, explored different project structures, and figured out what would work best for us. Our first commit to the repository established the foundational project structure, and we built it up further.

Bart: Could you give us an overview of the existing infrastructure setup and practices before you moved to GitOps?

Ángel: In 2018, Adidas had two people in charge of building the container platform. They envisioned managing everything as code across multiple repositories, even though there wasn't yet any Flux, DevOps, or multi-cluster configuration support. They had to get creative, developing a structure to manage multiple clusters in a large corporate setup with all its constraints.

Starting with five to ten on-prem clusters, they organized configurations with one Git repository per cluster, plus a centralized repository for shared configurations.

They created branches for different purposes. For example, ingress configurations were in separate branches and environments like development and production. Jenkins ran pipelines that merged

configurations and applied them to the clusters. It worked well and given the tools available in 2018, it was a solid solution—awe-inspiring work from those two team members, as it kept things running until 2022.

Bart: And what was the deployment strategy before the migration? Was it all GitOps?

Ángel: It was a push model where Jenkins merged and applied the configuration directly to the clusters, not GitOps. However, since the team had that "everything as code" approach from the beginning, transitioning to a GitOps structure was straightforward. While the configuration was all in code, the deployment process has significantly transformed with GitOps.

Bart: Did this setup expand naturally over time, or was there a structured plan guiding it from the beginning?

Ángel: There wasn't a structured approach from the start. The team initially set up five to ten on-prem clusters at Adidas. However, with the addition of AWS, our cluster landscape expanded from a single location in Germany to a global presence. Instead of a few large clusters, we have around a hundred smaller ones distributed across America, Europe, China, and other regions.

While the original configuration approach worked well for a handful of clusters, managing close to a hundred clusters became too much. When I joined in 2021, it was clear we needed a more scalable solution, so I connected

with R&D to explore alternatives.

Bart: What was the next step? Did you start brainstorming how the deployment process would work?

Ángel: I first collaborated with a colleague to define the requirements and constraints for this new DevOps approach. Being a global company, we needed to manage Adidas's regional specifics. We needed flexibility in our GitOps setup to accommodate different configurations for regions like China, which operates on a unique calendar with various restrictions, such as freeze periods.

Together, we outlined the essential features we needed in GitOps to match our legacy management process, drew up a plan, and established KPIs to measure our progress. A primary goal was to avoid the inefficiency of opening dozens of PRs to update multiple clusters—each requiring reviews. In the old setup, we had one repository per cluster, so any global change meant opening up to 50 PRs, which was time-consuming and prone to human error.

Another critical issue was that Jenkins struggled to run 50 pipelines simultaneously, leading to bottlenecks. If you were outside a maintenance window due to a full queue, your configuration could delay the update, causing inconsistencies across clusters. We set KPIs to reduce the number of PRs, minimize configuration

errors, and improve the scalability of our setup, aiming for a more resilient and efficient process.

Bart: It's one thing to set a plan and metrics, but what was it like putting everything into action? Did any tools or practices need to change along the way?

Ángel: The first significant shift was moving away from Jenkins for deployments. We still use Jenkins to validate configurations, checking that everything looks as it should and producing a difference between the intended configuration and what currently resides in the cluster. Now, Jenkins is strictly for CI tasks, focusing on validation rather than deployment. Moving away from Jenkins was a considerable effort, and the migration process was challenging, but now it's running smoothly, and we're delighted with the results.

We also began setting up new clusters using Flux and gradually transitioned older clusters to a GitOps model. It was a careful process; we wanted zero downtime or issues, and it went well. Since everything was already managed with Helm charts, knowing the chart states and values helped us switch from a push to a pull model with Flux. There were risks, especially with production workloads, but the transition went smoothly.

Bart: What advantages have you seen with the pull-based mechanism you transitioned to?

Ángel: The benefits are incredible. Now, we have

complete visibility into every change before it’s applied. Previously, our method of merging configurations from multiple repositories was somewhat improvised, which made it hard to track what would change. You'd open a pull request (PR) with a set of changes, but due to issues in the pipeline, it was challenging to be sure those changes would be applied as intended.

With this new setup, each pull request displays an exact "diff," or list of changes, thanks to continuous integration (CI) checks that compare the PR against the current cluster configuration. This transparency has been essential in demonstrating the value of the migration to our managers.

Another significant gain has been increased confidence in the deployment process. We're no longer relying on Jenkins to apply configurations to clusters. Instead, each cluster pulls its configuration directly from Git, making the process much faster and more reliable. Any sync issue immediately triggers an alert, something we didn't have before. We even have a dashboard that straightforwardly displays the platform's status—green means everything's running fine, and red means there's an issue. The shift to GitOps has proven to be more than worth the effort.

Bart: How has the new setup influenced transparency and accountability in your deployment process?

Ángel: Managers and stakeholders now have complete visibility on changes, which has boosted their confidence in our team. Previously, we had some friction whenever we needed to deploy urgent updates, like critical system upgrades. Without complete confidence in the Jenkins setup, conveying that assurance to stakeholders was hard, and it sometimes held up deployments.

With this new setup, we can give them precise details about what's changing, when, and how it will impact specific regions or components. With that insight, understanding and approving changes is much easier for them.

We also built a changelog around this process. Like any large company, Adidas requires detailed auditing records connected to our project management tools—in our case, Jira. Every change now has a timestamp, a person associated with it, and links back to the PR and diffs. This traceability was missing before. Everything is fully auditable, significantly benefiting security and business compliance.

Bart: Can you walk us through how you provision the clusters and integrate them into the GitOps workflow?

Ángel: It's pretty straightforward. We start by creating a managed cluster, whether on GKE or EKS—any provider works similarly. Once the cluster is ready, we inject secrets to access our private Git repository, so we

use tokens to secure access to BitBucket. After the secrets are in place, we install Flux with standard commands like `flux install`.

We then set up two key objects: the Git repository and the Kustomization. The cluster then syncs with GitOps, and that's it. There's still some room for improvement, though. For instance, connecting a cluster to the Adidas network requires a few manual steps, but we're working on automating that to make it part of the standardized setup. We'll close that gap in the next few weeks or months.

Bart: How about your multi-tenancy model—has it evolved since migrating to this setup?

Ángel: It mostly stayed the same. We have two different multi-tenancy approaches based on the team's needs. For instance, we have "power users" who control a dedicated cluster. These are usually expert teams, like the monitoring or API teams, who may need cluster-level access to test a new controller before rolling it out globally. They can request a whole cluster for their work. Then, we have the more common scenario: developers of various applications at Adidas. These teams only have access to specific namespaces rather than the entire cluster. We handle their permissions with role bindings, so they're limited to their assigned namespaces and don't see the underlying cluster configurations.

The setup doesn't interfere with developers' work—they focus on deploying their applications, leaving cluster management to us. We encourage them to adopt GitOps practices, but they ultimately choose whether to follow our guidelines or deploy directly from their laptops if they prefer.

Bart: How has the developer feedback been since they started using the platform for production deployments?

Ángel: Feedback from our internal developers and platform teams—our primary users—has been positive. Our end-users might not see the details of our cluster management efforts, but our platform users certainly do, and they're eager to adopt our approach. Their enthusiasm alone strongly signals that we're on the right path.

In platform engineering, we have various specialized teams—monitoring, API, and security, for instance—and they've all recognized the benefits of our cluster management setup. They see that it's more flexible, scalable, and reliable, which are critical needs in a large organization like ours.

Previously, teams managed their deployments independently, choosing their deployment strategies. Now, seeing the strengths of our standardized approach, they want to be onboarded into our system. For example, if the monitoring team wants to roll out a new version of

Prometheus across clusters, they can now follow a unified, streamlined deployment process.

These teams are joining our DevOps repository using the same container orchestration methods we use. They create pull requests, review changes, and follow the same standardized processes, which has brought a lot of alignment across platforms. This consistency makes workflows easier and strengthens alignment between technical and business stakeholders.

Bart: How did the business side respond to the migration? Was it hard to get their buy-in, and what was that experience like?

Ángel: It was immediately clear to everyone that we had a problem—periodic issues due to misconfigurations led to operational challenges. We needed a change and knew we had to approach it strategically.

With our previous setup, making a single change meant updating around 50 repositories and reviewing each pull request. With our "four eyes principle," two people review each PR. If each person spends 15 minutes on each PR, that's 30 minutes per PR. When multiplying that by 50, we needed roughly 24 to 34 hours of collective effort for one change. It made a compelling case when we showed management that we could cut that down from over 30 hours to just about 30 minutes.

Beyond cost and time, reliability was another selling

point. Building a more robust process meant everyone could see what would be applied, when, and exactly what changes would occur. Everything became auditable, bringing a level of flexibility and control that wasn't impossible—previously, deploying a change felt like crossing your fingers and hoping for the best. Now, we know what's happening every step of the way.

The business quickly saw the benefits, not only from a cost and efficiency perspective but also in terms of visibility and reliability. Once they understood the improvements, they bought into the migration very quickly.

Bart: Now that you've completed the migration to a GitOps-based cluster management system, is there anything you'd approach differently if you were to start this journey again?

Ángel: If I could do it over, I'd prioritize automation at the infrastructure level. We initially tried using Crossplane to automate network attachments for clusters, but it was still maturing and introduced issues in the control plane, so we paused full integration. Looking back, I'd push for other solutions to handle network connections and related infrastructure to replace the manual steps we're still using. Bringing everything into our GitOps model has been a complex task, especially when working across departments to prepare for Crossplane fully. In hindsight, I would have strongly

advocated for these automation efforts to streamline our setup.

Bart: We want to wrap up by getting to know you a little better. You've been working remotely for a while now. Do you see yourself returning to an office setup?

Ángel: The first rule is never say never, but honestly, I don't see myself returning to an office full-time. A daily commute isn't essential for tech roles if you can do it remotely. That's not to say there aren't benefits to in-person work, but it has to work for both the company and the employee. If being in the office boosts performance but takes away from work-life balance, it's a tough compromise.

The experience is also different depending on the type of company. Large organizations with established in-office routines may handle remote work differently than a remote-first startup. Hybrid models add complexity, especially when there's pressure to show up in person.

Remote work offers a balance everyone deserves, but you must also deliver equal or greater value remotely. If not, the company has a reason to bring you back to the office. When applying for a job, it’s essential to understand the company’s remote work culture—whether remote is the standard or if there are occasional in-office meetups.

Ultimately, office work should benefit both the company

and the individual. But in 2024, going in just for the sake of presence doesn’t add up.

Bart: What's next on your path—both professionally and personally?

Ángel: Right now, I focus on my role as an individual contributor at Adidas. But before joining, I was in engineering management and tech lead positions, where I got to mentor and support people, and I miss that part of the job. I'm deeply passionate about tech, so I'm not moving away from it, but my next step would ideally be a managerial role—whether next year or a bit further down the line. I enjoy designing features and watching a team bring a project to life.

I have a 10-month-old son, and watching him grow has been incredible. Life looks different now; I can't imagine it any other way.

Bart: Will he be learning Kubernetes basics soon?

Ángel: I've already got some Kubernetes books lined up for him!

Bart: If anyone wants to reach out, where can they find you?

Ángel: X is probably the best way to reach me.

Acknowledgments

We are deeply grateful to the brilliant minds who took the time to share their stories and insights, making this book a reality.

Pierre Mavro: Pierre's lessons on automating clusters at scale and the importance of effective observability and tool choices have been invaluable. He continues to push the boundaries of what's possible in Kubernetes management. Connect with Pierre on LinkedIn.

Dan Garfield: Dan's expertise in GitOps and the transformative power of Ingress continues to inspire engineers worldwide. His vivid storytelling and thoughtful reflections have enriched the Kubernetes community. He can be reached on Twitter or through Codefresh.

William Morgan: William's journey from AI to simplifying service mesh complexities is both enlightening and transformative. His work on demystifying eBPF and advancing secure, observable Kubernetes practices is crucial. Follow him on Twitter and explore his work at Linkerd.

Matthew Duggan: Matthew's advocacy for a Long-Term

Support model in Kubernetes provides deep insights into the challenges and opportunities for greater stability in the ecosystem. Stay connected with him on LinkedIn.

Artem Lajko: Artem's practical insights into multi-tenancy and efficient platform engineering have enriched the knowledge base of countless Kubernetes practitioners. His experience serves as a guide for those managing complex infrastructure. Reach him on LinkedIn.

Hillai Ben-Sasson & Ronen Shustin: Hillai and Ronen's research into Kubernetes security, including their insights into hacking Alibaba's cloud environment, are eye-opening and crucial for cloud safety. Follow Hillai on LinkedIn and Ronen on LinkedIn.

Stéphane Goetz: Stéphane's achievement of scaling Jenkins to 10,000 builds a week stands as a remarkable milestone in CI/CD. His strategies in cloud-native development are practical and forward-thinking. Connect with him on LinkedIn.

Ronald Ramazanov & Vasily Kolosov: Ronald and Vasily's journey from Docker Compose to Kubernetes is a must-read for anyone navigating cloud-native transformations. Their stories are full of actionable guidance for teams managing complex migrations. Ronald is available on LinkedIn, and Vasily on LinkedIn.

Kensei Nakada: Kensei’s pioneering work with Tortoise and his innovative approach to Kubernetes resource optimization, including exploring Wasm integration, highlight a forward-thinking vision. Connect with him on GitHub.

Ángel Barrera: Ángel’s leadership in transforming Adidas’s infrastructure with a GitOps-driven approach stands as a powerful example of platform engineering at scale. His work continues to inspire the broader cloud-native community. Find him on LinkedIn.

Thanks to the Guests

A heartfelt thank you to every guest who opened up about their experiences. Your willingness to share your victories, struggles, and insights has made this journey unforgettable.

This book would not be the same without your stories and expertise.

Your contributions shape the future.

A Special Thanks

A special thanks to Daniele Polencic at LearnK8s—the silent mastermind behind so much of what we do. Your leadership, vision, and quiet impact shaped this more than words can say.