ITNEXT · Follow publication

Featured

Building a Kubernetes Platform — Think Big, Think in Planes

Why a horizontal mindset matters for modern platform engineering and developer productivity

21 min read · 3 days ago

Artem Lajko

> ***Note:*** *In this blog I want to share a fresh perspective on building platforms by thinking in planes. If that sounds a bit abstract, don't worry. First we will go through what "planes" actually mean, and then we will look at* ***OpenChoreo****, an Internal Developer Platform that is secure by default and already built with this idea in mind.*
>
> *If you, like me, believe in Open Source, consider showing your support for the work behind it with a ⭐ on GitHub.*
> *If you are attending to the KubeCon NA 25, you can also provide your feedback directly to the OpenChoreo Team* ***(#Booth number 1243)****.*

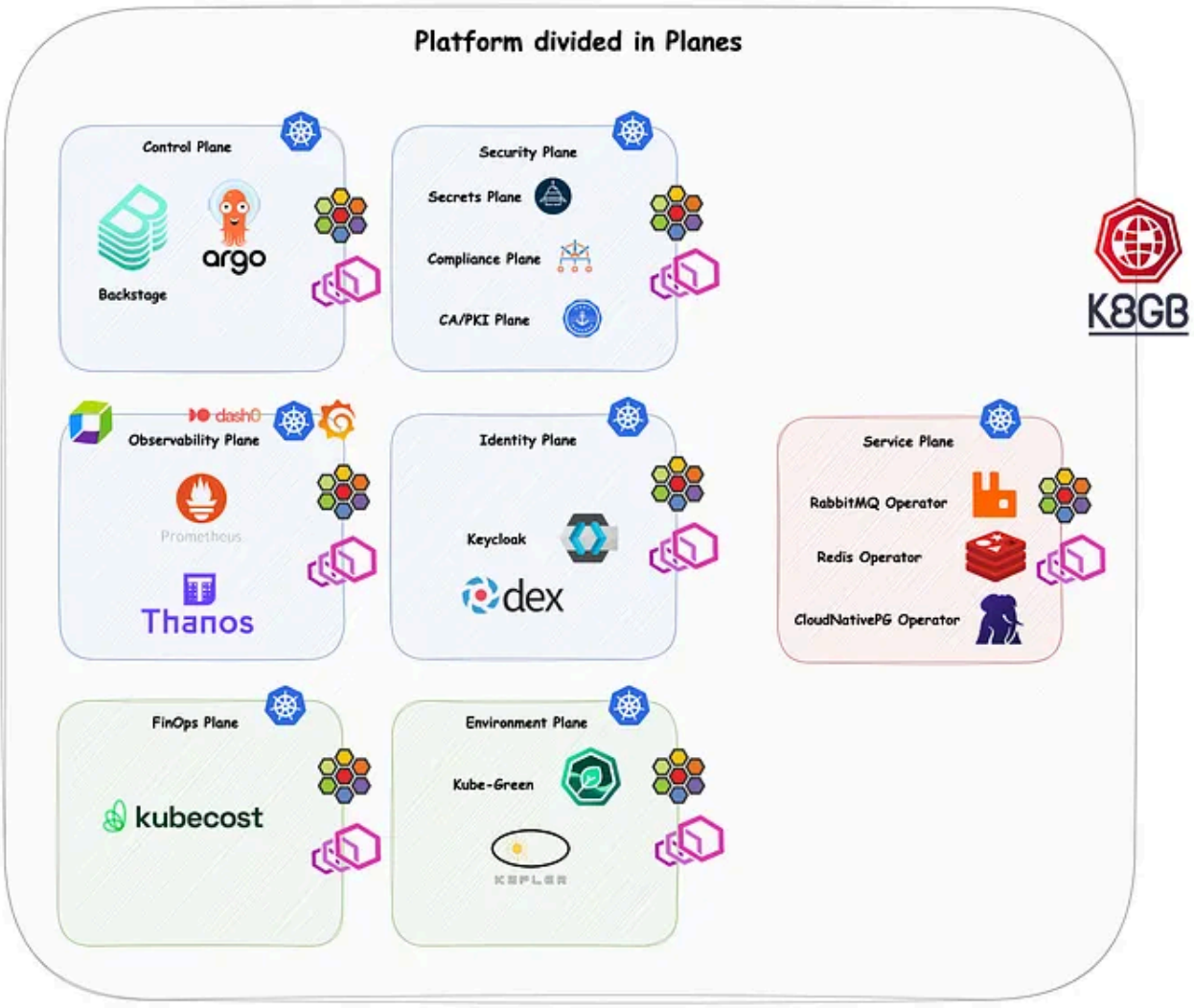

Fig. 0: Divide a Platform in Planes!

Introduction — Think Big, Think in Planes

This whole idea of thinking in planes started after a talk Maximilian and I gave at Cloud Native Days Austria.
The audience reaction was clear and honestly a bit surprising.
People came up to us saying things like *"We didn't even know approaches like this exist"*, or *"Thanks for sharing the tools behind it"*, or my personal favorite, *"Now we see those managed services from cloud providers with completely new eyes."*

That made us realize something important. This topic deserved more than a few slides and a short Q and A.
It needed a proper write up, something that connects the concepts, the reasoning, and the actual tools that make it all work. So here we are, turning that talk into a blog post.

You will see some of our original slides throughout, and if you make it to the end, you will find the full slide deck and a link to the recording of talk as well.

Now let us talk about this word I keep using, *planes*.

When you build a Kubernetes platform, it is easy to get lost in layers: infrastructure layers, service layers, security layers, you name it.
But here is the thing. Layers belong in cakes or burgers, not platforms.
If you really want to design a scalable Internal Developer Platform, you need to think in planes instead.

In this blog I want to share what that means, why it matters, and how projects like **OpenChoreo** already apply this concept in practice.
And just to set expectations right, this is not a "how to build it" tutorial.
You will not find a magic YAML that makes all your platform dreams come true.

What you will find instead is a new way to look at platform engineering, to understand where planes fit, and how some teams already solved these challenges with tools that are open and secure by default.

If that sounds like something your platform or your sanity might benefit from, keep reading.
But before we start thinking in planes, let us ask a simple and uncomfortable question:

Do you even need a platform in the first place? Just think about it 1–2 minutes.

Before You Jump In — Do You Really Need a Platform?

Let's be honest. Everyone in tech wants to "do Platform Engineering" these days It sounds great at conferences and looks even better on LinkedIn. DevOps is dead, and it is not fancy anymore. If you do not believe me, just look at the reports and see how many new Platform Engineering positions are being created.

But before you jump into the hype, ask yourself a simple question:

Does your business actually need a platform that scales?

If the answer is not a clear

Fig. 1: Are you committed to taking this path?

yes, then it is definitely a **NO**.

Building and maintaining an Internal Developer Platform is not a weekend project. A proper platform team will easily cost you around **one million** dollars a year.
You need at least four or five engineers for 24/7 coverage, maybe more if you run multiple environments. Multiply that by two to four years of setup and maintenance, and suddenly your "developer experience" initiative starts to look like a luxury item.

So think again. If your business itself is not scaling yet, then focus on scaling your business first, not your count of Kubernetes clusters.

Over the last year I have seen companies invest heavily in building internal platforms — portals, golden paths, fancy self-service provisioning — only to realize later that they did not really need them.
Sure, it was fun. We learned a lot. But it was also expensive fun.

If you are unsure, read this:
Why Companies Fail at Internal Developer Platforms or *You're not a platform team if*

you're just managing infrastructure
It is a brutal but honest reads.

From Layers to Planes — A New Mental Model

If you are building a managed Kubernetes service, thinking in layers makes sense. You start from hardware, move through virtualization, add containers, then orchestrate them, and finally hand everything to your users.
That is the traditional view — neat, hierarchical, and perfect for documentation and diagrams.

Fig. 2: From Layers to Planes — moving from vertical stacks to horizontal domains.

The problem is that real platforms do not behave like that.
They grow in every direction at once.
New tools, new teams, new pipelines, new compliance requirements.
Suddenly your perfect "layered" architecture looks more like a spaghetti diagram with YAML stuck in every corner.

So maybe layers are not the right mental model anymore.
Maybe we need something that lets us manage complexity horizontally — across domains, responsibilities, and teams.
That is where *planes* come in.

“Layers are static. Planes interact, evolve, and overlap without breaking everything else, mostly”

The Five Planes by Platform Engineering Org

The team at PlatformEngineering.org came up with a helpful way to describe modern platforms using five planes.

Fig. 3: Platform Tooling Landscape by Platform Engineering Org — 5 Planes.

Each plane represents a horizontal area of concern rather than a vertical technical stack.
Together, these planes define how your Internal Developer Platform works as a whole.

You do not have to follow this reference model exactly, but it gives you a solid mental map, much better than the CNCF landscape,

Fig. 4: A small extract from the actual CNCF Landscape

which has roughly a thousand logos and no clear starting point. But to be fair, that is not really its purpose. Its goal is to group tools into categories such as Runtime or Provisioning, and then break those down further into subgroups (imho).

I define the the the Platform Engineering tooling landscape like:

"If the CNCF Landscape feels like a zoo, this is the map that tells you where the animals live"

Let us look at each plane briefly before we dive deeper into the more technical ones later.

1. Developer Plane

The developer plane is the entry point for your users — the developers.
Here they interact with your platform through tools they already know, like VS Code, GitHub, GitLab, or Backstage.
Some companies build their own developer portal with frameworks like Backstage, Port, or Cortex.
Others use Software as a Service options that integrate governance, security, and compliance out of the box.

The main goal of this plane is to give developers an easy and secure entry point without locking them in.
You want to provide a *golden path*, not a *golden cage*.

Fig. 5: Developer Plane — where experience meets governance.

2. Integration and Delivery Plane

This plane is where most infrastructure or platform engineering teams live.
Here you find everything related to building, packaging, and deploying software.
CI pipelines that render Helm charts or apply Terraform modules, registries that store both container images and reusable artifacts, and GitOps tools that deploy them.

You might use Argo CD, Flux CD, or Humanitec's Platform Orchestrator, sometimes all at once.

The goal is to connect everything in a consistent, automated way so that the developer plane can provide true self-service.

Fig. 6: Integration and Delivery Plane — the factory floor of your platform.

3. Monitoring and Logging Plane

This one is straightforward. You probably already use some of it.
Whether it is Datadog, Dynatrace, Grafana Cloud, or a self-hosted LGTM or ELK stack, the goal is always the same: visibility.
This plane tells you what is happening across your clusters and applications, hopefully before your users do.

Fig. 7: Monitoring and Logging Plane — because “it works on my cluster” is not observability.

4. Security Plane

The security plane is not just about secrets.
It covers everything from identity and access management to network policies and compliance rules.
It is about making sure that your developers can deploy safely without accidentally opening every port on the planet.

You might see tools like Vault, Keycloak, Cilium, or Open Policy Agent here.
The goal is to protect without suffocating, to make security part of the flow instead of a roadblock.

Fig. 8: Security Plane — keeping your platform safe, your auditors happy, and your engineers slightly annoyed.

5. Resource Plane(s)

The resource plane defines where your workloads actually run.
It can include your compute clusters, databases, storage systems, or even external services like RabbitMQ or managed Postgres
You might have multiple resource planes grouped by cloud provider like AWS, Azure, GCP, StackIT, OVH, etc., each defining what services your platform can consume.

Fig. 9: Resource Plane(s))— where your YAMLs finally meet hardware.

This is where reality hits architecture: you need actual infrastructure for all those nice planes to exist on.

Why This Matters

Thinking in planes helps teams describe their platform in a simple, shared language. It creates a mental model that developers, architects, and managers can actually talk about without losing each other in buzzwords.

More importantly, it forces you to think horizontally — to see how things connect across teams and technologies instead of piling on more layers.
That is the mindset shift that turns a collection of tools into a platform.

Why Thinking in Planes Helps

By now you have seen what the five planes look like.
But let us talk about *why* this way of thinking actually helps.

Most teams that start with platform engineering make the same mistake.
They take the CNCF Landscape or the PE Tooling Landscape — a digital jungle full of logos — and try to “build something like that.”

Fig. 10: “Just build this!”

Then someone in the infrastructure team asks the obvious question:

Fig. 11: “Just build this!” — the classic reaction when theory meets reality.

“Build what, exactly?”

That is when the room goes quiet.

The truth is that the CNCF Landscape is useful, but it can be overwhelming if you do not already know what you are building or if you lack a clear architectural point of view.

You need a way to describe your own platform that makes sense for *your* organization — something your developers, managers, and security teams can all understand.
That is where thinking in planes, as introduced in the Platform Engineering reference model, becomes valuable: it gives you a shared mental model.

Personally, I use these landscapes in a simple way: the **CNCF Landscape** helps me stay aware of the available tools -> the **Platform Engineering Landscape** helps me understand how they fit together and express it to different stakeholders, and the -> **planes** help me design and implement them across multiple clusters and environments.

Fig. 12: From Tooling to Reference Architecture to Planes

It is not about collecting tools.
It is about describing what your platform actually *is* and what each part is responsible for.

When you describe your platform in planes, you stop talking about YAML, clusters, Helm charts, Tools or Projects and start talking about capabilities of the planes.
That is the moment when your internal discussions move from “how” to “why.”

Fig. 13: Speak Platform Wise to Me — finding a common language between tech and business.

A Different Point of View

Thinking in planes allows you to see your platform as a set of horizontal domains that can scale independently.
You can run different planes on the same cluster, or spread them across multiple clusters depending on size, security, or performance needs.

Instead of managing one huge shared environment, you manage connected but specialized planes.

Each one does its job, but all of them fit together to form the platform.

This horizontal view also helps when you need to talk about scaling and reliability. A single overloaded cluster is a risk; distributed planes let you spread that risk and manage growth more predictably.

And once you see it like that, you start noticing something interesting:
You are already using parts of this model without realizing it.

Fig. 14: Thinking in Planes — visualizing connected domains that scale horizontally.

You might already have a central control plane that manages configurations, a monitoring stack that collects data from all clusters, or a security system that manages identities across environments.
Those are planes. You just never called them that.

What Comes Next

Now that we understand *why* this mindset helps, let us look at how it actually works in practice.

In the next section, we will explore three of the most important planes in more detail:

- The **Control Plane**, where your management logic lives
- The **Observability Plane**, where you collect and centralize insight
- The **Service Plane**, where you enable developers to consume services on demand

Together with other planes they form the backbone of a modern platform that scales not only technically, but also organizationally.

So let us dive in.

Deep Dive — Control Plane, Observability Plane, and Service Plane

Now that we understand why planes make sense, let us take a closer look at how they actually work in practice.
You can think of a platform as a group of planes running on the same cluster or across many clusters.
Each plane serves a specific purpose, and together they create a system that scales without chaos.

We will look at three of them in more detail: the Control Plane, the Observability Plane, and the Service Plane.
These are the ones that will make or break your platform.

Control Plane

For most teams, the Control Plane is where everything starts.
It is the place where you manage your clusters, your configurations, and the add-ons that keep your ecosystem alive.
Here you deploy and control third party tools like Cert Manager, External DNS, or External Secrets Operator (ESO).

In this plane, you also decide how to structure your management topology.
Two common approaches are ***Hub and Spoke*** and ***Dedicated Instance per Cluster***.
Both allow you to manage multiple clusters from a central control point, but the role of that control point (e.g. central Helm Charts vs central Deployment + central Helm Charts) can shift depending on your setup and scale.

The pattern itself evolves over time. In this section, we will focus on the Hub and Spoke model.

Tools like Argo CD, Flux CD, or Sveltos can help you deploy and maintain add-ons (mostly Helm Charts) at scale.
For example, with Argo CD's ApplicationSet and Cluster Generator, you can deploy something like Cert Manager to one, ten, or a thousand clusters simply by labeling the targets.
With Sveltos you can achieve the same result through ClusterProfiles.

Fig. 15: Hub and Spoke Architecture — managing clusters at scale without losing control.

These tools support different topologies. Some are agentless (Argo, Flux, Sveltos), some are agent based (Sveltos, Argo).
With an agentless approach you push configurations from your control plane into your clusters, which gives you power but also more risk.
The control plane often needs high privileges, and scaling can become difficult as your environment grows.

Argo CD is currently working on an agent based model through Open Cluster Management (OCM), while Sveltos already supports both push and pull modes.

Agent based models can reduce privilege levels and scale better, especially for large fleets or edge environments.

And we are not speaking about...

Fig. 16: Inspired by cheezburger

I mean we have some in common we, want scale. horizontal.
Hopefully you get the idea and relate what can be defined as a Control Plane.

Observability Plane

Once your control plane is in place, the next thing you need is visibility.
You need to know what is happening across all those clusters and services.
That is where the Observability Plane comes in.

Fig. 17: Observability Plane — smaller agents collect, one central brain visualizes.

Most setups use small agents or collectors on every cluster.
These components gather metrics, logs, and traces and push them to a central observability endpoint.
This way you avoid deploying a full monitoring stack everywhere, which saves resources and reduces duplication.

You can combine tools like Prometheus, Fluent Bit, and OpenTelemetry to collect data, and use Thanos, Loki, or Tempo for centralized storage and querying. The central observability plane can then serve dashboards through Grafana, OpenSearch Dashboards or perses.

This architecture (fig. 17) gives you the best of both worlds.
Developers can still see their logs and metrics per environment, but operations can monitor the whole fleet from a single pane of glass.
And since the collectors talk to the central plane through secure channels, you can keep your data isolated without losing global insight.

Maximilian explained during our talk in the "Deep Dive — Observability Plane" Part, how you can achieve all of this with standard open source tooling.

Fig. 18: Building an Observability Plane

You do not need to reinvent observability , you just need to organize it.

I guess you will have a similar setup, but maybe with a different tool stack.

Service Plane

Now that you can see and control everything, you need to provide actual services. If you want to build a platform, not just managing infrastructure.
That is what the Service Plane is for.

Fig. 19: Service Plane — think like a cloud provider, but for your own developers.

Ask yourself this:
When you click “Create Database” in your favorite cloud console, what really happens behind the scenes?

A lot of cloud providers run managed services directly on Kubernetes using operators.
For example, CloudNativePG can provide a managed PostgreSQL instance that behaves almost exactly like a cloud database service.
The same applies for RabbitMQ as Message Broker Service, Redis as managed Cache Service, or even Harbor as a managed registry.

If you are building an internal platform with self service capabilities and your cloud provider does not support Crossplane or similar tools, operators can be your answer. They let you offer managed services inside your company with the same experience developers expect from the cloud.

In other words, the Service Plane allows you to deliver infrastructure as a service, but on your own terms.
It is how you build internal products that your developers actually want to use instead of being forced to.

When combined with the other planes, this approach lets you create a secure and scalable system where developers can request, deploy, and observe their workloads without waiting for tickets or approvals.

> ***Note:*** *If you are not sure yet how to make this work, for example how to provision a database on demand in the Service Plane and make it available to developers, do not worry.*
> *I have already covered this topic in another blog post together with Gianluca, the mastermind behind the Sveltos project.*

Summary

By separating your platform into functional planes, you can centralize what needs control and distribute what needs scale.
You can run lightweight agents instead of full stacks everywhere.
You can manage identity and access once, and reuse it across your environment.

The goal is not to over engineer, but to organize complexity in a way that grows with your company business.
Think like a cloud provider, but keep the focus on your internal users.

Putting It Together — Think Like a Cloud Provider

Once you start thinking in planes, something clicks.
You stop seeing your platform as a pile of disconnected tools and start seeing it as an internal ecosystem.
Each plane serves a purpose, and together they create a consistent experience for your developers.

A good platform is not just a bunch of YAML templates and Helm charts glued together with CI pipelines or GitOps.

It is an organized system that knows which tools belong where, what needs to be centralized, and what should stay local to a cluster or team.

Take a step back and think about how public cloud providers operate.
They do not deploy one monitoring stack per cluster, or spin up a new identity service for every environment.
They centralize what makes sense and distribute only what is needed.
You can do the same internally.

You do not need a separate Grafana instance for every cluster unless you enjoy switching dashboards all day.
You do not need to run a new Keycloak for every team, unless you want to simulate the pain of managing multiple Microsoft Entra IDs.
Instead, centralize those services in your control or observability plane, as long as you can do it securely.

That is the key idea behind this entire concept:
Centralize what needs control. Decentralize what needs scale.

Fig. 20: Think like a Cloud Provider — centralize control, distribute scale.

This approach is not new.
We already do it for observability, for identity management, and sometimes for security.
The trick is to apply the same logic consistently across your platform. If it makes sense!

When you design planes this way, you start thinking like a service provider with a product mindset, not just a team running infrastructure.
You create something that your developers actually want to use because it saves them time instead of adding friction.

And that is exactly what the next section is about.
There is already a project that puts this idea into practice — one that is open source, secure by default, and already built in planes.

Let us take a closer look at **OpenChoreo**, the Internal Developer Platform that shows what it means to truly think in planes.

Case Study — Applying the Plane Model in Practice: OpenChoreo

> ***Note:*** *In this part we will cover how OpenChoreo works. We will focus only on how it uses planes to provide an internal developer platform for developers and related parts to get understand of it. I will publish another blog in the next few days that focuses on OpenChoreo itself.*

We have discussed how thinking in planes helps to organize complexity in platform engineering. To see how this approach works in practice, let us look at a real implementation: **OpenChoreo**.

Fig. 21: OpenChoreo!

Background

OpenChoreo is an open source Internal Developer Platform created by WSO2 based on the enterprise solution Choroe.
It builds on more than twenty years of experience with IAM, API Management and

integration systems and brings those patterns into a modular, Kubernetes based architecture.
Rather than introducing new abstractions for their own sake, OpenChoreo demonstrates how the plane model can structure infrastructure, security, and delivery functions in a consistent way.

Fig. 22: OpenChoreo Planes — Control, Build, Data, Observability, and Identity working together.

Architecture Overview

OpenChoreo separates its responsibilities into several planes, each with a distinct role within the platform:

- **Control Plane** manages global coordination. It hosts the API server and controller manager and connects all other planes through defined APIs. Communication is secured through Cilium network policies and Envoy gateways.
- **Data Plane** represents a Kubernetes cluster where application workloads run. Each data plane is registered with the control plane and contains credentials,

TLS configuration, and cluster settings. Platform teams can manage multiple clusters — on premises or across clouds — through a single control plane.

- **Build Plane** provides dedicated infrastructure for continuous integration and image creation. It uses Argo Workflows to execute build pipelines in a Kubernetes native way and isolates heavy build tasks from production workloads.
- **Observability Plane** aggregates logs and metrics from all other planes. It is based on OpenSearch and receives log streams from Fluentbit agents deployed across the system. Once configured, it operates independently and offers authenticated access through the Observer API.
- **Identity System (Plane)** provides authentication and authorization. To secure APIs deployed on your platform.

Each plane can be deployed independently as a Helm chart, allowing platform engineers to scale and distribute them according to their operational or compliance requirements.

Fig. 23: Relationship between the Planes — how Control, Data, Build, Observability, and Identity interact.

Platform Abstractions

OpenChoreo defines several high level abstractions that map directly to Kubernetes concepts and enable self-service while maintaining governance:

- **Organization** is the highest level of tenancy and isolates resources across teams or customers through namespaces, network policies, and quotas.
- **Environment** defines a stage of the software delivery lifecycle such as development, staging, or production. Each environment maps to a data plane and includes its own policies and configuration.
- **Deployment Pipeline** represents the progression of applications through environments and defines promotion paths, approval requirements, and quality gates.
- **Class System** implements the Kubernetes Class pattern (similar to *GatewayClass* or *StorageClass*) to create higher level abstractions such as ServiceClass or WebApplicationClass. Classes encode organizational standards, and Bindings instantiate them in specific environments.

Fig. 24: Map your domain to OpenChoreo organizations, projects, and environments

These abstractions create a clear separation between platform governance and developer intent. Platform engineers define the standards through *Classes*, while developers express desired outcomes by declaring Services or *WebApplications* that reference those *Classes*.

Domain Model and Hierarchy

OpenChoreo resources follow a strict ownership hierarchy that defines isolation and traceability:

- **Organization** serves as the root container for all platform and application resources.
- **Projects** represent bounded domains within an organization and own **Components**, which are the deployable units such as web services or workers.
- Components generate **Builds**, which produce **Releases**, maintaining complete traceability from source code to running workloads.
- **Bindings** combine Classes, Workloads, and Environments to create concrete deployments.
- **Endpoints** and **Connections** define how components communicate. All traffic is routed through Envoy gateways and governed by Cilium network policies.

This structure provides strong multi-tenancy and controlled communication between components, matching the horizontal separation of concerns found in the plane model.

Fig. 25: Resource hierarchy in OpenChoreo

Security and Observability

OpenChoreo enforces zero trust security across all planes.
Each Project, represented as a **Cell based on** Cell-based Architecture (CBA), is an isolated, observable unit controlling internal and external communication through defined ingress and egress paths.
Cilium and eBPF enforce fine grained network policies, while Envoy gateways manage routing, rate limiting, and authentication.
All traffic is encrypted with mutual TLS, and every request is authenticated and authorized.

Fig. 26: Cell Based Architecture — mapping organizational domains into secure, isolated cells.

Observability is enabled by default.
Logging, metrics, and tracing are collected automatically **across planes** without manual setup. Fluentbit agents forward data to the central OpenSearch instance, which provides unified dashboards and search capabilities.

Why It Matters

The OpenChoreo example illustrates how the plane model can be applied to a real Internal Developer Platform.
Each plane operates independently yet contributes to a cohesive structure that scales horizontally.
The platform abstracts complexity through declarative APIs while preserving visibility, governance, and security.

For platform teams, this structure reduces operational overhead and clarifies ownership boundaries.
For developers, it simplifies the workflow — they can deploy, observe, and connect applications without dealing with the underlying infrastructure by using Backstage as provided Portal.

For platform teams it means fewer moving parts to manage.
For developers it means faster delivery and less waiting.

If you want to see OpenChoreo in action, check out the demo from our talk — from Backstage to the planes, all running in a single cluster setup, here 👈

Fig. 26: OpenChoreo in Action — from portal to planes in one consistent environment.

Conclusion — Keep It Simple, Keep It Open

We covered a lot.
We started with the idea that platforms are not built in layers but in planes.
We looked at what those planes are, how they connect, and why they make life easier for everyone who has ever tried to manage Kubernetes at scale.
And we saw how OpenChoreo turns this mindset into reality with an open, secure by default architecture that actually works.

Thinking in planes is not a new buzzword.
It is a way to organize complexity, to describe what a platform really is, and to help teams speak the same language.
It helps you stop stacking technology and start building structure.

If there is one thing I hope you take away from this blog, it is this:
Do not build something just because it looks cool on a slide.
Build it because it solves a problem that matters to your business.
And if it does, build it properly.
Not halfway. Not as an experiment that fades after one sprint.

Your platform should serve your developers, not the other way around.
It should make them faster, not busier.
That is what thinking in planes is really about combined with approaches like cell based architecture.

Final Thoughts

I hope this post helped you see why thinking in planes can make such a difference when building platforms.

If it did, and you share my belief in the power of open source, consider showing your support with a ⭐ on GitHub for projects like OpenChoreo that make this possible.
If this was your first time hearing about them, maybe this blog gave you a reason to take a closer look.
And if you are still not convinced, that is fine too — leave a comment and tell me why.
Your feedback helps all of us — me, the tool creators, and the community — to improve and keep moving forward!

Wrap Up

That is it.
Think big.

Think in planes.
And remember: if you build it, build it for humans first.

Addionatiol Ressources:

Watch the full video here ->

Video 0: Reverse Engineering a Kubernetes Platform — from UI to GitOps (Cloud Native Days Austria)

Here you can get the slides deck.

If you want to learn more about topics like this, you should definitely take a look at:

- **Platformless: How Choreo Built a Secure Kubernetes Platform with GitOps**
- **Sveltos: Argo CD and Flux CD are not the only GitOps Tools for Kubernetes**
- **Building Your Own Event-Driven Internal Developer Platform with GitOps and Sveltos**

Contact Information

Got questions, want to chat, or just keen to stay connected? Skip the Medium comments and let's connect on LinkedIn 👍. Don't forget to subscribe to the Medium Newsletter so you never miss an update!

Kubernetes | Cloud Computing | Containers | Platform Engineering | Open Source

Follow

Published in ITNEXT

76K followers · Last published 5 hours ago

ITNEXT is a platform for IT developers & software engineers to share knowledge, connect, collaborate, learn and experience next-gen technologies.

Follow

Written by Artem Lajko

3.5K followers · 159 following

Do something with cloud, kubernetes, gitops and all the fancy stuff https://www.linkedin.com/in/lajko

No responses yet

Bgerby

What are your thoughts?

More from Artem Lajko and ITNEXT

In ITNEXT by Artem Lajko

Manage Secrets of your Kubernetes Platform at Scale with GitOps

Learn how to manage secrets with the External Secrets Operator and plug it into Argo CD to power your Internal Developer Platform

Sep 1 213

In ITNEXT by Max Silva

The Real Cost of Server-Side Rendering: Breaking Down the Myths

There's a growing narrative in the web development community that Server-Side Rendering (SSR) is nothing more than an expensive burden on...

Oct 5 34 1

In ITNEXT by Robbie Cahill

Run a Raspberry Pi server at home in minutes with Tunnelmole

The Raspberry Pi is a marvel of accessible computing. This credit-card-sized device packs enough power to run a variety of applications...

Oct 10 132 2

In ITNEXT by Artem Lajko

kubriX: Your Out-of-the-Box Internal Developer Platform (IDP) for Kubernetes

Discover how kubriX integrates leading Open Source tools like Argo CD (GitOps), Kargo, and Backstage to deliver a fully functional IDP out...

Jul 13 344 6

See all from Artem Lajko

See all from ITNEXT

Recommended from Medium

In Data, AI and Beyond by Julius Nyerere Nyambok

Generate AWS architectural diagrams using this simple method

A combination of LLMs and MCP Servers.

Oct 14 193 2

In Stackademic by Heinancabouly

Kubernetes in Production: The Hard Truths Nobody Tells You

After 3 years running multi-region clusters: what matters for autoscaling, observability, security, and avoiding those 3 AM pages.

6d ago 25

In FAUN.dev() by Vishvadini Ravihari

Zero Trust with Cilium : Enforcing mTLS in Kubernetes

Kubernetes networking is highly flexible but this flexibility can introduce security risks because all pods can communicate with each...

Oct 20 7 2

In AWS in Plain English by PyOps

Kubernetes v1.35 Is About to Change Everything — Is Your Cluster Ready?

The next major release is stacking up big features and serious urgency.

Oct 22 15

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here's What Nobody Tells You About A2A and MCP.

Multi-agent systems are the future. Agent-to-Agent (A2A) communication and Anthropic's Model Context Protocol (MCP) are revolutionary. But...

Oct 16 1.3K 24

Salwan Mohamed

GitOps for Infrastructure Automation: Mastering Terraform and Flux CD

Part 8 of the “GitOps Mastery: From Zero to Production Hero” Series

Oct 21 3