More fun, more solutions, more DevOps productivity

Platform Engineering: Fundamentals and Concepts

by Marcus Greuel, Thomas Ganter, Michael Johann, Christian Hartung

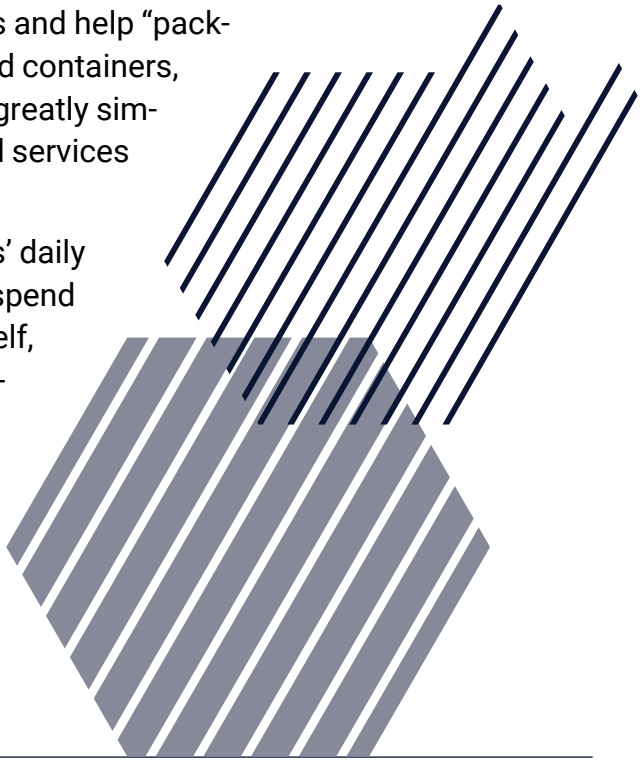
You often hear the provocative statement: “DevOps is dead.” This is a false claim, merely intended to draw attention to a current trend: platform engineering. In this article, we’ll shed light on the current problems facing DevOps and developers and describe platform engineering as a solution.

Once upon a time, developers were the ones who developed business functionality and system operators delivered the software. The time between handing over the “finished” software and delivering a release to users often took several months. IT organizations usually defined two main releases per year, with planned bug fix releases here and there in between. From the user's point of view, this could sometimes be unsatisfactory or even unthinkable for applications based on Software as a Service (SaaS). New features and bug fixes arrived too late.

From a developer's perspective, this meant a high degree of parallelism between multiple development, test, bug fix, and release versions. From an operational perspective, this complexity resulted in long bug fix cycles and the associated workarounds and “quick fixes,” along with a high degree of bureaucracy. System operators also had a decisive say in how the artifacts were delivered, which didn’t always support the necessary flexibility, often hindering innovation.

With the establishment of DevOps principles, these rigid processes became a thing of the past. Nowadays, every commit represents a potential release. Continuous delivery is the motto. Developers are much closer to operations than they used to be.

Developers have broader freedom when choosing programming languages and frameworks, as long as they also consider and support the specific requirements for deployment and operation. As a result, there is a huge proliferation of languages, frameworks, and tools for both developing software and operating applications. Docker, Chef, Ansible, Terraform, and Nix, are no longer exotic in the Infrastructure as Code environment. They ensure reproducible runtime environments and help “package” applications. With the advent of Docker and containers, the uniform provision of applications has been greatly simplified. Rolling out cloud-native applications and services reached its peak with Kubernetes.



But what does this mean for DevOps employees’ daily work? Roughly speaking, developers no longer spend most of their time on software development itself, but rather on providing infrastructure and cross-functional capabilities such as monitoring, logging, and alerting.

A typical scenario

The following story illustrates this point:

Hans is a software developer at a large company. He works with hundreds of developers who are increasingly moving away from the role of software developer and becoming DevOps engineers. In addition to all the meetings in an agile environment, more and more tasks from the operations area are being added. Hans realizes that he is only working on developing business functionality 20% of the time. The rest of his time is spent on operations and infrastructure. A few years ago, management decided that all legacy applications should be cloudified and run on Amazon AWS. Hans spent working hours and some of his free-time on learning AWS-specific services and he's very proud that he can now roll out and operate applications and services in the cloud. Of course, other employees want to do the same, so more and more is being invested in training. However, there aren't any standardized rollout processes yet, so some teams are developing frameworks and tools to make their daily work easier and become more productive using cloud services.

Meanwhile, competing cloud service providers like Microsoft Azure and Google Cloud Platform offer services that are better suited to specific applications and often more cost-effective, so management is considering a multi-cloud strategy. The aim is to reduce operating costs as much as possible, depending on the application. Unfortunately, this isn't possible if you've relied solely on AWS-specific services and haven't developed or used abstractions for database services, file services, and other infrastructure services. People specializing in Kubernetes can deploy their services independently of the hyperscaler, use smart abstractions, and draw on a wealth of information from the community.

The following description is based on a product service, and some readers may recognize parts of this example from their own environment. If so, it is definitely worth reading on. If not? Read on anyway.

Questions with lots of answers?

A few years later, Hans joins a department tasked with supporting new developers during onboarding. In the meantime, many cloud providers have set up their own portals so that DevOps staff aren't left too much to their own devices.

However, the onboarding process for new teams and projects now looks something like this: A team wants to develop a microservice based on Spring Boot and roll it out in a production environment.

It's great that they've already agreed on the programming language and frameworks. Now the team has other tasks to tackle before they can even start developing a single line of software. When new team members join, it's unclear how to proceed. Where do I get my accounts for AWS, GCP, Azure, etc.? Who is the contact person for security issues? Do I need approval from the council? What are the compliance requirements? Do we use Jira and do we need access or project structures for the project? Okay, so we use Git for version management. But do I use GitLab or GitHub, and in which substructures do we put our source code and artifacts? Alright, we want to roll out cloud-native. Which container registry should we use? Does the company have its own standards? Which substructures for container repositories should we consider?

And what about the pipelines for automated build, test, and deployment? Are there guidelines for which security scans need to be done? Who can tell me? Do we want to build and manage our own Kubernetes cluster? Or are there teams in the company that provide something like this? Which cost centers do we need to specify? How do we ensure compliance during runtime after we've rolled out the application? And what about observability, monitoring, logging, and tracing?

By now, this list of questions should be enough to illustrate how complex a project can be before the first line of code is even written. If you want to be able to answer all these questions for future projects, you need to stay up to date and deal with a whole host of issues that have little to do with the actual development of a product or service.



FIG. 1 The CNFC tool landscape

Cognitive overload

Once, DevOps emerged with “tools” as one of its core pillars. There was talk of “toolboxes,” and it worked as long as the number of tools stayed manageable. Anyone who takes a look at the Cloud Native Computing Foundation (CNCF) tool map today is overwhelmed (Fig. 1). Toolboxes with a handful of tools are long gone (perhaps they never even existed). Now we have tool ecosystems. Software development teams that take DevOps seriously and want to map all the value-added steps and deliver and operate software holistically face the challenge of an immense number of tool options.

Of course, as a team, you can try to stick to the de facto standards. But this approach cannot remedy the proliferation of tools beyond your team. Just when you’ve trained to become a tool specialist, the standard changes, different or new tool functionalities are required, or an open source project is discontinued. New tools regularly appear on the scene (which is welcome in principle, but always requires a greater solution). Every team uses its own tool stack, and team changes in flexible organizations, where project assignments take place regularly, lead to frustration. This means relearning other tools that, in most cases, don’t do anything different than the old ones. No developer can or wants to be a specialist in all these different, overlapping tools or services.

TRACK



PLATFORM ENGINEERING SUMMIT
by DevOpsCon

Build Once. Scale Everywhere. Real Developer Experience:

Why Platform Engineering Summit?

Platform Engineering is the backbone of modern Developer Experience—and the foundation for secure, scalable DevOps platforms. As AI becomes part of every toolchain, platforms must evolve to govern AI and accelerate delivery. At this, you’ll learn how teams build Internal Developer Platforms (IDPs) that reduce cognitive load, enable self-service, and embed AI guardrails from day one. Whether launching a new platform or scaling across teams, you’ll take away real-world patterns, best practices, and strategic insights to stay ahead.

Learn from Industry Leaders about:

- Treat platforms as products – designed for user value
- Reduce cognitive load & speed up feedback cycles
- Build scalable IDPs using CNCF tools & IaC templates
- Automate security & compliance from day one
- Prove ROI with DevEx metrics & data
- Learn from real-world case studies and patterns
- Best practices to accelerate DevEx + AI governance

This mess of extra knowledge and its necessity is also called cognitive load or, when there's too much of it, cognitive overload. From a business perspective, this is definitely not the ideal situation. Developers are usually much more interested in writing code than dealing with ticket ops, chat ops, or email ops. It's easy to imagine that estimating sprints and backlogs in these environments becomes inaccurate and isn't enjoyable.

Companies and their teams have been trying to answer these questions using their own order portals, function mailboxes, ticket systems, and training courses. But with more and more new topics (like artificial intelligence) emerging, new strategic paths are constantly being explored before other issues can be fundamentally clarified.

Platform engineering should finally provide clarity and solve many problems. Instead of countless entry portals and contacts, meaningful abstractions and corresponding control planes should significantly reduce the cognitive load of DevOps employees. The goal is to increase productivity with the help of self-services and integrated views, while putting fun back at the forefront of software development.

How is that possible?

First, it's important to identify best practices and "golden" paths for how efficient development can work. Golden paths can easily cover 80 percent of use cases and achieve maximum scaling effects – and then the question of individual solutions arises. This approach supports both because each team can develop and deliver services. The more customized a required service is, the less likely it is to already exist as a service provided by a team of specialists, and the more likely that the service will have to be implemented individually. This ultimately means less support from the inner source community. In other words, customized solutions are more complex, which is normal and okay, but it's just as feasible and worthwhile as use cases covering the majority.

Back to our example: Hans and his team are tasked with developing a product service. Ideally, the company offers an Internal Developer Platform (IDP) that supports all activities throughout the application lifecycle. An IDP is a centralized system in platform engineering that aims to provide developers with the necessary tools, services, and automations to increase efficiency and productivity in the software development process. It acts as an abstraction layer above the infrastructure and allows developers access resources without needing in-depth knowledge about the underlying infrastructure.

An IDP typically includes features such as developer self-services, continuous integration and delivery (CI/CD), infrastructure as code (IaC), monitoring, logging, tracing, security mechanisms, and, in the best case, cost accounting.

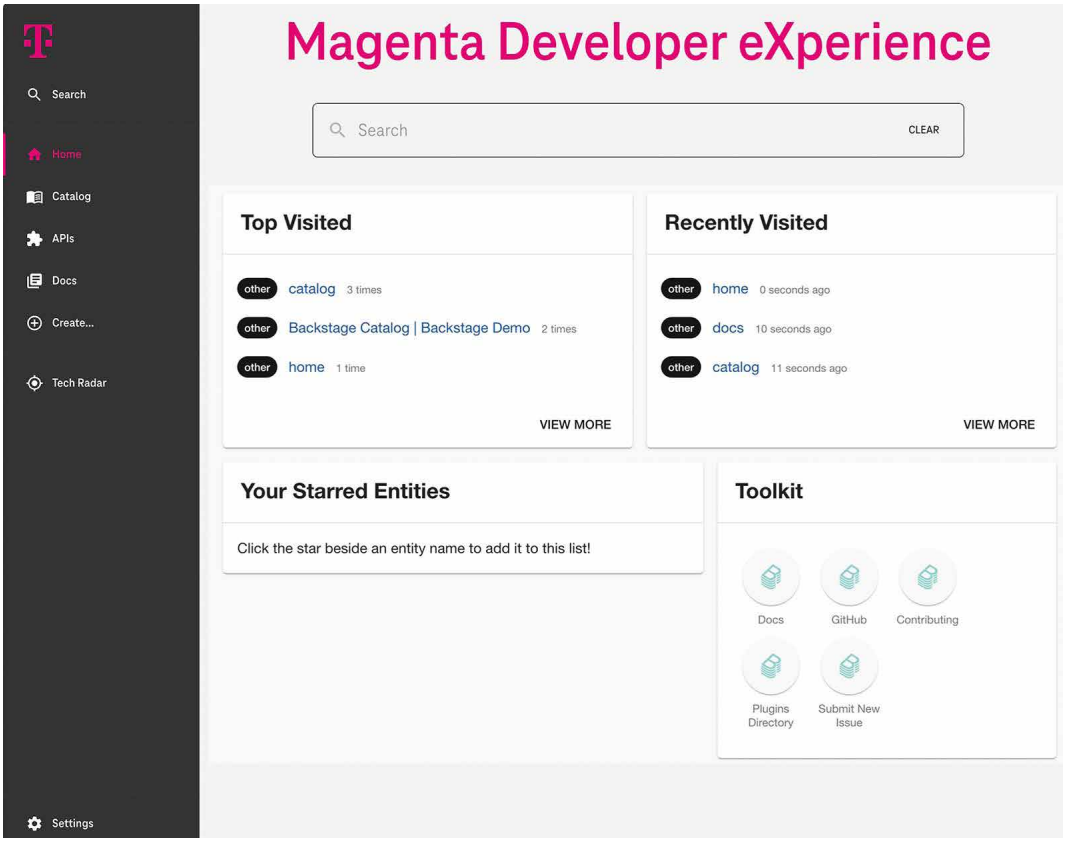


FIG. 2 Backstage as an introduction to the catalog

Hans sits down at his computer and opens the IDP developer portal to create a template for a typical microservice based on Spring Boot using a scaffolder. Examples of developer portals include Backstage and Port. Hans enters a few parameters like project name, the associated team, and frameworks used. The scaffolder converts the corresponding template into a Git project, which is automatically generated by the control plane with all conventions for creating substructures in GitLab/GitHub and created for the team with the respective roles. This largely frees Hans from other aspects like monitoring, pipelines, container images, runtime environments, security measures, etc., since the IDP abstracts all these points and keeps them away from his team. The control plane takes the desired parameters for the project from the developer portal, generates all components independently, and makes them available. The knowledge of platform engineers is incorporated into the platform, so Hans and his team no longer need detailed knowledge. This leaves more time for actual product service development.

If the product service requires a database, it is provisioned automatically. The same applies to monitoring, container images, logging interfaces, and even internal cost allocation. IDP's platform engineers implemented all conventions and best practices in the company in managed services and abstracted and synchronized them in the control plane so that Hans and his team ultimately receive a message that all required functionalities are available. Remember that previously, dozens of colleagues and order portals had to be identified and visited in a time-consuming process to gather all the necessary services. On top of that, they needed extensive knowledge to create the infrastructure needed for the service.

After successful completion of scaffolding in the Developer Portal, a Git repository, pipelines, container images, a runtime environment, and other non-functional services such as logging, monitoring, tracing, and compliance checks are now available. In the developer portal, this is visualized with a cumulative view of different data sources. Hans and his team only need one browser tab in the developer portal to stay informed about the current project status and can take action in case of problems during runtime.

The broker architecture also promotes competition among service providers, which can be beneficial from a business perspective. Many services can be improved over time through “natural selection” from the user's perspective. Whereas, more “exotic” services remain in use, depending on their usage, even though they don’t follow the golden path because they can solve a specific compliance problem.

Time for platform engineering

At first glance, this seems too good to be true. Right? But in the future, every company will be able to use platform engineering to increase productivity and achieve enormous savings. Ask your teammates if they’ve ever created resources for testing purposes with a cloud provider. If they say yes, ask them if they deleted those resources afterwards. Few will have done so. It's easy to imagine how much money is wasted this way. Platform engineering can delete resources just like it creates them. This makes it easy to implement a lifecycle for resources.

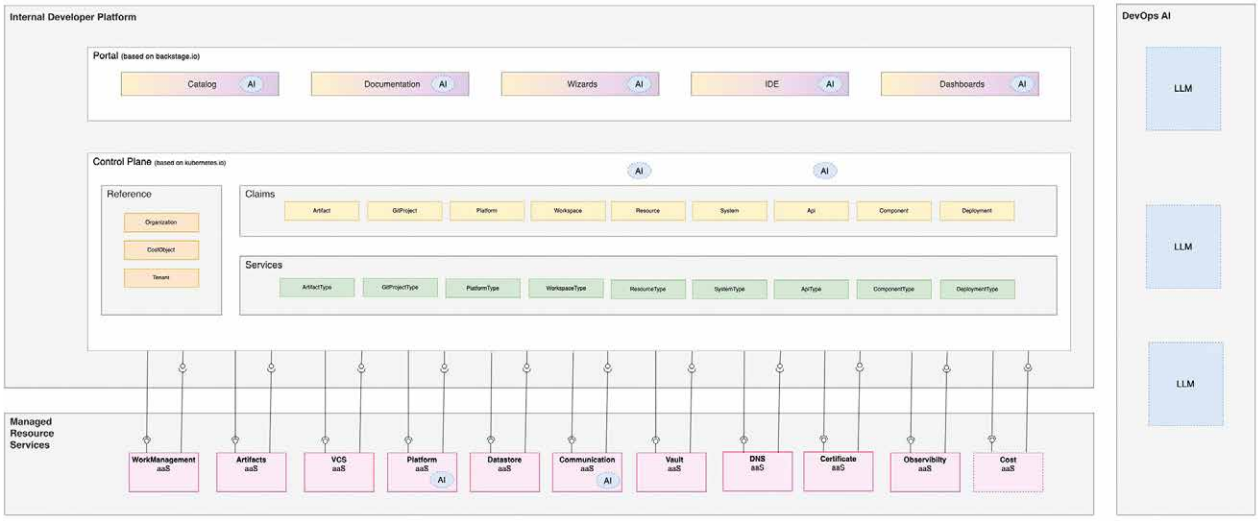


FIG. 3 Managed services as Kubernetes APIs

Architecture

We spent over a year developing an internal developer platform, including an open architecture, from the initial vision through prototypes to the currently defined production-ready, scalable, and maintainable architecture.

TRACK

DevOpsCon

Kubernetes & Cloud Native:
Containerize for a cloud-native approach.

Master containerization fundamentals, deployment strategies, and advanced orchestration to manage and scale containerized apps with tools like Kubernetes, Docker Swarm, and Apache Mesos. Gain expertise in running secure, cloud-native applications across public clouds, private clouds, or on-premises environments.

Learn from Industry Leaders about:

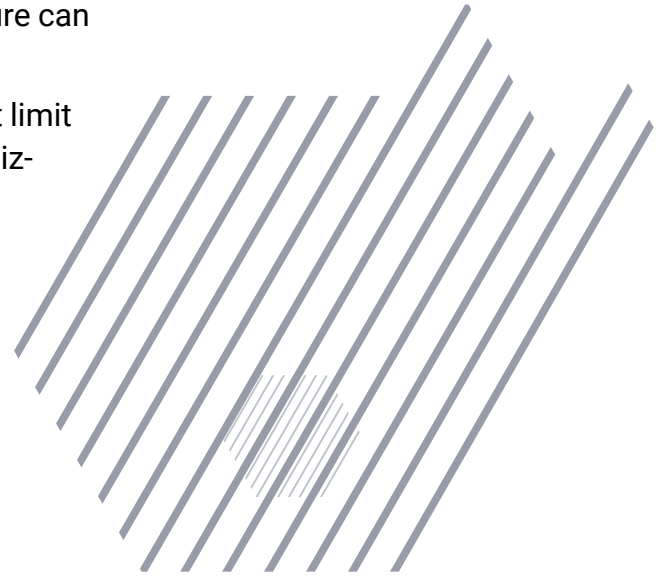
- **Containerization Basics:** Create and manage containers with Docker and Containerd for cloud-native deployments.
- **Kubernetes Core Concepts:** Master Kubernetes for orchestrating and scaling containerized applications.
- **Advanced Orchestration:** Explore multi-cluster management and high-availability setups with Kubernetes, Docker Swarm, and Apache Mesos
- **Deployment Techniques:** Learn rolling updates, blue-green, and canary deployments for containers.
- **Security Essentials:** Implement strategies to secure containerized environments across cloud and on-premises.
- **Scaling Containers:** Use Kubernetes Autoscaler and KEDA for smart, demand-driven scaling.

Initially, the idea was for a team of platform engineers to abstract all the services to be integrated and then make them available in the IDP. However, this was fraught with numerous problems. The main issue was the amount of manpower required. A relatively small team needed to devote itself intensively to planning, implementing, and integrating all services in the company. An IDP is not primarily about developing new services, but about abstracting existing services and integrating them into the platform. With hundreds of services, this isn't feasible for a small team.

Another problem is that not every team member has the in-depth expertise that other teams built when the services were developed. The question of how to provision complex dedicated Kubernetes clusters isn't easy to answer. Proprietary technologies and concepts also means that additional training is needed for other platform engineering employees. This specialized knowledge would only be used for the company itself.

Now, there are frameworks like Crossplane for abstracting services. These make it possible to abstract service provisioning, but even here, small platform teams aren't able to encapsulate all the necessary services and connect them to the platform. Another idea was to develop a proprietary domain-specific language (DSL) based on YAML, which would also have to be learned by other developers. All in all, this undertaking raises doubts about its implementation and gives pause for thought.

Therefore, in order to avoid vendor lock-in and be tied to a single manufacturer like AWS or Microsoft Azure, we looked at an abstraction layer that already exists and has huge community support: Kubernetes. Just like with Crossplane, Kubernetes is used as a generic API and can also do much more than classic REST APIs. For example, reconciling provided by Kubernetes for pods can also be used for general API handling, like retries, in the event of failed API calls. The Kubernetes architecture can also be easily extended to provide new resources.



Unlike Crossplane, our operator technology doesn't limit you to a single flow direction. Crossplane synchronizes resources from Kubernetes to external sources, whether Amazon AWS or GitLab. But it can also be used from Kubernetes to Kubernetes using a workaround. Moving from GitLab to Kubernetes is significantly more difficult. Crossplane can still be used by our service providers however, and we don't have to dictate how they should implement their services.

The path to production

We'd like to use the Uber platform for ride-hailing services as a comparison to visualize the IDP. Uber is viewed as a ride-hailing service broker. It provides value-added services such as billing, brokerage, ratings, and marketing for transportation providers. The IDP we designed works similarly. The IDP team provides a broker for Kubernetes API-based resources with added value to the actual services.

Managed services providers can use the platform to offer their services to other developers. Additional services like orchestration, internal billing, cost management, and visualization in the developer portal are provided by the IDP. The advantage for service providers is that they only need to provide their services as Kubernetes Custom Resources (CR). This usually involves creating a Custom Resource Definition (CRD), a YAML file with an Open API specification, and implementing it with a controller and operator that encapsulate the business logic. For more details, see the Custom Resource Definitions. The main focus is that services implemented as Kubernetes resources can be operated independently of an IDP, since they only need to comply with the Kubernetes standard. This eliminates the need for proprietary concepts, increasing acceptance within the company and among developers since they don't need proprietary knowledge and can adhere to open standards just like with all Kubernetes APIs.

Teams that provide services as Kubernetes resources also benefit from no longer having to implement and maintain their own ordering portals, since the IDP can generically visualize them in the portal. This frees up more capacity for optimizing existing functionalities and developing new features. All services registered in the IDP can be ordered with the uniform portal.

But let's get back to architecture. In principle, there are three main levels:

- Developer level
- Control level
- Managed services

Developer level

The developer plane refers to tasks controlled by a web interface and passed on to the control plane. This includes managing resources provided by managed services and scaffolding/order processing. In our example, we use Backstage.



Backstage is an open-source project originally developed by Spotify. It plays a central role in IDP implementation and management. It serves as a developer portal and allows teams to consolidate and manage their development processes, tools, and services. The most important IDP-related Backstage roles include:

- **Centralization and standardization:** Backstage provides a unified user interface that gives developers access to various tools, services, and documentation. This promotes standardization and simplifies the use and management of development resources. Backstage is implemented based on TypeScript, the React framework, and Material UI.
- **Plug-in architecture:** With its modular plug-in architecture, Backstage enables the integration of different tools and services that developers use in their daily work. This includes CI/CD pipelines, monitoring tools, cloud services, and much more. This data is provided by the control plane and used by Backstage for visualization.
- **Developer Experience:** Backstage improves Developer Experience with an intuitive user interface and easily accessible information. Developers can quickly access the resources they need, create and manage projects, and perform deployments.

TRACK

DevOpsCon

DevSecOps:
Shield the Production Cycle – Ship with Confidence.

Secure your SDLC with industry-leading solutions. Start threat modeling early and automate security testing throughout the lifecycle. Shift left with IaC scanning, software composition analysis, and other tests, while managing dependencies and securing cloud environments, open-source components, and Kubernetes deployments.

Learn from Industry Leaders about:

- **Shift Left Security Practices:** Integrate security into the early stages of development with IaC scanning, software composition analysis, and more.
- **Proactive Cloud Security:** Implement automated configuration management and security measures to protect cloud environments.
- **Open-Source Dependency Management:** Manage open-source dependencies and identify vulnerabilities using modern tools.
- **Generative AI for Enhanced Security:** Leverage AI for automating security tests and improving code quality during the development process.
- **Cloud Security Best Practices:** Safeguard your Kubernetes and cloud-native deployments with strategies for preventing vulnerabilities.
- **Security-Driven DevOps:** Learn how to integrate security tools and processes to maintain continuous security and compliance throughout your DevOps pipeline.

- **Cataloging and discovery:** Backstage provides a service catalog that catalogs all internal services and components. This makes it easier to find and reuse existing services and promotes collaboration between teams.
- **Automation and self-service:** Backstage supports automating recurring tasks and has self-service functions so developers can work without the support of infrastructure or DevOps teams. This accelerates the development cycle and increases productivity.
- **Documentation:** The TechDocs plug-in allows you to create centralized documentation. Documentation can be maintained for each resource in the catalog and searched globally.

When a user fills out and submits a template, a corresponding action ensures that further actions are executed in the control plane and the desired resources are created.

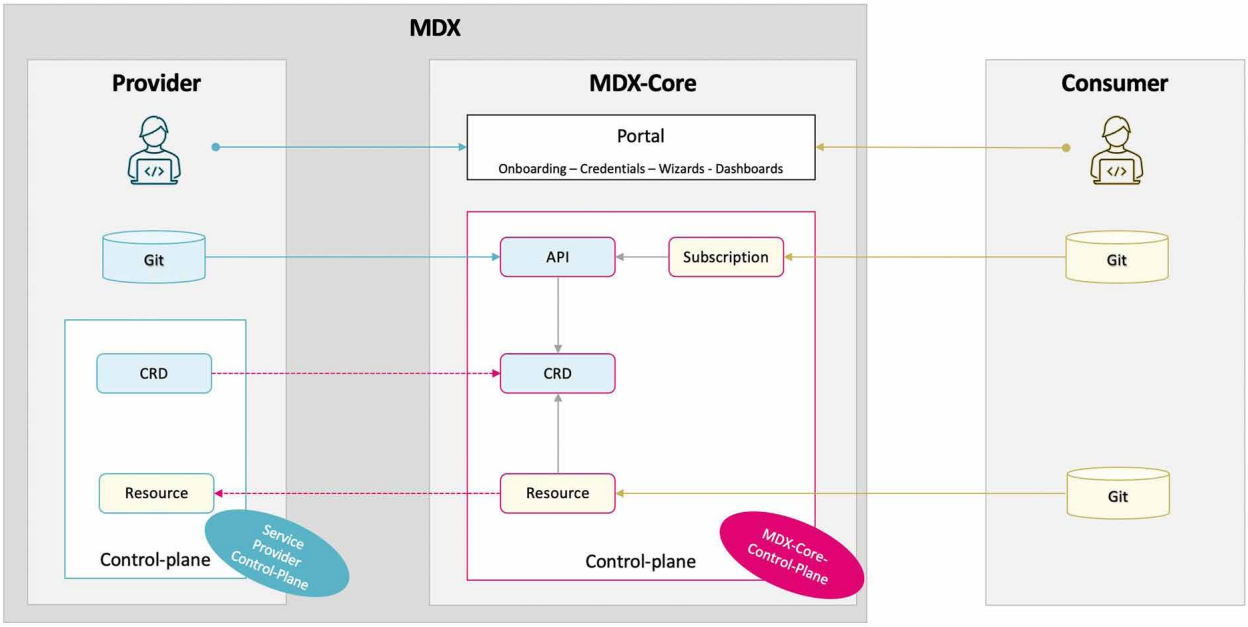


FIG. 4 Control planes are the core of IDP

The Control Plane

The actual central control plane orchestrates the services. The Kubernetes API specification forms the basis for everything. The control plane is a collection of Kubernetes APIs that provide value-added services for managed service providers. The core components of the control plane are core APIs and providers, which are Kubernetes APIs.

Consumers can use the APIs offered by the providers. Each provider, which basically offers its managed services as Kubernetes APIs, gets its own control plane and greatly simplifies scalability. The central control plane essentially acts as a broker for service mediation.

The providers are developed and deployed by the teams providing the managed services. In practical terms, the provider teams implement Kubernetes APIs instead of Rest APIs. Providers can integrate easily consumable services, allowing more complex services to be developed based on atomic services.

When onboarding new services, teams can take the necessary steps for registration with a self-service option in the developer portal. A Kubeconfig is delivered that providers can use to provide their APIs and subscribe to other services accordingly.

Providers continue to operate their managed services and exchange a Kubeconfig with the IDP for smooth operation. The IDP copies the provider's custom resource definitions to the central control plane. This establishes bidirectional communication between the IDP and the provider's control plane.

Admittedly, this sounds fairly complicated, but implementation is streamlined and can be done quickly with a little practice. As mentioned, one advantage is that the implemented Kubernetes APIs/providers can also be used independently of an IDP.

Workspaces

Another service for developers is an entire development environment based on Coder [1]. This is essentially a browser-based environment with Visual Studio Code and a terminal. The entire workspace configuration, which functions similarly to GitHub Codespaces, is achieved with a Nix configuration [2]. This allows internal and external project team members to be onboarded quickly and securely, significantly speeding up the process of getting started on a project.

Outlook

At the current IDP stage, the focus is on implementing a minimum viable product (MVP) that provides the IDP and the developer portal. Then, around a dozen managed services will be integrated into the IDP and available to developers. It should all grow organically. Provider teams must be given expertise and best practices by the core platform team. As a general rule, an IDP should be implemented in the same way as a commercial project. This includes marketing, sales, community management, and the appropriate budgets.

During the research phase, it was useful to always be open and proactive about everything currently in the pipeline. We drummed up internal support and invited external partners and companies to discuss the architecture. We plan to make the architecture and IDP components available as open source, as we see this as an opportunity to give something back to the community. Other companies are on the same wavelength, so we may end up with a community that identifies with the specific implementation using Kubernetes APIs and continues to expand it. Therefore, we encourage you to get in touch with us to discuss this!

Links & Literatur

[1] https://coder.com/
[2] https://nix.dev/



Thomas Ganter is an architect and product manager for developer tooling and self-service at Deutsche Telekom IT GmbH. Has successfully oversaw the transformation of the internal API architecture. Prior to this, he worked in a wide variety of roles along the software value chain in various industries and companies since the 1990s.



Marcus Greuel is Lead Software Architect and Platform Engineer for the Magenta Developer eXperience at Deutsche Telekom IT GmbH. Even with over 30 years of architectural experience, he still has a keen sense for upcoming IT trends.



Christian Hartung is a platform engineer in the core team of Magenta Developer eXperience at Deutsche Telekom IT GmbH. He primarily develops in the area of Kubernetes operators. Go is his preferred language for implementing cloud-native Kubernetes APIs. Furthermore, Christian is a member of the Kubernetes community and the Kubernetes Open Source Foundation.



Michael Johann is Principal Platform Engineer for the Magenta Developer eXperience at Deutsche Telekom IT GmbH. He focuses on designing the Developer Plane with Backstage. He was previously freelancing for over 30 years and has repeatedly reported on current trends and future standards at the JAX and WJAX conferences.

DevOpsCon

THE GLOBAL CONFERENCE SERIES FOR **DEVOPS & BUSINESS TRANSFORMATION**

