THE INTERNAL DEVELOPER PLATFORM
Bringing Unity to Platform Engineering

TABLE OF CONTENTS

3 Introduction

4 Who is the Prime Minister of Platform Unity (Internal Developer Platform)?

5 What Do Engineering Teams Actually Need from a Developer Portal?

6 What Makes a Developer Portal Actually Work

8 Internal Developer Portal vs Internal Developer Platform

10 DIY Pitfalls: When Teams Build Their Own Portal-Only Setup

12 What Actually Defines an Internal Developer Platform

16 Choosing the Right Architecture for Your Team

18 Cycloid: The Platform Layer Your Portal Is Missing

23 Frequently asked questions

INTRODUCTION

Welcome to the world of platform engineering, and your introduction to the Internal Developer Platform (IDP) that helps it all run smoothly.

Did you know that a group of owls is called a parliament? Well, we can tell you now it is. And as our logo is an owl, we saw an opportunity to talk to the world about platform engineering and parliaments, connected. Why? Well, let us tell you.

Imagine a **Parliament of Owls** in session as you saw on the front cover of this eBook - a metaphor for Cycloid's platform engineering features. In this eBook we will meet the **Prime Minister of Platform Unity** - otherwise known as the **Internal Developer Platform (IDP)**.

Cycloid's Parliament of Owls is a tableau on platform engineering, representing Cycloid's offering: a comprehensive platform that balances agility with governance, innovation with cost control, and automation with human oversight.

The eBook will guide you on how we see an IDP working for your organization. However, as parliaments reflect the societies they come from (just as the IDP reflects the organization it serves), each one is different, with different rules and methods of operation.

Who is the Prime Minister of Platform Unity (Internal Developer Platform)?

The Prime Minister of Platform Unity presides over the parliament, making sure all departments work in harmony under a single vision. This wise leader embodies Cycloid's Internal Developer Platform (IDP) - the central brain that unifies all DevOps tools and processes into one coherent whole.

Just as a real Prime Minister coordinates various ministries, Cycloid's IDP coordinates workflows between development, operations, security, and even finance and sustainability teams. The result is a stable "government" of DevOps where everyone, from developers to operators, collaborates efficiently under one roof with full visibility.

The Prime Minister Owl guarantees a cohesive strategy, so every feature and team in the platform moves in the same direction toward continuous delivery excellence.

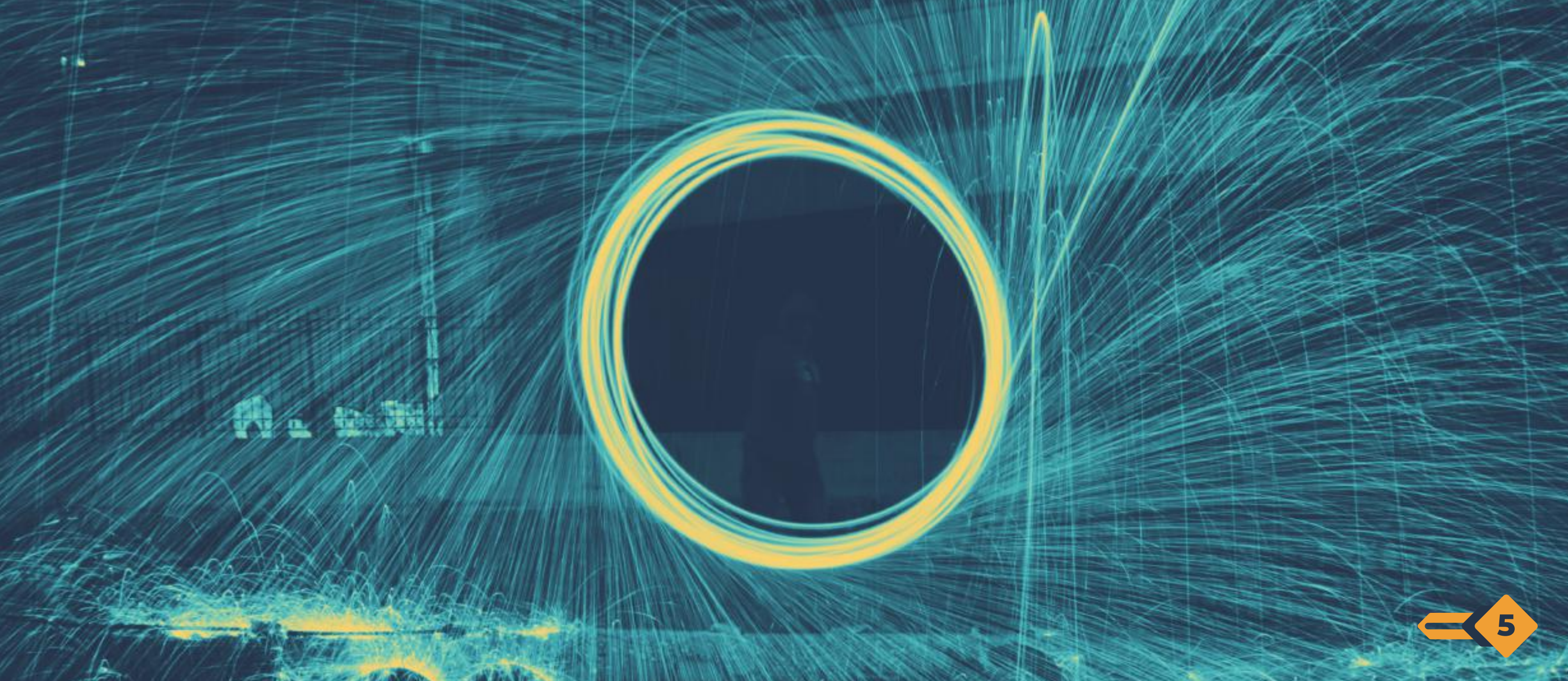
What Do Engineering Teams Actually Need from a Developer Portal?

Developer portals are only effective if they support engineering workflows - provisioning infrastructure, deploying code, and managing services. Many portals start strong with dashboards and service catalogs, but without integration into automation layers, they become read-only and irrelevant.

Static service metadata, manual triggers, and disconnected workflows lead to abandoned portals. Engineers default back to Slack and tickets when portals can't take action. A useful portal must do more than display metadata - it must connect with CI/CD, Git, and runtime systems to trigger real change.

Successful developer portals become control planes. They trigger workflows, enforce standards, and reduce developer friction. A portal should work less like a static directory and more like an active governing body driving work forward.

> ***An ineffective portal (one that cannot take action) is like a parliament that debates but never passes any laws - lots of discussion but no action.***



What Makes a Developer Portal Actually Work

1 Live, Accurate Service Catalogs

Real-time sync with Git, CI/CD, and incident tools ensures accurate metadata and reduces context switching. The catalog becomes a control point - not a static dashboard, but a well-stocked hub of services that everyone can trust.

2 Golden Paths

Standardized workflows for creating new services using approved templates, pre-wired pipelines, and default observability tools. No more reinventing setup per project.

3 True Self-Service

Trigger real actions from the UI - like provisioning infra, launching preview environments, or fetching secrets - without ticketing systems.

> ***"This is the same as cutting through bureaucratic red tape so developers can self-serve.***

4 Operational Scorecards

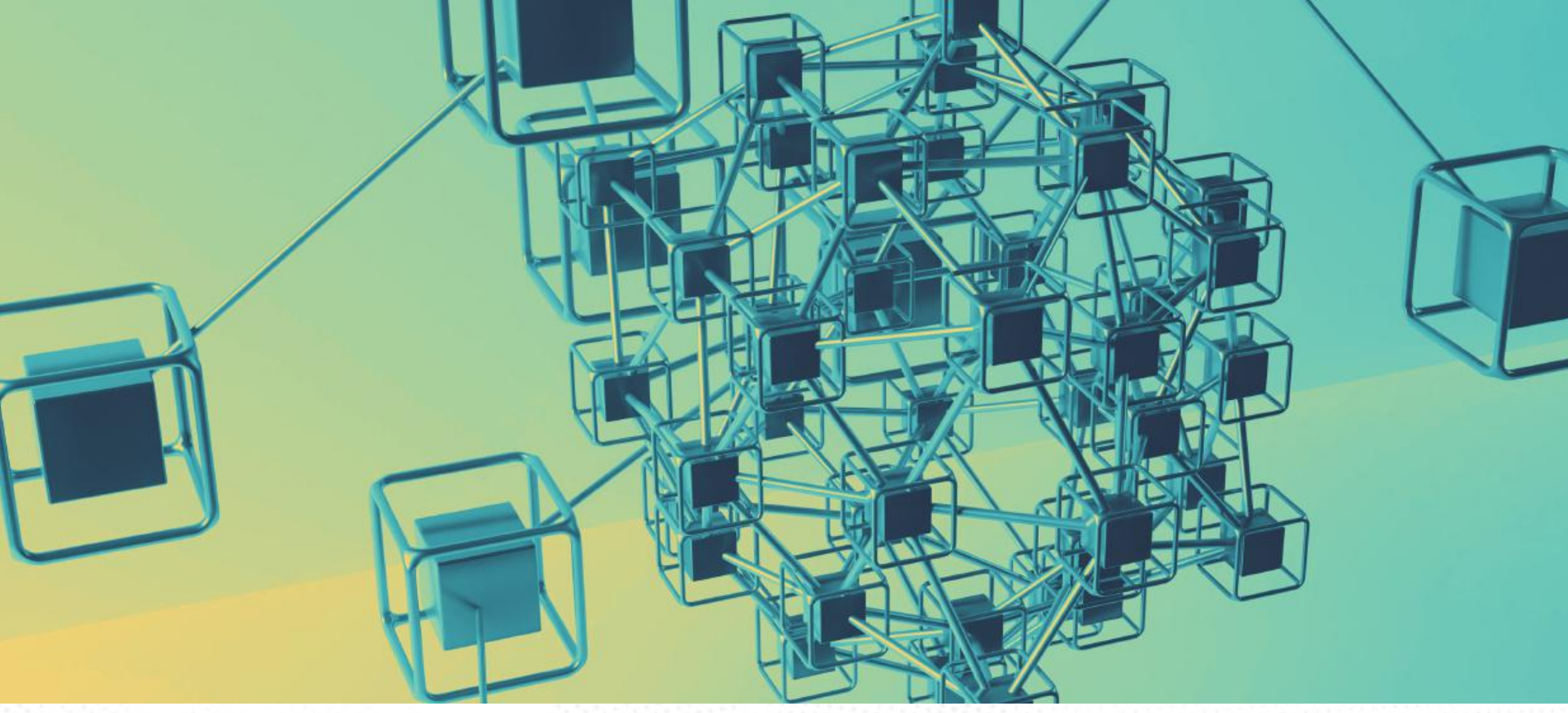
Real-time sync with Git, CI/CD, and incident tools ensures accurate metadata and reduces context switching. The catalog becomes a control point - not a static dashboard, but a well-stocked hub of services that everyone can trust.

5 Built-In Governance

Role-based access, approvals, policy-as-code, and quota enforcement keep teams safe while enabling autonomy.

6 Native Toolchain Integration

Connects to GitHub, Jenkins, Terraform, Vault, Datadog, and more - enabling real workflows in the tools developers already use.



Internal Developer Portal vs Internal Developer Platform

Let's clear up a common confusion - the difference between a developer portal and a developer platform. These terms often get used interchangeably, but they solve very different problems.

Focus	Developer Portal	Developer Platform
Purpose	Provide visibility, metadata and UI	Powers automation and workflows
Interface	Web-based UI for developers	APIs, pipelines, GitOps & IaC
Main Users	Developers and product engineers	Platform engineers & SREs
Key Capabilities	Service catalog, docs, ownership, scorecards	Infra provisioning, scaffolding, deployment, secrets management
Automation Support	Limited - needs external backend	Built-in - runs actual workflows
Setup Complexity	UI-focused, but often requires plugins	More backend-heavy but automates delivery
Example Failure Mode	Becomes a static dashboard	Becomes a black box without UI

Most open-source portals like Backstage focus on the UI and metadata layer. They help teams centralize service ownership, surface documentation, and track things like pipeline runs or on-call contacts. That's valuable, but it stops at visibility. The portal is essentially a user-friendly catalog (imagine a public bulletin board or an information minister making announcements) with no built-in execution power.

Let's say a developer wants to spin up a new feature environment. With just a portal, they'll likely:

- **Find a doc with instructions**
- **Clone a template repo**
- **Request infra via a Slack thread or a Jira ticket**
- **Wait on the platform team to approve and provision**

The portal might show metadata, like which team owns the staging cluster, but it can't actually deploy code or set up resources on its own.

> ***In other words, it lacks the "ministers" (automation components) to carry out the work.***

That's where internal developer platforms (IDPs) come in. They handle the execution layer:

- **Provisioning infrastructure with Terraform or**
- **Deploying apps with Helm or ArgoCD**
- **Injecting secrets from Vault or AWS Secrets Manager**
- **Enforcing RBAC and guardrails through policy-as-code**

So instead of asking for help, the developer clicks "Create Environment" in the UI, and the platform does the rest automatically.

It's also worth noting that Cycloid is both a portal and platform, reflecting this dual role.

> ***The platform doesn't just display data, it runs workflows safely and consistently, effectively acting as the executive branch that executes decisions.***

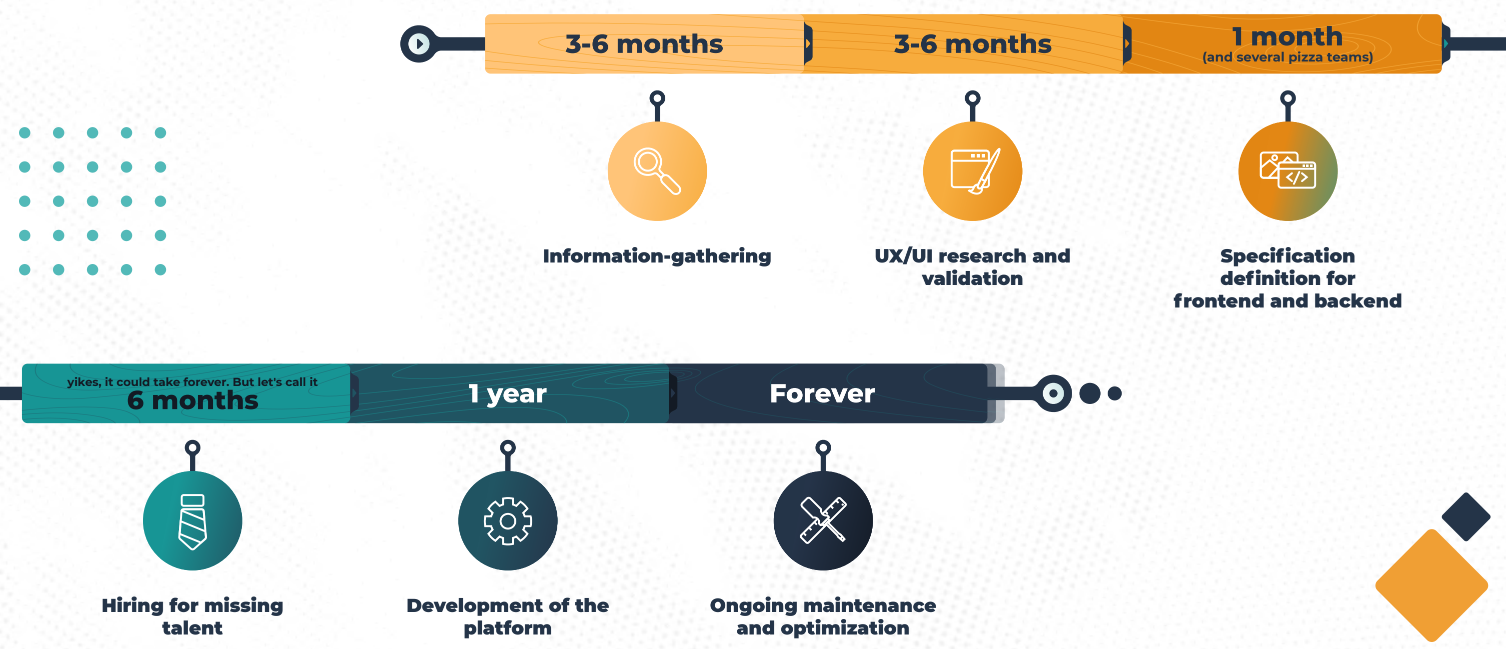
DIY Pitfalls: When Teams Build Their Own Portal-Only Setup

Many teams try to build both layers themselves, starting with open-source frameworks like Backstage. While the goal is to create a full internal developer platform, the reality is often more complex. What begins as a quick setup frequently turns into a multi-year engineering effort.

It can take more than three years and 20 developers to create a developer platform - which still leaves the problems of maintaining it, as reported in the State of DevOps report 2024.

Typically:

- **It takes months just to populate a working service catalog**
- **Custom plugins are needed to connect to tools like GitHub, ArgoCD, or Terraform, and maintaining them becomes a constant overhead**
- **Automation is still missing, so developers rely on tickets for infra requests**



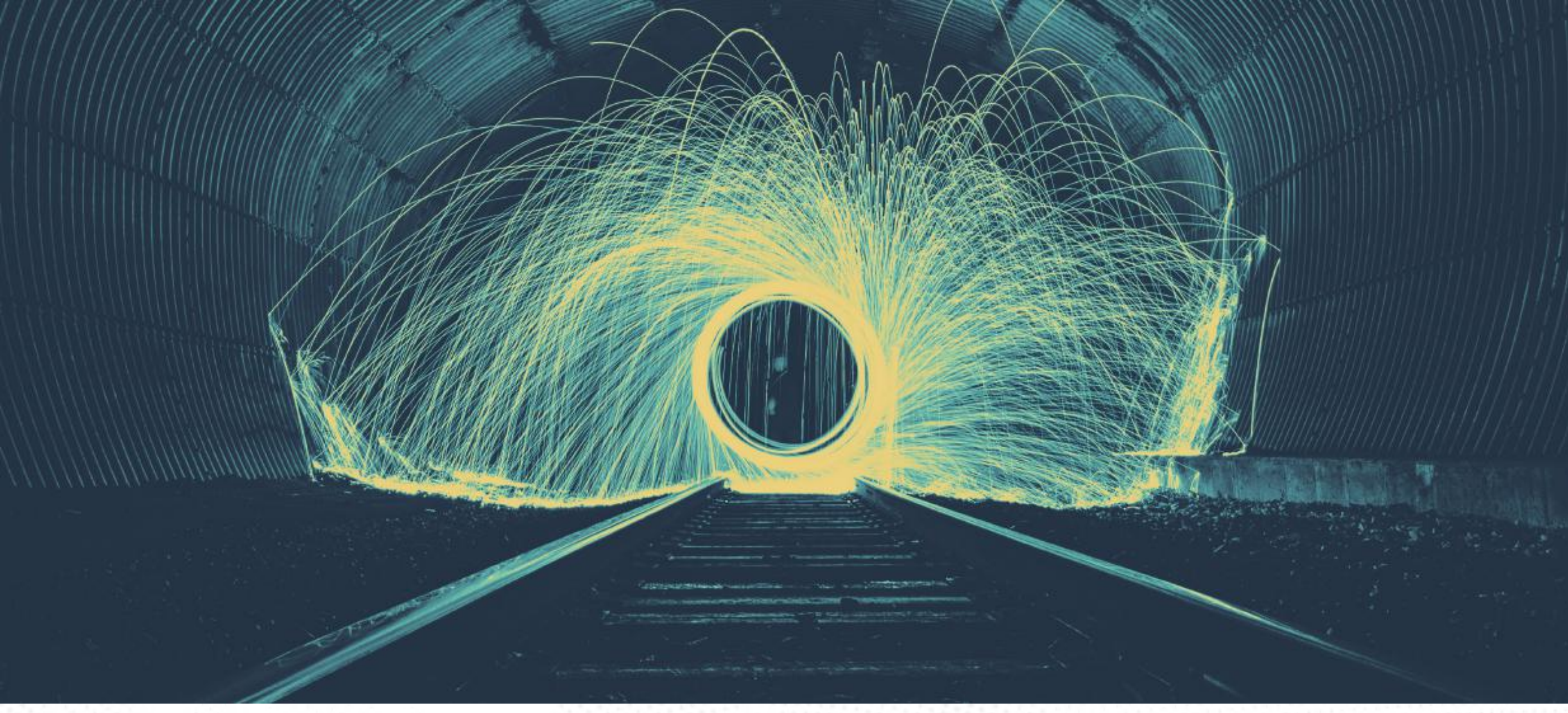
In discussions, a common pattern emerged: deep into rollout, many teams still don't have a usable portal. The reason? No automation layer underneath. The result? A portal that displays metadata but can't trigger workflows. It becomes a read-only mirror, not a real control plane.

Meanwhile, workflows remain disconnected. Developers still rely on Slack threads, internal docs, and ticket queues to request infra or access secrets. The portal shows data but doesn't help get work done.

> ***In our analogy, you've assembled a council that can observe (show data) but not govern (make changes).***

The portal is there, but without an automation layer beneath it, nothing gets executed. This is why many homegrown portal projects fail to gain traction. The lesson here is clear: an internal tool that only observes and catalogs, without the ability to act, will struggle to justify itself.

> ***You need the execution capabilities (the Owl Ministers carrying out tasks) to truly unlock developer self-service and efficiency.***



What Actually Defines an Internal Developer Platform

So far, we've seen why portals alone often fall short - and how many end up as just static dashboards. Now let's focus on the part that actually drives delivery: the platform layer.

An internal developer platform (IDP) isn't about showing metadata. It's about running things behind the scenes - provisioning infrastructure, managing environments, enforcing policies, and helping teams move fast without breaking things.

> ***If you go beyond the UI, here's what a real platform is responsible for (the key "ministries" of an IDP, so to speak)***

1 Infrastructure Orchestration

A core function of any internal developer platform is automating infrastructure provisioning - safely, consistently, and without manual steps. This goes beyond clickable UIs; it's about triggering reliable, code-defined workflows.

Most platforms integrate with tools like:

- **Terraform** for provisioning VPCs, databases, and IAM roles
- **Helm** for Kubernetes app deployments
- **Kubernetes APIs** for real-time control of workloads
- **Pulumi** an IaC tool to use the logic power of code
- **Ansible** to control the app layer

Instead of exposing raw IaC, platforms abstract these tools into reusable workflows. Developers request environments or services; the platform executes vetted modules behind the scenes, ensuring consistency, compliance, and speed.

2 On-Demand Environment Creation

A strong internal developer platform lets teams spin up isolated environments automatically - for testing, QA, or troubleshooting - without blocking shared resources or involving the platform team.

Typical use cases include:

- **Preview environments** per pull request, auto-created and destroyed
- **Dedicated QA/staging setups** with RBAC and isolation
- **Short-lived test setups** to control the app layer

Without this, teams rely on shared environments, leading to delays and conflicts. On-demand environments streamline testing and help developers validate changes quickly and safely.

> ***It's like having a rapid permitting office in your platform - granting "building permits" for new test environments whenever needed.***

3 Secure Secrets and Config Handling

Most deployments need credentials - API keys, database passwords, cloud tokens - and managing them manually is risky.

A robust platform should:

- **Integrate with Vault, Doppler, or AWS Secrets Manager** to manage secrets centrally and securely
- **Inject environment-specific configs dynamically** to prevent misdeployments and reduce drift
- **Log access and enforce RBAC,** so you know who accessed what, when, and why

Without this, teams rely on shared environments, leading to delays and conflicts. On-demand environments streamline testing and help developers validate changes quickly and safely.

> ***"It's like having a Minister of Security guarding the vault - developers get the secrets they need without ever handling them directly, and everything stays secure.***

4 Guardrails and Role-Based Automation

Speed without guardrails is risky. A mature platform enforces safety and compliance without slowing developers down. This is where the Minister of Governance (or Law & Order) steps in, ensuring developers move fast within safe boundaries.

Key capabilities include:

- **RBAC** to define who can access or deploy what
- **Approval workflows** for sensitive actions like production changes
- **Quotas and resource limits** to avoid overuse
- **Policy-as-code** with tools like OPA or Kyverno to enforce org-wide rules automatically

These controls empower developers to move fast - safely and within bounds - while giving platform teams visibility and control at scale. It's as if the platform has an internal rulebook that's always enforced, no matter who is using it.

5 Consistent Templates and Standards

Consistency is one of the biggest advantages of a real platform. Without it, teams build services differently, creating drift, delays, and security gaps.

A good platform provides:

- **Pre-configured service templates** with CI/CD, observability, and security baked in
- **Reusable pipelines** to standardize how services are built and deployed
- **Default logging, alerting, and health checks** to avoid misconfigurations

Without this, teams rely on shared environments, leading to delays and conflicts. On-demand environments streamline testing and help developers validate changes quickly and safely.

> ***Think of the Minister of Standards working hand-in-hand with the Minister of Continuous Delivery here: together they ensure every new service follows best practices and uses the approved pipelines and tools, right out of the gate.***

Choosing the Right Architecture for Your Team

There's no one-size-fits-all internal developer platform. The right setup depends on how your team operates, what bottlenecks you face, and how much time and engineering effort you can realistically invest.Here's a breakdown of three common scenarios and how to think about your IDP strategy:

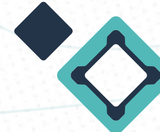
Questions to Guide Your Decision

Answering these will make it clear whether you need a portal, a platform, or both - and how deep you need to go to fix the bottlenecks that are slowing your team down.

Before choosing tools or building workflows, ask:

- **Where are our biggest delays - provisioning, visibility, or consistency?**
- **Do developers rely on Slack or tickets for basic tasks?**
- **Is our portal connected to automation, or is it just a dashboard?**
- **How often do services drift from standards? Do we even know?**
- **Can new services go live without platform team involvement?**

	Early-stage Startups	Scaleups	Enterprises
What they need	Fast setup Minimal tooling Low overhead	Standardization Automation across environments Better platform team communication	Strong governance Multi-cloud or hybrid cloud support Scalable RBAC & policy enforcement
Typical issues	No consistency across services Devs copy old repos or rely on tribal knowledge	Teams creating services differently Manual CI/CD pipelines & infra requests Ownership & maturity unclear	Compliance requirements Fragmented tooling across teams High coordination costs
Approach	Start with a lightweight developer portal for visibility Add templates and golden paths for scaffolding Hold off on full automation unless it's blocking delivery	A platform layer beneath the portal Enable true self-service for infra and deployment Enforce consistency with scorecards and golden paths	Prioritize platforms with integrated automation Implement policy-as-code and strong audit trails Connect to existing identity, secrets, and infrastructure systems



Cycloid: The Platform Layer Your Portal Is Missing

For many teams, building a portal helps surface information - like who owns which service, the status of recent CI runs, or links to documentation and dashboards. But beyond visibility, most portals don't offer direct interaction with the underlying infrastructure.

Tasks like provisioning a new database, deploying a feature branch, or spinning up a fresh environment still involve opening tickets, waiting for approvals, or coordinating over chat. There's no automation layer beneath the UI - so even with a sleek front-end, teams end up relying on manual handoffs to get work done.

> ***It's like having a beautiful city hall with no civil servants inside: plenty of form but no function.***

That's where [Cycloid] steps in. It gives you the automation backbone most portals are missing - effectively filling your portal with a capable team of Ministers ready to execute.

Cycloid is a production-ready internal developer platform that plugs directly into your Git repos, cloud accounts, and infrastructure code.

> *It doesn't replace your portal if you already have one - think of it like appointing all the necessary Ministers under an existing figurehead.*

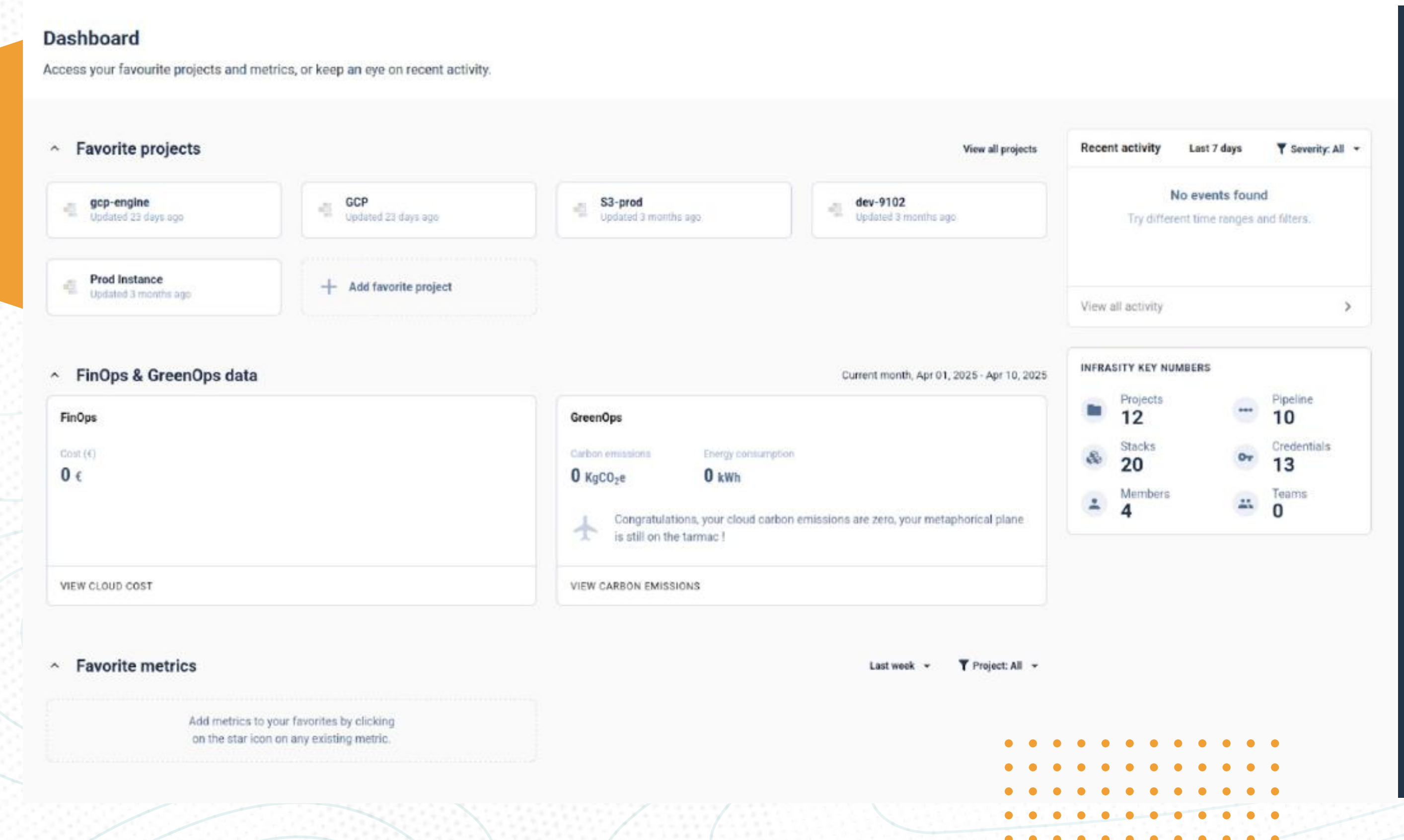
In other words, Cycloid gives your portal a real execution layer underneath.

What Platform Teams Use It For

Cycloid gives platform engineers a control plane that's actually usable out of the box. No need to write endless glue code or manage brittle plugin chains.

Here's what it looks like in practice:

- **You define reusable infrastructure stacks using Terraform, Helm, or Ansible. These can be anything from an S3 bucket to an entire EKS cluster.**
- **Those stacks are exposed to developers via a clean UI or API, with RBAC, tagging policies, quotas, and region restrictions baked in.**
- **You configure Git repos, secrets, environments, and pipelines once - then let teams launch and manage infra without needing you in the loop every time.**



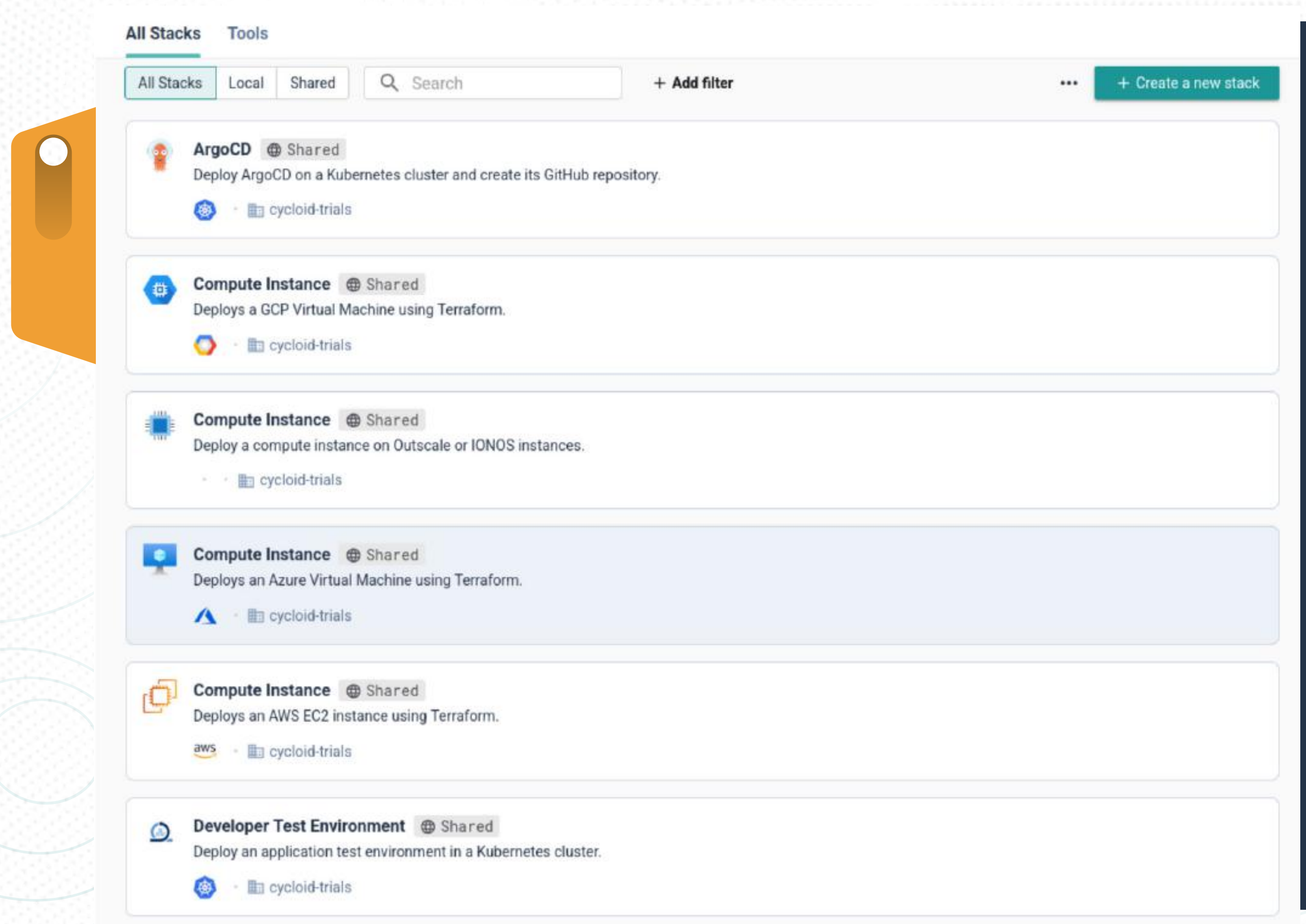
This is Cycloid's control panel for platform teams - live project status, cost usage, credentials, pipeline count, and security events, all visible without switching tools.

This setup replaces homegrown Jenkins jobs, Slack-based approvals, and tribal knowledge with a single system built to handle scale and complexity.

What Developers Actually Get

From a developer's point of view, Cycloid is a self-service interface that actually does something.

- **Need a new staging environment? Launch it via the UI or API-backed by Terraform and Kubernetes underneath.**
- **Want to deploy a feature branch? Run the pipeline straight from Git and watch it show up in the environment.**
- **Need a Redis instance or an S3 bucket? Choose from pre-approved stacks and launch without asking anyone.**



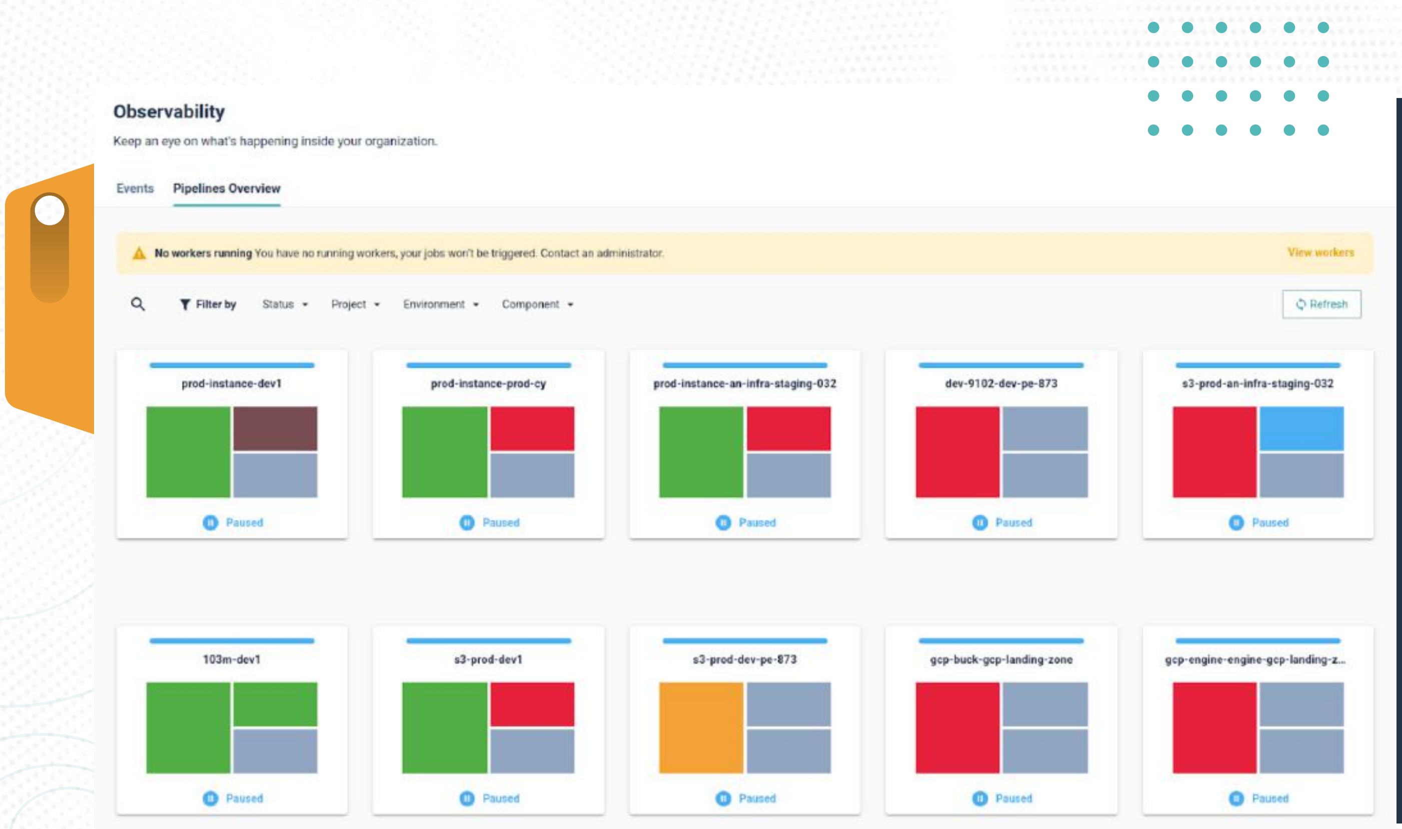
Here's where developers select from a catalog of infrastructure templates - each stack tied to automation and policy. No YAML writing. No guessing.

It works without needing devs to understand Terraform or Helm - just click, deploy, and go. The platform layer does the heavy lifting.

You Can Use Cycloid As-Is or Under the Hood

Cycloid works well as a standalone interface, but many teams plug it into their existing portals. If you're using Backstage or another front-end, you can point workflows back to Cycloid to do the actual automation.

> ***"It's like hiring Cycloid's expert "civil service" to operate behind the scenes of your current portal UI.***

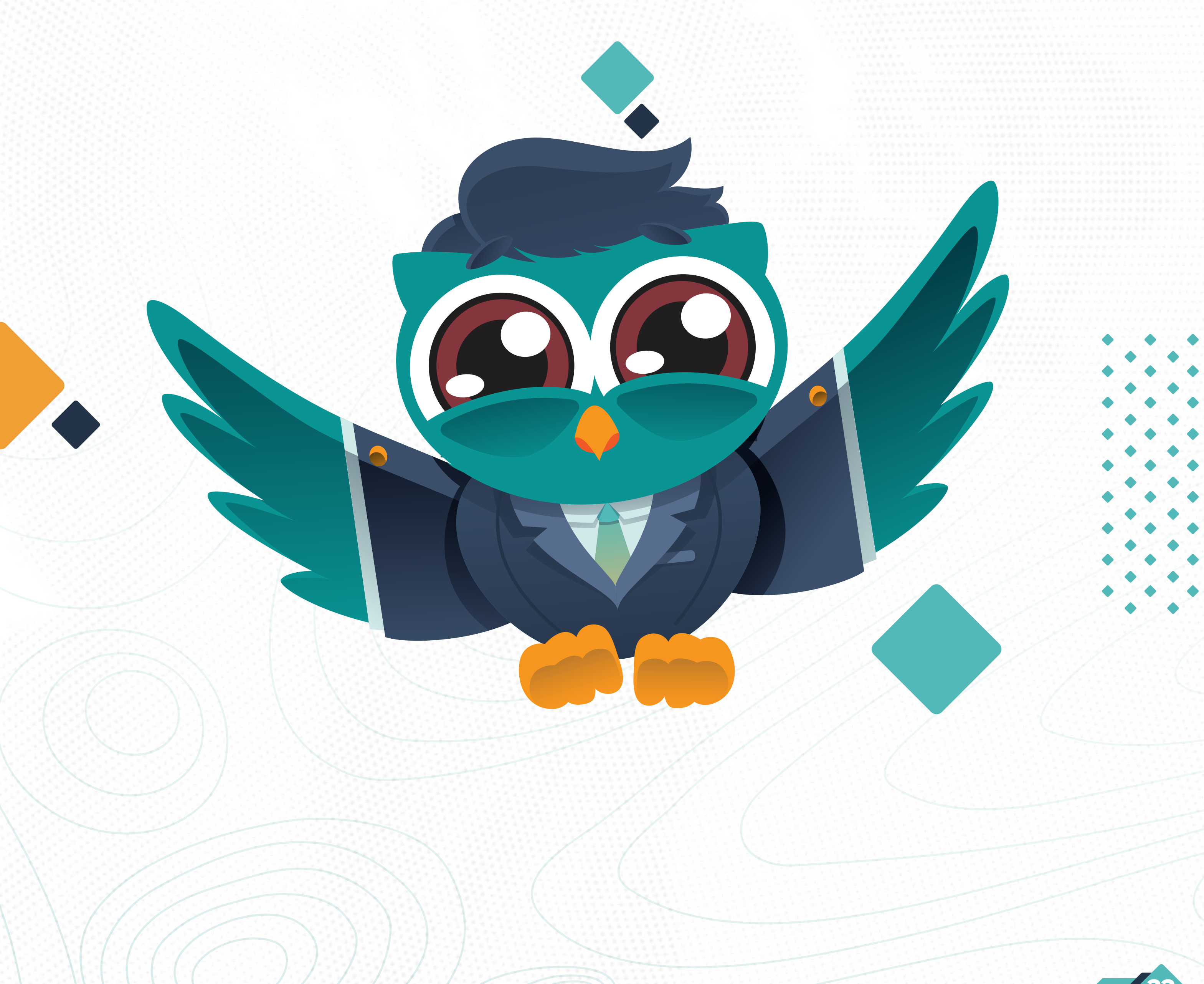


This view shows real-time pipeline status for running jobs across environments-color-coded for quick debugging and observability.

In fact, we've seen teams deploy Cycloid as the automation layer behind a portal that only handled metadata before. Once you hook it up, service creation, deployments, environment setup - it all just works.

What makes Cycloid different is that it's opinionated where it matters. You don't have to build a hundred integrations just to get started. You plug in Git, cloud credentials, and your IaC modules - and it gives you a functioning platform in a few hours, not months.

If your portal is already live but feels disconnected from how work actually gets done, Cycloid helps close that gap. And if you haven't built one yet, it gives you both layers - visibility **and** execution, from day one.



FREQUENTLY ASKED QUESTIONS

What's the difference between a developer portal and a developer platform?

A developer portal surfaces metadata - like ownership info, documentation links, and CI status. A developer platform runs actual workflows - like provisioning infrastructure, deploying applications, or managing secrets. Essentially, the portal is the front-end interface (the showroom or dashboard), whereas the platform is the engine behind the scenes that executes the work.

What makes a developer platform production-ready?

Being production-ready means having built-in orchestration and automation (so you're not manually operating it), support for Infrastructure as Code tools (Terraform/Helm, etc.), secure secrets management, guardrails via RBAC and policy enforcement, and the ability to onboard teams quickly without a ton of custom development. In short, a true IDP comes with the key capabilities (infrastructure, security, compliance, etc.) out-of-the-box so you don't have to build them all yourself.

Do I need both a portal and a platform?

Ideally, yes - if you want a great developer experience. The portal helps devs discover and understand resources (a bit like a helpful guide), and the platform lets them take action (actually provisioning or deploying those resources). You can have one without the other, but you'll get the most benefit when they work together.

What's the risk of building your own from scratch?

The biggest risk is time and complexity. Many DIY portal projects take 12-18 months just to deliver basic value. They require ongoing maintenance (especially as underlying tools change), and often they never achieve true self-service because building the automation layer is hard. You may spend a year or more and still end up with something that's essentially a fancy dashboard without the ability to execute workflows - which defeats the purpose of an IDP. Using an integrated solution or platform product can save a lot of this effort and get you to value much faster.