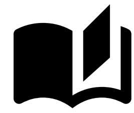
port

The practical guide to internal developer portals



The next big thing in DevOps is platform engineering, and the main tool it uses is the internal developer portal. Read this guide to understand what can be done with portals and why they matter.

Table of contents

INTRODUCTION

The era of platform .... 3 - 4

CHAPTER 1

Overview of internal developer portals .... 5 - 13

CHAPTER 2

Blueprints .... 14 - 16

CHAPTER 3

Software catalog 17-20

CHAPTER 4

Developer self-service actions .... 21 - 23

CHAPTER 5

Scorecards .... 24 - 28

CHAPTER 6

Automations .... 29 - 30

CHAPTER 7

Reporting and executive views .... 31 - 34

CHAPTER 8

Extensibility .... 35 - 37

CHAPTER 9

Adoption strategy .... 38 - 40

INTRODUCTION

The era of platform

Hello, platform engineering.

The world has changed.
The traditional DevOps approach is changing, giving way to platform engineering: an end user-focused approach to promoting developer autonomy, standardization, golden paths, and excellence in the engineering org. Platform engineering is exploding in popularity. According to research from Puppet Labs, 51% of organizations have adopted platform engineering or plan to do so in the next year.
In the world of platform engineering, developers are the "customers." And the goal is to empower them with tooling and resources to promote productivity and a culture of quality.

What's behind the shift?

The explosion and widespread adoption of microservices, APIs, and open source technologies in recent years has created greater opportunities than ever for innovation and collaboration. At the same time, devs are facing new challenges:

- The complexity of Kubernetes, which has imposed a staggering cognitive load to developers who aren't K8 experts
- The migration of developers from on-prem to SaaS development, requiring new toolkits and technologies
- Increasing fragmentation of the developer experience across a growing number of tools and frameworks

Many organizations realize that the benefits of developer autonomy and innovation need to be paired with guardrails and "golden paths": the notion that there's a single supported approach for a given enterprise to do things like build a backend service or create a data pipeline.

Enter the internal developer portal

Enter the internal developer portal ("portal"): a new product category with the potential to improve developer autonomy and productivity, set clear and consistent quality standards, and promote visibility into engineering activities and outcomes throughout the organization.
But…what exactly is an internal developer portal? And how should companies think about the role it can play for them – and adapt it to their needs?

We wrote this practical guide to introduce platform engineers, developer experience professionals, and DevOps leaders to the thought process that guides us at Port. This isn't a plug for our technology; it's an honest discussion of a category that has the potential to transform the way engineering organizations operate. Our goal is to highlight the opportunities, the key decision points, and the cultural and engineering transformation involved in adoption of a portal for your organization.

Here's how the practical guide is structured:

- We'll start with some basic disambiguation of internal developer platforms and internal developer portals before describing the four pillars of an internal developer portal – and laying out key design considerations (Chapter 1).
- We'll then do a deep dive into those pillars with dedicated chapters on blueprints, software catalogs, developer self-service actions, scorecards, and automations (Chapters 2-6).
- We'll delve into the importance of reporting and executive views (Chapter 7) and extensibility (Chapter 8) before concluding with recommendations on adoption strategy (Chapter 9).

Throughout the guide, we'll share examples to help flesh out basic building blocks with specifics.

We believe that providing the best developer experience is a prerequisite to recruiting and retaining top engineering talent – and that delivering quality software depends on developer productivity. In this environment, internal developer portals are no longer a nice-to-have. They're a basic necessity.

Let's get going.

CHAPTER 1

Overview of internal developer portals

Developer productivity + engineering excellence

Internal developer portals enable developer self-sufficiency through the use of self-service actions with baked-in guardrails. At the same time, they reduce cognitive load on developers by providing an abstraction of your software and infrastructure – so that developers can easily make sense of anything in the stack and the resulting interdependencies.

Portals have the potential to:

- Streamline and automate developer work, by helping developers get what they need from the underlying developer platform – without worrying about the details
- Improve quality of engineering practices, fostering innovation and collaboration
- Help you organize your DevOps "inventory," so you can get visibility into the tools, applications, and services in your stack that goes far beyond .csv's with a list of microservices

But internal developer portals are also a relatively new product category in the DevOps space, with a deep potential impact on developer culture and the DevOps toolkit. Before diving into the specifics, we need to start with a high-level overview of what internal developer portals are – and the problems they solve for engineering and DevOps teams.

In this section, we'll distinguish between two commonly-confused categories, internal developer **platforms and portals** – and lay out the four pillars of an internal developer portal, along with some of the key considerations involved in implementing an internal developer portal.

Every dev team is unique, but every portal project should begin with a basic understanding of how internal developer portals work. This includes thinking about how a portal's technical capabilities can support the "product" experience that you're tailoring to your dev team's structure, culture, and practices.

Choosing the right portal for an engineering organization requires a product-like approach – understanding your organization's specific tech stack, needs, and goals is critical.
You also need to think of team dynamics, responsibilities and the engineering's org developer culture. This is also called platform-as-product.

Internal developer platform or portal – what's the difference?

An internal developer platform (IDP) is a central hub made of up the tools required to build, deploy, and manage everything DevOps – while driving a higher-order developer experience. For example, an IDP offers unified access to internal APIs, microservices, SDKs, and other resources needed for development. It also provides integrated tooling like CLI tools, build automation, CI/CD pipelines, and infrastructure provisioning.

This means that the developer does not need to work directly within a third-party tool (e.g. incident management, Appsec, observability, cloud resource, CI/CD, container management). Instead the developer accesses these tools through the platform. So instead of using Jenkins for self-service actions, the developer will use a portal to trigger an action in Jenkins – which, in this case, will be part of the platform.

The main purpose of an internal developer platform is to reduce developers' cognitive load by centralizing resources and making them more accessible. But often, the sheer scope of an IDP can prevent wider adoption. For example, while Kubernetes and cloud resource provisioning would typically fall within an IDP, most engineering orgs wouldn't want developers to access these directly – for reasons of security, quality, and cognitive overload.

Additionally, developers often don't have the knowledge to interact with these resources directly, even in their abstracted platform representation. They need guardrails and guidance to set them up for success.

Many organizations realize that the benefits of developer autonomy and innovation need to be paired with guardrails and "golden paths": the notion that there's a single supported approach for a given enterprise to do things like build a backend service or create a data pipeline.

Further Reading:
Read: It's a Trap - Jenkins as Self-Service UI
Watch: A guide: setting up a Git-based internal developer portal and extending the data model with Kubernetes

So the question becomes: how can IDPs be further abstracted and "productized" to promote developer self-service, while reinforcing golden paths and providing relevant guardrails?

An internal developer platform is necessary but not sufficient. It centralizes everything DevOps – but still requires an interface that provides the right abstractions for developers and promotes golden paths.

Internal developer portals are the answer. They act as a user-friendly interface to the platform – abstracting the complexity of the software development environment by providing a single user interface that's built for the questions and needs of different dev teams.

A portal's software catalog provides developers with a strong foundation for understanding the software development lifecycle. This familiarizes developers with the tech stack, helps them understand who the owners of different services are, and lets them access documentation for each service directly from the portal – and keep track of deployments and changes made to a given service.

This information can also be invaluable to speed up the onboarding process for new developers, as well as make diagnosing and debugging incidents easier since the dependency between microservices and products within the company is clearer.

Further Reading:
Read: Internal Developer Portals: self-service actions using infrastructure as code and GitOps

Blueprints: the building blocks of your portal

Every organization has a unique engineering tech stack, applications and resources. And an internal developer portal has to be adapted and configured to fit with both the underlying IDP – and the use cases for both portal and platform. (For example: what are the issues the organization would like to solve first: access to cloud resources, scaffolding microservices, dealing with incidents, or AppSec?)
It can't be the other way around. In other words, the portal can't dictate the organization's use cases and priorities.

The first step to creating an internal developer portal is to clarify what you'd want developers to do with it. What are the dev team's goals and needs? The portal needs to adjust to these needs – rather than forcing a preconceived notion of what portals should do.A good starting point is supporting developers through the software development lifecycle (SDLC), but others can be justified too.

Blueprints are a critical piece in connecting the design of your internal developer portal to your organization's goals, team structures, and user needs.

Blueprints are the foundational building blocks for defining your internal developer portal. A blueprint is a metadata schema definition – each blueprint defines the assets that the portal can manage and track: basic SDLC, microservices, environments, packages, clusters, databases, as well as third party data from incident

management, feature flagging, and other apps.

Blueprints are connected by **relations**, which define logical connections and dependencies between them (one to one, one to many, etc). Once blueprints are defined, the portal can effectively visualize and manage the software development lifecycle, and auto-populate the software catalog.

Blueprints are also where you define self-service actions and scorecards, which are pillars of the internal developer portal.

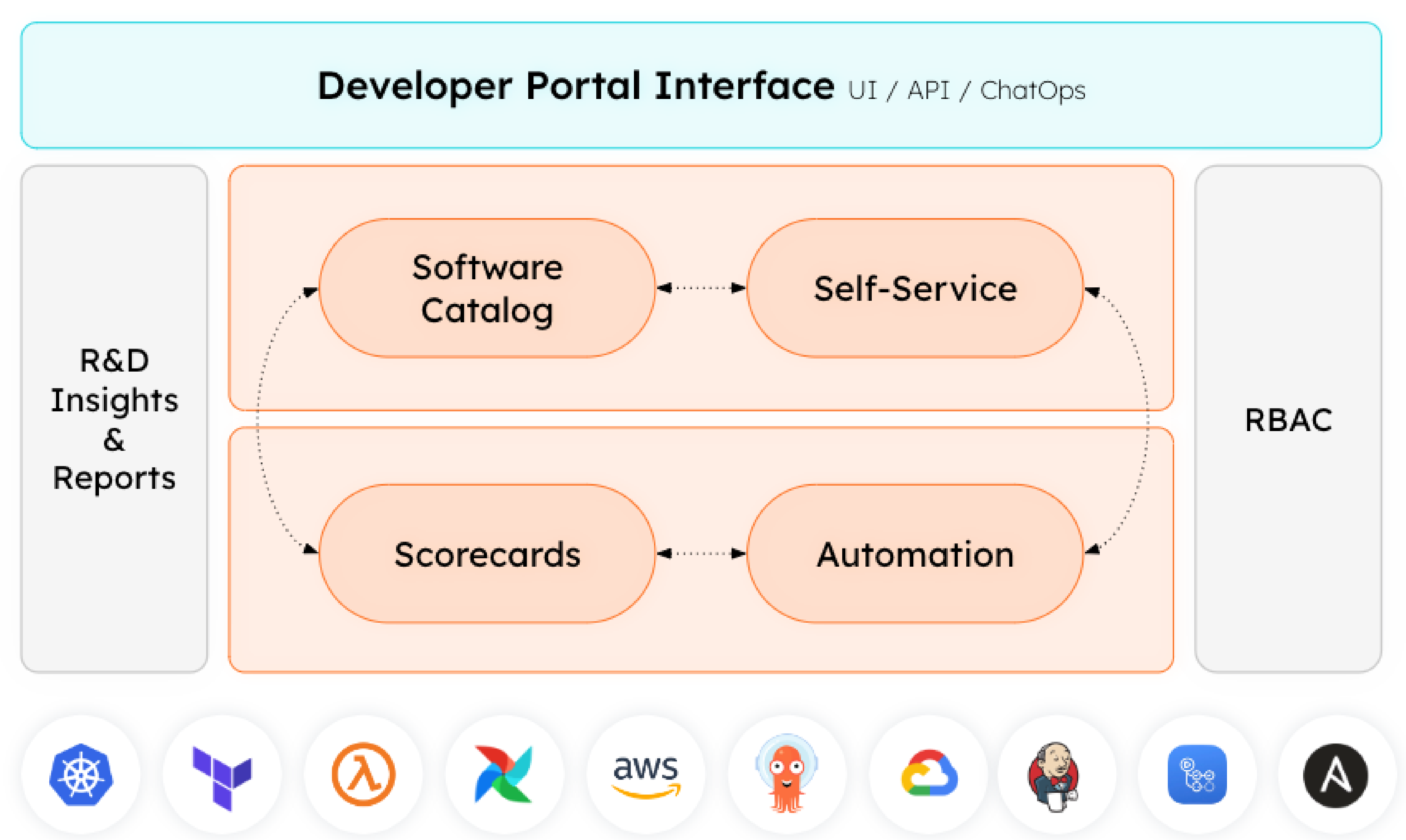
Blueprint examples

- K8s clusters
- Developer environments
- Package dependencies

Relations examples

- The packages that a microservice uses;
- The run history of a CI job;
- The Kubernetes clusters that exist in a cloud account

The four core pillars of an internal developer portal



1. Developer self-service actions

Self-service actions are perhaps the most heavily-used (and loved) feature among developer users of internal developer platforms. There's a reason for that: instead of having multiple open tabs for CI/CD, resources, Git etc, they can just use the internal developer portal with forms that simply and quickly get them what they need, even if it's temporary (an ephemeral environment with TTL) or requires manual approval. From the platform engineer's point of view, any self-service form is completely customizable to what abstractions developers need and what they don't. Think about it this way: when you order a ticket from an airline you don't want to choose the pilot and other flight staff. The same applies for developer self-service actions: the form should only include what developers care about (thank you, Kostis Kapelonis, for this idea).

In addition to streamlining workflows, actions often provide developers with new capabilities. For example, most organizations' DevOps would not allow a developer to provision a cloud resource on her own – requiring her instead to open a ticket to DevOps. But in a portal with self-service actions and proper guardrails, this becomes a possible, safe, and legitimate use case.

Forms are curated by the platform engineers to provide a product-like experience for developers.

Self-service action examples

- Self-service action examples
- Modify feature flag
- Spin up a developer environment for 5 days
- Modify cloud resource
- More here

2. Software catalog

The software catalog is a centralized metadata repository for software and its underlying resources, letting developers easily see and understand all entities – microservices, cloud resources, CI/CD data, Kubernetes clusters etc, developer environments, pipelines, deployments, and anything cloud tools, libraries, services, and components – in context. The catalog provides useful abstractions to developers, acting as a single pane of glass for everything software. The structure of the catalog is determined by the blueprints and relations defined in the data model (see Chapter 3 for more details).

The catalog is always kept up to date and reflects any changes, allowing developers to know what is going on and even serving as a source of truth for matters CI/CD and becoming an asset for pipelines that can query the catalog with questions such as TTL of an environment, service ownership, feature flags and more.

3. Scorecards

Scorecards are benchmarks for each element in the software catalog. This is where metrics for quality, production readiness, and developer productivity are defined and measured. Typically scorecards are created to set engineering standards and reflect to teams where they should be going. Scorecards can also be used to trigger alerts or to be checked by CI/CD pipelines (e.g. when a scorecard is degraded, something won't be deployed).

Further Reading:
Read: An Internal Developer Portal for Workflow Automation

Scorecards can also support initiatives – groupings of related scorecards, designed to drive strategic priorities (see Chapter 5 for more details) – and campaigns to improve production readiness or quality testing practices. Automations with scorecards might include tracking open Jira issues, opening Jira tasks, or blocking deployment of low production readiness services. Scorecards and initiatives are also perfect for DORA metrics.

Scorecard examples:

- Production readiness
- Service health
- DORA metrics
- Service operational readiness

4. Automation

If there's one thing you should be thinking about more in the context of internal developer portals, it's automations. Since portals are new, most people focus on the basics: catalog and self-service. But since the software catalog is a single source of truth for many tools, resources and software, it actually can support automations and alerts.

- Changes to scorecards, such as a degradation below a certain threshold
- Changes to software catalog entities

Automations can also be used to initiate actions, such as creating issues or triggering runbooks when a certain incident occurs. An automation might automatically terminate an ephemeral environment after its TTL (time to live) expires, or trigger an alert when scorecard assessments determine a service degradation.

Automation examples

- Incident escalation
- Security remediation
- Resource cleanup

Six portal considerations that (really) matter

Un-opinionated (vs. opinionated)

An internal developer portal can be either "opinionated" or "un-opinionated" in its design philosophy. **Opinionated** portals enforce a specific, predefined data model and provide a more guided experience, offering simplicity and consistency, but they are also more rigid and limiting by nature. **Un-opinionated** portals allow platform engineers to customize the data model to reflect the way the tech stack and use cases are in the specific engineering organization, offering more flexibility. They also let you build that data model as you grow, beginning with a minimum viable product (MVP) internal developer portal and growing it over time.

Read: How to Create an Internal Developer Portal MVP

Because every engineering organization has a unique tech stack, pain points, and degrees of visibility, opinionated portals – even if they're sound – can end up hemming the organization in. On the other hand, an un-opinionated portal, which is more extensible and open, will allow for growth and change over time, as well as the opportunity to add more data to the catalog as the portal grows. (Read more about data models for internal developer portals here.)

Extensible and open

An internal developer portal becomes more valuable the more data it has flowing into it – since it has a more complete software catalog, and supports a broader range of self-service actions and automations.

Extensible and open portals are not limited by specific data sources; they allow for extensions such as bringing in vulnerability data, incident data, and other sources (third-party or in-house). It may be helpful to think of the core software catalog and portal as an inner circle, and the many potential extensions (such as auxiliary sources of data) as the outer circle. Linking additional layers means that you can add more context and information to services within the software catalog – and drive more self-service actions.

Extensibility and openness also make the portal a complete, up-to-date, single pane of glass into your environment and SDLC metadata – making it ideal for observability and pipeline automation (e.g., CI/CD).

Loosely coupled

> When developer self-service actions are loosely coupled, it means they operate independently from the underlying infrastructure and tools, allowing for more adaptability when it comes to changes in the platform.

Why is a loosely coupled design for your portal widely considered a best practice in the industry?

- Most of the time, companies already have core tools and ways of working. They're looking for a portal that will organically connect and integrate with their existing tooling and workflows.
- Those that don't are looking to build best-practice platforms (e.g., they're using K8s native platforms for GitOps) – and don't need or want their portal to force them to make certain platform choices.

Loosely coupled portals also provide flexibility in case changes are made to the underlying platform – making them easier to maintain and scale.

API-first

An API-first portal will have a generic interface to model the software catalog, ingest data, invoke actions, query scorecards and more – providing an optimized experience for both humans and machines.

This single API for the entire catalog is used for:

- Building the catalog

Update the software catalog using a script, bringing any data into the portal, or importing an existing asset inventory from a .csv file or another portal. It can also be used to define the data model, permissions, views, settings, actions, and any portal configuration, as well as reporting the status of a running CI job (CI/CD catalog).

- Automations

Integrate the portal with your custom CI/CD and support programmatic automations and alerts (like Slack messages on software catalog changes).

An API-first portal will have an enhanced ability to provide automated decisions and actions based on the software catalog's real-time state. API-first is achieved by offering a well-defined and documented contract between the portal and external systems, allowing for workflow automation, triggering of DevOps flows, and seamless integration with CI/CD pipelines and other automation tools.

Portal-as-code

Portal-as-code describes an approach to configuring and customizing the portal itself through code. This can be done with tools like Terraform. It also promotes better visibility, as all changes are documented and traceable in the code repository. Portal-as-code is useful when you want to control the portal's stability and avoid unplanned change. If you're taking a portal-as-code approach you have a process in which the portal is defined, in one place, with all changes tracked and the ability to revert if needed.

Further Reading:
Watch: Developer portals: what developers want and what platform people should do about it

Customizable

A product-like experience ensures that developers will be able to answer the questions that matter to them – while interacting with the right abstractions, accessing the right self-service actions (with appropriate guardrails), and engaging with scorecards and reporting relevant to their roles.

Your portal should support platform engineering in building this product experience for your developer users. This requires customizability of views, visualizations, and UI – along with fine-grained role-based access controls – for different "personas" (teams and roles). It also requires the ability to deliver the right resources to specific users in context: for example, documentation to a developer drilling down into an individual service in the catalog.

Further Reading:
Read: Using an internal developer portal to make developers 10X and break conway's law

CHAPTER 2

Blueprints

Introduction

The goal of an internal developer portal is to provide a product-like experience to developers – so they can easily self-service with less cognitive load.

Blueprints are key in delivering this product-like developer experience. They enable platform engineers and DevOps folks to define the right abstractions for developers and to evolve over time as needs (and the technical environment) change.

What are blueprints and what can you do with them?

Blueprints are the foundational building blocks of the internal developer portal. They hold the schema of the entities you wish to represent in the software catalog. (For example, you might have a blueprint for packages, K8s clusters, microservices, and VMs – each one defining the metadata associated with these entities.)

Blueprints are completely customizable, and they support any number of properties the user chooses, all of which can be modified as you go.

Just as important as the individual blueprints is the way that they relate to one another ("relations"). Relations are the connections between blueprints, capturing the dependencies between them. Relationships can be single (e.g., a package version to a package) or multiple (e.g.,the packages used by a service), and they enable the representation of the software catalog as a dynamic graph database.

The backbone of the internal developer portal

Blueprints are critical for building the internal developer portal:

1. Blueprints describe the data you want inside the software catalog.

The process begins by defining detailed blueprints that describe the characteristics of different assets like microservices, environments, packages, clusters, and databases. The properties should be driven by questions that we'd like to answer (e.g., "who owns an asset?"). Then, real-world entities from your infrastructure are

associated with these blueprints, forming the software catalog entities through auto-discovery. You can also use templates inside the product or templates that come with the relevant Port integration (in some cases).

2. Blueprints define self-service actions for developers through the portal.

These actions cover a wide range of tasks, including provisioning, termination, and day 2 operations on any asset exposed in the software catalog. With blueprints, developers can easily interact with the platform, reducing the cognitive load on DevOps teams.

3. Blueprints define metrics and quality standards for catalog entities.

These metrics and standards are formalized via Scorecards. Scorecards can also indicate the health, quality, and readiness of specific elements within the platform. Having a "source of truth" for these metrics enables automated decision-making based on the status of entities, helping ensure compliance and standards. Scorecards also form the basis for driving engineering quality initiatives.

Blueprints support a product-like developer experience

So what makes blueprints powerful? In a nutshell, they help platform engineering and DevOps build a data model to answer the right questions for your engineering org – providing flexibility and context that support self-service.

Letting you define your own data model - Your data model should be driven by the specific questions you want to answer about different services and the use cases you want to enable for developers. What's the ideal data model for infrastructure as code (IaC), Kubernetes, GitOps, or alert management? The right answer is unique to your organization – and depends on the specific information that's most useful to your dev teams. This is where blueprints shine. Instead of imposing a rigid and opinionated data model, blueprints let platform engineers or DevOps define the ideal data model for your engineering org.

Being adjustable over time - Things change: your infrastructure, your applications, your team structure, the questions your engineering organization cares about, the use cases you want to cover in the portal. And blueprints are designed to change with you. The ability to continuously update blueprints – by extending or refining properties and metadata – means that you can easily support new services. And tools like data migration and versioning mean that you'll always be able to keep your software catalog aligned with your latest data model.

Providing context – Your code is not your app. How your app behaves depends on the environment it's running in and the ecosystem of tools and dependencies surrounding the end user. And a static software catalog that only includes metadata – and not runtime data – rarely provides developers with the context they need. Blueprints provide an easy way to enrich the data model with contextual "running service" data: references to the service, environment, and deployment, as well as real-time information such as status, uptime, and more. This provides critical context to help answer questions like, "What is the current running version of service x in environment y?"

Further Reading:
Read: Why "running service" should be part of the data model in your internal developer portal

CHAPTER 3

Software catalog

The software catalog is a central metadata store for everything application-related in your engineering, from CI/CD metadata through cloud resources, Kubernetes, services and more. As described in Chapter 2, its building blocks are blueprints and relations.

More than a static inventory

Far from being just a "flat" repository for static metadata (e.g., ownership, logs), the software catalog is continuously updated and enriched with context based on your specific data model. Software catalogs deliver value to the dev organization in several key ways:

- Help developers answer critical questions (e.g., "What is the version of the checkout service in staging vs. production?")
- Drive ownership and accountability. Port syncs with your identity provider to reflect software ownership from the team structure point of view.
- Offer a "single pane of glass" into your services and applications

Let's dive in.

Data models are driven by the needs of your "personas"

Imagine that you're the platform engineering team designing the software catalog for your organization. How would you go about it?

You'd probably start by thinking about the different personas that will be using the software catalog. Developers on the team would probably want to abstract away, say, Kubernetes complexity – for them, the ideal view would probably include production services, vulnerabilities, and a minimal amount of K8s. In contrast, DevOps users would want to understand the infrastructure more deeply, including cost and resource tradeoffs.

The point is that there's no "one size fits all" answer to what data goes inside the catalog and the views that different team members will use. It depends on the user personas and their needs.

This is where different data models come in.

> An internal developer portal isn't a static list of microservices; it's a dynamic graph of all your entities in context. How you choose to visualize that environment and those relationships should be driven by your organization's needs, use cases, and user personas. A good data model provides a graph of how everything is related to everything.

Base data models and extensions

Several base data models serve as foundational frameworks for structuring software catalogs. These models are designed to answer critical questions about common infrastructure and applications.

Here are some of the most common base models:

- **Classic (aka SDLC) Model:** This model encompasses three primary components: Service, Environment (representing different environments), and Running Service (depicting the live version of a service in a specific environment). Its goal is to make it easy to understand the interdependencies in the infrastructure and how the SDLC tech stack comes together. This helps answer questions such as how are cloud resources connected to a test environment on which a certain service is running, all within a larger domain and system.
- **C4 Model:** Port uses an adaption of the Backstage C4 Model, which provides a hierarchical approach to visualize software architectures built around "Context, Containers, Components, and Code." Context reveals the software catalog's broader position in the ecosystem, Containers identify major components, Components delve into internal structures, and Code showcases low-level details.

> **Further Reading:**
> Read: Using Backstage's C4 Model Adaptation to Visualize Software - Creating a Software Catalog in Port
> Read: Why "running service" should be part of the data model in your internal developer portal
> Read: A Quick Migration From Backstage to Port and What You Need to Know

And here are some extensions to base data models:

Primarily for DevOps

- **Kubernetes (K8s):** This expands the data model to represent all K8s data around infrastructure and deployment, utilizing Kubernetes objects like Pods, Deployments, Services, ConfigMaps, etc. to define system state and management.
- **API Catalog:** Adding API data where each API endpoint is part of the catalog, alongside its authentication, formats, usage guidelines, versioning, deprecation status, and documentation. This can support API route tracking and health monitoring.
- **Cloud Resources:** Expanding the model to encompass the entire technology stack involves representing both software components and the underlying cloud resources that support them. This approach provides a unified view of the software's technical context within the broader cloud environment.
- **CI/CD:** Including information about CI/CD pipelines and related tools augments the data model's scope. This offers a complete representation of end-to-end development and deployment workflows, enabling efficient management of software release processes.

For all developers

- **Packages & Libraries:** Extending the model to include packages and libraries facilitates improved software dependency management. This is crucial for maintaining software integrity and security by tracking and overseeing dependencies effectively.
- **Vulnerabilities:** Integrating security vulnerability information into the data model enables the identification and management of vulnerabilities present in software components or packages, bolstering security measures and risk mitigation.
- **Incident Management:** Integrating Incident Management information, such as data from tools like PagerDuty or OpsGenie, extends the data model to handle incidents, outages, and response processes. This inclusion provides a comprehensive view of how the software ecosystem responds to and recovers from unexpected events, contributing to overall reliability and rapid issue resolution. It also provides the ability to track historic incidents and create an internal knowledge base of incidents and their resolution.
- **Alerts:** Incorporating alerts into the data model provides timely insights into system performance, security, and health. This proactive feature empowers teams to take swift actions, ensuring a stable and reliable software ecosystem.
- **Tests:** Expanding the model to encompass tests, their status, and associated metadata creates a centralized view of testing efforts. This aids in monitoring testing progress, identifying bottlenecks, and promoting efficient quality assurance.
- **Feature Flags:** Bringing in data from external feature flag management systems allows for controlled and visible management of application features. This approach fosters an iterative and data-driven approach to feature deployment, enhancing flexibility and adaptability.

- **Misconfigurations:** Addressing misconfigurations by integrating them into the model helps prevent security vulnerabilities, performance issues, and operational inefficiencies. This ensures the software's operational health and stability.
- **FinOps:** Adding FinOps cloud cost data into your portal will instantly map it to developers, teams, microservices, systems, and domains. This simplifies the data, letting you easily break down costs by service, team, customer, or environment – helping FinOps, DevOps, and platform engineering teams efficiently manage costs and optimize spending without spending hours on basic reporting.

The software catalog should be stateful and updated in real time

Two important final considerations in the design of your software catalog.

First, your data model should support a stateful representation of the environment. For example: the running service in the classic model (see above) reflects the real world, where “live” services are deployed to several environments, such as development, staging or production environments. This is critical for providing context. (Again: your code isn’t your app.)

Further Reading:
Read: Why ”running service” should be part of the data model in your internal developer portal

Second: the software catalog should sync in real time with data sources to provide a live graph of your services and applications. For example, integrating with CI/CD tools ensures that your software catalog will be kept up to date by making updates directly from your CI/CD.

CHAPTER 4

Developer Self-service Actions

Self-service actions are often the first thing that developers associate with internal developer portals. There's a good reason for that: self-service actions make developers more productive (and happier), by promoting autonomy along with guardrails and golden paths.

Self-service actions promote developer productivity and engineering standards

Enabling self-service developer actions achieves several goals:

- Provides developers with autonomy to perform routine tasks themselves (e.g., spinning up a dev environment) without filing tickets or waiting for other teams
- Ensures compliance with organizational security policies and best practices
- Enables actions in context (e.g., a developer can see all runtime details on related microservices before scaffolding a new microservice)
- Provides a "golden path," reinforced through the UI and role-based access control
- Enforces guardrails around organizational policies (e.g., TTL for ephemeral environments, approval processes)
- Supports engineering quality standards by enforcing them in the scaffolding phase

Self-service actions drive efficiency and developer productivity by allowing developers to execute tasks independently and asynchronously, without having to rely on DevOps teams. This is achieved through a curated UI in the developer portal – with an abstraction of underlying workflows configured by the DevOps or platform engineering team, guided by organizational best practices.

Two keys to powerful self-service actions

Self-service actions should be **loosely coupled** (deliberately segregated) from the underlying infrastructure of the platform. This separation ensures that regardless of how the infrastructure evolves (e.g., if the underlying cloud resources, CI or CD systems are replaced), developers will still have the same experience built with a product mindset and designed to reinforce golden paths. For a variety of technical and security reasons, it's considered best practice for the portal to remain loosely coupled from underlying GitOps platforms.

Another important aspect of self-service actions is that they are **reflected in the catalog**, which is always kept up-to-date. In this sense, the catalog serves as a dynamic reference point—a repository for deployed services, environments, and other critical entities. Every self-service action affects catalog entities: creating, modifying, or deleting them.

By aligning self-service actions with the catalog's status, developers gain informed, real-time insights into the development environment.

Examples of self-service actions

- Microservices SDLC (examples: scaffold a new microservice, add secret, feature flag, lock deployments, add package version)
- Developer environments (examples: spin up developer environment for 5 days, ETL mock data to environment, invite developer to environment, extend TTL by 3 days)
- Cloud resources (examples: provision a cloud resource, modify cloud resource, get permissions to access cloud resource)
- SRE (examples: update Pod Count, update auto-scaling group, execute Incident Response Runbook automation)
- Data & ML (examples: train model, spin up remote Jupyter notebook, pre-process dataset, run Airflow DAG)

> Self-service actions may also be asynchronous. Two examples are long-running actions and ones that require manual approval. It's important that the portal be equipped with the right tooling to handle these – see "manual approval and run logs" below.

Manual approval and run logs

While self-service actions increase independence and efficiency of developers, certain critical processes may require oversight and approval from DevOps teams. In these cases, enabling manual approvals is crucial. Manual approvals require that developers ask for permission before taking certain actions, which helps ensure compliance over critical processes.

For all actions developers take (whether or not they require manual approvals), run logs allow for visibility and enable them to detect and fix any issues easily – and understand the status of a long-running developer self-service action which is performed asynchronously. Run logs also make it possible to abstract the complexity that usually lies within the complete CI/CD pipeline log, providing a curated log of actions that a developer might actually care about.

CHAPTER 5

Scorecards

Two fundamental goals of an internal developer portal are to foster a culture of engineering excellence and to encourage end-to-end developer ownership: a culture of "you build it, you own it."

Scorecards, a core element of a portal, provide clear standards of quality and maturity to drive alignment across engineering teams – and the toolkit to drive a variety of initiatives from continuous improvement through production readiness and more.

In this section, we'll define scorecards, explain why they matter, provide examples of the most common types of scorecards, articulate how scorecards are used in practice, and provide a step-by-step "quickstart" guide to new portal users for implementing scorecards.

What is a scorecard?

A scorecard is a way to measure and track the health and progress of each service and application within your software catalog. Scorecards establish metrics to grade production readiness, code quality, migration quality, operational performance, and more.

For example, a service maturity scorecard could track code coverage, ownership, and documentation for a critical customer-facing service. This provides visibility into whether practices are improving month-over-month.

Scorecards drive alignment, prioritization, and quality

There are three main reasons organizations use scorecards:

- **Organizational alignment:** Scorecards allow you to set clear, established standards and baselines for code quality, documentation, operational performance, and other metrics. This ensures consistency across teams and services. For example, you may set a standard for minimum code coverage across all services.
- **Alerting and prioritization:** By monitoring service scores against thresholds, you can be alerted when a service drops below acceptable baseline levels. This allows you to proactively identify services that need attention before problems occur. Scorecards enable you to prioritize which services need help first based on their scores. For example, an alert could notify if a service's uptime drops below 99%. Such alerting functions can also be tied to automations.

- **Driving quality improvements.** Initiatives (see "Initiatives roll it up," below) are groups of scorecards that all fit within a strategic focus area. Engineering leaders rely on initiatives to set organizational priorities – and invest team energy and focus in concerted improvements in a given area.

Capturing health and maturity throughout the SDLC

There are many types of scorecards that provide visibility into different aspects of your services and systems:

- **Operational readiness:** These scorecards check if services are production-ready by looking at factors like ownership, documentation, runbooks, and infrastructure setup. At Port, we use readiness scorecards to ensure services meet standards before launch. For example, requiring a runbook, alert setup, and monitored uptime.
- **Service maturity:** Service maturity scorecards grade factors like code coverage, versioning, README quality, and other code health metrics. For example, Port tracks service maturity with a scorecard that checks for ownership, test coverage, documentation, and more.
- **Operational maturity:** For services already in production, these scorecards measure operational health. Example metrics include test coverage, on-call activity, mean time to recovery, and deployment frequency.
- **DORA metrics:** Scorecards can track DORA metrics like deployment frequency, lead time for changes, and change fail rates. This provides insight into engineering efficiency and quality.
- **Migrations:** Migration scorecards provide visibility into progress, blocking issues, and other key metrics as you migrate services to new platforms. For example, tracking completion percentage and stalled tasks during a move to Kubernetes.
- **Health:** Health scorecards give high-level visibility into the overall status of services, often using a simple red/yellow/green scoring system. For example, services with no errors or incidents in the past week are green.
- **Code quality:** These scorecards analyze code on dimensions like test coverage, readability, extensibility, and reproducibility. For instance, tracking pylint scores over time.
- **Cloud cost:** Scorecards focused on cloud cost management (FinOps) track metrics like spend by team and time-to-live for cloud resources. For example, identifying unused S3 buckets or overprovisioned resources.
- **Security:** Security scorecards monitor for misconfigurations, unprotected resources, provisioning issues, and other risks. For instance, flagging unencrypted RDS instances or insecure bucket policies.

Initiatives roll it up

An "initiative" in the internal developer portal context refers to a strategic focus area that aligns developer

workflows and activities towards business goals and outcomes.

> Initiatives are concerted, organized efforts to improve engineering quality on a specific dimension. They contain collections of related scorecards representing priority areas like improving reliability, security, velocity, visibility etc. – driving developers to adopt practices and tooling that support those goals. For example, an "Improve Reliability" initiative may contain scorecards for crash-free releases, mean time to recover, end-to-end testing coverage etc.

Initiatives differ from individual scorecards in their goals and scope:

- Scorecards allow teams to benchmark themselves and get guidance on incremental steps to improve their scores. Scorecards may track automation, testing, monitoring, documentation, and other best practices.
- Initiatives align developer workflows to business KPIs. An "Accelerate Release Velocity" initiative could track deployment frequency, lead time, change failure rate, and other metrics.
- Initiatives connect dispersed efforts around a common cause. Developers have a purposeful path rather than ad-hoc tasks. Platform teams streamline tooling and workflows to support the initiative.
- By rallying the organization around targeted initiatives, internal developer portals drive lasting culture change, not just one-off optimizations. Initiatives create clarity, urgency and focus.

Ultimately, initiatives represent strategic programs to achieve business outcomes by guiding developers, project teams, and platform organizations towards practices and behaviors that matter most.

When the rubber hits the road

Organizations use scorecards in two ways:

As part of a periodical review of initiatives

Managers periodically review scorecards to check progress towards goals and identify areas that need attention (see below). For example, a CTO might review code quality scorecards across teams quarterly.

As alerts and automation triggers

- Alerts notify users via email, SMS, Slack, or other channels when thresholds are crossed. For example, a low operational readiness score could trigger an email to the service owner.
- Alerts can kick off automated workflows to resolve issues surfaced by scorecards. At Port, we've built workflows to auto-assign tasks to managers if scores for their services drop too low.

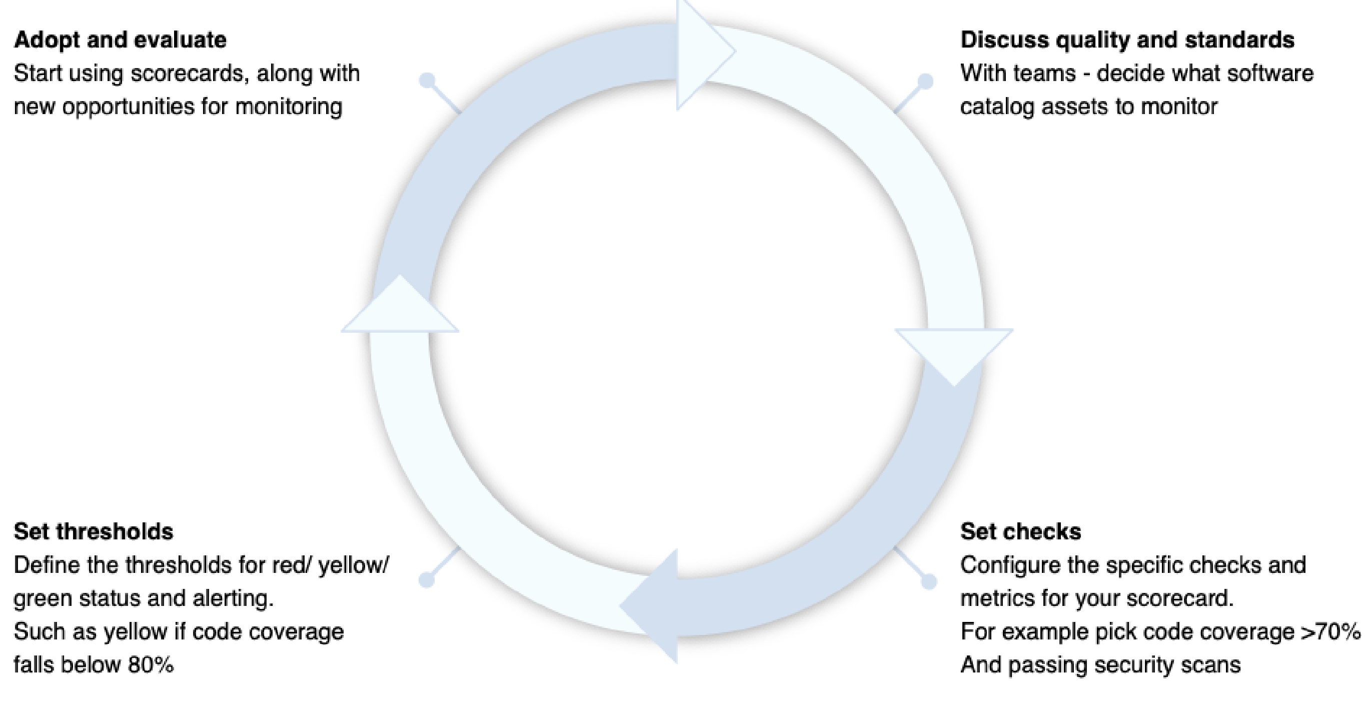
- CI/CDs can check the relevant scorecard (e.g., code coverage, security policy adherence) for the service deployment – and stop the deployment if the scorecard tier is too low.

Ultimately, initiatives represent strategic programs to achieve business outcomes by guiding developers, project teams, and platform organizations towards practices and behaviors that matter most.

Further Reading:
Read: An Internal Developer Portal for Workflow Automation

Create your own scorecard

Scorecards are created through a four-step process:



In their rush to implement scorecards, too many teams rush through the critical phase of discussing quality and standards with teams. What services or applications does the team want more visibility and measurement around? Understanding the needs and pain points of developers is a critical part of the platform engineering “product mindset.”

Further Reading:
Read: Setting Kubernetes standards with platform engineering

The final word

By providing standards, visibility, and automatic alerts, scorecards help engineering teams achieve consistency, improve practices, and proactively identify issues. They are an essential capability of portals like Port.

CHAPTER 6

Automations

Introduction

Internal developer portals have the potential to massively increase developer productivity.

That's because the portal's software catalog acts as a "single pane of glass" into all services and applications, enabling teams to automate DevOps workflows. A robust portal can – and should – be able to trigger workflows around resource management, remediation, provisioning, and more.

Automations trigger alerts or workflows

Automation refers to the ability to programmatically trigger alerts or DevOps workflows based on well-defined criteria. For example, an automation might automatically terminate an ephemeral environment after its TTL (time to live) expires, or notify teams via Slack when changes are made to the software catalog.

Automations are enabled through the portal's API, which enables machines to access real-time data about the software catalog. Automation use cases include the following:

- **Incident escalation:** Automated alerts that escalate unresolved incidents to on-call teams
- **Security remediation:** Auto-remediate security issues like excessive permissions and vulnerable configurations
- **Resource cleanup:** Terminate inactive cloud resources exceeding time-to-live policies to manage costs
- **Service monitoring:** Get notified of service degradations and anomalies, or disallow deployments when service scorecards degrade
- **Catalog updates:** Email subscribers when new services are added to the catalog or other changes in the software catalog

Anatomy of an automation

Every automation is defined by a "trigger" (what conditions initiate a workflow) and an "action" (the workflow that is automated as a result).

Examples of triggers	Examples of actions
↘ When an entity matches conditions	↘ Send an email
↘ When a form is submitted	↘ Send a Slack message
↘ When an entity is created	↘ Send a weekly digest
↘ When an entity is updated	↘ Run a job
↘ At a scheduled time	↘ Call an API
↘ When a scorecard degrades	↘ Disallow a deployment

By offering a "single pane of glass" into all services and applications, the internal developer portal's software catalog enables powerful automations: DevOps workflows that reduce manual work (like access provisioning and performance monitoring), streamline communication (around job failures, major releases, degraded services, and more), and enforce consistent practices to promote quality and standardization (auto-terminate resources, run CD flows, and more).

CHAPTER 7

Reporting and executive views

Reporting is almost always treated as an afterthought in conversations about internal developer portals.

This is a missed opportunity – but not a surprise. As we've discussed, one of the core goals of an internal developer portal is to improve developer productivity and autonomy. And historically, reporting (for example, views showing the adoption or maturity of different services, or DORA metrics by engineering team) are seen as catering to execs, rather than providing value to hands-on-keyboard devs. Thing is, reports can work both for execs and developers, especially if they are implemented into a personalised dashboard per developer or team, highlighting pertinent information in the portal or the catalog.

But you're leaving value on the table if you overlook a portal's reporting capabilities. Done right, reporting can:

- **Provide visibility** into platform adoption, usage, and impact to identify areas for improvement. For example, low usage of a new API could indicate issues with documentation that need to be addressed. Reports also surface troubled services with reliability problems or ineffective tools with low ROI.
- **Highlight trends** over time to inform strategic decisions. Analytics may reveal spikes in traffic to a core service, indicating a need to scale resources. Or broad adoption of a new programming language may inform training and hiring priorities.
- **Track progress** on business goals like productivity, time to market, operational efficiency. Reports can tie platform metrics directly to organizational KPIs to showcase impact.

> On the organizational level, strong reporting can improve the strategic impact of the engineering team by ensuring focus on the right priorities. And on the individual level, reporting can serve as a self-serve tool for developers to enhance productivity and identify opportunities for better engineering quality.

Reporting needs differ by persona

Just like a portal needs to provide dev teams with the right abstractions and self-service actions for their needs, reporting needs to be aligned to the use cases of different user "personas." For example, the right reporting views will ensure that:

- **Developers** can self-serve reports and dashboards to optimize their own productivity. For instance, seeing traffic sources and error rates for their APIs helps improve integrations. Monitoring runtime performance aids optimization.

- **Product managers** can track adoption of new tools and usage of existing features. This guides enhancement priorities and roadmaps. Low usage may indicate poor marketing rather than a bad product.
- **Executives** can have dashboards that guide prioritization of technical resources based on usage, health, and business impact reports. For example, if a core payments service shows spiking errors, additional staff could be allocated to improve resilience.
- **CIOs** can benchmark teams across usage of shared services, reuse of components, and productivity metrics. This highlights opportunities to promote best practices and engineering quality among groups.

Customization is critical

Customization of reporting is the key to ensuring that different personas in the organization are getting the information that's most valuable to them – when they need it.

Customizable reporting includes basic flexibility around elements of data reports, including:

- Metric definitions
- Visualizations
- Reporting frequency
- Supporting documentation

A platform engineering team should think of internal developer portal reporting through the product lens – with specific "customers" in the organization. (More on the product mindset and organizational adoption strategy in Chapter 9.)

Usage, health, productivity, and business impact

Once you've identified your primary personas for reporting and thought through the right customizations – it's time to think about what are the most common reporting "use cases"? In other words: what are the specific operational areas that the reporting should address and actions that might be taken as a result?

Usage reports show service adoption (e.g., how frequently APIs are called). **Health reports** measure the reliability and performance of services, **productivity reports** surface DORA metrics, and **business impact reports** track operational efficiency and identify the technology drivers of company-level financial performance. Other reports can track engineering quality initiatives, such as production readiness of microservices.

The table below summarizes each report type along with representative metrics – and why they matter.

Report type	Sample metrics	Why it matters
Usage	↘ API calls per day ↘ Peak concurrent users ↘ Tool adoption rates	↘ Prioritize documentation/support for low-traction APIs ↘ Rightsize infrastructure for highly utilized services ↘ Identify unused tools to deprecate
Health	↘ Uptime/availability ↘ Latency histograms ↘ Error rates	↘ Improve reliability for business-critical flows ↘ Optimize performance bottlenecks ↘ Prevent disruptions and outages
Productivity & velocity	↘ DORA metrics (deployment frequency, lead time for changes, and change fail rates)	↘ Enhance release planning and predictability ↘ Address code quality issues ↘ Optimize CI/CD pipelines ↘ Accelerate incident response and recovery
Business impact	↘ Revenue per API call ↘ Release frequency ↘ Mean time to resolution	↘ Identify highest value-add services ↘ Shorten iterations between releases ↘ Improve operational efficiency ↘ Enhance FinOps cost efficiency

Scorecards, discussed previously, are a critical input to reporting. A single report can draw from multiple scorecards to highlight different facets of, for example, service health.

Particularly valuable in the context of reporting is the concept of **initiatives**. As laid out in Chapter 5, initiatives represent collections of scorecards that point to common strategic priorities or investments (e.g., improving reliability). Reporting and dashboards are a crucial mechanism to provide updates on progress against organizational initiatives.

The final word

As a “single pane of glass” into a company’s applications and services, internal developer portals are uniquely situated to deliver reporting on the usage, health, and business impact of a company’s technology footprint.

Reporting built and automated through a company’s internal developer portal has the potential to unlock efficiency, productivity, and strategic insight at multiple levels of the organization. When done right, it can guide critical decisions at the C-suite level – and give developers self-service insights into key prioritization and resource allocation tradeoffs on a day-to-day basis.

CHAPTER 8

Extensibility

Introduction

Given its role as a "single pane of glass," it's no surprise that an internal developer portal is only as valuable as the data it contains. A portal on its own offers developer self-service, insight into the SDLC, documentation, and more. But its value multiplies exponentially when integrated with cloud resources, kubernetes, CI/CD data, and data coming from FinOps, AppSec, incident management, and similar tools.

By centralizing metadata from these tools into a unified catalog, developers and technical leaders can gain a holistic picture of the development lifecycle and status of services and applications. Extensions also broaden the range of developer self-service actions. For instance, executing a runbook when an incident occurs.

But how does one actually implement robust integrations to unite all these disparate data sources?

More data, more value

Out-of-the-box, a portal provides core services for documentation and self-service. But the range of possible integrations is endless:

- CI/CD platforms to surface build statuses, deployments, and releases
- Issue/project trackers to link tickets and epics to codebases and services
- Monitoring systems to associate alerts, traces, and telemetry data
- Cloud infrastructure to visualize storage and functions (e.g., S3, lambdas)
- Internal custom and legacy systems that provide additional context

This requires a flexible integration framework that can extract metadata from these systems and accurately sync it to the portal's searchable catalog in real-time. More importantly, the metadata should be stored in one place, so that visualizations, reporting and automation all work as they should.

For example, when you integrate alerts into your software catalog, you get a single pane of glass for everything alerts-related, in-context within the relevant software catalog entities, complete with all the information you need (like the service or resource owner). Beyond the convenience of not needing to check multiple alert tools, the fact that alerts are in context significantly reduces the cognitive load on developers. Each alert is linked to its origin, such as a production issue tied to a specific service, and it can even be associated with day-2 operations that help resolve the underlying problem.

Real-time, flexible, and secure-by-design matter

Every internal developer portal must define its own approach to extensibility. But broadly, everyone is trying to solve the following challenges in a reliable, performant, and secure manner:

- **Real-time continuous sync** - Data must be continuously streamed from sources to the software catalog to provide an accurate up-to-date system of record, rather than just periodic snapshots.
- **Bi-directional sync** - Beyond ingesting data, the portal should be able to propagate changes back to source systems after actions are taken, completing the loop. For example, the software catalog should get updated when I make changes in the source environment (e.g., if a pod in K8s is deleted, I need to make sure that the replica count is updated in my IDP). Equally, the software catalog should also get updated when I run a developer self-service action initiated from the IDP itself (e.g., provisioning an ephemeral environment).
- **Reconciliation** - The integration framework should handle reconciliation by treating the source system as the "ground truth" and seamlessly syncing additions, updates, and deletions. For example, if I want to display all the services from PagerDuty as part of the catalog, I also need an automatic process that removes a service from the portal's catalog if that service no longer exists in PagerDuty (thus keeping the catalog synced with the data in PagerDuty, or any other integration).
- **Secure by design** - A robust integration framework should eliminate the need to share sensitive credentials or secrets, as well as avoid the requirement to whitelist IPs. Data filtering and security measures should be implemented on the customer's side, ensuring data privacy and protection.
- **Flexible protocols** - The framework should support common protocols like webhooks, APIs, and message queues to integrate with any source system.

Why go with an open-source extensibility framework

While core integrations can be provided out-of-the-box, open sourcing an internal developer portal's integration framework unlocks community extensibility.
Open source principles align seamlessly with the concept of portal extensibility. Open source fosters collaboration, encourages contributions, and empowers developers to customize and extend the platform according to their unique requirements. An open source integration framework offers several advantages:

- **Community contributions:** An open source framework invites developers from diverse backgrounds to contribute integrations that cater to a wide array of use cases.
- **Flexibility:** Open source allows for the creation of custom integrations tailored to specific needs, giving organizations the freedom to enhance their portal's functionality.
- **Rapid innovation:** By sharing integrations as open source projects, the development community can collectively drive innovation and expand the capabilities of the portal.

Finally, transparent open source code enables greater trust and customization to meet each company's security policies, rather than opaque closed solutions.

The final word

Extensibility is a core driver of the value of an internal developer portal. And an open-source approach to integrations – paired with a framework that prioritizes real-time bi-directional connectivity, reconciliation, and security – ensures that a portal can integrate with critical data sources in a flexible and scalable way.

CHAPTER 9

Adoption strategy

Introduction

Over the course of this practical guide, we've covered everything related to internal developer portals, from capabilities and requirements to usage and reporting. But even the most advanced portal will fail without a sound adoption strategy.

This final chapter explores common adoption risks and proven strategies to mitigate them. We'll look at how to tailor your rollout based on team structures, gather feedback, and gradually extend usage. With the right approach, you can shift from simply launching a platform to actively accelerating development through it.

Common pitfalls in portal adoption

As builders ourselves, we developers have passionate opinions about the software tools we use. Perhaps it's no surprise then that developers (the end users of an internal developer portal) can make or break a portal implementation. Executive sponsorship is necessary – but far from sufficient – to ensure adoption.

When embarking on an internal developer portal initiative, there are a few common pitfalls that can derail success if not mitigated upfront:

- Developers reject the portal if it **doesn't integrate** into their workflows. For example, if the running services view in the portal does not reflect the real state, developers will only get confused by the information displayed in the portal
- Starting with **too many features** creates complexity and onboarding challenges. Faced with a cluttered portal, developers get overwhelmed and disengage. It's best to choose the MVP features that would make developer lives easier.
- **Lack of executive sponsorship** leads to low prioritization and engagement. If leaders don't reinforce the portal's value, developers doubt its usefulness

Adopting the product mindset

To drive engagement with your portal and avoid the common adoption risks, it's critical to adopt a product mindset. This means understanding the organizational structure and constraints of your users – and building an iterative, stepwise approach to rollout (as opposed to a "big bang" launch).

Step 1: Map and understand your team structure

Team topologies – outlined in the 2019 book Team Topologies: Organizing Business and Technology Teams for Fast Flow by Matthew Skelton and Manuel Pais – provide a model for organizing groups to optimize software delivery and collaboration.

The authors delineate four common team types:

- **Stream-aligned teams:** Focus on specific business capabilities and align to end-to-end value streams. (For example, a team responsible for the checkout workflow on an e-commerce site. They own the user experience from adding items to cart through payment and order confirmation.)
- **Enabling teams:** Provide capabilities like infrastructure to assist stream-aligned teams. (For example, a site reliability engineering team that provides coaching and tools to help product teams understand and reduce error budgets for their services.)
- **Complicated subsystem teams:** Specialize in complex technical domains like data or AI. (For example, the search infrastructure team maintains the highly complex algorithms, machine learning models, indexing pipelines, and query engines that power search across the company's apps and sites.)
- **Platform teams:** Offer shared internal tooling/services to accelerate stream-aligned teams. (For example, an internal platform-as-a-service team offers self-service APIs, CLI tools, and dashboards to allow developers to deploy containerized microservices to the cloud without infrastructure knowledge.)

Analyzing how these topologies interact in your organization shows workflow interdependencies. This enables tailoring adoption strategies accordingly. For example, onboarding platform teams first creates a foundation for stream-aligned teams.

Step 2: Tailor your rollout strategy to the goals/structures of your team

Adopting a product mindset serving your internal developer user base involves:

- Conducting developer interviews and surveys to grasp needs and pain points. This provides insights for prioritization.
- Selecting an initial focused use case, like high-value APIs for a critical frontend app. Avoid overwhelming everyone at once.
- Monitoring usage data to identify adoption obstacles early. For example, low utilization of a new tool may indicate UX issues rather than inherent low value.
- Gathering feedback through surveys and user testing. Rapidly iterate based on developer input.
- Leveraging successful initial teams as champions to encourage peer adoption through their credibility.
- Gradually onboarding more developers, APIs, and tools based on demand rather than a rigid timeline. Let organic growth guide expansion.

The final word

An effective internal developer portal requires more than just technical buildout – it demands cultivating users. By understanding team structures, goals, and pain points, you can craft tailored adoption strategies. This balances breadth of capabilities with simplicity. Through a product mindset, you can evolve your portal from launch to liftoff.

Conclusion: Parting thoughts

Ultimately, nearly every organization stands to reap the benefits of an internal developer portal: superior engineering quality, greater standardization and efficiency, and improved visibility of engineering output throughout the organization.

This practical guide has covered a lot of ground.

- We've discussed the definition of an internal developer portal and explored the basic considerations – including opinionated vs. un-opinionated, and degree of separation from underlying architecture – that go into selecting the right portal for your organization
- We've done a deep dive into the critical features of a portal – including blueprints, developer self-service actions, scorecards, and automations
- We've explored the importance of reporting and provided battle-tested recommendations on streamlining portal adoption for your organization

We hope that this practical guide has helped shape your thought process as you embark on your internal developer portal journey.

So...who are we?
Port is an open internal developer portal, owned by platform engineering teams and built for developers. Port consolidates everything developers need to know and execute to deliver software autonomously and to comply with organizational standards. You can open a free Port account at getport.io.

We'd welcome the chance to continue the conversation, get in touch

getport.io