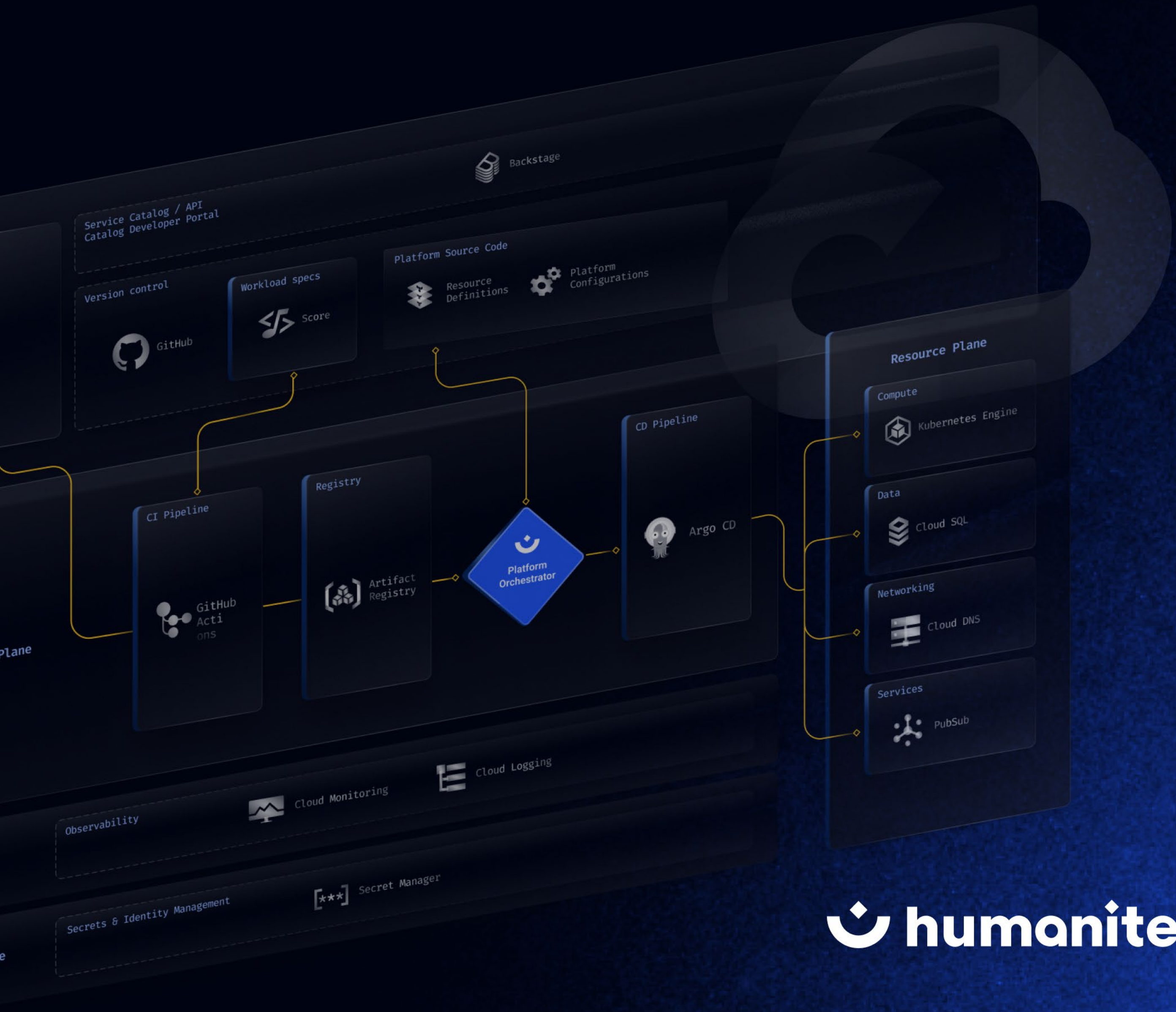
Reference architecture and implementation of an Internal Developer Platform built with Humanitec and Backstage on GCP



humanitec

humanitec

Table of Contents

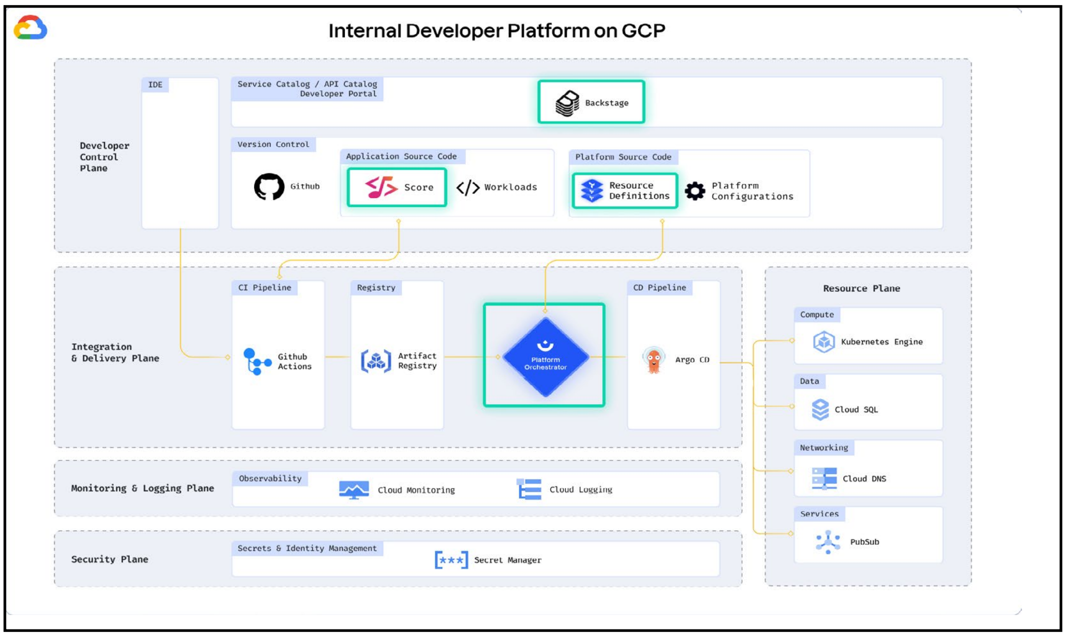
Introduction

To stay competitive, today's organizations must innovate fast and shorten time to market (TTM). This necessitates the transformation of static CI/CD setups into modern enterprise-grade Internal Developer Platforms (IDPs) that improve developer productivity and increase Ops efficiency by driving standardization and automation. A well designed IDP eliminates ticket-based workflows (ticket ops) and minimizes repetitive manual tasks having to be performed under time constraints, which causes DevOps burnout.

For developers, such an IDP reduces cognitive load and enables them to self-serve everything they need to be productive. Improving the developer experience leads to higher developer productivity which at the end results in shorter time to market.

But how exactly do you build an enterprise-grade IDP, and where do you start? As an industry, we need to move beyond buzzwords and provide real-life examples of modern IDPs. While every platform looks different, certain common patterns emerge. To help simplify things, a group of platform experts consolidated the platform designs of hundreds of setups into standard patterns based on real-world experiences, which have been proven to work effectively. They introduced these new reference architectures for IDPs on Amazon Web Services (AWS) and Google Cloud Platform (GCP) during a presentation at PlatformCon 2023, which inspired us at Humanitec to create our own IDP reference architectures for AWS, Azure, and GCP-based setups.

By adopting these patterns, organizations can create IDPs so they can innovate and deliver applications faster than ever before, and stay ahead of the competition.



In this whitepaper you'll learn key design principles for a reference architecture on GCP including how a platform is structured into different plane levels, the core components, and how they are integrated into the end-to-end architecture.

In total Humanitec provides you with a prepackaged platform as code:

- The reference architecture
- The implementation code per vendor for a reference architecture on GCP
- The template to create your own reference architectures
- Resource Packs for GCP to connect preconfigured resources to the Humanitec Platform Orchestrator.
- A learning path where you'll find hands-on tutorials that run you through how to set up the reference architecture in GCP.

You can use this pre-packaged platform to get started and build your own Minimum Viable Platform (MVP) in no time at all. This approach (explained in more detail later) allows you to start simple and small, and prove value fast, just as 100s of top-performing platform teams have already successfully done.

Please note that definitions and wording that are used in this whitepaper will reference internaldeveloperplatform.org, which usually follows official Gartner definitions, and that this reference architecture is for enterprises with more than 50 full-time application developers focused on cloud-native technologies.

Problems this IDP design aims to solve

Enterprise-grade IDPs can help solve a number of problems faced by organizations today.

↳ Slow time to market

When it comes to launching business applications, fast time to market (TTM) is critical for organizations to stay competitive. This puts great pressure on developers to run what they build. Unfortunately, this often involves the overwhelming overhead of dealing with infrastructure and DevOps-related tasks. To relieve this pressure, platform engineering teams build IDPs that drive standardization by design and enable developers to self-serve the tech and tools they need to work more efficiently without waiting for Ops to help. This helps remove bottlenecks and boosts developer experience and productivity. By radically simplifying how teams deliver software, organizations can increase deployment frequency and speed up lead time which results in the ability to slash TTM, drive revenue growth, and ultimately, respond faster to a competitive landscape.

↳ Ops inefficiency

Platform engineers, infrastructure, and Ops teams mostly want to get rid of repetitive tasks, but with peace of mind knowing that developers aren't breaking stuff. Often they get bogged down with spending time manually reviewing or maintaining output from developers, which can cause frustration, delays, and decreased productivity. An enterprise-grade IDP allows developers to perform daily deployment activities like deploy, debug, rollback, observe, apply minor changes to configs, and add or remove resources, all without having to wait for Ops to do this for them. Developers can work with autonomy and speed, moving fast with guardrails they can trust when they do go fast. By enabling the automation of repetitive tasks, IDPs also allow Ops to easily optimize default configs and add frequently used functionality to their platform. IDPs also offer a high degree of flexibility so platform engineers are no longer restricted in the way automations and configs are set. A platform should conform to the exact needs of their organization and their security practice, and drive standardization of app and infrastructure config management, enforcing them as users consume them.

↳ Poor developer productivity

For developers, too much time spent switching between tools and interfaces can result in frustration and additional cognitive load which slows productivity and innovation. An enterprise-grade IDP can provide a unified interface (golden paths) and formats, and reduce the complexity developers face when building and running their apps. When used with an open-source workload specification (such as Score), an IDP enables developers to request the resources their workload depends on in a declarative way. Developers can run the same workload on completely different technology stacks, without needing to be an expert in any one of them. There's no context-switching, it's easy to learn, and it removes the need to worry about tech and tools when promoting workloads from local to production. Developers can focus on writing and deploying code. All this helps improve DevEx, kills shadow ops, and in the end, leads to higher developer productivity.

↳ Lack of standardization

In current static CI/CD setups, there can be dozens of ways to reach the same goal, such as spinning up a new Postgres Database or deploying to production. To do this, developers need to know and maintain the environment-specific config before any deployment. They also need to define and maintain a separate connection string for a database in each environment. In this way, static setups require teams to operate an ever-increasing number of config files, hindering productivity and increasing change failure rate. With a Platform Orchestrator platform engineers can ensure standardization by design across infrastructure provisioning and app config management, and build golden paths that developers can consume with low cognitive load. Developers only need to operate a single file per service, which heavily reduces cognitive load, while being able to choose their level of abstraction. They simply describe what their workload requires to run, and the optimal resource will be matched or created. The number of config files a team needs to maintain can be reduced by 95%, reducing infrastructure sprawl, minimizing risk of human error, and ensuring compliance and consistency across environments and infrastructure.

Design principles

We follow several proven principles when designing IDPs.

01 Treat your platform as a product: We believe in taking a platform as a product approach. This is where a platform team builds, maintains, and continuously works to improve the platform by following product management principles and best practices. The goal is to ensure an IDP is built that developers actually want to use and that can deliver value to the organization.

02 Enable standardization by design: Shifting the focus onto self-service requires platform engineers to define ways to vend resources and configuration. This means that every resource is built securely, compliant, and well-architected.

03 Build golden paths over cages: Creating golden paths that provide best practice guidance and recommended approaches help lower cognitive load and improve security and standardization. IDPs that apply smart layered abstractions give developers the choice to follow the golden paths, be free to leave, or change lower-level configs if the security posture allows.

04 Assume a brownfield situation: We assume that we'll encounter a brownfield situation, where a team is confronted with several CI systems, registries, IaC format, and clouds. The integration of the status quo should be fast, seamless, and not require major change effort.

05 Leave platform interface choice to the developer: We believe in giving developers the freedom to use the interfaces they're most comfortable with and that best meet their needs. Because of this, we offer a wide range of frameworks, libraries, and interfaces, and never enforce a choice. We also opt for code-based workflows by default while providing the option to use an OSS workload specification like Score, a portal (GUI), CLI, or API.

06 Never break developer workflows: Anything that changes daily habits is considered a distraction. We believe in designing platforms that maintain stability and compatibility, ensuring developers can work safe in the knowledge their workflow will not be disrupted.

07 Keep design code-first: We consider code as the single source of truth which helps maintain transparency, increase reliability, and simplify maintenance. An IDP that's code-first at its core allows for disaster recovery, versioning, and structured product development principles. While the architectural design offers interface choice and features a UI, CLI, or API, we ensure primary developer interaction is also code-first so anyone can understand and contribute to the platform, as well as make improvements.

08 Take an 80/20 attitude to platforming: While we believe in standardization, we understand every organization has unique needs. To adapt to different situations, meet diverse needs, and benefit from evolving technologies, staying open-minded is key. Our platform design doesn't try to cover every technology on earth or convince every developer in a user base. We do not assume that we'll be able to please 100% of developers. Instead, we consider achieving 80% a great win.

Architectural components

We're thinking about the different areas of the platform architectures as "plane levels" that cluster certain functionalities. Let's zoom in on the plane levels we have to take care of and see what technologies fulfill each function in each of these levels.

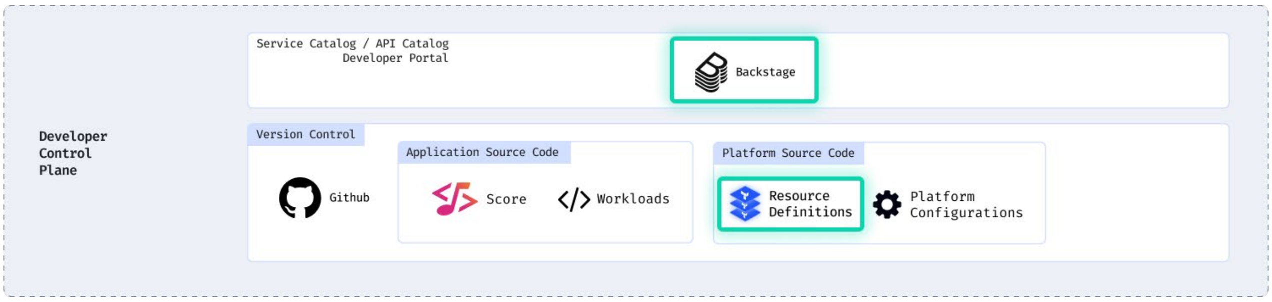
Developer Control Plane

Under the term Developer Control Plane, we cluster the primary "interfaces" developers can choose to use when interacting with the platform or applying any change. As discussed in the "Design principles" section, we leave interface change to the developer on a workload-by-workload basis. We also, by default, recommend not to break the current workflow of the developer. Which is why we default to code wherever possible.

Components used in this architecture

This plane is the platform users' primary configuration layer and interaction point. It harbors the following components:

- A Version Control System is a prominent example, but this can be any system that contains two types of repositories:
    - Application source code
    - Platform source code, e.g. using Terraform
- Workload specifications. The reference architecture uses Score.
- A portal for developers to interact with. This can be the Humanitec Portal, but it can also be Backstage or any other portal on the market. This reference architecture uses Backstage



Following the design principle "everything as code" both the app and platform source code are stored in Git. The platform source code represents the configuration of the platform and is maintained using the IaC framework Terraform. Terraform is used for both managing the Humanitec Resource Definitions using the Humanitec Terraform provider, and for configuring the different automation systems. The primary interaction method for developers is designed to be code-driven using the Workload spec Score, to describe the Workload and dependent Resources in abstract terms. Git integrates with the IDE, the CI pipeline, and the portal using the GitHub API.

The portal layer offers a user interface on top of all platform capabilities that acts like a single pane of glass, including shortcuts to scaffolding new services, metrics, service catalogs, and additional self-service actions. The portal integrates with the VCS through its API, using plugins where available, and equally to the Platform Orchestrator. It might also pull additional data directly from the CI pipelines or project management systems like JIRA.

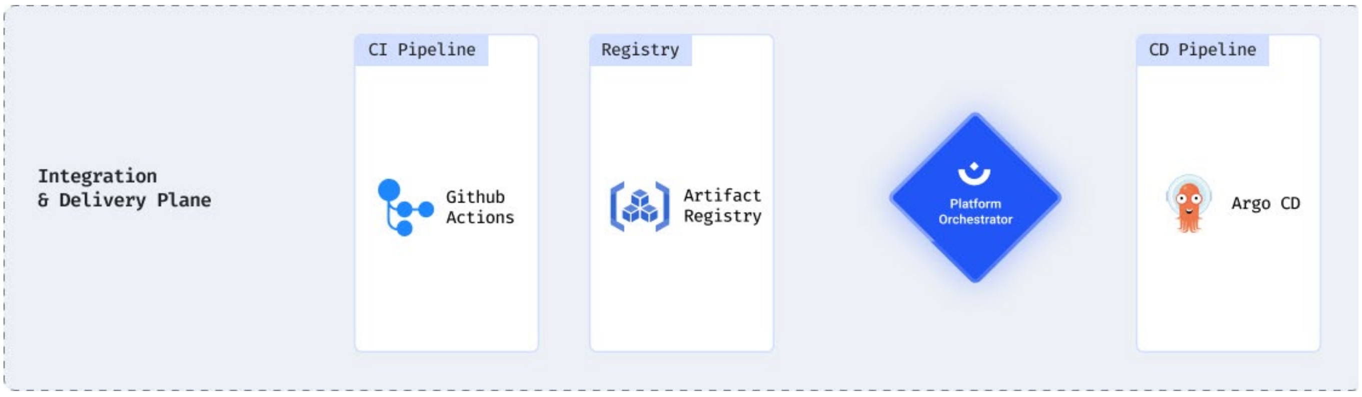
Integration and Delivery Plane

This plane is about building and storing the image, creating app and infra configs from the abstractions provided by the developers, and deploying the final state. It's where the domains of developers and platform engineers meet.

Components used in this architecture

This plane usually contains four tools:

- A CI pipeline. This can be GitHub Actions or any CI tooling on the market.
- The image registry holding your container images. Again, this can be any registry on the market.
- A Platform Orchestrator, which in our example is the Humanitec Platform Orchestrator.
- The CD system, which can be the Platform Orchestrator's deployment capabilities, an external system triggered by the Platform Orchestrator using pipelines, or a setup in tandem with GitOps operators like Flux or ArgoCD.



The job of the CI pipeline is to build and test the developed code. It then pushes new images to the image registry.

At the end of its run, the pipeline may also notify the Platform Orchestrator about the new image and submit the Workload specification (Score) to trigger a deployment using the newly generated image. An example pipeline in the form of a GitHub workflow is available in our architecture documentation. The CI pipeline passes deployment metadata to the Platform Orchestrator, including image path, tags, and a deployment delta (basically the converted Score file in a format the Platform Orchestrator can interpret).

The registry itself does not have a direct integration with the Platform Orchestrator. You may choose to have the Platform Orchestrator provide registry credentials to CI/CD systems, or to workloads running in your Kubernetes cluster by injecting registry credential secrets. See the integration options for details.

Monitoring and Logging Plane

The integration of monitoring and logging systems varies greatly depending on the system. Some require using sidecar containers included in all your Workloads, adding labels and annotations to your Application configurations, or by some other method.

Monitoring & Logging Plane
Observability
Cloud Monitoring
Cloud Logging

Security Plane

The security plane of the reference architecture focuses on the secrets management system. The secrets manager stores configuration information such as database passwords, API keys, or TLS certificates needed by an Application at runtime. It allows the Platform Orchestrator to reference the secrets and inject them into the Workloads dynamically. You can learn more about secrets management and integration with other secrets management in our security documentation.

The reference architecture sample implementations use the secrets store attached to the Humanitec SaaS system.

Resource Plane

This plane is where the actual infrastructure exists and includes clusters, databases, storage, or DNS services. The configuration of the Resources is managed by the Platform Orchestrator which dynamically creates app and infrastructure configurations with every deployment and creates, updates or deletes dependent Resources as required.

Check the list of specific Resources created for each of the reference architecture implementations in the respective README. Additionally, we open sourced Resource Packs, a curated collection of Resource Definitions as code, featuring best practice configurations for both infrastructure-based elements (like S3 buckets) and logical elements (such as IAM Policies). They are tested and ready to be seamlessly integrated with the Platform Orchestrator. Alongside AWS, Azure, GCP, there's also a special Resource Pack with definitions for in-cluster Resources if you want them to be provisioned inside your Kubernetes cluster.

Components used in this architecture

- Google Kubernetes Engine (GKE)- a fully managed Kubernetes service on GCP.
- Cloud SQL - a managed relational database service.
- Cloud DNS - a highly available and scalable DNS service.

Your platform interfaces

The reference architecture allows the user or the organization to choose their preferred interface with your platform, which will offer varying degrees of flexibility and abstraction. How these abstractions are resolved depends on the context and the rules set by the platform team. Our reference architecture comes with the following interface options:

An Abstract code-based Workload specification (Score):

A code-based, abstract configuration of the Workload and its dependencies. This specification prevents workflow breaks and can facilitate self-service actions such as changing Environment variables or removing/adding Resources. A Score specification may also be used for promoting an Application to the next Environment, spinning up a PR Environment, and adding meta-data to your deployment. It does this by adding or removing parts of the code and varying deployment metadata. This can then be interpreted by the Platform Orchestrator, which executes the request.

Specific IaC and Resource Definitions (Terraform):

Code-based, specific Resource configurations and templates. These are usually maintained by a central Ops/platform team and are ideally applied across Applications to achieve standardization by design. Only if you need a very specific configuration for a Resource would you modify those files.

Orchestrator API:

The Orchestrator API acts as the de facto platform API that allows you to pull data and intel about what's going on inside the platform, and automate and trigger actions. The API covers 100% of the Platform Orchestrator capabilities and can be used for actions like spinning up Environments, rolling back, pulling information, running diffs between deployments, and much more.

Orchestrator CLI:

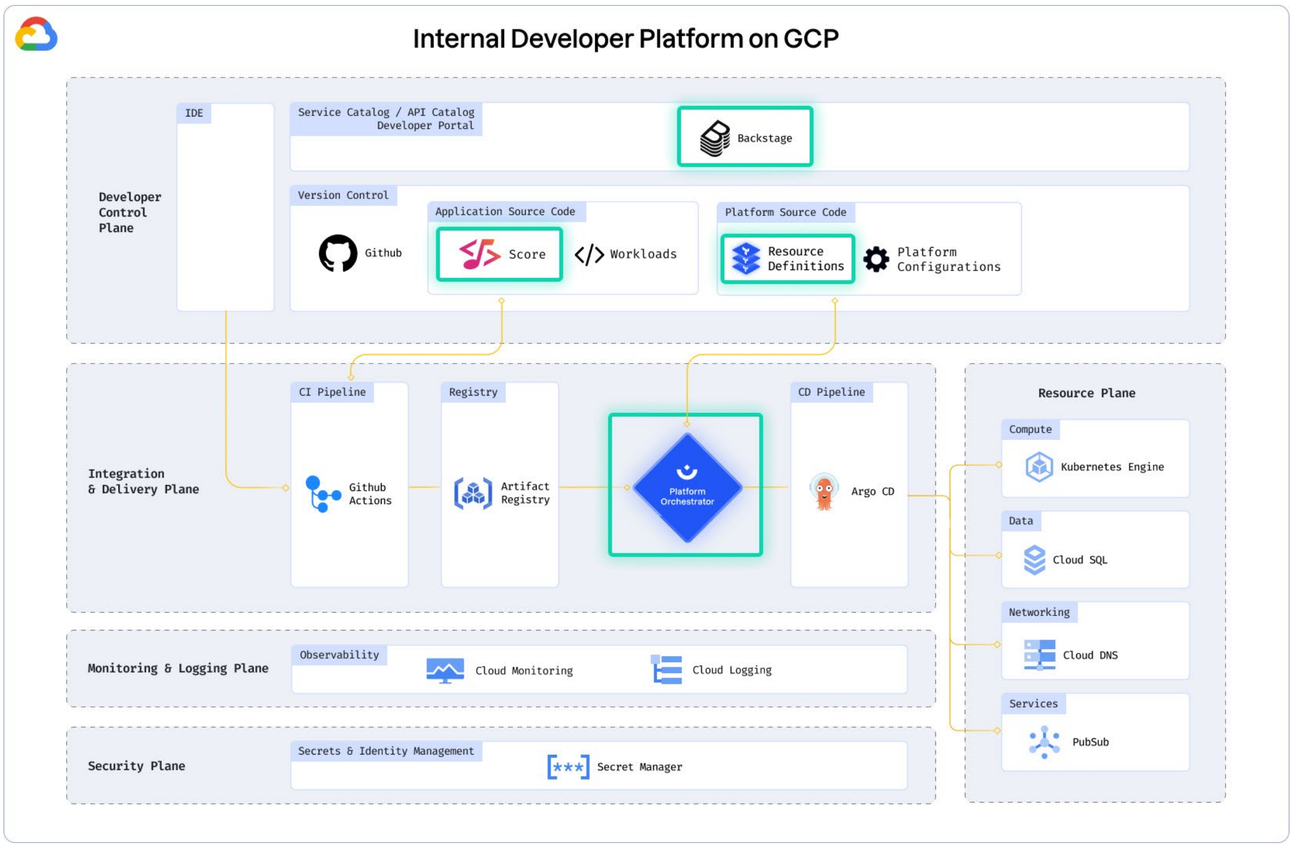
The API and most commonly used combinations of requests are also packaged as a CLI.

Portal:

Per default, you can use the Humanitec Portal. The UI also allows you to perform most self-service actions and automation such as rolling back, running diffs, or adding Applications and Resources. You can also build your own user interface using a base product like Backstage, working the Platform Orchestrator API behind the scenes or you adopt any other available portal solution. The reference architecture uses Backstage.

The end-to-end architecture result

Now that we've defined the different plane layers, let's put all of it together into an end-to-end architecture. The interplay of some layers is intuitive, others need slightly more deep-dive. Ultimately, our assembled architecture will look as follows:



How platform engineers or Ops teams build, operate, and maintain a platform

As the name suggests, platform engineers or Ops teams build the platform. This means they are ultimately responsible for designing the individual layers and the interplay of those planes. Depending on the organization's team structure, the responsibility for the different planes varies. Here's a list of the most common patterns we've observed in the enterprise; note that often the distinction is pure naming convention:

Plane	60% of orgs	20% of orgs	20% of orgs
Developer Control Plane	Platform engineering	Ops	Developer experience
Integration and Delivery Plane	Platform engineering	Ops	CI/CD
Resource Plane	Platform engineering	SRE/Infrastructure	Ops
Security Plane	Security	Platform engineering	Ops
Observability Plane	Platform engineering	Ops	Developers

Platform engineers, infrastructure, and Ops people mostly want to get rid of repetitive tasks, but with peace of mind knowing that hordes of devs aren't breaking stuff. Here's what they are primarily looking for:

01 Automation of repetitive tasks: They want to optimize default configs and add frequently used functionality to their platform. They don't want to spend time manually reviewing or maintaining output from developers.

02 High degree of flexibility: Platform engineers don't want to be restricted in the way automations and configs are set. The platform should conform to the exact needs of their organization and their security practice.

03 Driving standardization: The Internal Developer Platform setup should drive standardization of app and infrastructure config management, and enforce them as users consume them.

In the vast majority of cases teams already have an existing setup when building their IDP, so it's often about remodeling to ensure their setup matches the design principles. Here's how:

01 Design the individual planes. Start with the Resource Plane because it dictates other design decisions on other layers. We usually propose the following order:

- Resource Plane (you probably already have Resources. In this case, decide which are supported by your platform as a default).
- Integration and Delivery Plane: Pipeline design, configs of the Orchestrator etc.
- Security Plane.
- Monitoring and Logging Plane.
- Developer Control Plane: This heavily depends on the design choices of the other planes and should always come last.

02 Wire the individual components of the planes to each other as well as one plane to the other and test the raw end-to-end flows.

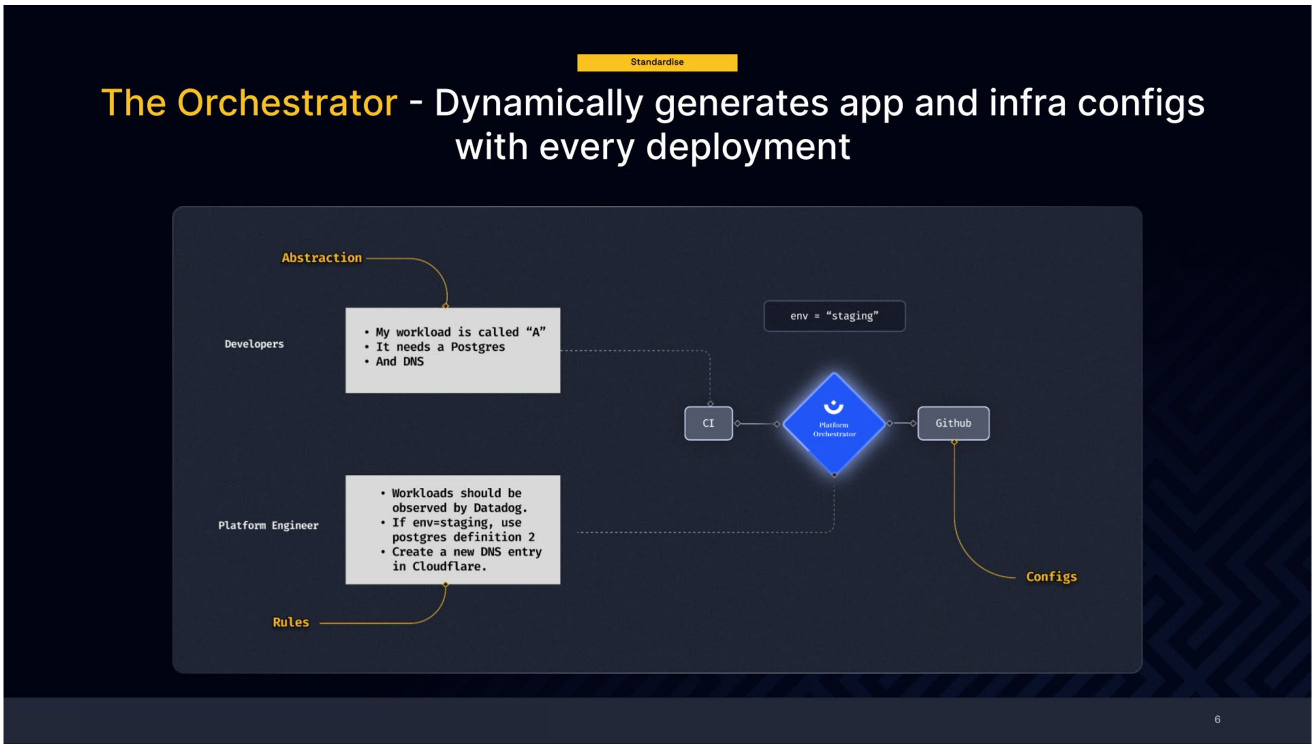
03 Set baseline configs for app and infrastructure configs (more details below).

04 Set deployment and environment progression automation.

05 Design and implement Roles Based Access Control (RBAC).

06 Onboard your first application.

We've covered the planes and their design, let's next zoom in on the baseline configs and automations. This means the next job to be done for the platform engineering team is to set those app and infrastructure config defaults.



Standardise

The Orchestrator - Dynamically generates app and infra configs with every deployment

- My workload is called "A"
- It needs a Postgres
- And DNS

- Workloads should be observed by Datadog.
- If env=staging, use postgres definition 2
- Create a new DNS entry in Cloudflare.

The core of the IDP is the Platform Orchestrator. With every git-push, the Platform Orchestrator interprets what resources and configs are required for a workload to run. It creates app and infrastructure configs based on rules defined by the platform team and executes them following a "Read"-"Match"-"Create"-"Deploy" pattern:

- Read: interpret workload specification and context.
- Match: identify the correct configuration baselines to create the application configurations and identify what resources to resolve or create based on the matching context.
- Create: create application configurations; if necessary, create (infrastructure) resources, fetch credentials and inject credentials as secrets.
- Deploy: deploy the workload into the target environment wired up to its dependencies.

One part of the rules are baseline configs, the other part is automation.

Baseline configuration

Platform engineers define baseline configurations in the Humanitec Platform Orchestrator, typically centralized and maintained by platform teams or senior developers. These baseline configurations include several key components:

01 Workload Profiles: These are application configuration baselines containing essential details such as CPU minimum allocations, labels, and annotations. They serve as the foundation for creating the final application configuration, analogous to empty Helm charts.

02 Resource Definitions: These define how to wire existing resources or create new ones using Infrastructure as Code tools like Terraform or Humanitec Drivers. This component outlines the approach for integrating and managing Resources within the infrastructure. Check our Resource Packs, a curated collection of Resource Definitions as code, featuring best practice configurations for both infrastructure-based elements (like S3 buckets) and logical elements (such as IAM Policies). They are tested and ready to be seamlessly integrated with the Platform Orchestrator.

03 Available Resources and their Matching Criteria: This component specifies how to select or create Resources based on certain conditions. For example, if a workload specification requires a database of type Postgres and the context matches the criteria “Environment Type = production or development,” the Humanitec Platform Orchestrator would use the driver humanitec/postgres-cloudsql to connect the workload to an existing database in a CloudSql instance.

These baseline configurations are crucial for the Humanitec Platform Orchestrator to dynamically create application and infrastructure configs with every git push. By applying abstract requests (workload specifications) to these baseline configs, the platform engineering team enables the dynamic and efficient setup of applications and infrastructure. For more detailed information on how these baseline configs are utilized within the Humanitec ecosystem, head to the documentation on Baseline Configuration and How Humanitec’s Products Work.

Automation

While the baseline configurations care for how the abstract ask of a developer is answered with concrete infrastructure representations by the platform, the deployment pipelines answer how the workloads enter and progress between environments. The Platform Orchestrator is usually triggered by a CI pipeline but can also be triggered manually or by API. These are the functionalities and integrations that need to be detailed and configured at this step:

- Integrate your CI pipeline with the Orchestrator
- Define promotion among environments
  - Pre-requisites for promotion like:
    - Executed automated tests
    - Policy checking
    - Sign-offs
    - Pre-deployment notifications
  - Post promotion activities like:
    - Execute automated tests
    - Policy checking
    - Post deployment notifications
    - API updates of other systems like e.g. CMDB
  - Create pipelines for environment promotion
  - Create pipelines for deployments if specific steps beyond defaults are needed
    - Automated
    - Manual
    - Artifact update
    - Scrutinize with your manual intervention and disaster processes in mind

Roles and Role-Based Access Control (RBAC)

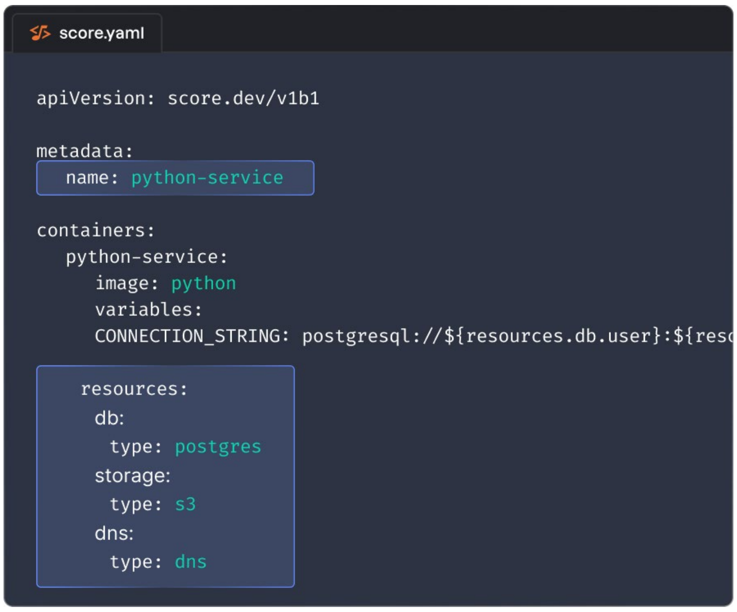
The RBAC of the VCS and the Platform Orchestrator allows your organization to define the correct levels and control for your developers across roles. How these RBACs are set up depends on the security posture of your organization. Are developers allowed to change infrastructure resources and or templates? Who's allowed to deploy to production? Who's permitted to change the baseline templates? We usually think about an RBAC setup before onboarding applications at scale.

How developers use such a platform

Holding true to our design principle of leaving interface choice and opting for code first, the answer is: It depends! The proposed architecture leaves that choice to a workload-by-workload basis.

The primary interaction method (by far the most used) is the code-based one. Developers prefer to stay in their usual workflow, in the VCS, and within their integrated development environment to "indicate" what their workloads require, spin up new services, add Resources etc. This is where a workload specification like Score comes into play. It provides a code-based "specification" to describe how the workload relates to other workloads and their dependent Resources. Adding a Resource Definition to the Score file will tell the Platform Orchestrator to automatically create a new Resource or wire an existing one. We'll explain how this works in the next section.

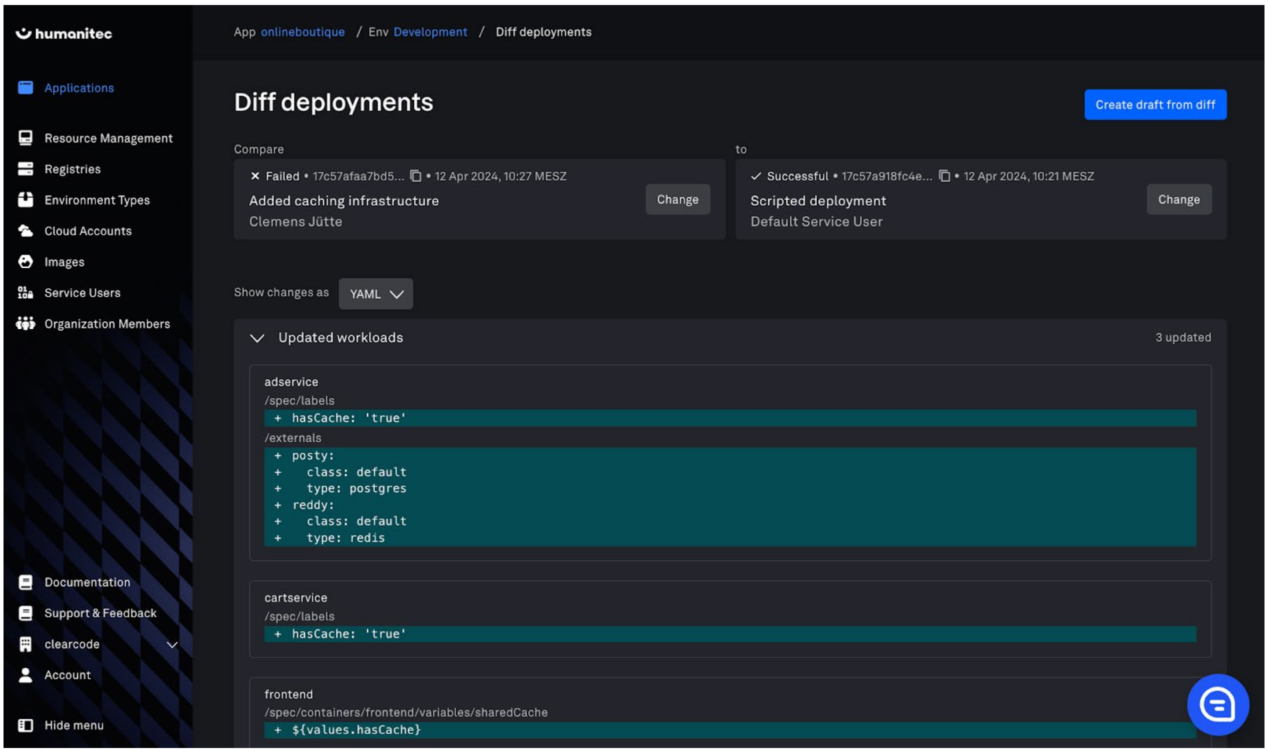
This is a simple example of how a Score file looks:



```yaml
apiVersion: score.dev/v1b1

metadata:
  name: python-service

containers:
  python-service:
    image: python
    variables:
    CONNECTION_STRING: postgresql://${resources.db.user}:${reso

   resources:
    db:
      type: postgres
    storage:
      type: s3
    dns:
      type: dns
```

We can see that the developer requires a database type Postgres, a storage of type S3 and a DNS of type DNS. For the vast majority of use cases, this code-based format should be entirely sufficient and is the preference for most developers.

For specific situations (like running diffs, rolling back, or spinning up new environments) developers might prefer to use the Platform Orchestrator UI, CLI, or API.



App onlineboutique / Env Development / Diff deployments

Diff deployments

Compare
✕ Failed • 17c57afaa7bd5... • 12 Apr 2024, 10:27 MESZ
Added caching infrastructure
Clemens Jütte

to
✓ Successful • 17c57a918fc4e... • 12 Apr 2024, 10:21 MESZ
Scripted deployment
Default Service User

Show changes as YAML

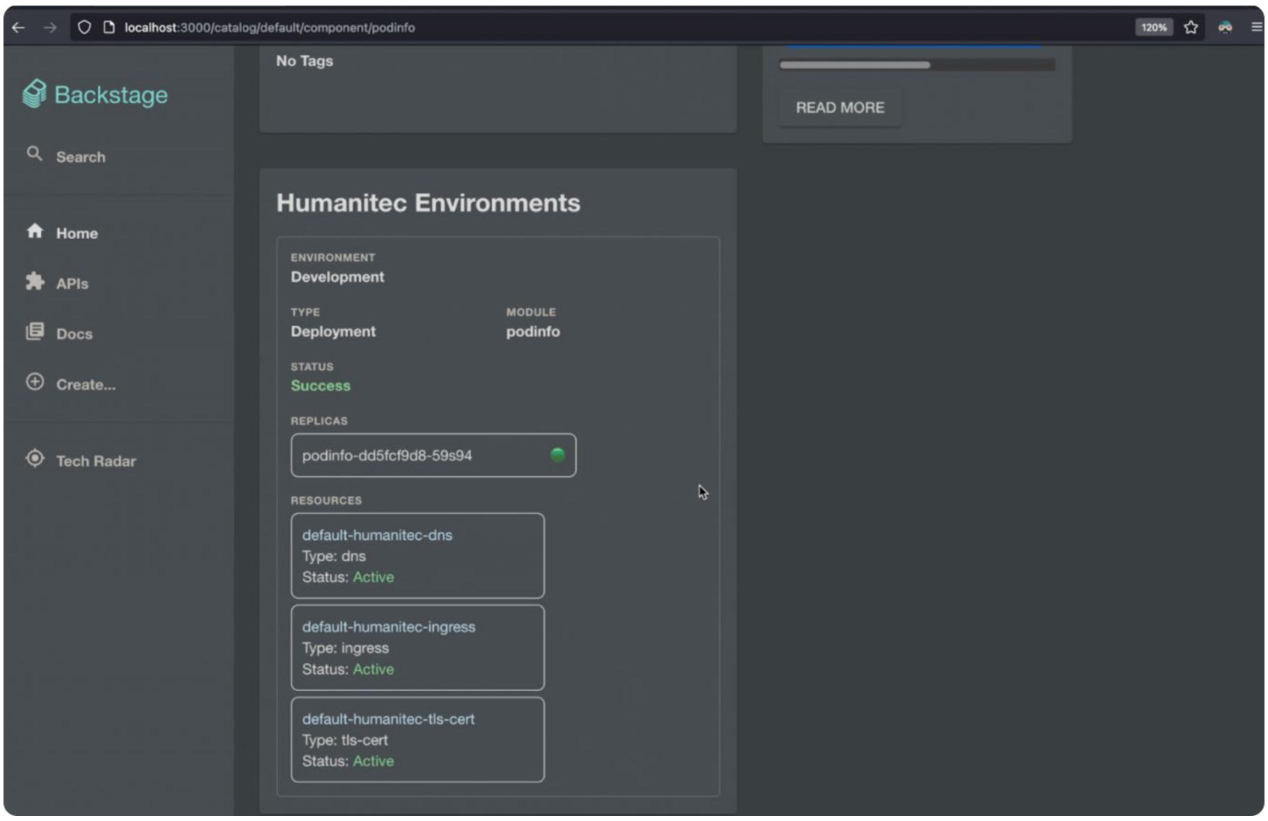
Updated workloads — 3 updated

```
adservice
/spec/labels
 + hasCache: 'true'
/externals
 + posty:
 +   class: default
 +   type: postgres
 + reddy:
 +   class: default
 +   type: redis
```

```
cartservice
/spec/labels
 + hasCache: 'true'
```

```
frontend
/spec/containers/frontend/variables/sharedCache
 + ${values.hasCache}
```

Portals and service catalogs are primarily used for consolidation by product managers/engineering managers, as well as onboarding and orienting new developers.



No Tags

Humanitec Environments

ENVIRONMENT
Development

TYPE
Deployment

MODULE
podinfo

STATUS
Success

REPLICAS
podinfo-dd5fcf9d8-59s94

RESOURCES
default-humanitec-dns
Type: dns
Status: Active

default-humanitec-ingress
Type: ingress
Status: Active

default-humanitec-tls-cert
Type: tls-cert
Status: Active

Here's a list of some activities a developer performs using an IDP and what interface they usually choose:

Activity type	Predominant interface choice
Deploy	Terminal/IDE
Change configuration	Code: Workload specification (Score)
Add/remove resource	Code: Workload specification (Score)
Roll back/Diff	Orchestrator UI
Configure resource in detail	Code: IaC
Spin up a new environment	Orchestrator UI
See logs/error messages	Orchestrator UI
Search service catalog	Portal/Service catalog
Inner source use case	Portal/Service catalog
Service create case	Portal/Service catalog or templating in VCS

Zooming in on a “golden path” to understand the interplay of all components

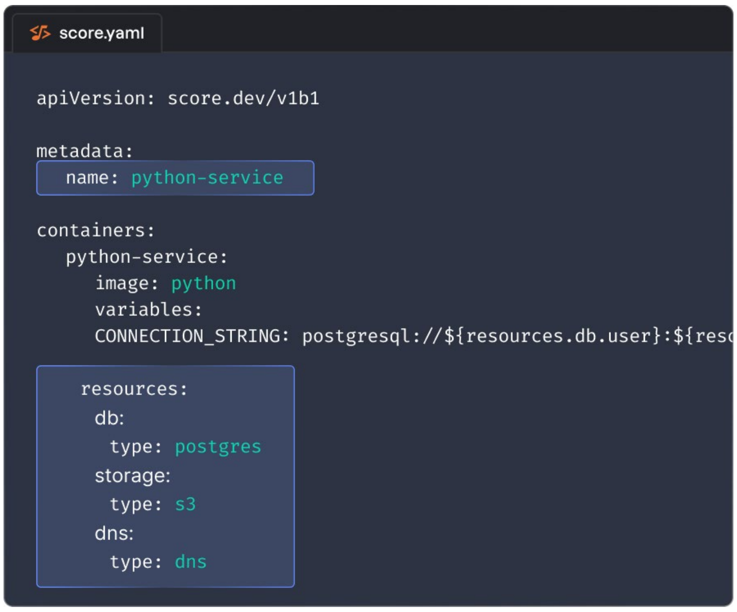
Now we understand the integration points for the different planes, let’s look at how a developer deployment request would flow through the platform step-by-step. We call these standard flows “golden paths”. From the perspective of a developer, they can be certain that nothing will break if they stay on the path, while at the same time, they are able to go off-path if they so desire, however without the same guarantees that nothing will break.

Below are three (of many) standard flow examples the IDP provides golden paths for and another example how to maintain such golden paths.

Golden path 1: Simple deployment to dev

Let’s start with a very simple example. A developer has changed something on their work-load and now deploys to a dev environment. As we discussed earlier, the primary interaction method for devs would be code. They would git-push their change, and the CI pipeline would pick it up and run. It would then push the built image to the image registry. At this point we have the service built, but we don’t have the configs yet (remember, we’re opting for Dynamic Configuration Management).

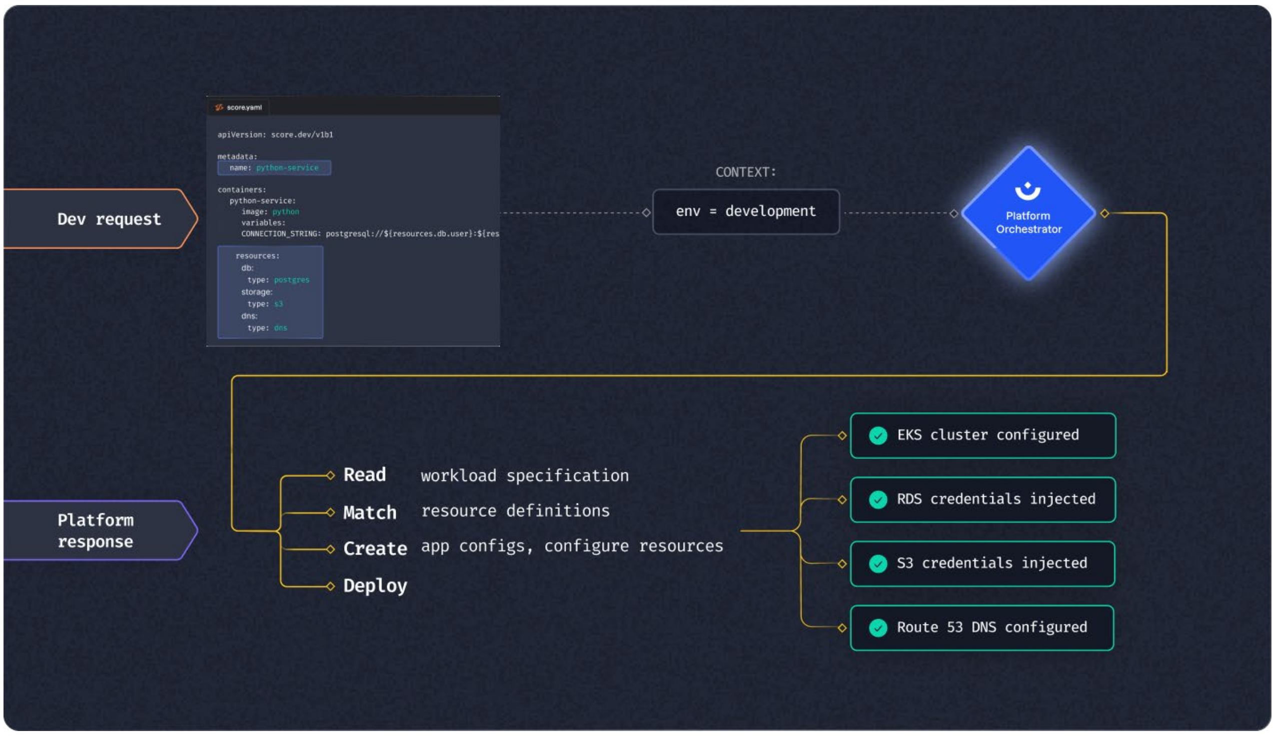
The workload source code contains the workload specification (Score), which as shared above, in this case might look like this:


score.yaml

```yaml
apiVersion: score.dev/v1b1

metadata:
  name: python-service

containers:
  python-service:
    image: python
    variables:
    CONNECTION_STRING: postgresql://${resources.db.user}:${reso

    resources:
      db:
        type: postgres
      storage:
        type: s3
      dns:
        type: dns
```

We can see that the developer requires a database type Postgres, a storage of type S3, and a DNS of type DNS.

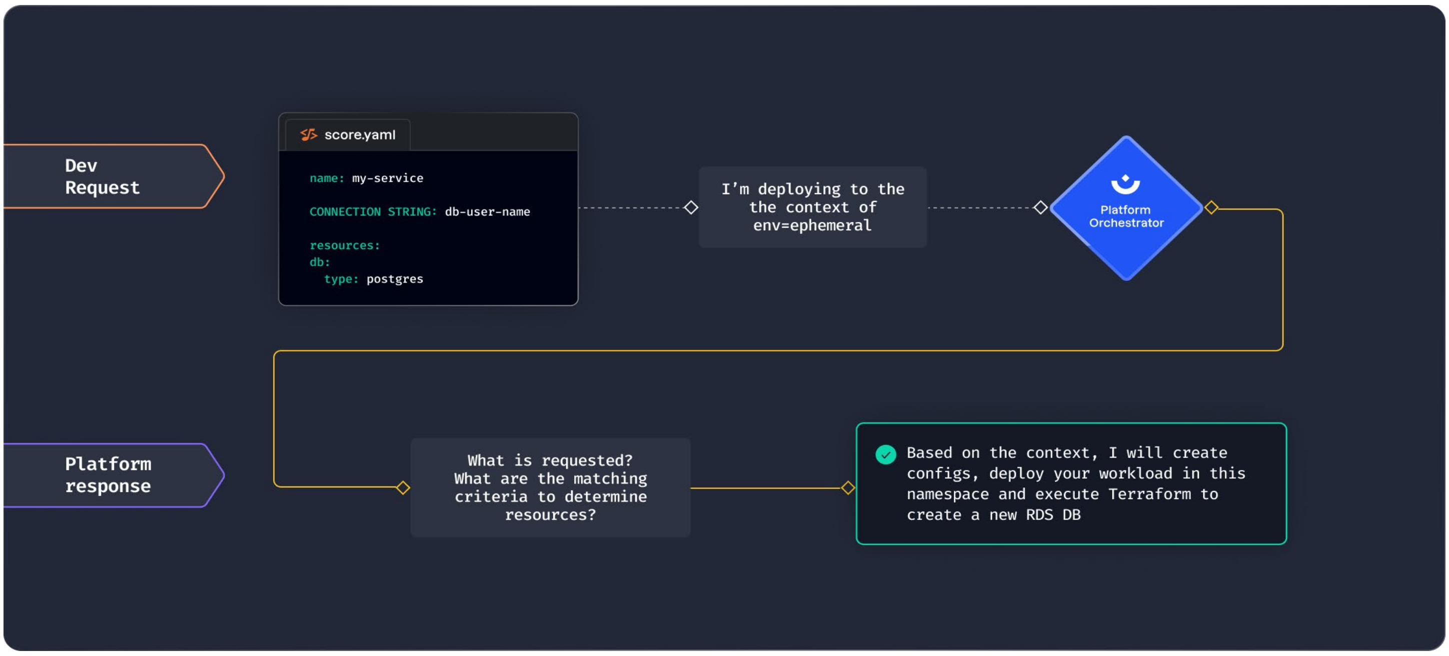
So after the CI has been built, the Platform Orchestrator picks up the context and looks up what Resources are matched (in our case, it’s maybe the context “environment = development”). It checks whether the Resources are already created (which is likely in this case because it’s just a deployment to an existing dev environment) and reaches out to the GCP API to retrieve the Resource credentials. It then creates the application configs in the form of manifests because our target compute in this architecture is Amazon EKS. Once this is done, the Platform Orchestrator deploys the configs and injects the secrets at runtime into the container (utilizing Vault).



```
apiVersion: score.dev/v1b1

metadata:
  name: python-service

containers:
  python-service:
    image: python
    variables:
    CONNECTION_STRING: postgresql://${resources.db.user}:${res
   resources:
    db:
     type: postgres
    storage:
     type: s3
    dns:
     type: dns
```

Golden path 2: Create a new environment

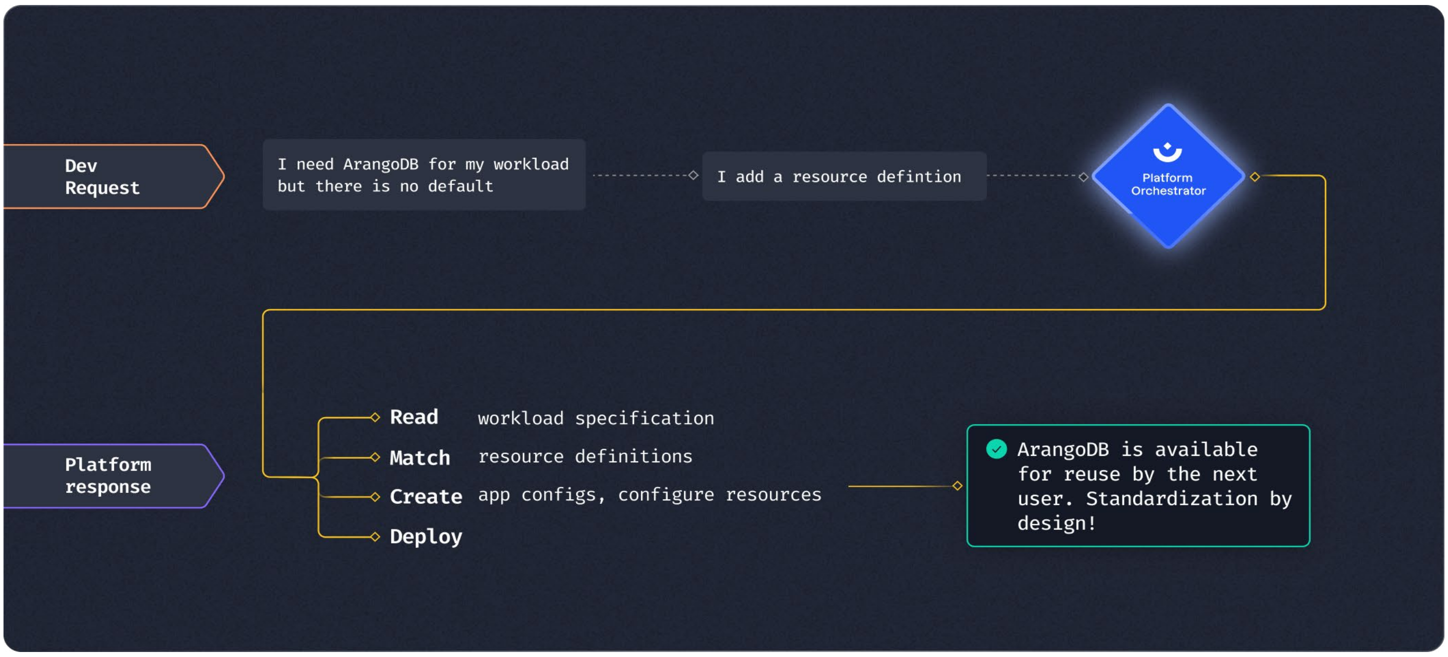
Let’s repeat the same procedure but spin up a new environment. The developer experience is as simple as it gets; they push the same repository and the same workload specification (because it works across all environments). By setting the context to “ephemeral” (through the tag), the Platform Orchestrator will again now interpret the workload specification. It will realize that Postgres doesn’t exist yet and that it should create one using a specific Driver. The Platform Orchestrator will then create the configs, inject the dependencies, and serve.


Dev Request
score.yaml
I’m deploying to the the context of env=ephemeral
Platform Orchestrator
Platform response
What is requested? What are the matching criteria to determine resources?
Based on the context, I will create configs, deploy your workload in this namespace and execute Terraform to create a new RDS DB

```
name: my-service

CONNECTION STRING: db-user-name

resources:
db:
  type: postgres
```

Golden path 3: Create a new resource based on the developers’ request

An interesting case is when the developer sends a request the system doesn’t know yet. This is where many approaches fail, except when applying Dynamic Configuration Management (DCM). Because everything is repository-based, the approach allows developers to extend the set of available resources or customize them to their liking. Let’s play through the scenario where a developer needs an ArangoDB, but this isn’t known to the setup so far. By adding a Resource Definition to the general baselines of the organization, the developer can easily extend the setup in a way that can be reused by the next developer.


Dev Request
I need ArangoDB for my workload but there is no default
I add a resource defintion
Platform Orchestrator
Platform response
Read workload specification
Match resource definitions
Create app configs, configure resources
Deploy
ArangoDB is available for reuse by the next user. Standardization by design!

Maintaining a golden path: A platform engineer updates the dev Postgres resource to the latest Postgres version

This is a great example of how platform engineers utilize the platform to maintain a high degree of standardization. Let’s explore how one would go about updating the dev Postgres resource to the latest Postgres version. Given our architectural choice, let’s assume all resources use Terraform for Infrastructure as Code. To make the example more compelling, let’s also assume that development environments all have their own instance of a Postgres. This approach is so expensive that in reality, we would likely share instances across several dev environments. So how would we go about updating all our Postgres resources across the workloads that use them?

01 The “thing” we need to update is the Resource Definition of dev Postgres resources. Assuming we have our Postgres configured in Terraform, we’ll simply update the Terraform module. Otherwise, we would actually adopt the driver.

02 Should we have to apply changes to the in and outputs, we would need to update the Resource Definition itself in the Terraform provider of the Orchestrator.

```
resource "humanitec_resource_definition" "postgres" {
  id          = "db-dev"
  name        = "db-dev"
  type        = "postgres"
  driver_type = "humanitec/postgres-cloudsql-static"

  driver_inputs = {
    values = {
      "instance" = "test:test:test"
      "name"     = "db-dev"
      "host"     = "127.0.0.1"
      "port"     = "5432"
    }
    secrets = {
      "username" = "test"
      "password" = "test"
    }
  }

criteria = [
    {
      app_id = "test-app"
    }
  ]
}
```

03 We then need to find out what workloads are currently depending on our Resource Definition of “dev Postgres”. The answer can be found in our “rules engine”, the Platform Orchestrator. Simply because this is where the “decision is made” regarding what resources to use to wire the workload up, and in what context. We can do this by pinging the Orchestrator API or looking at the user interface in the Resource Definition section: “Usage”.

04 Once identified, we can auto-enforce a deployment across all workloads that depend on the resource type “Postgres dev”, and the new version is rolled out across all workloads and applications.

Benefits of this Architecture

The presented architecture has a significant impact on reducing lead time and thus decreasing time to market. On average, fully rolled-out platforms show a decrease of 30% in lead time. The impact differs from team to team:

Key impact for Platform or Ops teams

The impact of Internal Developer Platforms (IDPs) on Platform and Ops Teams can be significant in a number of ways. First, by **enabling developer self-service**, manual work can be reduced, which in turn allows teams to focus more on improvements. Second, Dynamic Configuration Management (DCM) can **drive standardization by design**, which creates a more efficient and effective process. Third, standardization enables the **rapid onboarding** of new developers, which can be a significant advantage for companies looking to grow their teams quickly. Fourth, **golden paths** can **streamline your setup**, reduce maintenance and allow team members to work more effectively. All of these factors combine to improve the development process, which can lead to better results and **happier team members overall**.

Key impact for developers

The impact of IDPs on application Developers can be significant in many ways. The first advantage is that self-service **reduces dependencies and waiting times.** Developers can easily access the resources they need and work on their projects without having to wait for the IT department to assist them. This can lead to **faster development** and a higher deployment frequency, which is a crucial aspect of modern software development.

Another benefit of IDPs is **streamlined config management, which reduces cognitive load.** Developers can focus on writing code instead of worrying about infrastructure, which can be a complex and time-consuming task. With an IDP, developers can simply select the re-sources they need and configure them as required, freeing up more time for coding.

IDPs also offer new superpowers that can **boost productivity.** For example, developers can use Score as a workload spec, which allows them to specify the desired performance characteristics of their application. They can also spin up PR environments, which can be used to test and debug code changes before merging them into the main codebase.

Furthermore, the diff functionality for debugging allows developers to **quickly identify and fix issues**, while secure infrastructure self-service ensures that the entire development process remains secure.

In conclusion, IDPs can have a **significant impact on the productivity and efficiency of application developers.** By reducing dependencies and waiting times, streamlining config management, and offering new superpowers, developers can focus on delivering high-quality applications.

Key impact for executives

While the most obvious impact IDPs have on executives and the wider business is the **significant reduction in time to market** (TTM), the benefits are far more multifaceted and far-reaching. One of the biggest advantages is the high degree of automation they provide, which in turn reduces failure rates and increases security. This is achieved through the use of advanced tools and technologies that are constantly improving and evolving.

In addition to the automation benefits, IDPs also enable developer self-service which **reduces waiting times and skyrockets productivity.** This allows for **faster innovation cycles** and enables organizations to stay ahead of the competition. Moreover, dynamic IDPs require less full-time Ops employees per every application developer, which helps organizations **streamline overall operations** and **reduce costs**.

Another important benefit of IDPs is **cost control**. By reducing cloud bills and optimizing resource allocation, organizations can invest saved money in other areas of the business. This is especially important in today's highly competitive landscape, where every dollar counts.

Overall, IDPs have the potential to revolutionize the way organizations develop and deploy software. By leveraging automation, developer self-service, and other advanced technologies, IDPs can help organizations stay ahead of the curve and achieve their goals more quickly and efficiently than ever before.

Ready to build your enterprise-grade IDP?

Done well, IDPs drive standardization by design and empower developers to self-serve the tech and tools needed to work more efficiently. They remove the need to wait for Ops help (ticketops), and the complexity of building and running apps which reduces developer cognitive load. Ops can work more efficiently knowing developers can move fast with autonomy, speed, and guardrails they can trust. All this helps improve DevEx, and drives both developer and Ops productivity. Optimizing how software is delivered in this way enables you to innovate at speed and cut lead time, which puts you in the best position to slash TTM and outpace the competition.

With the IDP reference architecture on GCP discussed in this whitepaper, you now have a starting point to build your enterprise-grade IDP in the fastest way possible. The reference architecture is part of our total prepackaged platform as code, which includes the implementation code for this reference architecture on GCP and resource packs for GCP to wire up preconfigured resources to the Humanitec Platform Orchestrator. You can also use our learning path where you'll find hands-on tutorials on how to set up your reference architecture in GCP.

This pre-packaged platform can be used to get started and build your own Minimum Viable Product (MVP) within minutes. And with our MVP program, you get a guided playbook that results in a clear path from a simple MVP to an enterprise-grade IDP, enabling you to start realizing tangible benefits fast.

If you want to get started right away and avoid the common pitfalls of platform engineering. Reach out to our platform architects for expert help.

PlatCo GmbH

(dba Humanitec)

Wöhlertstraße 12-13, 10115 Berlin, Germany
Phone: +49 30 6293-8516

Humanitec Inc

228 East 45th Street, Suite 9E,
New York, NY 10017

Humanitec Ltd

3rd Floor, 1 Ashley Road
Altrincham, Cheshire WA14 2DT
United Kingdom

E-mail: info@humanitec.com
Website: https://www.humanitec.com

CEO: Kaspar von Grünberg
Registered at Amtsgericht Charlottenburg, Berlin: HRB 196818 B
VAT-ID according to §27a UStG: DE318212407
Responsible for the content of humanitec.com ref. § 55 II RStV: Kaspar von Grünberg