HashiCorp

The 6 pillars of platform engineering

Workflows and checklists for building developer-centric platforms

WHITE PAPER

Contents

Platform engineering is about developer experience

Platform engineering is the discipline of designing and building toolchains and workflows that enable self-service capabilities for software engineering teams. These tools and workflows comprise an internal developer platform, often referred to as just "a platform." The goal of a platform team is to increase developer productivity, facilitate more frequent releases, improve application stability, lower security and compliance risks, and reduce costs.

HashiCorp has supported many organizations as they scale their cloud operating model through platform teams, and in order for these teams to meet their goals they must provide a satisfying developer experience. We have observed two common themes among companies that deliver great developer experiences:

1. **Standardizing on a set of infrastructure services to reduce friction for developers and operations teams:** This empowers a small, centralized group of platform engineers with the right tools to improve the developer experience across the entire organization, with APIs, documentation, and advocacy. The goal is to reduce tooling and process fragmentation, resulting in greater core stability for your software delivery systems and environments.
2. **A platform-as-a-product practice:** Heritage IT projects typically have a finite start and end date. That's not the case with an internal developer platform. It is never truly finished. Ongoing tasks include backlog management, regular feature releases, and roadmap updates to stakeholders. Think in terms of iterative agile development, not big upfront planning like waterfall development.

No platform should be designed in a vacuum. A platform is effective only if developers want to use it. Building and maintaining a platform involves continuous conversations and buy-in from developers (the platform team's customers) and business stakeholders. This guide is intended as a starting point for those conversations by helping platform teams organize their product around six technical elements, or "pillars", of the software delivery process along with the general requirements and workflow for each one.

The 6 pillars of platforms

What are the specific building blocks of a platform strategy? In working with customers in a wide variety of industries, HashiCorp has identified six foundational pillars that comprise the majority of platforms:

1. Security
2. Pipeline (VCS, CI/CD)
3. Provisioning
4. Connectivity
5. Orchestration
6. Observability

The following sections outline the definitions, workflows, requirements, dependencies, and considerations involved in implementing each of the six pillars.

Platform pillar 1: Security

The first things a developer asks when they start using any system is: "How do I create an account? Where do I set up credentials? How do I get an API key?" Even though version control, continuous integration, and infrastructure provisioning are fundamental to getting a platform up and running, in order to promote a secure-by-default platform experience from the outset.

Historically, many organizations invested in network perimeter-based security, often described as a "castle-and-moat" security approach. As infrastructure becomes increasingly dynamic, however, perimeters become fuzzy and challenging to control without impeding developer velocity.

In response, leading companies are choosing to adopt identity-based security, identity-brokering solutions, and modern security workflows, including centralized management of credentials and encryption methodologies. This promotes visibility and consistent auditing practices while reducing operational overhead in an otherwise fragmented solution portfolio.

Leading companies have also adopted "shift left" security; implementing security controls throughout the software development lifecycle, leading to earlier detection and remediation of potential attack vectors and increased vigilance around audit and regulatory control implementations. This approach demands automation-by-default instead of ad hoc enforcement.

Enabling this kind of DevSecOps mindset requires tooling decisions that support modern identity-driven security. There also needs to be an "as code" implementation paradigm to avoid ascribing and authorizing identity based on ticket-driven processes. That paves the way for traditional privileged access management (PAM) practices to embrace modern methodologies like just in time (JIT) access and zero trust security.

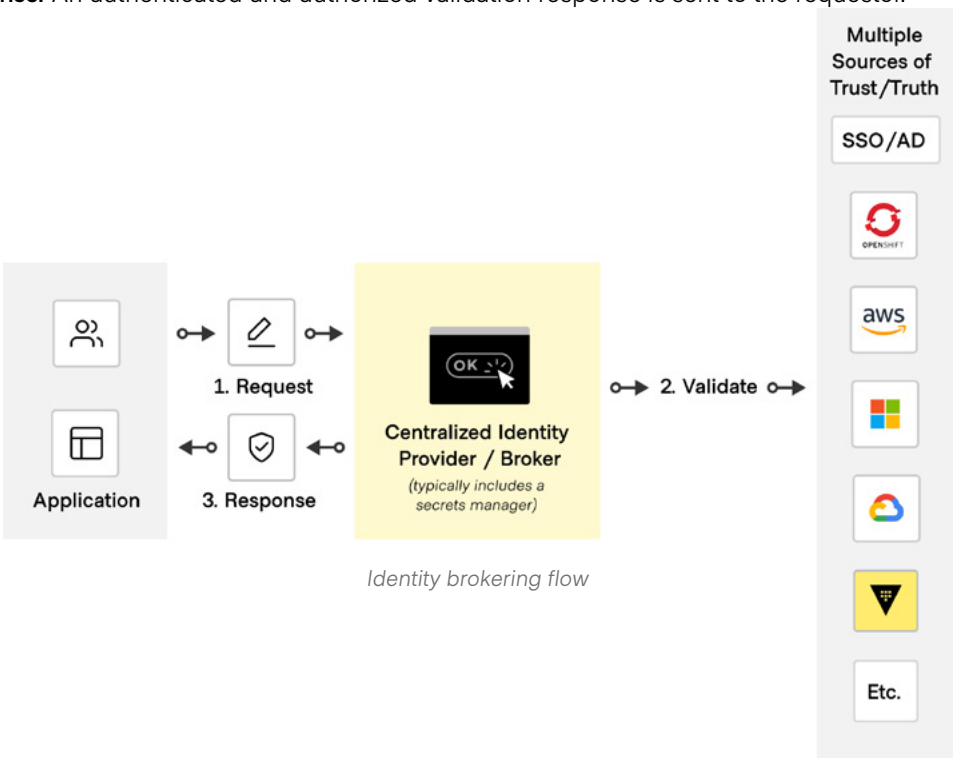
Identity brokering

In a cloud operating model approach, humans, applications, and services all present an identity that can be authenticated and validated against a central, canonical source. A multi-tenant secrets management and encryption platform along with an identity provider (IdP) can serve as your organization's identity brokers.

Workflow: Identity brokering

In practice, a typical identity brokering workflow might look something like this:

- **Request:** A human, application, or service initiates interaction via a request.
- **Validate:** One (or more) identity providers validate the provided identity against one (or more) sources of truth/trust.
- **Response:** An authenticated and authorized validation response is sent to the requestor.



Identity brokering flow

Identity brokering requirements checklist

Successful identity brokering has a number of prerequisites:

- ☐ All humans, applications, and services must have a well-defined form of identity.
- ☐ Identities can be validated against a trusted IdP.
- ☐ Identity systems must be interoperable across multi-runtime and multi-cloud platforms.
- ☐ Identity systems should be centralized or have limited segmentation to simplify audit and operational management across environments.
- ☐ Identity and access management (IAM) controls are established for each IdP.
- ☐ Clients (humans, machines, and services) must present a valid identity for AuthN and AuthZ).
- ☐ Once verified, access is brokered through deny-by-default policies to minimize impact in the event of a breach.
- ☐ AuthZ review is integrated into the audit process and, ideally, is granted just in time.
    - ☐ Audit trails are routinely reviewed to identify excessively broad or unutilized privileges and are retroactively analyzed following threat detection.
    - ☐ Historical audit data provides non-repudiation and compliance for data storage requirements.
- ☐ Fragmentation is minimized with a flexible identity brokering system supporting heterogeneous runtimes, including:
    - ☐ Platforms (VMware, Microsoft Azure VMs, Kubernetes/OpenShift, etc.)
    - ☐ Clients (developers, operators, applications, scripts, etc.)
    - ☐ Services (MySQL, MSSQL, Active Directory, LDAP, PKI, etc.)
- ☐ Enterprise support 24/7/365 via a service level agreement (SLA)
- ☐ Configured through automation (infrastructure as code, runbooks)

Access management: Secrets management and encryption

Once identity has been established, clients expect consistent and secure mechanisms to perform the following operations:

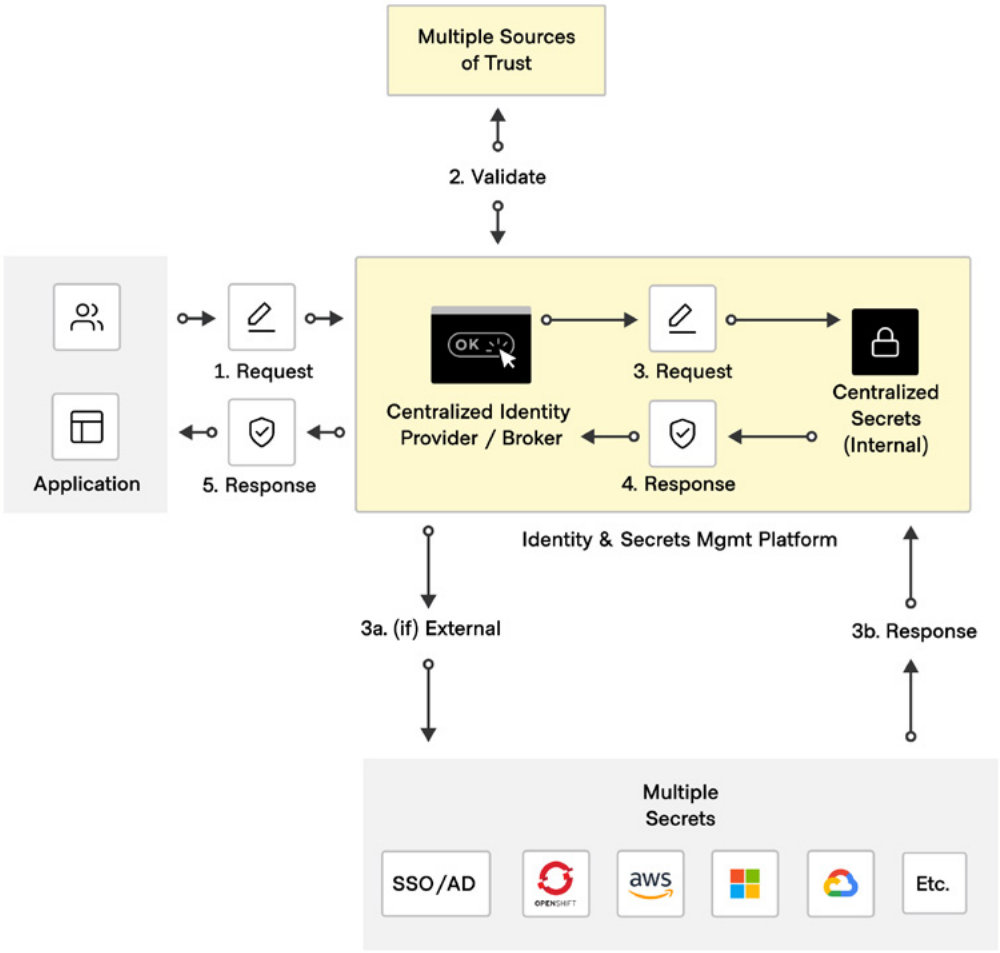
- Retrieving a secret (a credential, password, key, etc.)
- Brokering access to a secure target
- Managing secure data (encryption, decryption, hashing, masking, etc.)

These mechanisms should be automatable — requiring as little human intervention as possible after setup — and promote compliant practices. They should also be extensible to ensure future tooling is compatible with these systems.

Workflow: Secrets management and encryption

A typical secrets management workflow should follow five key steps:

1. **Request:** A client (human, application, or service) requests a secret.
2. **Validate:** The request is validated against an IDP.
3. **Request:** A secret request is served if managed by the requested platform. Alternatively:
   - The platform requests a temporary credential from a third party.
   - The third-party system responds to the brokered request with a short-lived secret.
4. **Broker response:** The initial response passes through an IAM cryptographic barrier for offload or caching.
5. **Client response:** The final response is provided back to the requestor.



Secrets management flow

Access management: Secure remote access (human to machine)

Human-to-machine access in the traditional castle-and-moat model has always been inefficient. The workflow requires multiple identities, planned intervention for AuthN and AuthZ controls, lifecycle planning for secrets, and complex network segmentation planning, which creates a lot of overhead.

While PAM solutions have evolved over the last decade to provide delegated solutions like dynamic SSH key generation, this does not satisfy the broader set of ecosystem requirements, including multi-runtime auditability or cross-platform identity management. Introducing cloud architecture patterns such as ephemeral resources, heterogeneous cloud networking topologies, and JIT identity management further complicates the task for legacy solutions.

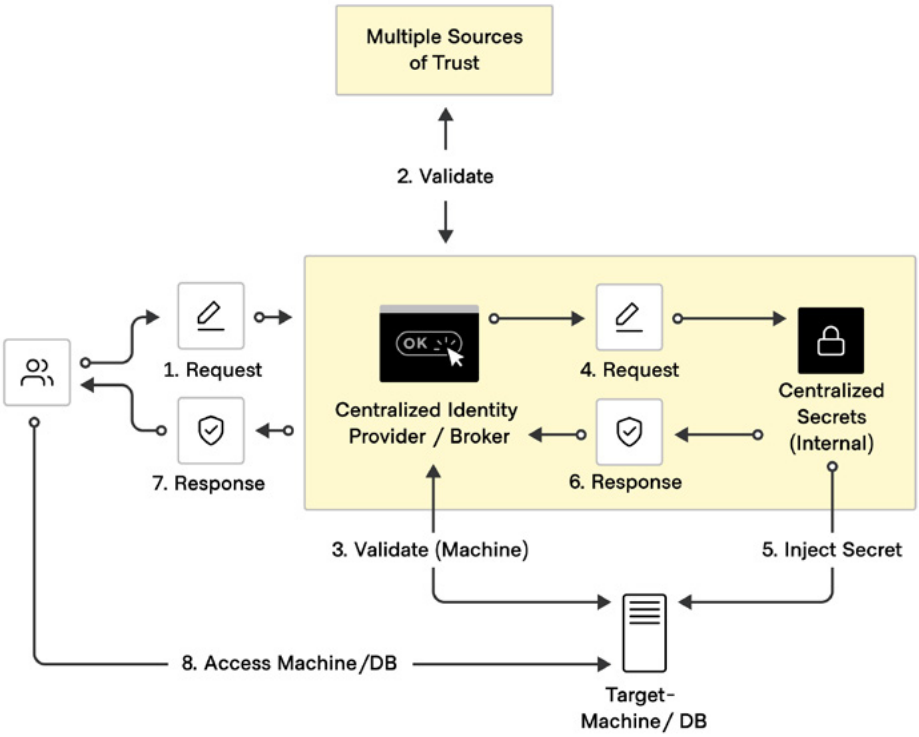
A modern solution for remote access addresses the challenges of ephemeral resources and the complexities that arise with ephemeral resources such as dynamic resource registration, identity, access, and secrets. These modern secure remote access tools no longer rely on network access such as VPNs as an initial entry point, CMDBs, bastion hosts, manual SSH, and/or secrets managers with check in/out workflows.

Enterprise-level secure remote access tools use a zero trust model where human users and resources have identity. Users connect directly to these resources. Scoped roles — via dynamic resource registries, controllers, and secrets — are automatically injected into resources, eliminating many manual processes and security risks such as broad, direct network access and long-lived secrets.

Workflow: Secure remote access (human to machine)

A modern remote infrastructure access workflow for a human user typically follows these eight steps:

1. **Request:** A user requests system access.
2. **Validate (human):** Identity is validated against the trusted identity broker.
3. **Validate (to machine):** Once authenticated, authorization is validated for the target system.
4. **Request:** The platform requests a secret (static or short-lived) for the target system.
5. **Inject secret:** The platform injects the secret into the target resource.
6. **Broker response:** The platform returns a response to the identity broker.
7. **Client response:** The platform grants access to the end user.
8. **Access machine/database:** The user securely accesses the target resource via a modern secure remote access tool.



Secure remote access flow

Access management requirements checklist

All secrets in a secrets management system should be:

- ☐ Centralized
- ☐ Encrypted in transit and at rest
- ☐ Limited in scoped role and access policy
- ☐ Dynamically generated, when possible
- ☐ Time bound (i.e. defined time-to-live — TTL)
- ☐ Fully auditable

Secrets management solutions should:

- ☐ Support multi-runtime, multi-cloud, and hybrid-cloud deployments
- ☐ Provide flexible integration options
- ☐ Include a diverse partner ecosystem
- ☐ Embrace zero-touch automation practices (API-driven)
- ☐ Empower developers and delegate implementation decisions within scoped boundaries

- ☐ Be well-documented and commonly used across industries
- ☐ Be accompanied by enterprise support 24/7/365 based on an SLA
- ☐ Support automated configuration (infrastructure as code, runbooks)

Additionally, systems implementing secure remote access practices should:

- ☐ Dynamically register service catalogs
- ☐ Implement an identity-based model
- ☐ Provide multiple forms of authentication capabilities from trusted sources
- ☐ Be configurable as code
- ☐ Be API-enabled and contain internal and/or external workflow capabilities for review and approval processes
- ☐ Enable secrets injection into resources
- ☐ Provide detailed role-based access controls (RBACs)
- ☐ Provide capabilities to record actions, commands, sessions, and give a full audit trail
- ☐ Be highly available, multi-platform, multi-cloud capable for distributed operations, and resilient to operational impact

Security pillar: HashiCorp solutions

HashiCorp Vault Enterprise or **HCP Vault**, and **HCP Boundary**: Many organizations have adopted these commercial versions of the community editions (HashiCorp Vault and HashiCorp Boundary), as industry standards for extensible identity brokering, and centralized secrets management, encryption as a service, and secure remote access management.

References:

- HashiCorp Vault use cases (Secrets management, Kubernetes secrets, database credential rotation, automated PKI, data encryption and tokenization, key management)
- HashiCorp Boundary use cases (Zero trust, multi-cloud, SSO remote access management and monitoring)

Platform pillar 2: pipeline (VCS, CI/CD)

One of the first steps in any platform team's journey is integrating with and potentially re-architecting the software delivery pipeline. That means taking a detailed look at your organization's version control systems (VCS) and continuous integration/continuous deployment (CI/CD) pipelines.

Many organizations have multiple VCS and CI/CD solutions in different maturity phases. These platforms also evolve over time, so a component-based API platform or catalog model is recommended to support future extensibility without compromising functionality or demanding regular refactoring.

In a cloud-native model, infrastructure and configuration are managed as code, and therefore a VCS is required for this core function. Using a VCS and managing code provide the following benefits:

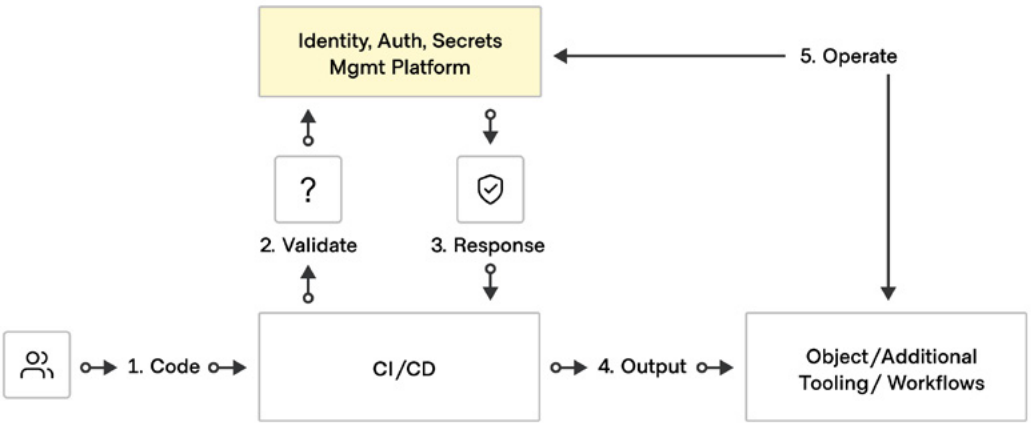
- Consistency and standardization
- Agility and speed
- Scalability and flexibility
- Configuration as documentation
- Reusability and sharing
- Disaster recovery and reproducibility
- Debuggability and auditability
- Compliance and security

VCS and CI/CD enable interaction and workflows across multiple infrastructure systems and platforms, which requires careful assessment of all the VCS and CI/CD requirements listed below.

Workflow: VCS and CI/CD

A typical VCS and CI/CD workflow follows five steps:

1. **Code:** The developer commits code to the VCS and a task is automatically submitted to the pipeline.
2. **Validate:** The CI/CD platform submits a request to your IdP for validation (AuthN and AuthZ).
3. **Response:** If successful, the pipeline triggers tasks (e.g. test, build, deploy).
4. **Output:** The output and/or artifacts are shared within platform components or with external systems for further processing.
5. **Operate:** Security systems may be involved in post-run tasks, such as deprovisioning access



VCS and CI/CD pipeline flow

VCS and CI/CD requirements checklist

Successful VCS and CI/CD solutions should deliver:

- ☐ A developer experience tailored to your team's needs and modern efficiencies
- ☐ Easy onboarding
- ☐ A gentle learning curve with limited supplementary training needed (leveraging industry-standard tools)
- ☐ Complete and accessible documentation
- ☐ Support for pipeline as code
- ☐ Platform agnosticism (API-driven)
- ☐ Embedded expected security controls (RBAC, auditing, etc.)
- ☐ Support for automated configuration (infrastructure as code, runbooks)
- ☐ Support for secrets management, identity, and authorization platform integration
- ☐ Encouragement and support for a large partner ecosystem with a broad set of enterprise technology integrations
- ☐ Extended service footprint, with runners to delegate and isolate span of control
- ☐ Enterprise support based on an SLA (e.g. 24/7/365)

Note: VCS and CI/CD systems may have more-specific requirements not listed here.

As platform teams select and evolve their VCS and CI/CD solutions, they need to consider what this transformation means for existing/legacy provisioning practices, security, and compliance. Teams should assume that building new platforms will impact existing practices, and they should work to identify, collaborate, and coordinate change within the business.

Platform teams should also be forward-looking. VCS and CI/CD platforms are rapidly evolving to further abstract away the complexity of the CI/CD process from developers. HashiCorp looks to simplify these workflows for developers by providing a consistent way to deploy, manage, and observe applications across multiple runtimes, including Kubernetes and serverless environments with HashiCorp Waypoint.

Pipeline pillar: HashiCorp and partner (VCS and CI/CD) solutions

VCS: GitHub, GitLab, BitBucket

CI/CD: Jenkins, CircleCI, GitHub Actions

- These solutions are among the most popular in their respective categories. HashiCorp does not provide VCS or CI/CD solutions as there have been well-established leading tools in these spaces for more than a decade.
- What HashiCorp has innovated on is a new control layer above each platform pillar, designed for platform engineers to truly unify the Platform-as-a-Service (PaaS) experience (for Kubernetes, Amazon ECS, and other platforms) into one workflow. This layer is built with a tool called HashiCorp Waypoint, which gives developers a consistent workflow to build, deploy, and release applications across any platform. Waypoint enables developers to get their applications from development to production in a single file and deploy using a single command: `waypoint up`.

References:

- Integrate Terraform with existing CI/CD workflows
- HashiCorp Waypoint
- Find a HashiCorp technology partner

Platform pillar 3: Provisioning

In the first two pillars, a platform team provides self-service VCS and CI/CD pipeline workflows with security workflows baked in to act as guardrails from the outset. These are the first steps for software delivery. But once you have application code to run, the next question is where will you run it?

Every IT organization needs an infrastructure plan at the foundation of its applications, and platform teams need to treat that plan as the foundation of their initiatives. Their first goal is to eliminate ticket-driven workflows for infrastructure provisioning, which aren't scalable in modern IT environments. Platform teams typically achieve this goal by providing a standardized shared infrastructure provisioning service with curated self-service workflows, tools, and templates for developers. Then they connect those workflows with the workflows of the first two pillars.

Building an effective, modern infrastructure platform hinges on the adoption of infrastructure as code. When infrastructure configurations and automation are codified, even the most complex provisioning scenarios can be automated. The infrastructure code can then be version controlled for easy auditing, iteration, and collaboration. There are a few solutions for adopting infrastructure as code, but the most common is HashiCorp Terraform: a provisioning product that is more widely used than competing tools by a wide margin.

Terraform is the most popular choice for organizations adopting infrastructure as code because of its large integration ecosystem. This ecosystem helps platform engineers meet the final major requirement for a provisioning platform: Extensibility. An extensive plugin ecosystem helps platform engineers quickly adopt new technologies and services that developers want to deploy, without having to write custom code.

Provisioning: Modules and images

Building standardized infrastructure workflows requires platform teams to break down their infrastructure into reusable, and ideally immutable, components. Immutable infrastructure is a common standard among modern IT that reduces complexity and simplifies troubleshooting while also improving reliability and security.

Immutability means deleting and re-provisioning infrastructure for all changes, which minimizes server patching and configuration changes, helping ensure that every service iteration initiates a new, tested, and up-to-date instance. It also forces runbook validation and promotes regular testing of failover and canary deployment exercises. Many organizations put immutability into practice by using Terraform, or another provisioning tool, to build and rebuild large swaths of infrastructure by modifying configuration code. Some also build golden image pipelines, which focus on building and continuous deployment of

repeatable machine images that are tested and confirmed for security and policy compliance (golden images).

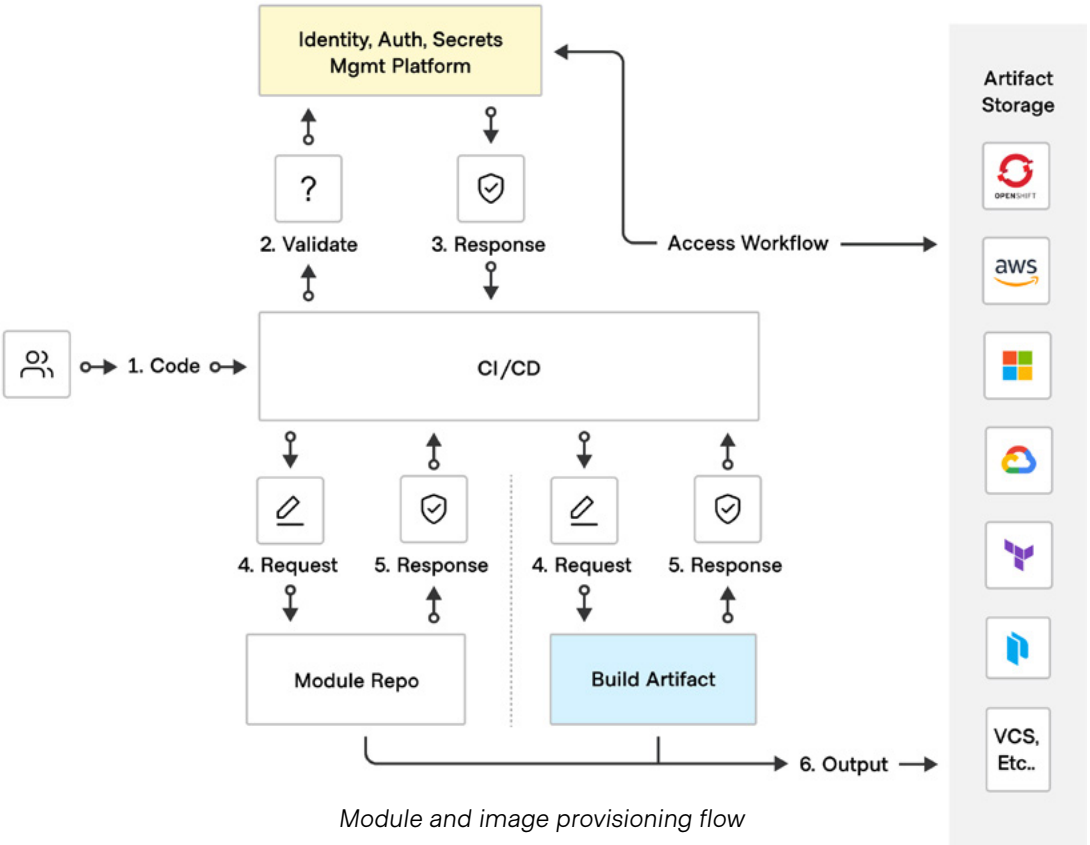
Along with machine images, modern IT organizations are modularizing their infrastructure code to compose commonly used components into reusable modules. This is important because a core principle of software development is the concept of not "reinventing the wheel," and it applies to infrastructure code as well. Modules create lightweight abstractions to describe infrastructure in terms of architectural principles rather than discrete objects. They are typically managed through version control and interact with third-party systems, such as a service catalog or testing framework.

High-performing IT teams bring together golden image pipelines and their own registry of modules for developers to use when building infrastructure for their applications. With little knowledge required about the inner workings of this infrastructure and its setup, developers can use infrastructure modules and golden image pipelines in a repeatable, scalable, and predictable workflow that has security and company best practices built-in on the very first deployment.

Workflow: Provisioning modules and images

A typical provisioning workflow will follow these six steps:

1. **Code:** A developer commits code and submits a task to the pipeline.
2. **Validate:** The CI/CD platform submits a request to your IdP for validation (AuthN and AuthZ).
3. **IdP response:** If successful, the pipeline triggers tasks (e.g. test, build, deploy).
4. **Request:** CI/CD automated workflow to build modules, artifacts, images, and/or other infrastructure components.
5. **Response:** The response (success/failure and metadata) is passed to the CI/CD platform.
6. **Output:** The infrastructure components such as modules, artifacts, and image configurations are deployed or stored.



Module and image provisioning flow

Provisioning: Policy as code

Agile development practices have shifted the focus of infrastructure provisioning from an operations problem to an application-delivery expectation. Infrastructure provisioning is now a gating factor for business success. Its value is aligned around driving organizational strategy and the customer mission, not purely based on controlling operational expenditures.

In shifting to an application-delivery expectation, we need to shift workflows and processes. Historically, operations personnel applied workflows and complaints to the provisioning process by leveraging tickets. These tickets usually involved validating access, approvals, security, costs, etc. The whole process was audited for compliance and control practices.

This process now must change to enable developers and other platform end users to provision via a self-service workflow. This means that a new set of codified security controls and guardrails must be implemented to satisfy compliance and control practices.

Within cloud native systems, these controls are implemented via policy as code. Policy as code is a practice that uses programmable rules and conditions for software and infrastructure deployment that codify best practices, compliance requirements, security rules, and cost controls.

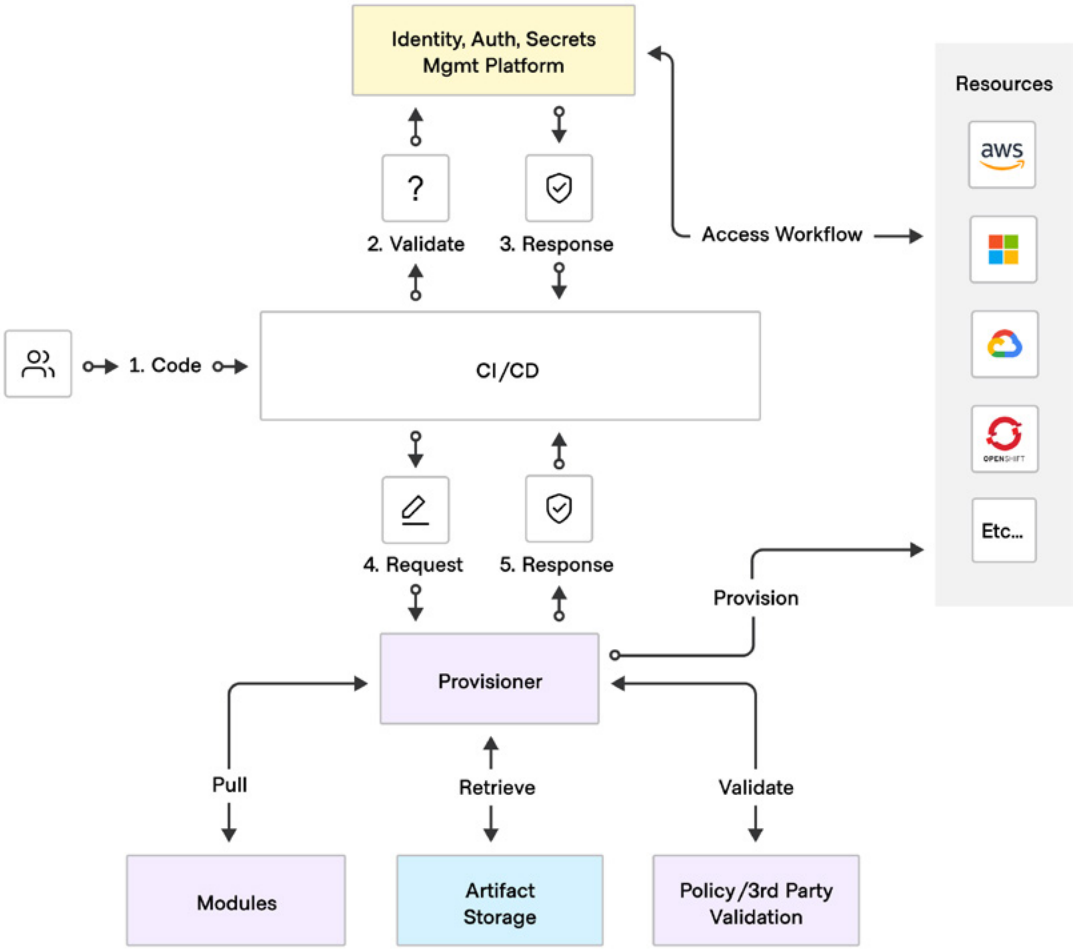
Some tools and systems include their own policy system, but there are also higher-level policy engines that integrate with multiple systems. The fundamental requirement is that these policy systems can be managed as code and will provide evaluations, controls, automation, and feedback loops to humans and systems within the workflows.

Implementing policy as code helps shift workflows "left" by providing feedback to users earlier in the provisioning process and enabling them to make better decisions faster. But before they can be used, these policies need to be written. Platform teams should own the policy as code practice, working with security, compliance, audit, and infrastructure teams to ensure that policies are mapped properly to risks and controls.

Workflow: Policy as code

Implementing policy as code checks in an infrastructure provisioning workflow typically involves five steps:

1. **Code:** The developer commits code and submits a task to the pipeline.
2. **Validate:** The CI/CD platform submits a request to your IdP for validation (AuthN and AuthZ).
3. **IdP response:** If successful, the pipeline triggers tasks (e.g. test, build, deploy).
4. **Request:** The provisioner runs the planned change through a policy engine and the request is either allowed to go through (sometimes with warnings) or rejected if the code doesn't pass policy tests.
5. **Response:** A metadata response packet is sent to CI/CD and to external systems from there, such as security scanning or integration testing.



Provisioning flow with policy as code

Provisioning requirements checklist

Successful self-service provisioning of infrastructure requires:

- ☐ A consolidated control and data plane for end-to-end automation
- ☐ Automated configuration (infrastructure as code, runbooks)
- ☐ Pre-defined and fully configurable workflows
- ☐ Native integrations with VCS and CI/CD tools
- ☐ Support for a variety of container and virtual machine images required by the business
- ☐ Multiple interfaces for different personas and workflows (GUI, API, CLI, SDK)

- ☐ Use of a widely adopted infrastructure as code language — declarative language strongly recommended
- ☐ Compatibility with industry standard testing and security frameworks, data management (encryption), and secrets management tools
- ☐ Integration with common workflow components such as notification tooling and webhooks
- ☐ Support for codified guardrails, including:
  - ☐ Policy as code: Built-in policy as code engine with extensible integrations
  - ☐ RBAC: Granularly scoped permissions to implement the principle of least privilege
  - ☐ Token-based access credentials to authenticate automated workflows
  - ☐ Prescribed usage of organizationally approved patterns and modules
- ☐ Integration with trusted identity providers with single sign-on and RBAC
- ☐ Maintenance of resource provisioning metadata (state, images, resources, etc.):
  - ☐ Controlled via deny-by-default RBAC
  - ☐ Encrypted
  - ☐ Accessible to humans and/or machines via programmable interfaces
  - ☐ Stored with logical isolation maintained via traceable configuration
- ☐ Scalability across large, distributed teams
- ☐ Support for both public and private modules
- ☐ Full audit logging and log streaming capabilities
- ☐ Financial Operations (FinOps) workflows to enforce cost-based policies and optimization
- ☐ Well-defined documentation and developer enablement
- ☐ Enterprise support based on an SLA (e.g. 24/7/365)

Provisioning pillar: HashiCorp solutions

HashiCorp Terraform: The industry standard for infrastructure provisioning. Many organizations have adopted Terraform Cloud or Terraform Enterprise to provide a centralized provisioning workflow and guardrails.

HashiCorp Packer: The de facto standard solution for building golden images. HashiCorp Cloud Platform (HCP) Packer provides enhanced functionality for image metadata management, automated compliance enforcement, and global image querying. HCP is HashiCorp's managed cloud service.

References:

- Infrastructure as code
- Terraform modules as building blocks for infrastructure
- Policy as code: Sentinel
- Multi-cloud provisioning
- Multi-cloud compliance and management
- Drift detection
- Golden image pipeline

Platform pillar 4: Connectivity

Networking connectivity is a hugely under-discussed pillar of platform engineering, with many legacy patterns and hardware still in use at many enterprises. It needs careful consideration and strategies right alongside the provisioning pillar, since connectivity is what allows apps to exchange data and is part of both the infrastructure and application architectures.

Traditionally, ticket-driven processes were expected to support routine tasks like creating DNS entries, opening firewall ports or network ACLs, and updating traffic routing rules. This caused (and still causes in some enterprises) days-to-weeks-long delays in simple application delivery tasks, even when the preceding infrastructure management is fully automated. In addition, these simple updates are often manual, error-prone, and not conducive to dynamic, highly fluctuating cloud environments. Without automation, connectivity definitions and IP addresses quickly become stale as infrastructure is rotated at an increasingly rapid pace.

To adapt networking to modern dynamic environments, platform teams are bringing networking functions, software, and appliances into their infrastructure as code configurations. This brings the automated speed, reliability, and version-controlled traceability benefits of infrastructure as code to networking.

When organizations adopt microservices architectures, they quickly realize the value of software-driven service discovery and service mesh solutions. These solutions create architectures where services are discovered and automatically connected based on centralized policies in a zero trust network — if they have permissions, otherwise the secure default is to deny service-to-service connections. In this model, service-based identity is critical to ensuring strict adherence to common security frameworks.

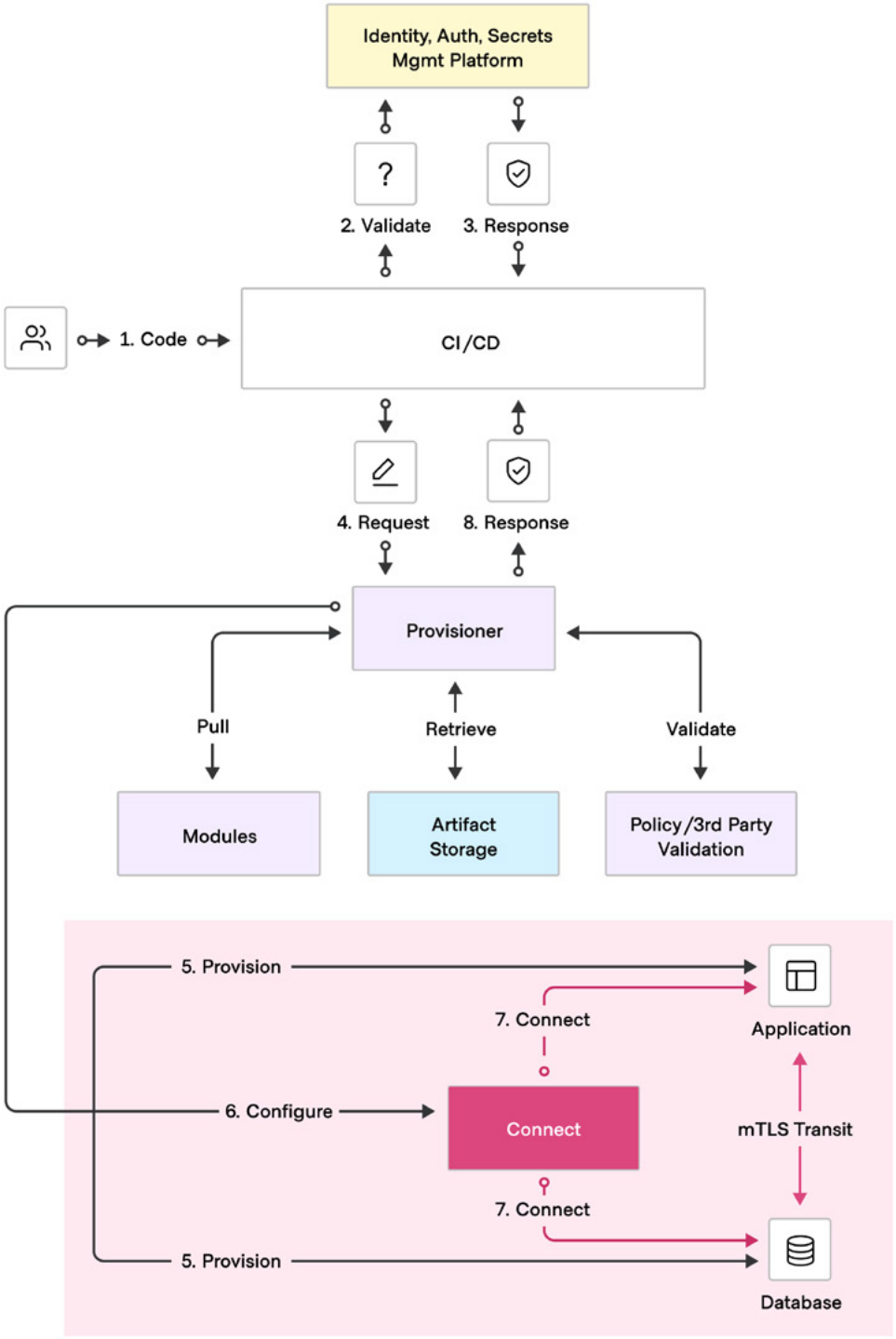
An organization's choice for its central shared registry should be multi-cloud, multi-region, and multi-runtime — meaning it can connect a variety of cluster types, including VMs, bare metal, serverless, or Kubernetes. Teams need to minimize the need for traditional networking ingress or egress points that bring their environments back toward an obsolete "castle-and-moat" network perimeter approach to security.

Workflow: Connectivity

A typical network connectivity workflow should follow these eight steps:

1. **Code:** The developer commits code.
    - **Note:** Developers may have direct network control plane access depending on the RBACs assigned to them.

2. **Validate:** The CI/CD platform submits a request to the IdP for validation (AuthN and AuthZ).
3. **IdP response:** If successful, the pipeline triggers tasks (e.g. test, build, deploy).
4. **Request:** The provisioner executes requested patterns, such as building modules, retrieving artifacts, or validating policy against internal and external engines, ultimately provisioning defined resources.
5. **Provision:** Infrastructure is provisioned and configured, if not already available.
6. **Configure:** The provisioner configures the connectivity platform.
7. **Connect:** Target systems are updated based on defined policies.
8. **Response:** A metadata response packet is sent to CI/CD and to external systems that perform actions such as security scanning or integration testing.



Connectivity flow (the Connect box includes service mesh and service registry)

Connectivity requirements checklist

Successful network connectivity automation requires:

- ☐ A centralized shared registry to discover, connect, and secure services across any region, runtime platform, and cloud service provider
- ☐ Support for multiple interfaces for different personas and workflows (GUI, API, CLI, SDK)
- ☐ Health checks
- ☐ Multiple segmentation and isolation models
- ☐ Layer 4 and Layer 7 traffic management
- ☐ Implementation of security best practices such as defense-in-depth and deny-by-default
- ☐ Integration with trusted identity providers with single sign-on and delegated RBAC
- ☐ Audit logging
- ☐ Enterprise support based on an SLA (e.g. 24/7/365)
- ☐ Support for automated configuration (infrastructure as code, runbooks)

Connectivity requirements: HashiCorp solutions

HashiCorp Consul provides advanced service-based networking capabilities, supporting common use cases like service discovery and service mesh. Consul solves multi-platform application connectivity challenges to bridge workloads across heterogeneous environments (private and public clouds) and runtimes (mainframes, microservices, traditional VMs, or bare-metal infrastructure).

References:

- Network infrastructure automation
- Application networking
- What is service discovery
- What is a service mesh?
- HashiCorp Consul documentation

Platform pillar 5: Orchestration

When it comes time to deploy your application workload, if you're working with distributed applications, microservices, or generally wanting resilience across cloud infrastructure, it's going to be much easier using a workload orchestrator.

Workload orchestrators such as Kubernetes and HashiCorp Nomad provide a multitude of benefits over traditional technologies. The level of effort may vary to achieve these benefits. For example, rearchitecting for containerization to adopt Kubernetes may involve a higher degree of effort than using an orchestrator like HashiCorp Nomad that is oriented more toward supporting a variety of workload types. In either case, workload orchestrators enable:

- Improved resource utilization
- Scalability and elasticity
- Multi-cloud and hybrid cloud support
- Developer self-service
- Service discovery and networking (built-in or pluggable)
- High availability and fault tolerance
- Advanced scheduling and placement control
- Resource isolation and security
- Cost optimization

Orchestrators provide optimization algorithms to determine the most efficient way to allocate workloads into your infrastructure resources (e.g. bin-packing, spread, affinity, anti-affinity, autoscaling, dynamic application sizing, etc.), which can lower costs. They automate distributed computing and resilience strategies without developers having to know much about how it works under the hood.

As with the other platform pillars, the main goal is to standardize workflows, and an orchestrator is a common way modern platform teams are unifying deployment workflows to eliminate ticket-driven processes.

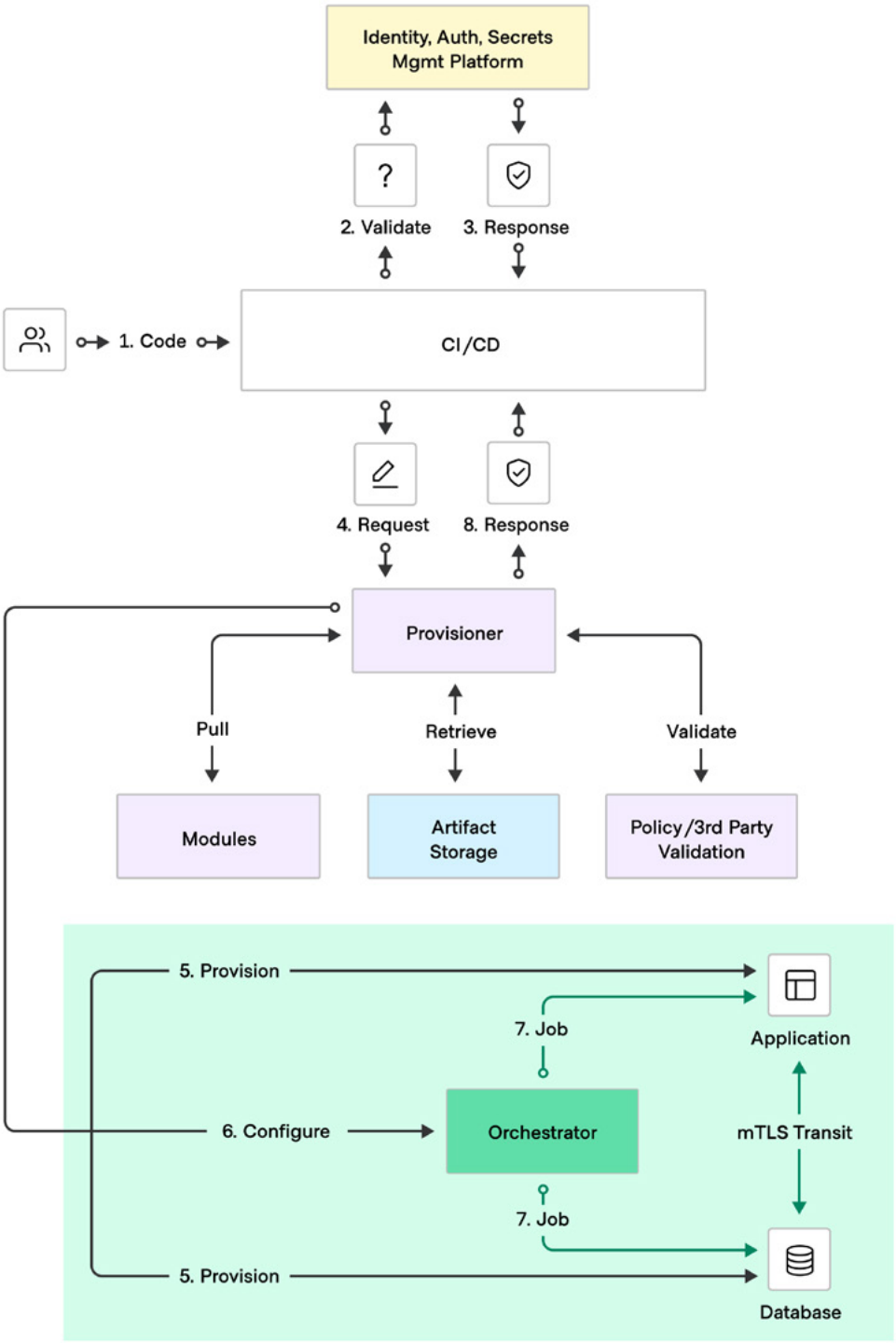
When choosing an orchestrator, it's important to make sure it's flexible enough to handle future additions to your environments and heterogeneous workflows. It's also crucial that the orchestrator can handle multi-tenancy and easily federate across multiple on-premises datacenters and multi-cloud environments.

It is important to note that not all systems — such as vendor-provided monolithic appliances or applications — can be containerized, or shifted to a modern orchestrator, so it is important for platform teams to identify opportunities for other teams to optimize engagement and automation for orchestrators as per the other platform pillars. Modern orchestrators provide a broad array of native features. While specific implementations and functionality vary across systems, there are a number of core requirements.

Workflow: Orchestration

A typical orchestration workflow should follow these eight steps:

1. **Code:** A developer commits code.
    - **Note:** Developers may have direct network control plane access depending on the RBACs assigned to them.
2. **Validate:** The CI/CD platform submits a request to the IdP for validation (AuthN and AuthZ).
3. **IdP response:** If successful, the pipeline triggers common tasks (test, build, deploy).
4. **Request:** The provisioner executes requested patterns, such as building modules, retrieving artifacts, or validating policy against internal and external engines, ultimately provisioning defined resources.
5. **Provision:** Infrastructure is provisioned and configured, if not already available.
6. **Configure:** The provisioner configures the orchestrator resource.
7. **Job:** The orchestrator runs jobs on target resources based on defined tasks and policies.
8. **Response:** Completion of the provisioner request is provided to the CI/CD platform for subsequent processing and/or handoff to external systems that perform actions such as security scanning or integration testing.



Orchestration flow

Orchestration requirements checklist

Successful orchestration requires:

- ☐ Service/batch schedulers
- ☐ Flexible task drivers
- ☐ Pluggable device interfaces
- ☐ Flexible upgrade and release strategies
- ☐ Federated deployment topologies
- ☐ Resilient, highly available deployment topologies
- ☐ Autoscaling (dynamic and fixed)
- ☐ An access control system (IAM JWT/OIDC and ACLs)
- ☐ Support for multiple interfaces for different personas and workflows (GUI, API, CLI, SDK)
- ☐ Integration with trusted identity providers with single sign-on and delegated RBAC
- ☐ Functional, logical, and/or physical isolation of tasks
- ☐ Native quota systems
- ☐ Audit logging
- ☐ Enterprise support based on an SLA (e.g. 24/7/365)
- ☐ Configuration through automation (infrastructure as code, runbooks)

Orchestration requirements: HashiCorp solutions

HashiCorp Nomad is a lightweight and robust orchestrator/scheduler that can be used in multiple runtime environments. Check out these resources to learn more about what Nomad provides, its architecture, its UX, and how it helps operators at various companies.

References:

- A Kubernetes user's guide to HashiCorp Nomad
- A Kubernetes user's guide to HashiCorp Nomad secret management
- Nomad as a supplement to Kubernetes
- Nomad as an alternative to Kubernetes
- Workload orchestration

Platform pillar 6: Observability

The last leg of any platform workflow is the monitoring and maintenance of your deployments. You want to build observability practices and automation into your platform, measuring the quality and performance of software, services, platforms, and products to understand how systems are behaving. Good system observability makes investigating and diagnosing problems faster and easier.

Fundamentally, observability is about recording, organizing, and visualizing data. The mere availability of data doesn't deliver enterprise-grade observability. Site reliability engineering, DevOps, or other teams first determine what data to generate, collect, aggregate, summarize, and analyze to gain meaningful and actionable insights.

Then those teams adopt and build observability solutions. Observability solutions use metrics, traces, and logs as data types to understand and debug systems. Enterprises need unified observability across the entire stack: cloud infrastructure, runtime orchestration platforms such as Kubernetes or Nomad, cloud-managed services such as Azure Managed Databases, and business applications. This unification helps teams understand the interdependencies of cloud services and components.

But unification is only the first step of baking observability into the platform workflow. Within that workflow, a platform team needs to automate the best practices of observability within modules and deployment templates. Just as platform engineering helps security functions shift left, observability integrations and automation should also shift left into the infrastructure coding and application build phases by baking observability into containers and images at deployment. This helps your teams build and implement a comprehensive telemetry strategy that's automated into platform workflows from the outset.

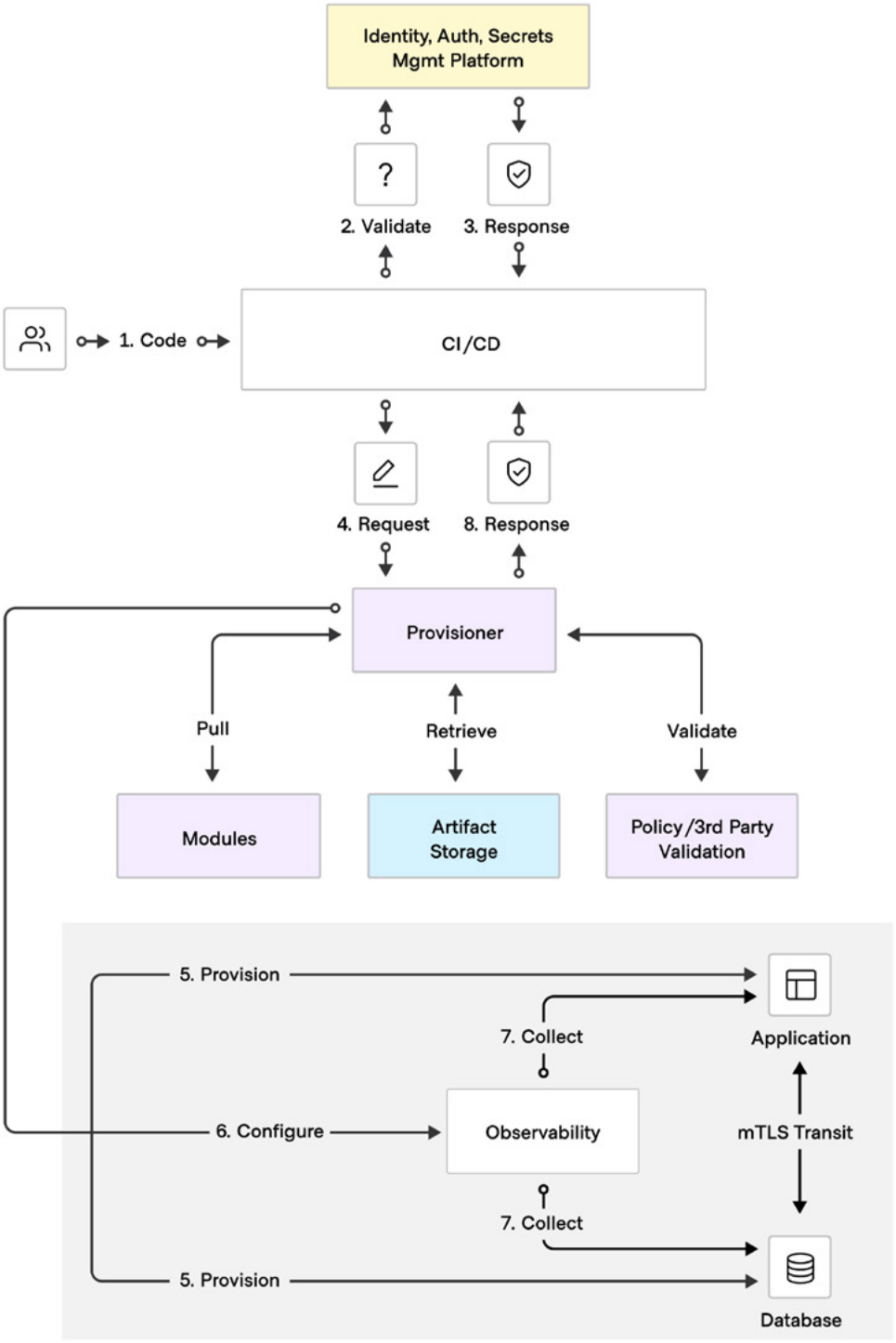
The benefits of integrating observability solutions in your infrastructure code are numerous: Developers can better understand how their systems operate and the reliability of their applications. Teams can quickly debug issues and trace them back to their root cause. And the organization can make data-driven decisions to improve the system, optimize performance, and enhance the user experience.

Workflow: Observability

An enterprise-level observability workflow might follow these eight steps:

1. **Code:** A developer commits code.
    - **Note:** Developers may have direct network control plane access depending on the RBACs assigned to them.

2. **Validate:** The CI/CD platform submits a request to the IdP for validation (AuthN and AuthZ).
3. **IdP response:** If successful, the pipeline triggers tasks (e.g. test, build, deploy).
4. **Request:** The provisioner executes requested patterns, such as building modules, retrieving artifacts, or validating policy against internal and external engines, ultimately provisioning defined resources.
5. **Provision:** Infrastructure is provisioned and configured, if not already available.
6. **Configure:** The provisioner configures the observability resource.
7. **Collect:** Metrics and tracing data are collected based on configured emitters and aggregators.
8. **Response:** Completion of the provisioner request is provided to the CI/CD platform for subsequent processing and/or handoff to external systems, for purposes such as security scanning or integration testing.



Observability flow

Observability requirements checklist

Enterprise-level observability requires:

- ☐ Real-time issue and anomaly detection
- ☐ Auto-discovery and integrations across different control planes and environments
- ☐ Accurate alerting, tracing, logging, and monitoring
- ☐ High-cardinality analytics
- ☐ Tagging, labeling, and data-model governance
- ☐ Observability as code
- ☐ Scalability and performance for multi-cloud and hybrid deployments
- ☐ Security, privacy, and RBACs for self-service visualization, configuration, and reporting

Observability requirements: HashiCorp partner solutions

- HashiCorp observability partner: Datadog
- HashiCorp observability partner: Splunk
- HashiCorp observability partner: New Relic
- HashiCorp observability partner: Honeycomb

Next steps and technology selection criteria

Platform building is never totally complete. It's not an upfront-planned project that's finished after everyone has signed off and started using it. It's more like an iterative agile development project rather than a traditional waterfall one.

You start with a minimum viable product (MVP), and then you have to market your platform to the organization. Show teams how they're going to benefit from adopting the platform's common patterns and best practices for the entire development lifecycle. It can be effective to conduct a process analysis (current vs. future state) with various teams to jointly work on and understand the benefits of adoption. Finally, it's essential to make onboarding as easy as possible.

As you start to check off the boxes for these six platform pillar requirements, platform teams will want to take on the mindset of a UX designer. Investigate the wants and needs of various teams, understanding that you'll probably be able to satisfy only 80% - 90% of use cases. Some workflows will be too delicate or unique to bring into the platform. You can't please everyone. Toolchain selection should be a cross-functional process, and executive sponsorship at the outset is necessary to drive adoption.

Key toolchain questions checklist:

- **Practitioner adoption:** Are you starting by asking what technologies your developers are excited about? What enables them to quickly support the business? What do they want to learn and is this skillset common in the market?
- **Scale:** Can this tool scale to meet enterprise expectations, for both performance, security/compliance, and ease of adoption? Can you learn from peer institutions instead of venturing into uncharted territory?
- **Support:** Are the selected solutions supported by organizations that can meet SLAs for core critical infrastructure (24/7/365) and satisfy your customers' availability expectations?
- **Longevity:** Are these solution suppliers financially strong and capable of supporting these pillars and core infrastructure long-term?
- **Developer flexibility:** Do these solutions provide flexible interfaces (GUI, CLI, API, SDK) to create a tailored user experience?
- **Documentation:** Do these solutions provide comprehensive, up-to-date documentation?
- **Ecosystem integration:** Are there extensible ecosystem integrations to neatly link to other tools in the chain, like security or data warehousing solutions?

For organizations that have already invested in some of these core pillars, the next step involves collaborating with ecosystem partners like HashiCorp to identify workflow enhancements and address coverage gaps with well-established solutions.

HashiCorp