

The cloud-native



adoption



playbook



From containerization
to optimization on
Red Hat OpenShift



Introduction 03

Chapter 5: 41
Stage 3—Modernize

Chapter 1: 04
The application
innovation platform

Chapter 6: 53
Stage 4—Optimize

Chapter 2: 11
Understanding cloud-native
adoption maturity

Chapter 7: 63
Building your platform
engineering practice

Chapter 3: 16
Stage 1—Containerize

Chapter 8: 69
Navigating your journey

Chapter 4: 29
Stage 2—Operationalize

Reference 77



Introduction

Every enterprise faces the same fundamental question: How do you deliver software faster without sacrificing security, reliability, or operational control?

The answer is cloud-native technologies. Containers, Kubernetes, microservices, and DevOps practices have moved from experimental to essential. But adopting these technologies is not as simple as installing a platform and declaring victory. The organizations that succeed with cloud-native adoption—those running thousands of applications in production, deploying multiple times per day, and scaling across hybrid environments—share a common trait. They followed a deliberate, staged approach to adoption.

This e-book distills the lessons learned from the most successful Red Hat® OpenShift® customers into a practical playbook for cloud-native adoption. It provides:

- ▶ **A maturity framework** based on real-world patterns observed across organizations running thousands of applications on Red Hat OpenShift. This framework helps you understand where your organization is today and what it takes to reach the next level.
- ▶ **An adoption roadmap** that moves your organization beyond Day 1 installation to widespread adoption across development teams, business units, and application portfolios.
- ▶ **A map of key supporting patterns**, the application architecture patterns—platform capabilities, and organizational practices—that unblock cloud-native adoption at each stage.

Whether you are beginning to containerize your 1st workloads or looking to optimize a mature microservices platform, the chapters ahead provide practical guidance, real customer examples, and a clear picture of what success looks like at each stage.

The journey unfolds across 4 stages:



1. **Containerize**



2. **Operationalize**



3. **Modernize**



4. **Optimize**

This playbook is written for platform engineers, architects, and engineering leaders who are responsible for leading cloud-native adoption within their organizations.

The path to cloud-native innovation is not a single leap. It is a series of deliberate steps, each building on the last. Let this playbook be your guide.

Chapter 1:

The application innovation platform

Why organizations need an application platform 05

Application platforms in the age of AI 06

Red Hat OpenShift: Accelerate innovation, reduce complexity, run anywhere 08

1 platform, any infrastructure 09

The platform advantage: Integrated, not assembled 10

The foundation for your cloud-native journey 10



Why organizations need an application platform

Enterprise software development has evolved. Applications are no longer monolithic systems deployed quarterly on dedicated servers. They are composed of dozens, sometimes hundreds, of services, built by distributed teams, deployed continuously across multiple environments, and expected to scale on demand. The infrastructure beneath these applications must keep pace.

Yet many organizations still operate with fragmented toolchains. Development teams choose their own continuous integration and continuous delivery (CI/CD) tools. Operations teams manage separate clusters for different business units. Security teams bolt on scanning and compliance checks as an afterthought. The result is sprawl: inconsistent deployment practices, duplicated effort, and security gaps that widen as the portfolio grows.

The repercussions can be costly. Without a unified platform, tracking which teams use which base images becomes nearly impossible, rollback procedures vary by team, and a vulnerability discovered in a base image may never be fully remediated across the portfolio.

An [application platform](#) solves this problem by providing a unified foundation for building, delivering, and running applications. It's more than a container orchestrator; it's the integrated set of developer tools, delivery pipelines, security controls, and operational capabilities that an organization needs to move from idea to production, consistently, with security, and at scale.

The most effective application platforms:

- ▶ Abstract infrastructure complexity so developers can focus on code rather than configuration.
- ▶ Embed security into the development lifecycle rather than treating it as a gate at the end.
- ▶ Provide self-service capabilities that give development teams autonomy while maintaining organizational guardrails.
- ▶ Run consistently across any infrastructure, including on-premise datacenters, public clouds, and locations at the edge of the network.

The distinction between a container orchestrator and an application platform matters. Kubernetes provides the engine, including pod scheduling, service networking, and storage orchestration. But running it in production requires additional investment: CI/CD pipelines, container registries, observability stacks, security scanners, certificate management, identity integration, multicluster management, and developer tooling. Organizations that adopt raw Kubernetes often spend their 1st year building these capabilities from scratch, assembling open source tools into a bespoke platform. An application platform provides these capabilities out of the box, integrated and tested together, so organizations can focus on delivering applications rather than building infrastructure.

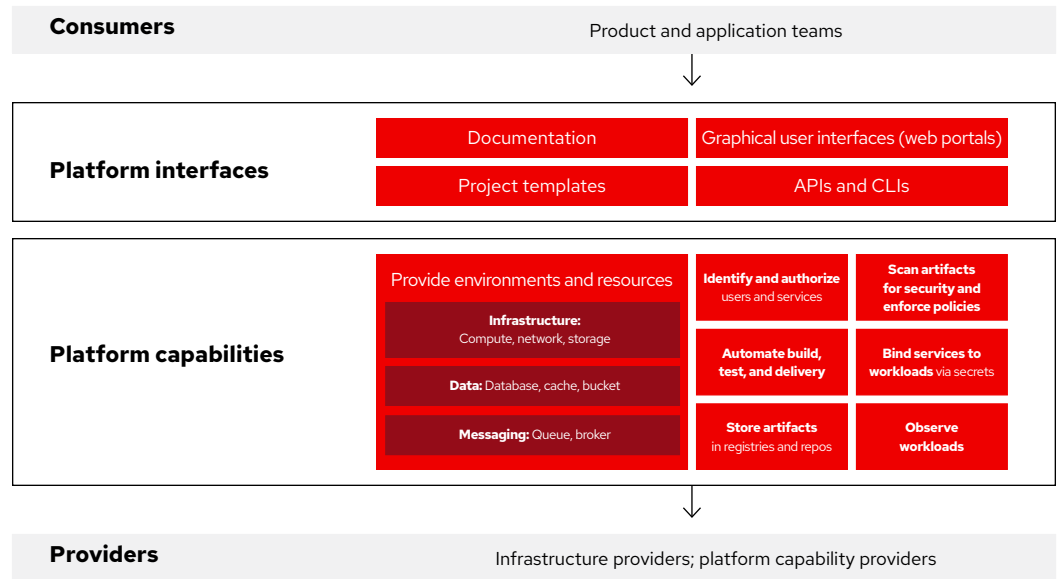


Figure 1. Components of an application platform.

Application platforms in the age of AI

A common reaction to the rise of AI code assistants and agentic development tools is to question whether application platforms are still necessary. The answer is a resounding yes. AI-assisted development makes application platforms more essential, not less. Here's why.

Writing code is only a single phase of software delivery

Code must be built into container images, scanned for vulnerabilities, signed for provenance, stored in a trusted registry, deployed with the correct resource constraints, observed for regressions, and rolled back when something goes wrong. An AI agent can generate a Dockerfile in seconds, but somebody, or something, still needs to build that image with a focus on security to help make sure it doesn't introduce critical Common Vulnerabilities and Exposures (CVEs), deploy it with appropriate memory limits, and alert the on-call engineers when it consumes 10 times the expected central processing unit (CPU) at 2 AM.

AI-assisted development also means that more services are deployed more frequently. Ten services requiring bespoke operational configuration is manageable. One hundred services, each deployed and updated multiple times per week by AI-assisted teams, is not. The platform provides the abstractions (such as ingress controllers, service mesh, centralized observability, or automated certificate management) that prevent operational complexity from growing linearly with the application portfolio. AI agents accelerate code creation. The platform handles everything else.

Higher velocity demands automated guardrails and security

When humans write all the code, the pace of change is naturally throttled by the speed of human review and coordination. AI agents remove that hurdle, generating container configurations, pipeline definitions, Infrastructure as Code (IaC) templates, and application code in parallel. Organizations cannot, and should not, extend the same level of trust to AI-generated code that they extend to code written and reviewed by experienced engineers. AI agents can produce code that is syntactically correct, passes unit tests, and appears functional, while still containing subtle security issues such as overly permissive security contexts, base images with known vulnerabilities, hard-coded credentials, or misconfigured network policies. The volume of AI-generated code exceeds what human review alone can validate.

The application platform provides this enforcement layer. Security controls reject noncompliant deployments before they reach the cluster. Pipeline gates mean images must pass vulnerability scans and compliance checks before reaching production. Image signing, software bill of materials (SBOM) generation, and runtime security policies create an automated trust boundary independent of who, or what, wrote the code. These controls exist in the platform, operate automatically, and scale with the volume of changes.

AI agents are consumers of the platform, not replacements for it

AI agents need the same structured interfaces as human developers to generate correct output. To scaffold a new microservice correctly, an agent must discover the organization's approved base images, follow the standard project layout, generate a CI pipeline that matches security requirements, and produce Kubernetes manifests that comply with namespace quotas, network policies, and resource limits. Without a platform that codifies these standards in machine-readable form, through software templates in a self-service developer portal, policy definitions, and pipeline configurations, the AI agent is guessing. And guessing at scale produces inconsistency at scale.

Whether 10 developers or 10,000 AI agents are deploying workloads, someone needs to enforce resource quotas, manage namespace isolation, prevent noisy-neighbor problems, and make sure that there is fair scheduling across teams and projects. The platform serves as this arbiter. Without platform-level controls, an AI agent that autoscales aggressively could starve neighboring services of resources.

The platform is the multiplier for AI-assisted development

The most productive AI-assisted teams will be those whose agents operate within a well-structured application platform defined by clear software templates, [Golden Paths](#), automated validation, and guardrails. Investing in a scalable, well-governed platform foundation today creates the framework that allows AI-assisted development to scale safely and adapt continuously.

Red Hat OpenShift: Accelerate innovation, reduce complexity, run anywhere

[Red Hat OpenShift](#) is a comprehensive application platform built on Kubernetes and enriched with developer, operational, and security capabilities. It provides everything organizations need to build, deliver, and scale applications across any infrastructure.

Red Hat OpenShift's value is organized around 3 pillars: **Build** applications with security from the start, **accelerate** delivery across hybrid cloud, and **scale** any application on any infrastructure.

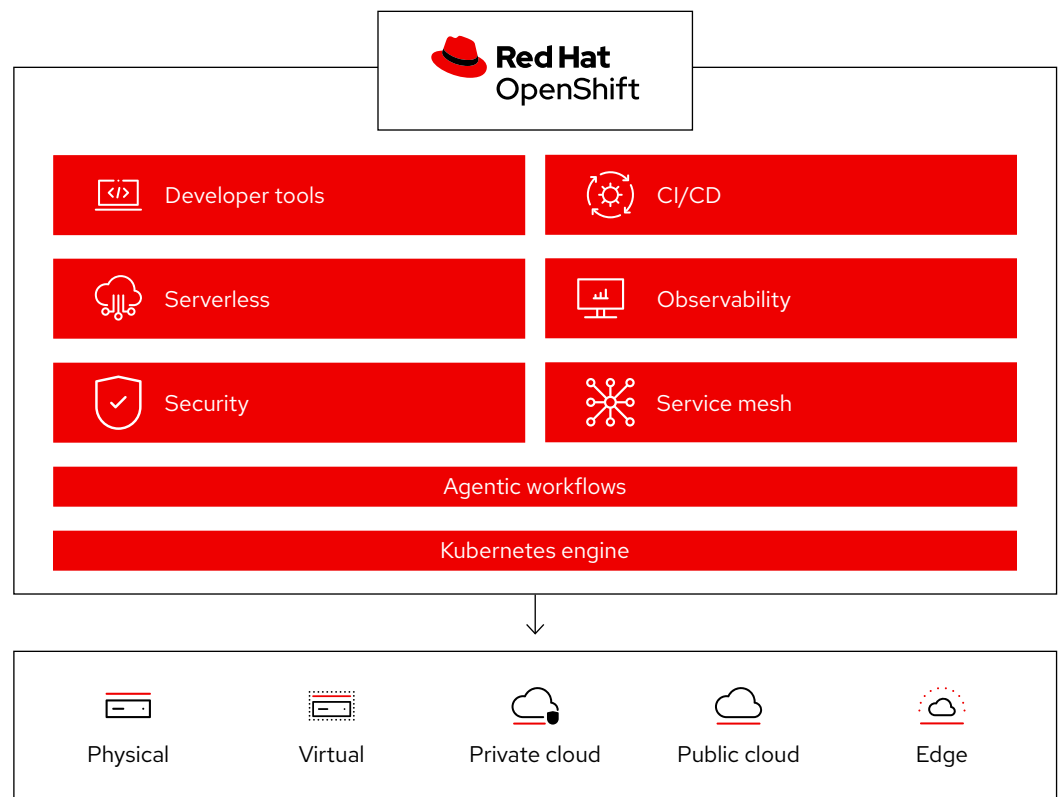


Figure 2. Red Hat OpenShift provides the foundation needed to build, deliver, and scale applications across any infrastructure.

Build applications with security from the start

Red Hat OpenShift simplifies the developer experience by integrating security from the initial line of code. By embedding vetted images, vulnerability scanning, and security controls directly into workflows, the platform has tools to help protect applications without slowing innovation. Developers gain the flexibility to build microservices in their preferred languages, while operations teams manage them from a centralized console, supported by a broad ecosystem of Red Hat-maintained tools.

Accelerate delivery across hybrid cloud

To move code from workstation to production safely and quickly, Red Hat OpenShift provides integrated CI/CD, GitOps, and observability capabilities. The platform's operator framework supports containers, virtual machines (VMs), and serverless architectures across disparate environments. With embedded continuous security, AI-ready infrastructure—including graphics processing unit (GPU) slicing—and intelligent assistance, Red Hat OpenShift reduces cognitive load, accelerates time to market, and maintains operational integrity across the hybrid cloud.

Scale any application on any infrastructure

Red Hat OpenShift delivers a consistent platform for modern applications across datacenters, public clouds, and the edge. By supporting the modernization of legacy applications, VMs, and serverless workloads alongside containers, organizations can reduce operational complexity. Automated services like Red Hat Advanced Cluster Management for Kubernetes and Red Hat Advanced Cluster Security for Kubernetes allow teams to enforce consistent policies, ensure reliable deployments, and maintain visibility across multicluster environments.

1 platform, any infrastructure

One advantage of Red Hat OpenShift is infrastructure flexibility. The same platform, the same application programming interfaces (APIs), and the same operational model run consistently across:

- ▶ **Physical servers:** Bare-metal deployments for maximum performance and control, common in regulated industries and high-performance computing scenarios.
- ▶ **Virtual infrastructure:** Deploy on VMware, [Red Hat OpenShift Virtualization](#), or other hypervisors to benefit from existing investments while adopting cloud-native practices.
- ▶ **Private cloud:** Run Red Hat OpenShift on private cloud infrastructure like [Red Hat OpenStack® Services on OpenShift](#), providing self-service cloud capabilities within the datacenter.
- ▶ **Public cloud:** Deploy as a managed service with [Red Hat OpenShift Service on AWS](#), [Microsoft Azure Red Hat OpenShift](#), [Red Hat OpenShift Dedicated on Google Cloud](#), or [Red Hat OpenShift on IBM Cloud](#). Managed services reduce operational burden while maintaining full Red Hat OpenShift capabilities.
- ▶ **Edge:** Extend the platform to edge locations with lightweight, single-node, or 3-node configurations. [Edge deployments](#) bring compute closer to where data is generated, which helps low-latency applications in retail, manufacturing, and telecommunications.

Organizations run workloads across multiple infrastructure types simultaneously. A financial services firm might run its trading platform on bare metal for performance, its customer-facing applications on public cloud for elasticity, and its branch systems at the edge for latency. Red Hat OpenShift provides a single platform that spans all of these environments, reducing the operational complexity of managing multiple, segmented infrastructures.

The platform advantage: Integrated, not assembled

The difference between an integrated platform and a collection of tools becomes apparent at scale. Consider the security workflow: a developer commits code; the CI pipeline builds a container image; that image is scanned for vulnerabilities, signed for provenance, pushed to a registry, deployed through GitOps, and monitored in production for runtime threats. In an integrated platform, these steps share identity, policy, and telemetry. A vulnerability discovered during scanning is linked to the specific build, the specific source commit, and the specific deployment, creating a complete chain of custody.

In a tool-assembled approach, each step produces its own artifacts, logs, and reports. Correlating a production vulnerability back to its source requires manual investigation across multiple systems, and at hundreds of applications, this becomes impractical.

The same integration advantage applies to developer experience. A platform that connects the integrated development environment (IDE), the build system, the deployment pipeline, and the observability tooling provides end-to-end visibility. A developer who commits code can watch it build, deploy, and serve traffic from a single interface, tightening the feedback loop between writing code and understanding its impact in production.

The foundation for your cloud-native journey

The platform is the lever that multiplies the effectiveness of every development team in the organization, and increasingly, every AI agent working alongside them.

An application platform isn't just infrastructure. It's the foundation upon which your organization's entire software delivery capability is built. The platform's capabilities directly influence how fast your teams can ship software, how secure your applications are in production, and how efficiently you use infrastructure resources.

Organizations that treat the platform as a strategic investment consistently deploy more frequently, recover from incidents faster, and ship new features with greater confidence.¹ The platform is the lever that multiplies the effectiveness of every development team in the organization, and increasingly, every AI agent working alongside them.

The chapters that follow show how to progressively unlock the full value of this platform, moving from basic containerization through operational maturity, application modernization, and platform-level optimization. The journey starts with understanding where you are today.

¹ Cloud Native Computing Foundation. "[CNCF Annual Cloud Native Survey](#)." January 2026.

Chapter 2:

Understanding cloud-native adoption maturity

Why a maturity model matters 12

The 4 stages of cloud-native adoption maturity 12

Stage 1: Containerize ▶ Container adopters 13

Stage 2: Operationalize ▶ Platform adopters 13

Stage 3: Modernize ▶ Cloud-native practitioners 13

Stage 4: Optimize ▶ Platform innovators 14

How to assess where your organization stands today 14

The roadmap at a glance 15



Why a maturity model matters

Installing an application platform is the beginning, not the end, of the cloud-native journey. Many organizations invest substantial resources in standing up Red Hat OpenShift clusters, only to find that adoption stalls after the 1st few workloads are containerized. The platform is operational, but the promise of faster delivery, greater resilience, and lower costs remains unrealized.

This pattern is remarkably common: an organization provisions its initial cluster, migrates a handful of applications, and declares success, but 6 months later, those same 5 applications are the only workloads on the platform. Developers find it confusing. Operations teams are overwhelmed with manual processes. Leadership begins to question the investment.

The gap between installation and value is not a technology problem. It is an adoption problem. Organizations that achieve the full benefits of cloud-native architecture do so by deliberately evolving their applications, teams, and platform capabilities over time. They don't try to leap from traditional infrastructure to fully optimized microservices in a single bound. Instead, they progress through a series of stages, each building the skills, patterns, and platform maturity needed for the next.

The maturity model presented in this chapter is built from lessons learned from the most successful Red Hat OpenShift customers, organizations running thousands of applications in production. It provides a framework for understanding where your organization stands today, what the next stage looks like, and what patterns and practices will get you there.

This model is not a rigid prescription. Not every application will progress through every stage. A stable internal tool may remain a containerized monolith indefinitely, while a customer-facing platform might advance to event-driven architecture. The model also serves as a communication tool, giving platform teams a shared vocabulary for discussing adoption progress with leadership, architects, and development teams. For example, saying "our customer-facing applications are at Stage 2 and we need to reach Stage 3 by Q4" is far more useful than "we need to modernize."

The 4 stages of cloud-native adoption maturity

The maturity model defines 4 stages, each characterized by distinct organizational behavior, application architecture patterns, and platform capabilities. Organizations typically move through these stages sequentially, though different applications within the same organization might sit at different stages simultaneously.

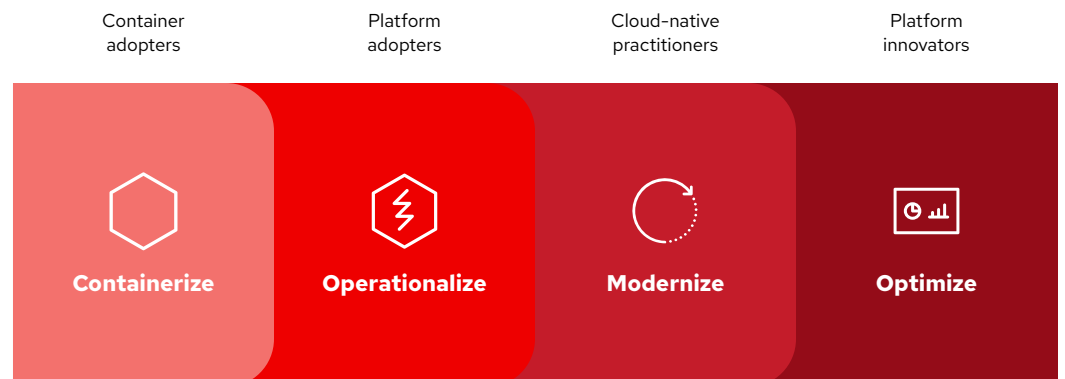


Figure 3. 4 stages of cloud-native adoption.

The 2026 CNCF annual report uses a similar cloud-native maturity grouping: explorers, adopters, practitioners, and innovators.

[Visit their report to see their findings.](#)¹



Stage 1: Containerize

Business outcome: Faster deployment and operational consistency.

Organizations at this stage are beginning their cloud-native journey. The primary focus is transitioning from traditional infrastructure to container-based deployments, using a lift-and-shift approach for initial migrations. The goal is to establish the operational foundation for running container workloads reliably while demonstrating the value of the platform.

This stage is about building confidence. The organization is learning a new operational model, and the application patterns adopted here, such as health checks, externalized configuration, and log forwarding, create the foundation for everything that follows. See [Chapter 3](#) for more details.



Stage 2: Operationalize

Business outcome: Zero-downtime deployments and operational efficiency.

With container viability proven, the focus shifts to operational maturity. Multiple production workloads are running on the platform, and now the goal is running them efficiently and reliably at production scale.

This stage introduces the operational discipline that separates proof-of-concept deployments from production-grade platforms. Observability instrumentation, automated scaling, graceful shutdown procedures, and deployment standardization with tools like Helm and Kustomize become essential practices. See [Chapter 4](#) for more details.



Stage 3: Modernize

Business outcome: Keep pace with business priorities.

Organizations actively modernize their application architecture, decomposing monoliths into microservices, adopting cloud-native frameworks, and building the service governance capabilities needed to manage a distributed architecture.

This stage is where cloud-native agility is realized—and where complexity increases dramatically—making service discovery, API gateway patterns, and deployment orchestration essential. See [Chapter 5](#) for more details.

¹ Cloud Native Computing Foundation. [“CNCF Annual Cloud Native Survey.”](#) January 2026.



Stage 4: Optimize

Business outcome: Elastic cost efficiency with predictable performance at scale.

This is where the platform delivers its highest return. Event-driven architectures reduce costs by scaling precisely to demand. Service mesh provides zero trust security and resilience without code changes. Advanced patterns like Command Query Responsibility Segregation (CQRS) and event sourcing handle the data complexity that comes with truly distributed systems. See [Chapter 6](#) for more details.

How to assess where your organization stands today

Most organizations do not sit neatly within a single stage, and that is expected. The maturity model applies to individual applications and workload portfolios, not the entire organization uniformly. To assess where a specific workload or team sits, consider these diagnostic questions:



You are likely at Stage 1 if:

- ▶ Applications are being containerized for the 1st time.
- ▶ Deployments are largely manual or script-based.
- ▶ Observability is limited to basic infrastructure monitoring.
- ▶ The platform team is focused on cluster setup and maintenance.
- ▶ Developers are still learning container fundamentals.



You are likely at Stage 2 if:

- ▶ Multiple applications run in production.
- ▶ CI/CD pipelines exist but aren't standardized across teams.
- ▶ Observability includes application-level metrics and alerts.
- ▶ Scaling is partially automated but not based on application-specific metrics.
- ▶ The platform team is building self-service capabilities.



You are likely at Stage 3 if:

- ▶ Applications are being decomposed into microservices.
- ▶ Multiple teams deploy independently to the same platform.
- ▶ Service discovery and API management are active concerns.
- ▶ Deployment strategies include canary releases or blue-green deployments.
- ▶ The platform team operates a developer portal with Golden Paths.



You are likely at Stage 4 if:

- ▶ A large number of microservices communicate through service mesh.
- ▶ Event-driven architecture is used for critical workloads.
- ▶ Resilience patterns like circuit breakers are implemented at the platform level.
- ▶ Scaling responds to business events, not just CPU and memory.
- ▶ The platform provides comprehensive self-service for most development lifecycle needs.

The roadmap at a glance

Progression is not always linear. The model accommodates this reality; it measures workloads and teams, not the organization as a monolithic unit.

Progression is not always linear. An organization might advance some applications to Stage 3 while still onboarding new ones at Stage 1. A major acquisition might bring in a portfolio that starts from Stage 1 even as the existing portfolio operates at Stage 3. The model accommodates this reality; it measures workloads and teams, not the organization as a monolithic unit.

The common thread across all stages is the platform engineering team. Regardless of where a particular workload sits, the platform team provides the tools, automation, and self-service capabilities that make progression possible. [Chapter 7](#) addresses this team, its evolving role alongside AI-assisted development, and the practices that make it effective.

Chapter 3:

Stage 1—Containerize



Faster deployment and operational consistency	17
Organization maturity at this stage	17
Application patterns in depth:	18
Containerized monolith	18
Log to standard outputs	19
Health check endpoints	20
Init-container pattern	21
Externalizing configurations	22
High-performance local volumes	23
External Database-as-a-Service	23
Security at this stage	24
Common mistakes to avoid	24
Red Hat OpenShift capabilities that accelerate progression	25
Customer spotlights: Advent One, Emirates NBD	28



Faster deployment and operational consistency

The cloud-native journey begins with a deceptively simple step: packaging existing applications into containers and running them on Red Hat OpenShift. Doing this well requires attention to patterns that will either accelerate or hinder every stage that follows.

Containerization addresses 3 persistent challenges. The 1st is deployment speed. In traditional environments, deploying across development, staging, and production can stretch from days to weeks. Containers collapse this timeline: a containerized application carries its dependencies and deploys through a single declarative manifest, turning coordinated manual effort into a repeatable, automated process.

The 2nd is operational consistency. Traditional environments drift over time as configuration changes accumulate and patches are applied unevenly. Containers solve this structurally: the same immutable image runs in every environment, every time, with differences handled through externalized settings, not by modifying the application. Fewer “works on my machine” surprises means less time troubleshooting and more time delivering value.

The 3rd is environment availability. In traditional infrastructure, developers often wait days or weeks for operations teams to provision a new environment. Kubernetes namespaces and resource quotas turn environment provisioning into a self-service, on-demand operation, removing a major bottleneck from the development cycle and freeing operations teams to focus on platform capabilities rather than fielding provisioning requests.

Organizations at this stage are **container adopters**. The approach is primarily lift-and-shift, and the focus is on proving viability and demonstrating platform value.

Organization maturity at this stage

Maturity at Stage 1 consists of:

- ▶ **Early containerization** of existing applications, often without major architectural changes. Applications are packaged into containers and deployed on the platform as-is.
- ▶ **A mostly lift-and-shift approach**, where the immediate goal is operational consistency rather than architectural optimization. Applications gain the benefits of container orchestration, automated restarts, resource isolation, and declarative deployment, without requiring code changes.
- ▶ **Creation of the platform team**, focused on setting up the basic container platform. The team is learning Kubernetes concepts and establishing initial operational procedures.
- ▶ **Developers learning container basics**, including how to write Dockerfiles, understand pod lifecycles, and troubleshoot in a containerized environment.
- ▶ **A focus on proving viability** and demonstrating platform value to stakeholders. Early wins, such as faster deployment times or improved consistency across environments, build organizational confidence.

Some organizations arrive at this stage with workloads already running as VMs on Red Hat OpenShift Virtualization, having migrated from an existing virtualization platform. This is a valid starting point that allows teams to build familiarity incrementally.

Application teams at this stage are typically asking: How do we containerize our existing applications? How do we know if our containers are healthy? How do we manage configurations across different environments? What storage options do we need for stateful workloads?

The key challenge at this stage is demonstrating value quickly. Leadership needs to see that the platform investment pays off before committing to broader adoption. Select initial workloads that are visible enough to matter but manageable enough to succeed. Early wins, such as faster deployment times, improved consistency, reduced environment drift, and self-service environment provisioning, build organizational confidence and create internal advocates who make the case for expanding platform adoption.

Application patterns in depth

The following patterns are the building blocks of containerized applications. Taken together, they create applications that are observable, configurable, resilient, and ready for the next stage of maturity.

Containerized monolith

The initial step for most organizations is to containerize existing applications as-is. A containerized monolith is not the end state; it's the starting point that delivers immediate value while establishing the foundation for future modernization.

The key is resisting the temptation to refactor and containerize simultaneously. These are separate concerns. A monolithic Java application in a container is still a monolith. But it can now be deployed consistently, scaled horizontally (if stateless), and managed through the same platform tooling as every other workload. Start by focusing on getting the application running reliably in a container: identify dependencies, package the runtime, configure the entry point, and validate behavior. Architectural modernization comes in later stages. The strangler fig pattern described in [Chapter 5](#) provides a proven, incremental approach to that future decomposition.

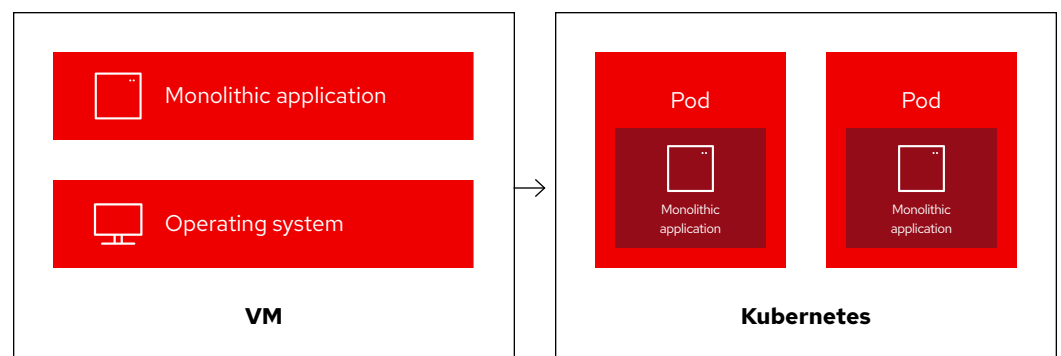


Figure 4. Monolithic application on a VM versus a containerized monolithic application.

Log to standard outputs

Traditional applications write logs to files on disk. Containerized applications should write logs to standard output (stdout) and standard error (stderr) so the platform can capture, aggregate, and forward them to log management systems.

When your application writes to stdout, developers can view logs through the platform console or command line interface (CLI), and operations teams can forward them to centralized systems for analysis, alerting, and long-term storage. It also simplifies troubleshooting: logs written to files inside a container are lost when the container restarts or is rescheduled, while stdout-based logging ensures data is captured regardless of pod lifecycle events.

For applications that cannot easily be modified to write to stdout—traditional applications with hard-coded file paths or frameworks that require file-based logging—a sidecar container can tail log files and forward them to stdout as an interim solution. (See [Chapter 5](#) where the sidecar pattern is explored in depth.) Most modern frameworks (Spring Boot, Quarkus, Node.js) default to console output and require minimal configuration changes.

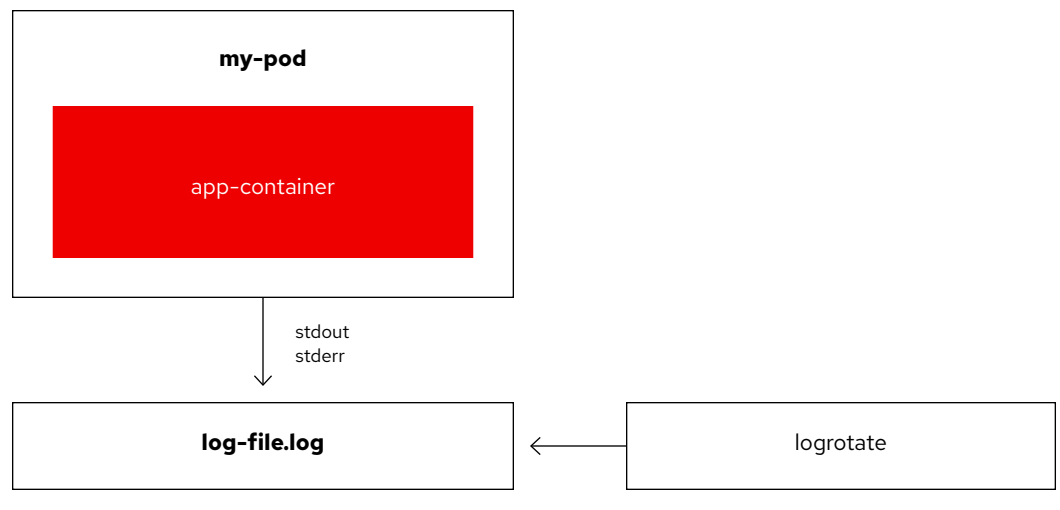


Figure 5. Application log to standard output and standard error.

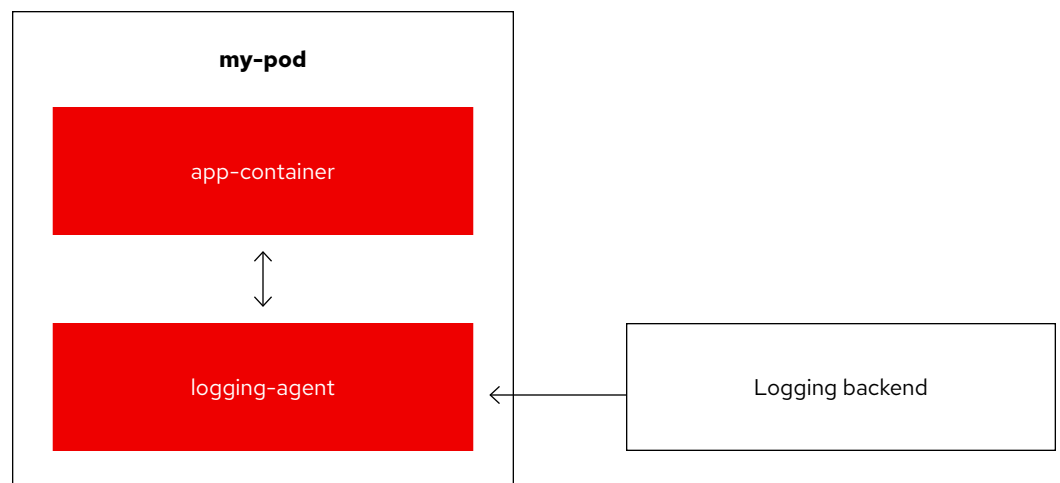


Figure 6. Sidecar forwarding hard-coded application logs to the logging backend.

Health check endpoints

Kubernetes needs to know whether your application is live and ready to serve traffic. Without this information, the platform cannot make intelligent decisions about routing, scaling, or restarting your application.

Three probe types serve distinct purposes. Liveness probes tell Kubernetes whether the application process is still functioning. If there's a failure, Kubernetes restarts the container, handling situations like deadlocks or corrupted state. Without a liveness probe, a deadlocked application sits idle indefinitely, consuming resources and dropping requests.

Readiness probes tell Kubernetes whether the application is ready to accept traffic. If it fails, Kubernetes removes the pod from the service's load balancer, preventing traffic from routing to pods still initializing or warming connections. During a rolling deployment, readiness probes ensure that traffic only shifts to new pods after they are fully initialized.

Startup probes are useful for applications with long initialization times. They prevent Kubernetes from killing slow-starting containers by deferring liveness checks until the startup probe succeeds.

Health checks should be lightweight and test meaningful application state, not just return HTTP 200 unconditionally. A liveness check might verify that the main application thread is responsive. A readiness check might verify that database connections are established and required caches are loaded. Avoid expensive health checks (such as full database queries), as probes execute frequently and shouldn't add heavy load to the application or its dependencies.

Health checks are foundational for the graceful shutdown pattern in [Chapter 4](#), where readiness probes play a critical role in achieving zero-downtime deployments.

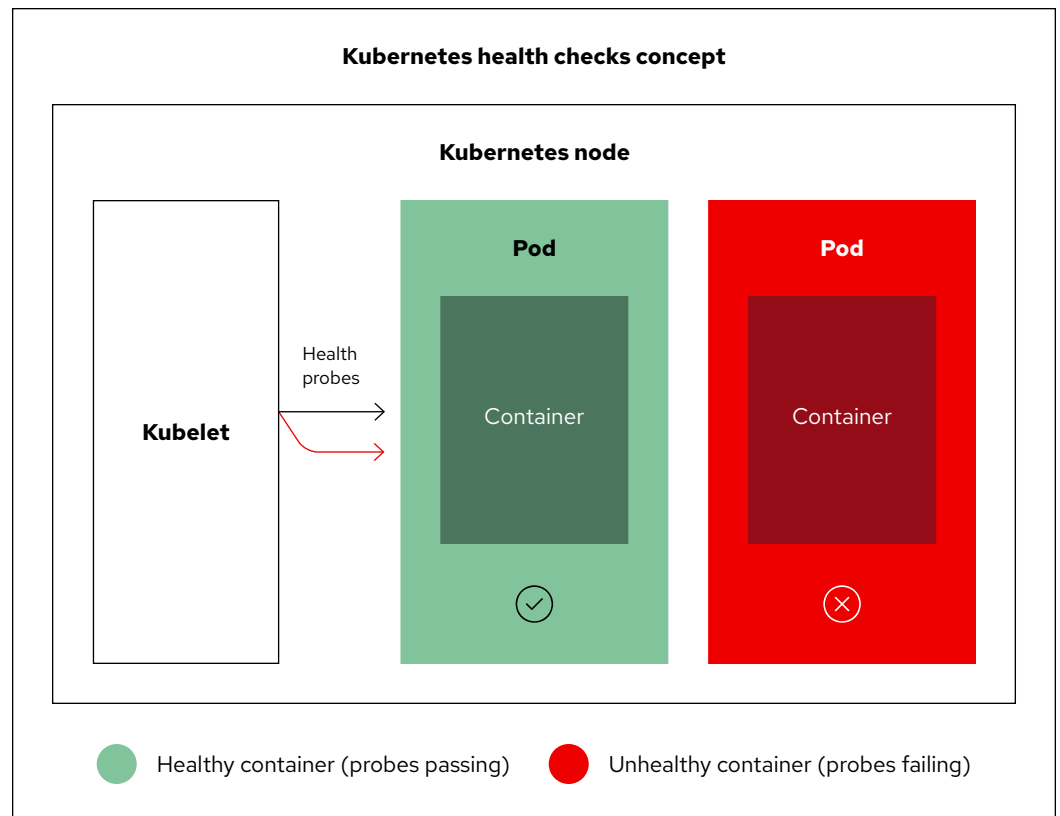


Figure 7. Kubernetes health checks.

Init-container pattern

Many applications need to perform initialization tasks before starting, like database schema migration, configuration file generation, secret retrieval, or waiting for dependent services. Embedding this logic in the application itself creates tight coupling between initialization and runtime concerns, making both harder to maintain and test independently.

Init containers run to completion before the main application container starts, sharing the same pod volumes. The main container does not start until all init containers complete successfully, ensuring that the application never starts in a partially initialized state.

Common uses for init containers include running database migrations, fetching secrets from an external vault and writing them to a shared volume, waiting for dependent services, and generating environment-specific configuration files. Because each init container can use a different image, initialization tools don't need to be bundled into the application image. A Flyway migration runner or Vault agent can handle their respective tasks while the main container stays focused on the application itself.

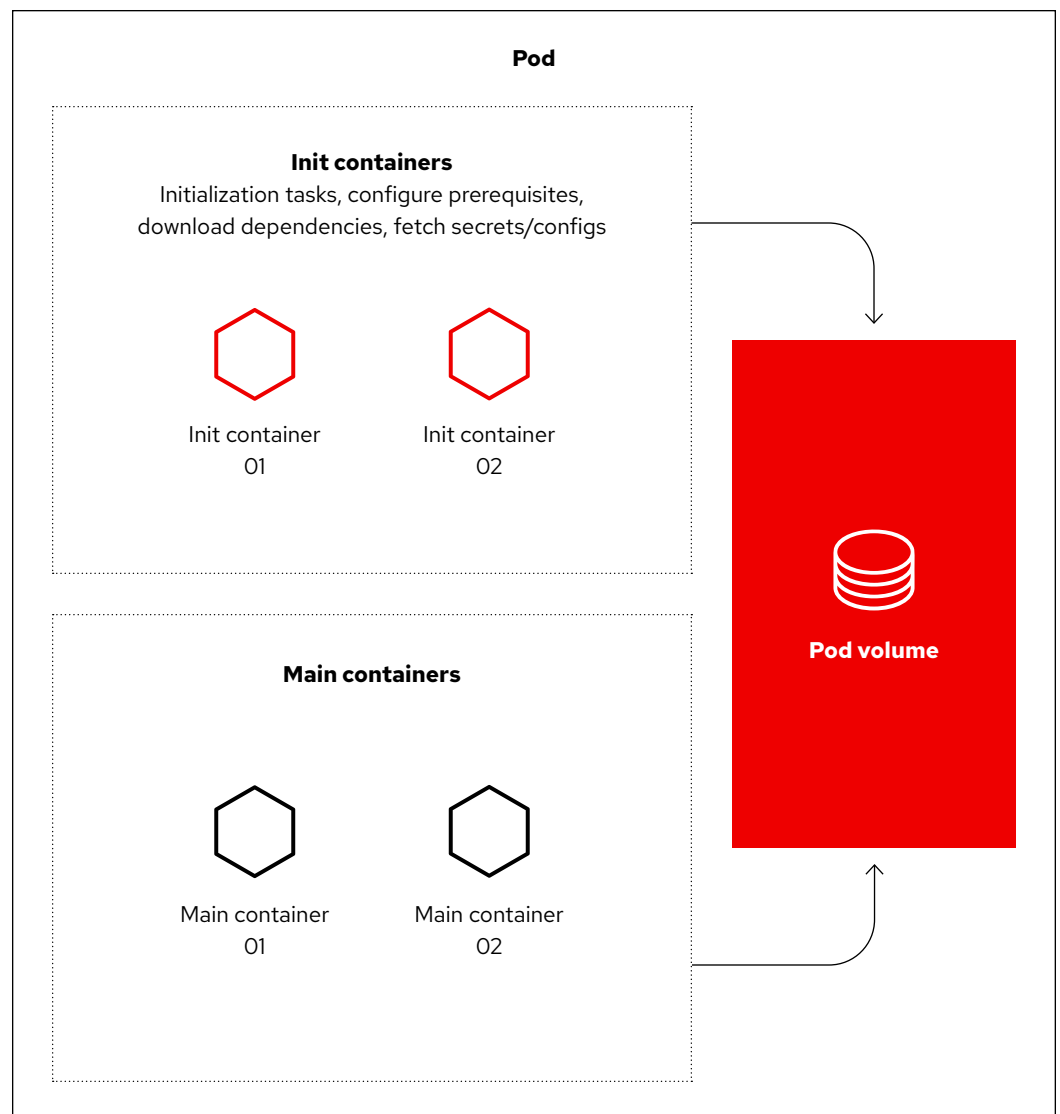


Figure 8. Init container running application initialization task before the application starts.

Externalizing configurations

Hard coding configuration values, such as database connection strings, feature flags, API endpoints, and timeouts, into container images is a common mistake at this stage. It forces different images for each environment, defeats the purpose of container portability, and creates security risks: credentials embedded in image layers persist in registry history even after the image is updated.

Instead, externalize configuration using Kubernetes ConfigMaps and Secrets. Configuration values are injected as environment variables or mounted files, so the same container image runs in development, staging, and production. ConfigMaps are appropriate for nonsensitive settings: feature flags, logging levels, service endpoints, and application-specific settings. For sensitive configuration, such as database passwords, API keys, and transport layer security (TLS) certificates, do not rely on Kubernetes Secrets, which are merely base64-encoded and readable by anyone with namespace access. Instead, use an external [secrets manager](#) integrated through a Kubernetes-native operator or container storage interface (CSI) driver that synchronizes credentials from your vault into pods at runtime. These integrations provide centralized secret lifecycle management, automatic rotation, and audit trails that Kubernetes Secrets fundamentally lack.

This pattern is essential for CI/CD pipelines in later stages. [Chapter 4](#) covers Kustomize overlays and Helm chart value overlays, structured approaches for managing environment-specific configuration that build directly on this foundation.

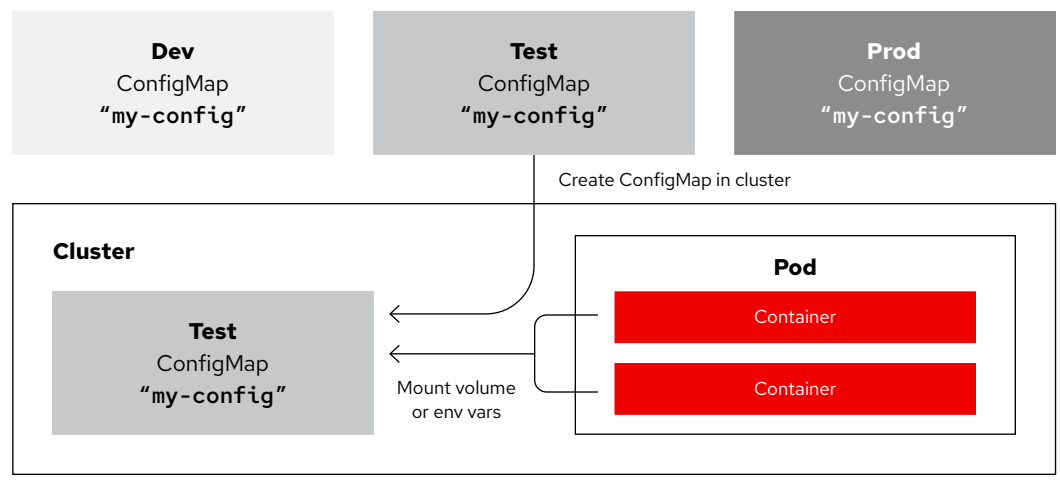


Figure 9. Configurations externalized as separate ConfigMaps for each deployment environment.

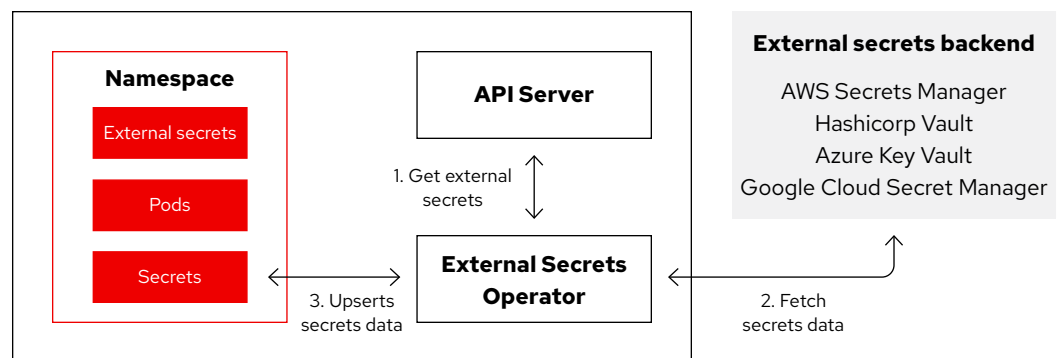


Figure 10. Credentials managed in external secret managers and injected into the namespace by the External Secrets Operator.

High-performance local volumes

Stateful workloads, like databases, caches, and data-intensive applications, require high-performance storage that survives pod restarts and rescheduling. The key consideration is the trade-off between performance and durability. Local volumes deliver the highest throughput and lowest latency but do not survive node failures. Software-defined storage replicates data across nodes, ensuring availability even when individual nodes fail.

Kubernetes provides a pluggable storage model through the CSI, allowing organizations to use existing storage investments, whether Storage Area Network (SAN) arrays, Network Attached Storage (NAS) filers, or software-defined storage, without rearchitecting their infrastructure. Choose storage class based on workload characteristics: block storage for low-latency database access, file storage for shared volumes across multiple pods, and object storage for unstructured data at scale. Assuming all applications are stateless, and neglecting to provision persistent storage, leads to data loss when pods are rescheduled.

External Database-as-a-Service

Managing databases on Kubernetes adds operational complexity; backup, restore, failover, patching, and performance tuning all require specialized expertise. For organizations at Stage 1, still building Kubernetes proficiency, using external Database-as-a-Service offerings is often the pragmatic choice. These can be cloud provider managed services, such as Amazon Relational Database Service (RDS), Azure Database, or Google Cloud SQL, or databases managed by an internal database administrator team on dedicated infrastructure.

This pattern decouples the database lifecycle from the application lifecycle. Application teams focus on writing code, and database specialists handle operations. Connection details are injected through Kubernetes Secrets, keeping the application portable and credentials protected with external secrets management systems. As the organization matures, [Kubernetes Operators for databases](#) can bring database management closer to the platform with automated provisioning, backup, and failover, as described in [Chapter 4](#). At Stage 1, simplicity and reliability should take precedence over architectural elegance.

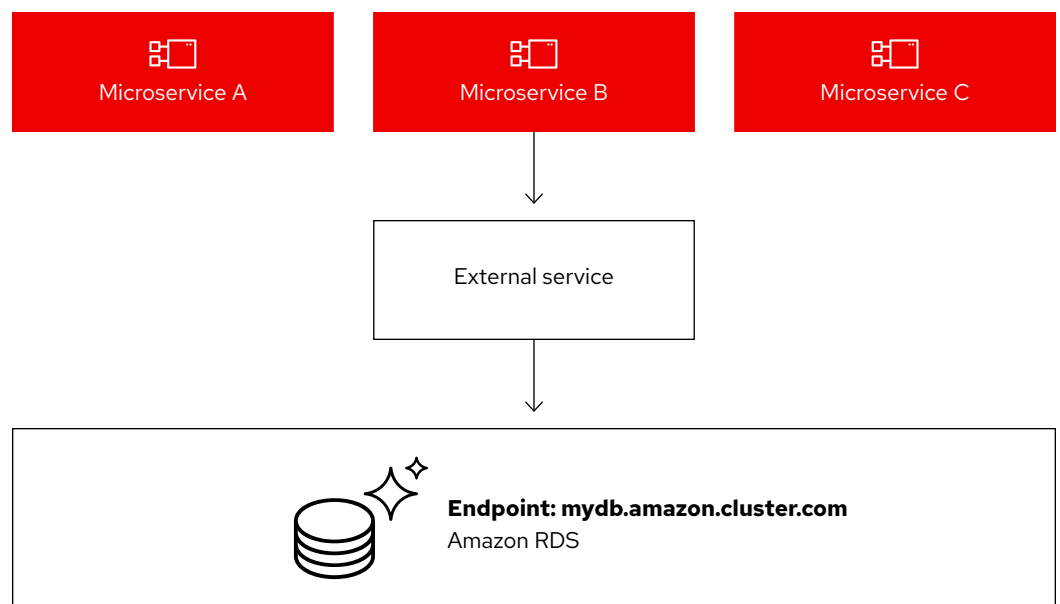


Figure 11. Application data in an external database on Amazon RDS.

Security at this stage

Security at the containerization stage focuses on establishing a strong foundation, including trusted images, nonroot execution, and externalized secrets. The right foundation prevents entire categories of vulnerabilities from reaching production and becomes the baseline every subsequent stage builds on.

Start with trusted base images

Use [Red Hat Universal Base Image \(UBI\)](#) as the foundation for container images. UBI images are tested, signed, and continuously updated with security patches. Beyond UBI, Red Hat OpenShift ships with a [curated catalog of container images](#) for common language runtimes. These images follow Red Hat's security hardening practices and are maintained with regular CVE remediation.

To reduce the size of the possible attack surface, [Red Hat Hardened Images](#) provide micro-sized, vendor-neutral container images built with a near-zero known vulnerability posture at release, each shipping with a validated security profile and an integrated [SBOM](#). The [Red Hat Ecosystem Catalog](#) provides hundreds of certified third-party products, from databases and monitoring tools to AI frameworks, that have been tested and validated to run on Red Hat OpenShift. Starting from trusted, certified images rather than pulling unvetted images from public registries reduces the chance of supply chain vulnerabilities before a single line of application code is written.

Run containers as nonroot, no exceptions

A compromised container running as root can escape its sandbox, access host resources, and pivot to other workloads on the same node. Configure security contexts to drop all Linux® capabilities, set read-only root file systems, and explicitly prevent privilege escalation. Red Hat OpenShift enforces this through [Security Context Constraints \(SCCs\)](#), and the default SCC already prohibits running as root. If a workload claims to require root access, treat that as a defect to fix, not an exception to grant.

Avoid Kubernetes Secrets, when possible

[Kubernetes Secrets](#) are base64-encoded, not encrypted. Anyone with API access to the namespace can read them in plain text, and they persist unencrypted in etcd unless you explicitly configure encryption at rest. Do not treat them as a secure storage mechanism. Instead, integrate with external secrets management systems using the [External Secrets Operator](#) or [Secrets Store CSI Driver](#). They inject secrets directly from the vault into pods as in-memory tmpfs volumes. Secrets never pass through the Kubernetes API or persist in etcd, and they disappear when the pod terminates.

Common mistakes to avoid

Skipping application-centric health checks

Relying on “is the process running” is insufficient. Kubernetes cannot distinguish between a running application and a functional application without proper liveness and readiness probes. Equally problematic are overly aggressive health checks. A readiness probe that queries a downstream database on every check can cascade failures when that database is under load, taking healthy pods out of rotation and amplifying the outage it was meant to detect.

Hard coding configuration values in container images

This forces rebuilds for each environment, and makes it impossible to use the same image across development, staging, and production. It also risks embedding secrets in image layers where they persist in registry history even after updates.

Treating the containerized monolith as the end state

Containerizing an application is a starting point, not the goal. Organizations that stop here miss the efficiency, resilience, and scalability benefits of later stages. Resist the opposite mistake of trying to refactor and containerize applications simultaneously. Be deliberate about which applications are suitable for refactoring from the start.

Ignoring storage requirements for stateful applications

Assuming all applications are stateless leads to data loss when pods are rescheduled. Choosing the wrong storage class after data is in production means a painful migration.

Using unvetted base images from public registries

Pulling a popular image from Docker Hub because it has millions of downloads does not make it secure. Many public images run as root, bundle unnecessary packages, and go months without security updates.

Deploying AI-generated Dockerfiles without review

AI code assistants can generate functional Dockerfiles quickly, but they might select inappropriate base images, run as root, or miss security hardening steps. Always review AI-generated container configurations against your organization's security policies.

Red Hat OpenShift capabilities that accelerate progression

Curated runtimes and frameworks

[Red Hat Runtimes](#) provides supported, container-optimized runtimes for Java (Spring Boot, Quarkus), Node.js, Python, .NET, Go, and Ruby. They're tested on Red Hat OpenShift, regularly patched, and tuned for container environments. For organizations with existing Apache Tomcat-based applications, Red Hat JBoss Web Server provides a supported distribution that simplifies the lift-and-shift path. Quarkus deserves particular mention for organizations planning long-term modernization. Its fast startup times, low memory footprint, and native compilation make it purpose-built for containers and position teams well for the patterns described in later stages.

Red Hat Hardened Images

[Red Hat Hardened Images](#) are micro-sized, vendor-neutral container images containing only the packages necessary to run an application, with no shells, package managers, or unnecessary utilities that attackers could exploit. Each ships with a validated security profile and an integrated SBOM. Red Hat tracks upstream sources and targets a near-zero known vulnerability state at release, reducing scanner noise and accelerating audit cycles. Red Hat Hardened Images run on any Linux distribution,

container engine, or Kubernetes platform—not just Red Hat OpenShift—providing portability alongside security. When combined with Red Hat’s curated runtime images and [Red Hat Universal Base Images](#), Red Hat Hardened Images provide a layered approach to trusted base images for your workloads.

Red Hat build of Podman Desktop

The [Red Hat build of Podman Desktop](#) is a graphical interface for building, testing, and managing containers locally. Podman runs without a daemon and supports rootless containers with SELinux enforcement, mirroring the security posture of production Red Hat OpenShift environments. Developers can build images, inspect containers, view logs, manage registries, and deploy to local and remote Red Hat OpenShift clusters, iterating without waiting for cluster access or CI pipeline runs. The tool supports Open Container Initiative (OCI) and Compose standards, so existing Docker Compose files work without modification.

Red Hat OpenShift web console

The [OpenShift web console](#) is the unified interface for both developers and operations teams, and the primary onboarding tool for those new to Kubernetes. Developer views organize workloads by application, providing a topology view of components, services, routes, and builds without requiring CLI proficiency. Guided flows for common tasks, like deploying from Git, configuring health checks, and adding storage, embed platform team standards by default. Vulnerability scan results appear directly alongside running workloads.

Builds for Red Hat OpenShift

Based on the open source Shipwright project, [builds for Red Hat OpenShift](#) provide a Kubernetes-native API for building container images on the cluster, ensuring consistent, reproducible builds that follow organizational security policies. Three build strategies are supported out of the box: Buildah for Dockerfile-based builds, Source-to-Image (S2I) for developers who want to provide source code and let the platform handle containerization, and Cloud-Native Buildpacks for automated framework detection. Platform teams can enforce base image policies, inject credentials securely, and produce multiarchitecture images from a single build definition. The integrated experience in the OpenShift web console makes it accessible to developers still learning container concepts.

Red Hat OpenShift observability and logging

Red Hat OpenShift includes an integrated [observability toolset](#) that captures metrics, logs, and traces from every workload, with no additional configuration required from application teams. This is critical at this stage because developers are transitioning from local debugging tools to a platform where containers are ephemeral and can be rescheduled at any time. The monitoring architecture is built on Prometheus, and LokiStack backed by Vector provides logging, capturing every container’s stdout and stderr automatically. The Red Hat build of OpenTelemetry and Tempo provide distributed tracing, and Korrel8r connects metrics, logs, alerts, and traces to help teams identify root causes without switching tools.

Migration toolkit for applications

[Red Hat’s migration toolkit for applications](#) (MTA) helps organizations assess and prioritize which applications to containerize. Its assessment module uses extensible questionnaires to evaluate applications across technical complexity, business criticality, and containerization readiness, replacing gut-feel estimates with structured, evidence-based analysis. The analysis engine scans source code and binaries to identify specific migration issues and maps each to a migration path: containerization, JBoss

EAP migration, Spring Boot to Quarkus, or Oracle JDK to OpenJDK. MTA's intelligent assistant adds AI-assisted refactoring in the IDE, combining static code analysis findings with a knowledge base of past migrations to generate contextually accurate code fixes. Developers review explanations, inspect code diffs, and apply fixes with a single click. An agent mode automatically detects new issues introduced by code changes, helping preserve application functionality throughout the migration.

Red Hat OpenShift Virtualization

Not every workload can be containerized on Day 1. [Red Hat OpenShift Virtualization](#) runs traditional VMs alongside containers on the same cluster, managed through the same console, CLI, and operational workflows. There is no separate infrastructure to maintain and no operational separation between virtualized and containerized workloads. VMs and containers share networking and storage, so a virtualized database can serve a containerized front-end without crossing infrastructure boundaries. Over time, components can be extracted into containers incrementally using the [strangler fig pattern described in Chapter 5](#), turning modernization from a single migration into a gradual progression.

Secrets management

Red Hat OpenShift provides a layered secrets management portfolio that integrates with enterprise vault systems, giving organizations flexibility to match their compliance requirements.

- ▶ [External Secrets Operator for Red Hat OpenShift](#) synchronizes credentials from external vaults, like [HashiCorp Vault](#), AWS Secrets Manager, Azure Key Vault, and CyberArk, managing the full lifecycle including automatic secrets rotation.
- ▶ [Secrets Store CSI Driver Operator](#) mounts secrets directly from the vault into pods as in-memory tmpfs volumes, bypassing the Kubernetes API entirely. This is the strongest option for organizations whose compliance rules mandate credentials remain exclusively in a central store.
- ▶ [cert-manager automates](#) TLS certificate lifecycle, generation, renewal, and distribution across certificate authorities including internal PKI, Let's Encrypt, and Venafi.

OpenShift APIs for Data Protection

[OpenShift APIs for Data Protection](#) provide application-level backup and restore built on the open source Velero project. It backs up Kubernetes resources and persistent volume data, which allows recovery from accidental deletion, cluster migration, or disaster scenarios. As organizations move stateful applications onto the platform for the 1st time, this provides the safety net that gives teams confidence.

Red Hat OpenShift Data Foundation

A software-defined storage platform, [Red Hat OpenShift Data Foundation](#) provides persistent block, file, and object storage on the same cluster as your applications. Built on Ceph, Rook, and NooBaa, it provides a consistent storage interface across bare metal, private cloud, and public cloud. For workloads requiring the lowest latency, the [Local Storage Operator](#) provisions local nonvolatile memory express (NVMe) or solid state drive (SSD)-backed volumes. Red Hat OpenShift also integrates with a broad ecosystem of certified storage partners via CSI, including enterprise vendors such as NetApp, Pure Storage, Dell, Fujitsu, Hitachi, and IBM, and native cloud provider storage on AWS, Azure, and Google Cloud.

Customer spotlight: Advent One

“We have reduced patching times from 2 hours to just 20 minutes.”

Talor Holloway
CTO, Advent One²

Advent One was managing a fragmented mix of VMs across its hybrid IT environment, facing rising virtualization costs and growing operational overhead that made day-to-day infrastructure management increasingly difficult. By adopting Red Hat OpenShift Virtualization, along with Red Hat Ansible® Automation Platform and Event-Driven Ansible, Advent One unified management of its containers and VMs, cut its VM count by up to 50%, and built a proven, repeatable blueprint it now uses to guide clients through their own modernization journeys.²

Customer spotlight: Emirates NBD

“Automating deployments with Argo CD slashed release times from 45 minutes to 3, accelerating our digital services.”

Puneet Sharma
Platform Team,
Emirates NBD³

Emirates NBD faced the challenge of modernizing its infrastructure to power digital banking services across 13 countries, while keeping legacy systems security-compliant and running without disruption. By deploying Red Hat OpenShift on Oracle Cloud Infrastructure and embracing GitOps, Argo CD, and OpenShift Virtualization, the bank unified VMs and containers on a single platform. Release windows dropped 94%, from 45 minutes to just 3.³

Moving to the next stage

The patterns in this chapter establish the operational foundation for everything that follows. Before advancing to Stage 2, verify that your containerized applications meet these criteria:

- ☐ Health check endpoints are implemented for both liveness and readiness.
- ☐ Logs are written to stdout/stderr and captured by the platform’s logging system.
- ☐ Configuration is externalized through ConfigMaps and external secrets management.
- ☐ Persistent storage is provisioned for stateful workloads.
- ☐ Container images are built from curated, hardened base images.
- ☐ Security contexts are configured to run as nonroot with minimal privileges.

Once these foundations are in place, you are ready to build the operational maturity described in [Chapter 4](#).

² Red Hat case study. [“Advent One modernizes IT with Red Hat OpenShift Virtualization.”](#) February 2026.

³ Red Hat customer video. [“Red Hat and Emirates NBD, a journey to Hybrid Cloud.”](#) OpenShift Commons Denver, May 2024.

Chapter 4:

Stage 2—Operationalize



Zero-downtime deployments and operational efficiency	30
Organization maturity at this stage	30
Application patterns in depth:	31
Observability instrumentation	31
Autoscaling on application metrics	31
Graceful shutdown	32
Kustomize overlays	33
Helm chart value overlays	33
Operator pattern	34
Security at this stage	35
Common mistakes to avoid	36
Red Hat OpenShift capabilities that accelerate progression	37
Customer spotlights: Centris AG, SWIFT	39



Zero-downtime deployments and operational efficiency

Your applications are running in containers. The platform is operational. Now comes the stage that separates proof-of-concept deployments from production-grade operations.

Organizations at this stage are **platform adopters**, focused on operational maturity, including comprehensive observability, automated scaling, zero-downtime deployments, and standardized delivery pipelines. The goal is to remove manual overhead and ensure workloads are resilient, observable, and efficiently managed.

This is where the platform begins to pay dividends. A deployment that drops connections costs revenue. An application that cannot scale with demand costs customers. Poor observability that turns a 10-minute diagnosis into a multihour incident erodes organizational trust.

The shift is fundamentally from manual to automated operations. In Stage 1, teams deployed, watched dashboards, and scaled by hand. In Stage 2, the platform makes those decisions, scaling on real-time demand, rolling out updates without dropping requests, and alerting only when human judgment is needed.

Organization maturity at this stage

Maturity at this stage consists of:

- ▶ **Multiple production workloads** running on Red Hat OpenShift. The platform is no longer experimental; it's a critical part of production infrastructure.
- ▶ **A platform team focused on operational efficiency**, standardization, and reducing toil. Self-service capabilities are introduced so development teams can deploy independently.
- ▶ **Developers who are increasingly autonomous**, using standardized pipelines and deployment templates rather than manual processes.
- ▶ **A growing need for standardization and governance** as more teams adopt the platform. Without standards, each team invents its own deployment process, leading to inconsistency and risk.
- ▶ **Pressure to reduce operational costs and toil** as the platform scales. Manual processes that worked for 5 applications break down at 50.

Application teams are typically asking: How do we proactively identify issues before they affect users? How do we handle application updates without downtime? How do we scale applications based on actual demand? Should we manage databases ourselves or use managed services? How do we standardize deployments across teams and environments?

The key challenges at this stage are inconsistent deployment practices across teams, reactive rather than proactive incident response, manual scaling that leads to over-provisioning or under-provisioning, and the difficulty of maintaining multiple environment configurations.

Application patterns in depth

Observability instrumentation

Observability is not the same as monitoring. Monitoring tells you when something is wrong; observability tells you why and where to look. Applications at this stage should emit 3 types of telemetry: metrics (request rates, error rates, response latencies, queue depths—instrumented via OpenTelemetry or Prometheus client libraries) that power dashboards, alerts, and autoscaling decisions; logs that are now structured using JavaScript Object Notation (JSON), include correlation IDs for request tracing, and are forwarded to centralized logging; and traces that follow a request across service boundaries. Even in a monolith, adding tracing now prepares you for the microservices architecture in Stage 3, where understanding inter-service dependencies becomes essential.

The key principle is that instrumentation should help you take action. Every metric should tie to an alert or a scaling decision. Every log entry should provide enough context to diagnose an issue. A practical approach is defining Service Level Objectives (SLOs), for example, “99.9% of requests complete within 500 ms,” and instrumenting the environment to measure and alert against those objectives. SLO-based alerting focuses attention on what matters to users rather than arbitrary thresholds, reducing alert fatigue while improving detection accuracy. This groundwork also prepares you for service mesh observability in [Chapter 6](#), where the mesh captures request-level metrics, traces, and service topology automatically.

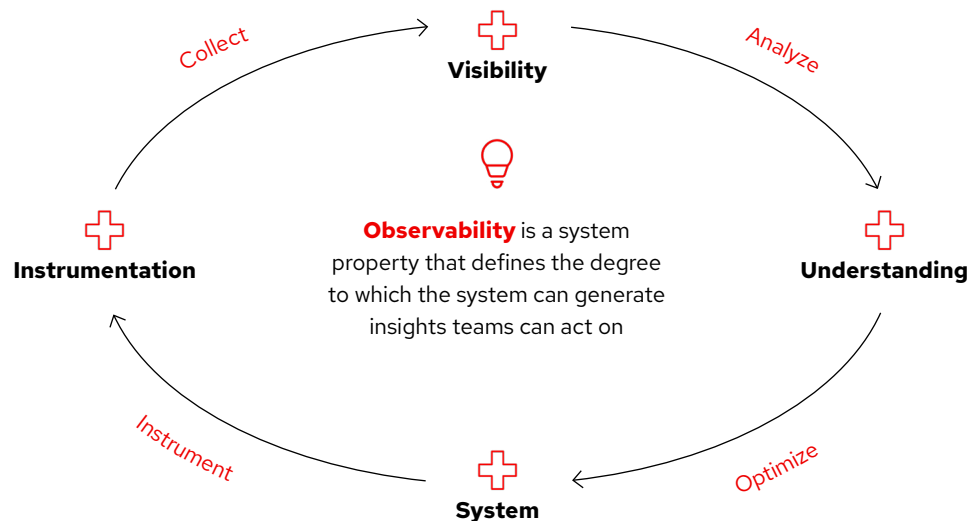


Figure 12. Observability pillars.

Autoscaling on application metrics

Kubernetes provides the [Horizontal Pod Autoscaler \(HPA\)](#) for scaling applications based on resource use. At Stage 1, teams typically scale manually or use basic CPU-based autoscaling. At Stage 2, scaling should be based on application-specific metrics that reflect actual user experience. Consider an e-commerce flash sale: CPU-based scaling is a lagging indicator; by the time CPU spikes, users are already experiencing latency. Scaling on request queue depth or response latency is a leading indicator that triggers before users notice degradation. That’s the difference between managing incidents and preventing them.

An event-driven autoscaler exposes your application's custom metrics, such as request rate, queue depth, and active connections, to the HPA, adjusting pod replicas as demand changes. For event-driven workloads, this same mechanism can scale based on external event sources like Kafka topic lag, message queue depth, including scaling to 0 for workloads with intermittent traffic. This is a meaningful cost optimization for batch processing or services with limited active hours. [Chapter 6](#) explores this in depth as the foundation of an entire architectural approach.

Effective autoscaling also requires setting appropriate resource requests and limits on pods. Without them, the scheduler cannot make intelligent placement decisions, and scaling might over- or under-provision capacity.

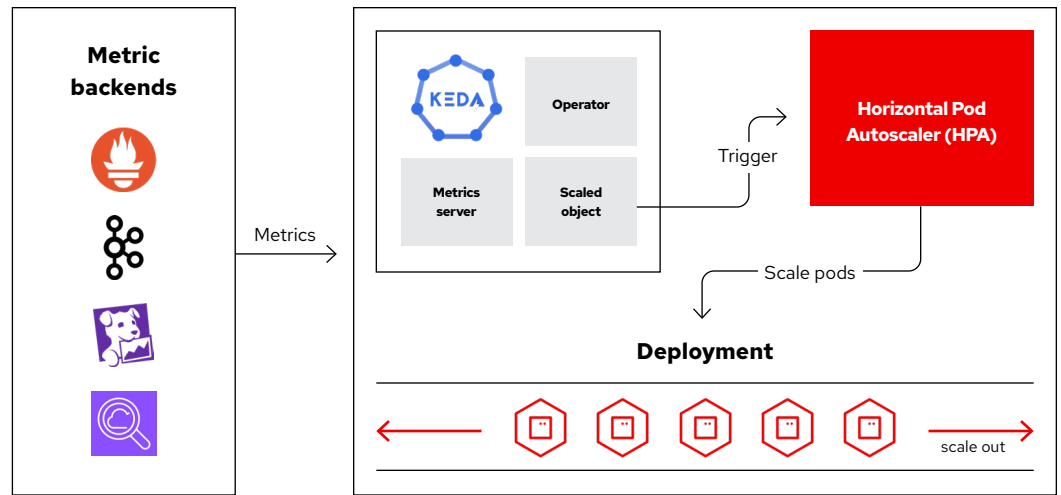


Figure 13. Kubernetes Event-Driven Autoscaling (KEDA) pods based on external metrics (Prometheus, Kafka, Datadog, and others).

Graceful shutdown

[Zero-downtime deployments](#) require applications that shut down gracefully. When Kubernetes terminates a pod, during a rolling update, scale-down, or node drain, it sends a SIGTERM signal to the application and waits for a configurable grace period (default 30 seconds) before sending SIGKILL, which forcibly terminates the process.

A well-behaved application responds to SIGTERM by: stopping acceptance of new requests (the readiness probe should immediately start failing, removing the pod from the load balancer); completing in-flight requests (waiting for active HTTP requests to return, or finishing the current message and committing its offset); closing connections cleanly (database connections, write buffers, file locks, service registry deregistration); and exiting with a zero exit code to signal intentional shutdown.

A pre-stop hook adding a short delay (5-10 seconds) before SIGTERM gives the load balancer time to remove the pod from rotation, preventing the race condition where traffic is still being routed to a pod already shutting down. For long-running processes, like batch jobs, file uploads, and database transactions, increase the `terminationGracePeriodSeconds` so in-flight work isn't lost to a premature SIGKILL.

This pattern depends directly on the health check endpoints from [Chapter 3](#): readiness probes that fail immediately on SIGTERM are what prevent traffic from reaching a terminating pod. Without graceful shutdown, rolling updates cause dropped connections. This kind of disruption erodes confidence in the platform and pushes teams back to manual, scheduled deployment windows.

Kustomize overlays

As applications move across environments, configuration differences multiply. Replica counts, resource limits, image tags, environment variables, and ingress rules all change between environments. Without a structured approach, teams maintain separate manifest copies for each environment, leading to copy-paste errors and drift.

[Kustomize](#) manages these differences without duplicating manifests: a base configuration holds common elements, and overlays specify only what changes per environment. It's built into **kubectl** and **oc** CLIs, requires no additional tooling, and works with plain Kubernetes YAML. There's no templating language to learn, making it accessible to teams still building Kubernetes proficiency. Kustomize transformations cover the most common variations: patching fields, adding labels and annotations, generating ConfigMaps and Secrets from files, and setting image tags.

Kustomize is the recommended starting point for multienvironment configuration management at Stage 2. Teams needing more advanced capabilities, such as conditional logic, loops, or dependency management, might adopt Helm, but many organizations find Kustomize sufficient through Stage 3 and beyond.

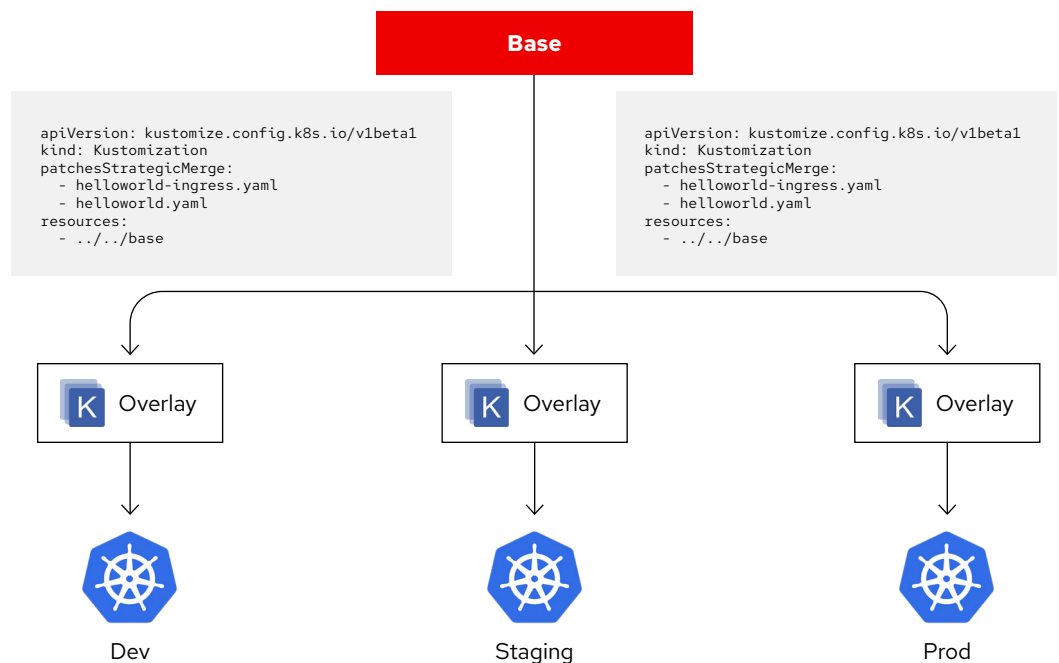


Figure 14. Environment-specific deployment manifests using Kustomize overlays.

Helm chart value overlays

For more complex applications, or organizations standardized on Helm, [value overlays](#) provide an alternative. A **base values.yaml** file holds defaults, and environment-specific files (**values-staging.yaml**, **values-production.yaml**) override specific settings. Helm adds several advantages:

- ▶ **Packaging** bundles all Kubernetes resources into a single, versioned artifact consumable by CI/CD pipelines.
- ▶ **Templating** via Go templates handles conditional logic and loops for complex configuration scenarios, such as conditional sidecars.

- ▶ **Dependency management** lets charts declare dependencies on other charts, managing a full configuration system as a single coordinated unit.
- ▶ A large **ecosystem** of community and vendor charts via OperatorHub provides prepackaged deployments for common infrastructure.

The trade-off is complexity. Helm's templating has a learning curve, and poorly structured charts become difficult to debug. Invest in chart linting, schema validation, and cross-environment testing before promotion. [Chapter 5](#) builds on this foundation with library charts for sharing common patterns across teams and umbrella charts for orchestrating multiservice deployments.

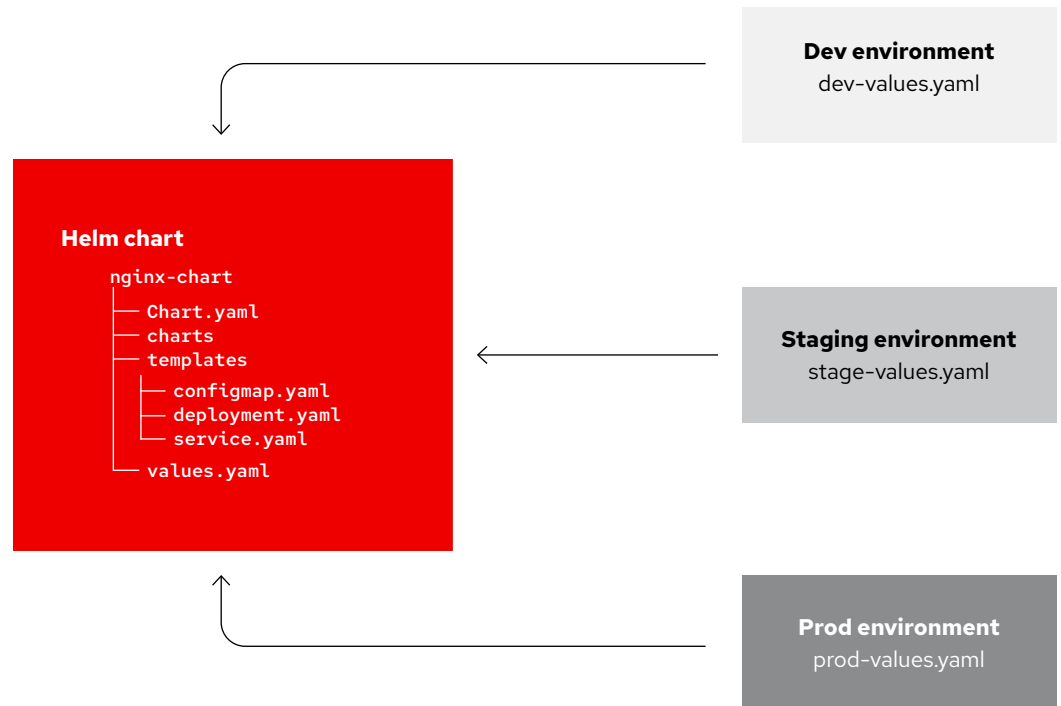


Figure 15. Environment-specific deployment manifests using a Helm chart and value overlays.

Operator pattern

[Kubernetes Operators](#) extend the platform with domain-specific operational knowledge. An Operator is a controller that watches for custom resources and reconciles desired state with actual state, the same pattern Kubernetes uses internally for pods and deployments. Where Kubernetes knows how to manage containers, an Operator knows how to manage a specific application: its backup schedule, failover behavior, scaling characteristics, and upgrade path.

At this stage, Operators are primarily used rather than authored. Installing an Operator for PostgreSQL, for example, gives you automated backup, failover, and scaling, without the manual effort and domain expertise that would otherwise require. This is where the external Database-as-a-Service pattern from [Chapter 3](#) can evolve. Teams now comfortable with the platform can bring database management on-cluster, gaining tighter integration with observability, security, and lifecycle management while retaining automated operations.

For organizations building custom Operators, the Operator Framework and [Operator Lifecycle Manager](#) (OLM) provide the tooling to build and manage them at runtime, encoding institutional runbooks and operational procedures into software that executes consistently.

Security at this stage

Security shifts from establishing foundations to embedding security into the delivery pipeline and enforcing runtime policies automatically, making security continuous rather than a point-in-time check.

Pipeline security

CI/CD pipelines must include security gates that scan images for vulnerabilities at build time, verify base image provenance through signatures, and run static analysis on application code. These gates should block promotion when vulnerabilities are detected rather than merely reporting them. Image signing with cosign or Sigstore establishes a chain of trust from build to deployment, and Red Hat OpenShift can be configured to reject unsigned images at admission time. Tekton Chains in [Red Hat OpenShift Pipelines](#) automatically sign every pipeline run and its resulting artifacts, making them verifiable.

Continuous image scanning

Images in the registry must be continuously rescanned for newly discovered vulnerabilities. [Red Hat Quay](#) scans every image on push and continuously rescans stored images as new CVEs are disclosed. [Red Hat Advanced Cluster Security for Kubernetes](#) extends this with policy-based enforcement, flagging or blocking running workloads that violate security policies, including vulnerability severity, exposed ports, privilege escalation, and network exposure. Together, they create a closed loop where every image in the registry and every workload in production is both trusted and continuously validated.

Role-based access control (RBAC) maturation

As more teams adopt the platform, access control must become more granular. Teams should have full control over their namespaces but limited access to shared resources and cluster-level configurations. Service accounts should follow least privilege, with each workload using a dedicated account rather than the namespace default. Audit logging tracks who accessed what and when, providing the accountability trail compliance teams require.

Centralized identity management

Managing authentication independently in each application becomes inconsistent as more applications move onto the platform, each with separate user stores, different password policies, and no single sign-on (SSO). [Red Hat build of Keycloak](#) provides centralized identity and access management with SSO via OpenID Connect and SAML, giving security teams a single place to enforce password policies, multifactor authentication, and identity federation with external providers like LDAP and Active Directory. This foundation becomes critical in [Chapter 5](#), where microservices decomposition multiplies the number of services requiring consistent authentication.

Network segmentation

Implement Kubernetes [NetworkPolicies](#) to restrict pod-to-pod communication. By default, every pod in a Kubernetes cluster can communicate with every other pod, an open network that lets a compromised workload reach any service on the cluster. NetworkPolicies enforce deny-by-default, where pods can only communicate with explicitly permitted endpoints.

Common mistakes to avoid

Implementing observability without alerts you can act on

Every alert should have a clear owner, a runbook, and a defined threshold for paging versus waiting for business hours. Teams that instrument everything but define no SLOs end up with hundreds of unreviewed alerts and a monitoring system that is effectively ignored. Start with a small number of SLO-based alerts tied to user-facing outcomes, availability, latency, error rate, and expand from there. Five meaningful alerts serve a team better than 500 unreviewed ones.

Causing request failures during deployments

Rolling updates without proper readiness probes, graceful shutdown, and pre-stop hooks cause dropped connections. The most common failure is a race condition where the load balancer routes traffic to an already-terminating pod. Test deployment strategy under realistic load, not just with synthetic health checks. For higher-risk deployments, Argo Rollouts provides canary and blue-green strategies that shift traffic gradually and automatically roll back if metrics degrade.

Scaling only on CPU and memory instead of application metrics

Infrastructure metrics are lagging indicators; application-specific metrics like latency and queue depth are leading ones. Configure the Custom Metrics Autoscaler to scale on what actually reflects user experience, and set accurate resource requests and limits so the scheduler can make intelligent decisions.

Leaving default network policies open

Without NetworkPolicies, a single compromised container can reach any database or service on the cluster. Retrofitting deny-by-default policies onto a running cluster with dozens of services is far harder than establishing them early. Start by isolating namespaces, then refine policies within them.

Skipping security gates in CI/CD pipelines

Treating scanning and signature verification as optional under delivery pressure means vulnerabilities reach production. A vulnerability blocked at build time costs minutes to fix; the same vulnerability in production costs an incident response.

Allowing each team to build its own deployment process

Without standardized pipelines, every team invents its own approach, with different tools, promotion strategies, and configurations. This works with 3 teams but collapses at 30, leaving no common playbook when an incident occurs. Internal developer portals provide Golden Path templates that encode standards into self-service workflows without removing team autonomy.

Trusting AI-generated pipeline and chart definitions without testing

AI agents can generate syntactically valid Helm charts and CI pipelines that fail in specific environments due to missing value overrides or incorrect resource assumptions. Validate all AI-generated definitions across target environments and put them through the same linting and policy checks as human-authored configurations.

Red Hat OpenShift capabilities that accelerate progression

Red Hat OpenShift Pipelines and Builds

[Red Hat OpenShift Pipelines](#) provides Tekton, Kubernetes-native CI/CD that runs as pods on the cluster, declarative, version-controlled, reusable across teams, and with the same isolation and scheduling as application workloads. Each step runs in its own container with its own security context, and pipeline execution produces verifiable provenance metadata. Red Hat uses Tekton as the backbone for its Supply-chain Levels for Software Artifacts (SLSA) Build Level 3 pipelines, ensuring tamper-resistant build provenance. Tekton Chains automates signing of runs and resulting images. Red Hat's curated catalog of prebuilt tasks (for image building, vulnerability scanning, and signing) gives platform teams a trusted starting point to extend with organization-specific shared tasks. Combined with Argo CD, Tekton pipelines form the foundation of a fully automated, auditable software supply chain.

Red Hat OpenShift GitOps

Argo CD keeps cluster state synchronized with Git repositories, continuously reconciling the cluster to match the desired state. Each change is a Git commit, providing a complete audit trail. [Red Hat OpenShift GitOps](#) extends Argo CD with complementary components. [Argo Rollouts](#) adds canary and blue-green deployment strategies; [ApplicationSets](#) generate Argo CD Applications across clusters or teams from a single template; [Argo CD Notifications](#) delivers real-time deployment status to Slack, email, or webhooks; [Argo CD Image Updater](#) watches registries and updates Git automatically when new image versions appear. Together, this solves the configuration drift problem that emerges as more teams deploy to the platform. Without GitOps, ad-hoc `kubectl apply` changes accumulate and cluster state diverges from what teams believe is deployed.

Red Hat Developer Hub

Based on the Backstage project, [Red Hat Developer Hub](#) is a self-service developer portal that provides software templates, a service catalog, and integrated documentation. Templates encode Golden Paths so developers can scaffold new projects with organizational standards built in from the start. Developers select a template, fill in parameters, and receive a fully wired project with source repository, CI pipeline, Argo CD application, and observability configuration. The service catalog gives a centralized view of all services, owners, and dependencies as the platform grows.

Custom Metrics Autoscaler and Red Hat OpenShift Serverless

The [Custom Metrics Autoscaler](#), based on Kubernetes Event-Driven Autoscaling (KEDA), extends Kubernetes autoscaling beyond CPU and memory to application-specific metrics and external event sources. It integrates with Prometheus and supports more than 60 event-source scalers (Kafka, RabbitMQ, Amazon Simple Queue Service (AWS SQS), cron schedules), including scaling to 0. [Red Hat OpenShift Serverless](#) provides a developer-focused experience for building event-driven autoscaled functions and containers directly from code. Together they transform scaling from manual and reactive into automated and metrics-based, reducing over-provisioning while preventing user-facing degradation.

OpenShift distributed tracing and Red Hat build of OpenTelemetry

The [Red Hat build of OpenTelemetry](#) provides a vendor-neutral, standards-based collection of traces, metrics, and logs through the OpenTelemetry Collector. It runs as a sidecar or daemonset, receiving telemetry from instrumented applications (via SDKs for Java, Node.js, Python, Go, and .NET) and forwarding it to Tempo for scalable storage, with the Red Hat OpenShift console providing integrated

visualization correlated with metrics and logs. Adding [distributed tracing](#) now (even for monoliths) is an investment in future readiness because the infrastructure will already be in place when services are decomposed in Stage 3.

Red Hat Advanced Cluster Security for Kubernetes

[Red Hat Advanced Cluster Security for Kubernetes](#) provides continuous security across build, deploy, and runtime. It scans images, evaluates deployments against policy, monitors runtime behavior for anomalies, and provides fleet-wide network segmentation visibility from a single console. It integrates directly into Tekton pipelines to block builds with critical vulnerabilities and monitors running workloads for unexpected process execution or network connections, replacing periodic manual security review with continuous, automated security assessment.

Red Hat Quay

An enterprise container image registry, [Red Hat Quay](#) provides built-in vulnerability scanning on every push and continuous rescanning as new CVEs are disclosed, plus RBAC, image signing, repository mirroring, and audit log storage. The [Quay Bridge Operator](#) integrates Quay directly with Red Hat OpenShift; the [Container Security Operator](#) surfaces Clair vulnerability scan results in the Red Hat OpenShift web console, showing which running pods use vulnerable images and what remediation is required. Combined with [Tekton Chains](#) for image signing and [Red Hat Advanced Cluster Security](#) for admission policy enforcement, Quay completes the secure software supply chain from build to runtime.

Cost management

The [Cost Management Metrics Operator](#) provides cost visibility and analysis across hybrid cloud and multicloud environments, aggregating spending across AWS, Azure, Google Cloud, and on-premise clusters and correlating it to cluster and namespace consumption. Tag-based cost mapping associates spending with clusters, projects, or applications, giving platform teams the data for showback reporting and accountability.

Red Hat Advanced Cluster Management for Kubernetes

As the number of clusters grows across environments and regions, managing them individually becomes unsustainable. [Red Hat Advanced Cluster Management for Kubernetes](#) provides a single control plane for the fleet: provisioning, upgrading clusters with controlled rollout policies, and decommissioning across bare metal, private cloud, and public cloud. It delivers multicloud observability by aggregating metrics, alerts, and health status from every cluster into a single console, and delivers policy-based governance that enforces security configurations, quotas, and required operators consistently, with new clusters automatically inheriting baseline configuration.

Red Hat Lightspeed

[Red Hat Lightspeed](#) is an AI-powered system management tool that uses predictive analytics to identify risks, optimize performance, and monitor subscriptions across Red Hat OpenShift clusters. It surfaces recommendations like pods with resource requests far above actual use, nodes with low resource use, or security vulnerabilities that need remediation. Combined with cost management data, Lightspeed helps platform teams make data-driven decisions about capacity planning and right-sizing.

Customer spotlight: Centris AG

“Building an automated deployment pipeline with the Red Hat OpenShift operators has significantly accelerated our time to market, while the increased stability gives developers the confidence to deliver releases more regularly and predictably.”

David Studer

Product Owner,
Centris AG⁴

Centris was managing more than 40 environments across highly customized customer configurations, making release coordination complex and slowing time to market. By adopting Red Hat OpenShift Pipelines and Red Hat OpenShift GitOps, alongside Red Hat Developer Hub, Centris moved from manual, highly coordinated releases to a fast, reproducible deployment process, reducing time to market by 75% and increasing release frequency by 300%.⁴

Customer spotlight: SWIFT

“Our zero trust, zero-privilege approach to OpenShift fleet management, combined with GitOps automation using Argo CD, allows us to deliver software faster, securely, and at scale across our air-gapped environments.”

Praveen Siddu

Senior Enterprise Architect,
SWIFT⁵

SWIFT’s financial messaging network underpins millions of daily transactions across more than 11,000 institutions worldwide. Facing the dual challenge of accelerating delivery while maintaining strict security and regulatory compliance across air-gapped environments, SWIFT adopted a zero trust, zero-privilege security model built on Red Hat OpenShift, and automated cluster provisioning with GitOps, Tekton, Argo CD, and Red Hat Advanced Cluster Management. Deployment times dropped from weeks to minutes, multicluster fleet management brought consistency across datacenters, and operational efficiency improvements delivered 7-figure financial savings.⁵

⁴ Red Hat case study. [“Centris uses Red Hat OpenShift and Red Hat Developer Hub to build a modern developer portal.”](#) 23 June 2026.

⁵ Red Hat customer video. [“Disconnected OpenShift, A zero-touch provisioning approach with SWIFT.”](#) OpenShift Commons Salt Lake City, 15 Nov. 2024.

Moving to the next stage

The operational patterns in this chapter transform your container platform from a deployment target into a mature operational environment. Before advancing to Stage 3, verify that your operations meet these criteria:

- ☐ Applications emit metrics, structured logs, and traces.
- ☐ Alerts let you take action, and are tied to runbooks.
- ☐ Autoscaling is based on application metrics, not just CPU and memory.
- ☐ Deployments achieve zero downtime through graceful shutdown and readiness probes.
- ☐ CI/CD pipelines are standardized across teams using Tekton and Argo CD.
- ☐ Environment-specific configuration is managed through Kustomize or Helm overlays.
- ☐ A developer portal provides self-service capabilities for common operations.

With these operational foundations in place, you're ready to tackle the architectural challenge of Stage 3: modernizing monolithic applications into microservices.

Chapter 5:

Stage 3—Modernize



Keep pace with business priorities	42
Organization maturity at this stage	42
Application patterns in depth:	43
Cloud-native frameworks (Quarkus, Spring Boot)	44
API gateway	44
Service discovery	45
Sidecar pattern	45
Library chart	46
Helm umbrella chart	46
External cache	46
Security at this stage	47
Common mistakes to avoid	48
Red Hat OpenShift capabilities that accelerate progression	49
Customer spotlights: Dutch Tax Authority, Tel Aviv Stock Exchange	51



Keep pace with business priorities

The initial 2 stages built a solid operational foundation: applications are containerized, observable, automatically scaled, and deployed through standardized pipelines. Stage 3 is the architectural shift that allows teams to fully achieve the agility of a cloud-native approach: decomposing monolithic applications into microservices.

Organizations at this stage are cloud-native practitioners, actively moving from monolithic to service-based architecture. Multiple teams work on independent services, and the organization needs service governance, API management, and sophisticated deployment orchestration to manage the growing complexity.

The business case for modernization is velocity. Monolithic applications force every change through a single deployment pipeline; a small UI fix and a major database migration ship together or not at all. When the payment team cannot deploy a fraud detection update until the inventory team finishes testing an unrelated change, the organization is paying an architectural tax on every release. Microservices eliminate this coupling, as independent teams deploy independent services on independent schedules aligned with business priorities rather than release calendar constraints.

Velocity is not the only factor. Microservices allow for granular scaling, a checkout service under load scales independently without provisioning capacity for the entire application. They support technology diversity, so teams can choose the right language and framework for each service. They also allow fault isolation, so a failure in 1 service does not cascade across the application. These benefits compound as the organization grows.

Organization maturity at this stage

Maturity at this stage consists of:

- ▶ **Moving from monolithic to microservices architecture**, identifying bounded contexts within monoliths and extracting them into independently deployable services.
- ▶ **Multiple teams working on independent services**, with organizational structure evolving toward a team-per-service or domain-team model.
- ▶ **A need for service governance and API management**. As the number of services grows, managing dependencies, API versions, and communication standards becomes critical.
- ▶ **Complex deployment orchestration requirements** with canary releases and rollback capabilities that go beyond simple rolling updates.
- ▶ **A platform team providing self-service capabilities** through developer portals, software templates, and Golden Paths.

Application teams at this stage are typically asking: How do services discover and communicate with each other? How do we manage traffic routing and API versioning? How do we decompose the monolith without a big-bang rewrite? How do we handle caching for performance? How do we ensure each service can evolve independently?

Application patterns in depth

Modernizing the monolith

The [strangler fig pattern](#) is a well established, lower risk approach to decomposing a monolith. Named after the tropical fig tree that gradually envelops its host tree, this pattern extracts functionality from the monolith incrementally, 1 bounded context at a time, while keeping the monolith running. The approach:

1. **Identifies a bounded context**, a coherent set of functionality with clear data ownership and well-defined interfaces. Good starting candidates have frequent change requirements, clear data boundaries, or distinct scaling needs.
2. **Builds the new service** alongside the monolith, implementing the bounded context's functionality from scratch using cloud-native patterns.
3. **Routes traffic** to the new service using an API gateway or reverse proxy; the monolith still handles everything else.
4. **Migrates data** from the monolith's database to the new service's data store. Often the most challenging step, it requires careful planning to avoid data inconsistencies.
5. **Removes the old code** once the new service is proven in production.
6. **Repeats** for the next bounded context.

The strangler fig pattern reduces risk because the monolith continues to function throughout the migration. If a new service has problems, traffic can be routed back. Each extraction is an independent, reversible step that builds team confidence and reduces the monolith's scope. Over time, the monolith shrinks until it can be retired entirely or reduced to a thin routing layer. This is the natural evolution of the containerized monolith pattern from [Chapter 3](#).

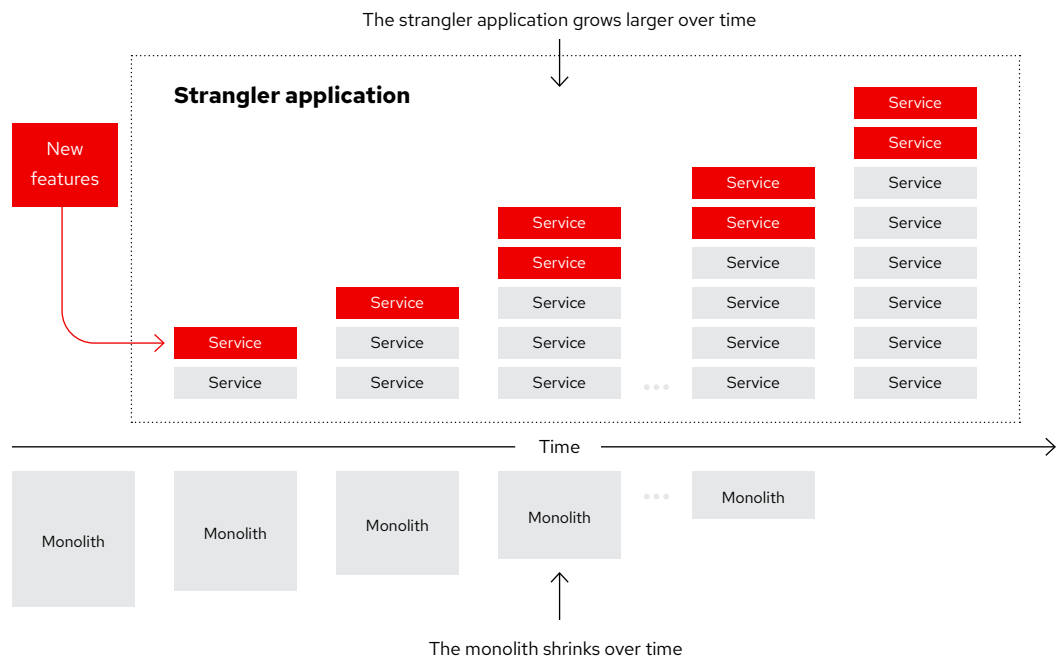


Figure 16. Strangling the monolith over time into a services-based architecture.

Cloud-native frameworks

The choice of framework shapes development velocity, resource consumption, and operational behavior, differences that compound as the number of deployed services grows.

[Quarkus](#) is a Kubernetes-native Java framework for containers. It provides fast startup times (often under 1 second) and lower memory consumption compared to traditional Java frameworks. Build-time optimization moves work that traditional frameworks do at runtime (classpath scanning, reflection, proxy generation) to the build phase. In a microservices environment where dozens or hundreds of instances run simultaneously, the aggregate resource savings are substantial. Quarkus supports both imperative and reactive programming models with first-class support for Hibernate, RESTEasy, and Vert.x, and includes dev mode with live reload for rapid iteration.

[Spring Boot](#) remains a widely adopted Java framework for microservices. Its extensive ecosystem and large developer community mean most Java developers are already familiar with it, making adoption straightforward for organizations with existing Spring expertise. The choice typically comes down to resource efficiency and startup speed (Quarkus) versus ecosystem and breadth (Spring Boot). Both are available as curated, regularly patched runtimes on Red Hat OpenShift.

API gateway

Without a unified entry point, external clients must know the location and protocol of every individual service, a coupling that becomes unmanageable as the architecture evolves. An API gateway sits at the edge of your service architecture and provides a single entry point for external traffic, decoupling clients from the internal service topology.

The gateway handles: routing (directing requests based on path or headers), authentication and authorization (validating tokens before requests reach backend services), rate limiting and throttling (protecting backend services from spikes and abuse), API versioning (routing traffic to the appropriate version so services can evolve without breaking consumers), and protocol translation (converting between external protocols and internal protocols). It also provides a natural point for API analytics, understanding which APIs are most used, which consumers generate the most traffic, and which endpoints have the highest error rates.

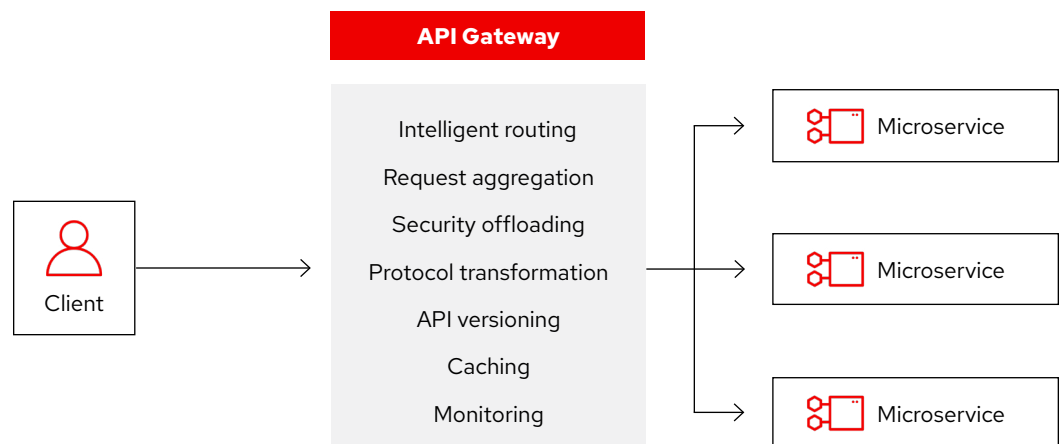


Figure 17. API gateway pattern.

Service discovery

In a monolith, function calls happen within a single process. In microservices, those same calls become network requests between services running on different nodes, scaling dynamically, and deploying independently. Kubernetes provides built-in service discovery through DNS. Each service gets a stable DNS name that other services can use to communicate, with traffic automatically routed to healthy pods, regardless of where they're running.

For more advanced needs, like load balancing across service versions, canary routing, and cross-cluster resolution, a service mesh provides a comprehensive solution. The mesh maintains a service registry and distributes routing information to sidecar proxies without application code changes. [Chapter 6](#) covers the service mesh pattern in depth.

The key principle: never hard code service URLs or IP addresses. Services are ephemeral; they scale, restart, and move between nodes. Let the platform handle discovery, and your services will remain resilient to the dynamic nature of the environment.

Sidecar pattern

The sidecar pattern attaches a secondary container to the application pod to provide supporting functionality, such as logging, monitoring, security, or networking, without modifying the application itself. Both containers share the same network namespace and can share volumes, making them tightly coupled at the infrastructure level while remaining loosely coupled at the code level.

A common sidecar is the service mesh proxy (Envoy), which intercepts all inbound and outbound traffic, enabling mutual TLS (mTLS) encryption, traffic routing, retry logic, and distributed tracing without a single line of application code change. Other common sidecars include log forwarders that transform application logs into structured formats, secret agents that refresh credentials from external vaults without restarting the application, and configuration reloaders that watch for ConfigMap changes and trigger dynamic updates.

The sidecar pattern is effective because it separates infrastructure concerns from application logic. The application team focuses on business logic, and the platform team manages the sidecars. Infrastructure capabilities can be upgraded or reconfigured across all services simultaneously by updating the sidecar, without touching any application code or coordinating releases across application teams.

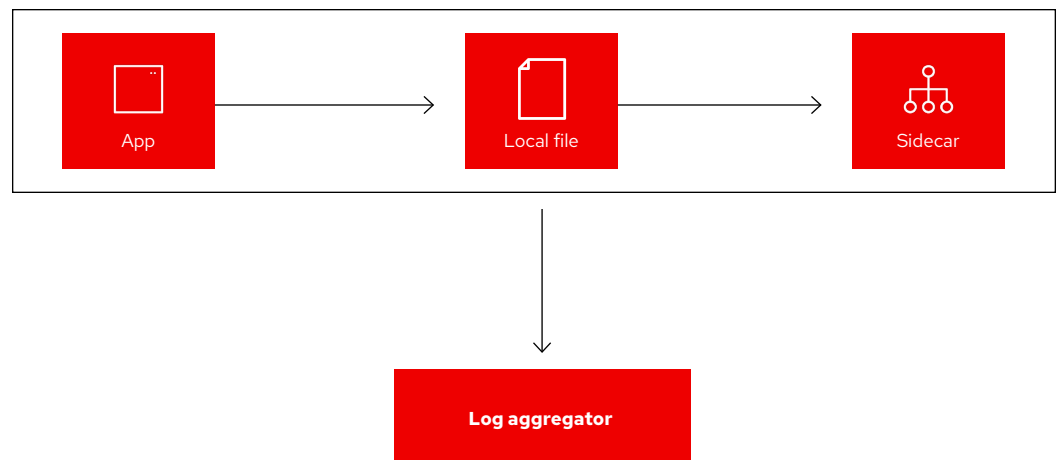


Figure 18. Sidecar forwarding logs from a local log file to a centralized log system.

Library chart

As [Helm](#) charts multiply across teams, shared configuration patterns emerge. Every service needs the same label conventions, security contexts, monitoring annotations, and resource limit structures. Without a sharing mechanism, each team implements these independently, introducing inconsistencies and making organization-wide updates difficult.

A library chart contains reusable template definitions but produces no Kubernetes resources of its own. Other charts import it as a dependency and call its templates, inheriting standardized patterns without duplicating code. Library charts are valuable for standardizing deployment boilerplate, service and ingress configurations, ConfigMap and Secret patterns, and Prometheus annotations. When the platform team updates the standard once, adding a new required label, changing a default security context, all consuming charts inherit the change on their next deployment. Library charts are a key tool for scaling governance as services grow.

Helm umbrella chart

A Helm umbrella chart declares other charts as dependencies, providing a single deployment artifact for a complete application architecture (application, database, cache, message broker) with a unified configuration interface. A single `helm install` deploys the entire configuration in the correct order with compatible versions, with a unified `values.yaml` configuring all components and passing shared values (like database connection strings) to multiple sub-charts.

The trade-off is coupling. An umbrella chart ties component versions together, which can slow teams that deploy services independently. If the frontend team needs to ship a fix but the umbrella chart pins a specific backend version, it becomes a bottleneck. Use umbrella charts for tightly coupled service groups that genuinely deploy together, not for entire application portfolios. For loosely coupled services that deploy independently, use Argo CD ApplicationSets or individual Helm releases.

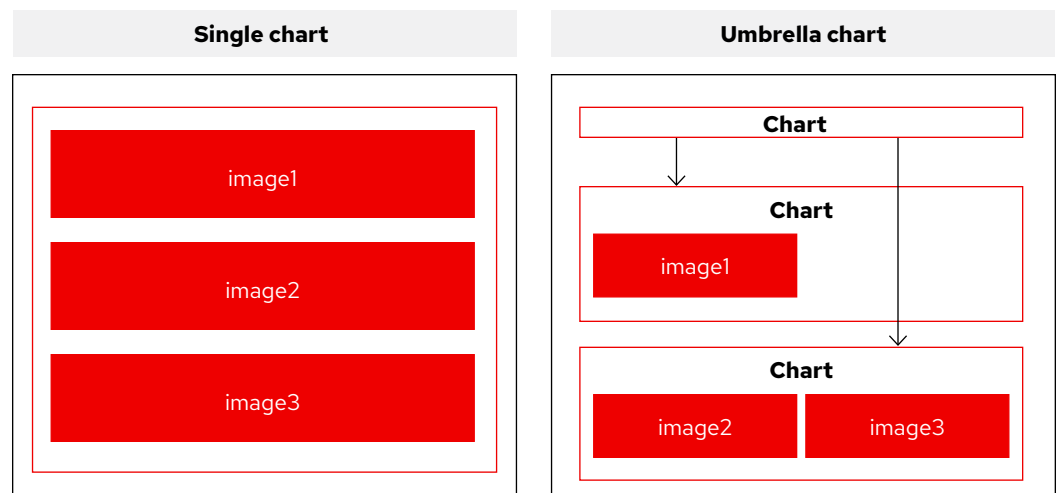


Figure 19. Single Helm chart vs. an umbrella chart.

External cache

As traffic grows and microservices multiply, each service making independent database queries for frequently accessed data creates redundant load that slows response times and can overwhelm the database during traffic spikes. An external cache serves frequently accessed data from memory, reducing database round trips and improving response times.

Unlike pod-local caches, an external distributed cache survives restarts, is shared across all service instances, and scales independently. This is critical in a microservices environment where pods are ephemeral and frequently rescheduled. Three considerations matter: invalidation (time-based expiration is simple but can serve stale data; event-based invalidation is more accurate but adds complexity); warming (populate critical caches before accepting traffic, using readiness probes to gate until loaded); and resilience (fall back to direct database queries if the cache is unavailable—it's an optimization, not a hard dependency).

Security at this stage

Microservices multiply the attack surface. Instead of a single application, you now have dozens of services communicating over the network, each with its own dependencies, container images, and deployment pipelines. Security at this stage extends the pipeline gates and continuous scanning from [Chapter 4](#) to cover the entire software supply chain and every service-to-service interaction.

Software supply chain security

Every container image running in production should be cryptographically signed and accompanied by provenance attestations verifying it was built by a trusted pipeline from trusted source code. [Red Hat Trusted Artifact Signer](#) provides this capability integrated with Red Hat OpenShift, signing every image and generating attestations automatically without requiring teams to manage signing keys or infrastructure. If configured to do so, admission policies in Red Hat Trusted Artifact Signer are designed to be rejected if they are not verified.

Software composition analysis

As services multiply, so does the aggregate dependency tree and exposure to third-party vulnerabilities. [Red Hat Trusted Profile Analyzer](#) provides continuous software composition analysis, scanning dependencies against known vulnerability databases and generating SBOMs for every service. Trusted Profile Analyzer also tracks dependency provenance, flagging components with unclear origins or unmaintained upstream projects. [Red Hat Advanced Cluster Security](#) complements Trusted Profile Analyzer by generating SBOMs for workloads already running on the cluster, giving security teams visibility into what is deployed in production, not just what was built. Together, Trusted Artifact Signer, Trusted Profile Analyzer, and Red Hat Advanced Cluster Security work together so that images are signed, dependencies are tracked, and known vulnerabilities are surfaced from build through runtime.

Centralized identity and access management

Each service must verify user identity and enforce authorization; implementing this independently in every service is fragile and inconsistent. [Red Hat build of Keycloak](#), introduced for SSO at Stage 2, now becomes the centralized identity provider for the entire microservices architecture, issuing OAuth 2.0 and OpenID Connect tokens that services validate independently. The API gateway validates tokens at the edge, rejecting unauthenticated requests before they reach backend services. Keycloak also manages service-to-service authentication through the client credentials flow, which lets backend services authenticate using short-lived tokens rather than shared secrets. Combined with mutual TLS from Red Hat OpenShift Service Mesh, this creates layered identity verification at both transport and application layers.

Service-to-service communication security

Every network call between services is a potential attack vector. [Red Hat OpenShift Service Mesh](#) provides mTLS transparently, encrypting all traffic and verifying the identity of both caller and receiver without application code changes. The mesh manages identity certificates automatically, rotating them without team intervention, establishing zero trust networking at the infrastructure layer: services prove

their identity on every request, and the mesh enforces authorization policies that control which services can communicate with which endpoints. This foundation scales into the comprehensive zero trust networking described in [Chapter 6](#).

Common mistakes to avoid

Implementing microservices without service discovery

Hard coding service URLs breaks the moment pods are rescheduled, services scale, or cluster topology changes. Service discovery must be automatic. Use Kubernetes DNS-based resolution as the baseline, and adopt a service mesh for sophisticated capabilities like load balancing, circuit breaking, or traffic shifting. Services should reference logical names, never physical addresses. Without it, every scaling event risks broken connections and cascading failures.

Decomposing the monolith without proper API boundaries

Services that share databases, rely on internal data structures, or make synchronous calls to many other services are a distributed monolith, containing all of the complexity of microservices and none of the benefits. Define clear API contracts specifying interface, data formats, versioning strategy, and error handling before extracting any service. Use domain-driven design to identify bounded contexts with clear data ownership. The API contract becomes the only dependency between services, and changes behind that contract can be made independently. Without this discipline, extraction creates a distributed system harder to operate than the monolith it replaced.

Sharing databases between microservices

A shared database is a common form of hidden coupling. When 2 services read from and write to the same tables, schema changes in 1 can break the other, deployments must be coordinated, and independent scaling is impossible. Each service should own its data exclusively, communicating through APIs or events. During early decomposition, services can use separate schemas within a shared instance, provided each has exclusive table ownership and no cross-service joins occur, then migrate to independent instances as the architecture matures.

Creating too many small (nano) services

A service that cannot be deployed and tested independently, or that always changes in lockstep with another service, is too small. Services should align with business capabilities, such as “order management” or “payment processing,” not technical functions like “validation” or “logging.” When in doubt, keep services larger and split later when a clear need emerges. The cost of merging prematurely split services is far higher than splitting after a service has grown too large.

Relying on in-memory caches without external backing

In-memory caches are lost when pods restart or are rescheduled. For data that must survive pod lifecycle events, like session state, reference data, or computed results, use an external distributed cache. Without it, every pod restart triggers a cold-cache performance penalty, and scaling a service splits the cache across replicas with no coordination. Design for externalized caching from the start; retrofitting it is more work.

Building complex deployment scripts for multiservice applications

Imperative scripts for deploying multiservice applications become brittle and impossible to review at scale. Use Helm umbrella charts to package related services as a single versioned release, or Argo CD

ApplicationSets for declarative management across services, environments, and clusters. Declarative definitions are auditable, repeatable, and compatible with GitOps workflows. When a deployment fails, a single Git revert replaces a frantic manual recovery process.

Letting AI agents decompose monoliths without architect-defined service boundaries

AI tools can analyze code structure, identify coupling patterns, and suggest decomposition points, but they lack the domain knowledge to define proper bounded contexts. A code-level analysis might suggest splitting by package structure, which rarely aligns with business capabilities or data ownership. Architects should define service boundaries based on domain analysis, establish API contracts, and then use AI-assisted analysis to accelerate implementation within those boundaries. AI accelerates the mechanical work of refactoring, but the strategic decisions must come from people who understand the business.

Red Hat OpenShift capabilities that accelerate progression

Red Hat build of Quarkus

[Quarkus](#)'s build-time optimization produces sub-second startup times and lower memory consumption than traditional Java frameworks.⁶ These efficiencies compound directly into reduced infrastructure costs as dozens or hundreds of microservice instances run simultaneously. Dev mode with live reload accelerates iteration. Red Hat also provides curated Spring Boot runtimes for organizations with existing Spring expertise.

Migration toolkit for applications

[Migration Toolkit for Applications](#) extends its containerization assessment with deeper analysis for microservices decomposition, scanning monolithic code bases to identify coupling patterns, dependency graphs, and bounded contexts. The analysis maps internal module dependencies, database access patterns, and API call chains, highlighting areas of high cohesion (good service candidates) and high coupling (areas needing refactoring before extraction). [Developer Lightspeed for migration toolkit for applications](#) adds AI-assisted refactoring, generating contextually accurate code changes that developers review and apply directly in their IDE. This combination of automated analysis and AI-assisted implementation accelerates the strangler fig pattern, reducing the manual effort of each service extraction.

Red Hat OpenShift Dev Spaces

[Red Hat OpenShift Dev Spaces](#) provides cloud-based development environments that run as containers on the OpenShift cluster, provisioned in minutes through a devfile that defines the workspace configuration as code. As independent services multiply, OpenShift Dev Spaces solves the “works on my machine” problem at scale, giving developers an identical, production-like environment. When a developer switches from a Java/Quarkus order service to a Node.js notification service, they open a new workspace and are immediately productive with an identical, production-like environment.

Red Hat OpenShift GitOps (Argo Rollouts)

[Argo Rollouts](#) extends the GitOps deployment model with canary releases that shift configurable traffic percentages to new versions and automatically promote or roll back based on metric analysis (for example, promoting only if error rate stays below 1% and p99 latency less than 500 ms). For a system where each service has its own deployment lifecycle, this automated progressive delivery is essential.

6 Quarkus blog. “[Quarkus Insights #255: Faster, Leaner, Predictable - Quarkus in the Real World.](#)” 3 Aug. 2026.

Red Hat OpenShift Service Mesh

[Red Hat OpenShift Service Mesh](#), based on Istio, provides a dedicated infrastructure layer for managing service-to-service communication. It handles mTLS encryption, traffic routing, load balancing, circuit breaking, and retry policies, all configured declaratively without application code changes. The mesh provides deep observability into inter-service traffic through service topology graphs, and supports sophisticated traffic management (canary routing, A/B testing, and cross-cluster failover), without requiring application teams to implement these capabilities in their code. The platform team manages mesh policies centrally, applying consistent security and routing policy across services.

Red Hat Connectivity Link

[Red Hat Connectivity Link](#) provides API gateway and management for external traffic, including authentication, rate limiting, API versioning, protocol translation, and an integrated developer portal for API consumers to discover and subscribe to services. The combination of Connectivity Link for north-south traffic and OpenShift Service Mesh for east-west traffic provides unified governance of both external API management and internal service communication through a single, platform-integrated approach.

Red Hat Data Grid

[Red Hat Data Grid](#) is a distributed, in-memory data store that provides low-latency caching, session management, and data partitioning, managed by an Operator that handles deployment, scaling, backup, and upgrades. It supports replicated mode for small, frequently accessed data sets and distributed mode for large data sets partitioned across nodes. Common use cases include caching frequently queried reference data, storing user session state externally so any pod replica can serve any user, and buffering event data for high-throughput pipelines. By externalizing cache state, Data Grid helps prevent a cold-cache performance penalty when scaling, restarting, or redeploying a service.

Red Hat build of Keycloak

[Red Hat build of Keycloak](#) provides centralized identity and access management for the microservices architecture, supporting OpenID Connect, OAuth 2.0, and SAML. It acts as the single identity provider across all services, issuing tokens services validate independently, managing user sessions, and enforcing authentication policies. It supports identity federation with external providers (LDAP, Active Directory, and social logins), fine-grained authorization policies, and multifactor authentication. Keycloak removes the need for each service to implement its own authentication logic.

Red Hat Trusted Artifact Signer

[Red Hat Trusted Artifact Signer](#) provides cryptographic image signing and provenance attestation using Sigstore standards, integrated into the CI/CD pipeline without requiring teams to manage signing keys. Images produced by team pipelines carry a verifiable proof of origin. Combined with admission policies rejecting unsigned images, Trusted Artifact Signer creates a trust gate at the cluster boundary.

Red Hat Trusted Profile Analyzer

[Red Hat Trusted Profile Analyzer](#) provides continuous software composition analysis, scanning application dependencies against known vulnerability databases and generating SBOMs for every service. When a new CVE is disclosed, Trusted Profile Analyzer identifies affected services in minutes. With Trusted Artifact Signer, rules work to make sure that every image is signed, every dependency tracked, and every vulnerability surfaced before it reaches production.

Lightwell

A joint initiative between Red Hat and IBM, [Lightwell](#) helps enterprises strengthen the security posture of the open source software supply chain when normal upgrades create compatibility, certification, release, audit, or production risk. Using specialized AI agents paired with human expertise, Lightwell serves as a trusted clearinghouse for vulnerability management, triage, patching, backporting, and upstream collaboration. It also provides curated and certified package registries for ecosystems, such as Java and Python, helping organizations use open source software confidently.

Customer spotlight: Dutch Tax Authority

“OpenShift gives us a container platform where we bring all development resources into a single, self-service Project-as-a-Service.”

Jerry van Hulst

Product Owner,
Belastingdienst⁷

The Dutch Tax Authority (Belastingdienst), with 4,000 IT staff and hundreds of applications, faced fragmented development environments where teams lacked a unified way to provision namespaces and deploy applications, leading to inconsistent workflows and long onboarding lead times. By standardizing on Red Hat OpenShift Pipelines and Red Hat OpenShift GitOps for CI/CD, alongside Red Hat OpenShift Observability and Keycloak, Belastingdienst built a self-service Project-as-a-Service platform, reducing environment provisioning from weeks to just 2 minutes.⁷

Customer spotlight: Tel Aviv Stock Exchange

“Red Hat OpenShift means our developers could, in theory, write code in the morning and deploy it in the afternoon.”

Efraim Glatt

DevOps Team Leader,
Tel Aviv Stock Exchange⁸

As Israel’s sole public stock exchange, the Tel Aviv Stock Exchange (TASE) faced intense pressure to accelerate service delivery without compromising strict security and compliance requirements. By adopting Red Hat OpenShift with Argo CD for continuous delivery and Red Hat Advanced Cluster Security embedded throughout the CI/CD pipeline, TASE cut time to market by 30% and scaled from quarterly releases to daily deployments.⁸

⁷ Red Hat customer video. “[Ghost in the platform: How the Dutch Tax Authority built a service to scale K8s to 99+ production applications](#).” KubeCon EU, March 2026.

⁸ Red Hat case study. “[Tel Aviv Stock Exchange brings new services to customers faster](#).” May 2025.

Moving to the next stage

Before advancing to Stage 4, verify that your architecture meets these criteria:

- ☐ Monolith decomposition follows the strangler fig pattern with clear bounded contexts.
- ☐ Services communicate through well-defined APIs with proper versioning.
- ☐ Service discovery is handled by the platform, not hard-coded URLs.
- ☐ Deployments use progressive delivery strategies (canary, blue-green).
- ☐ Supply chain security includes image signing and provenance verification.
- ☐ External caching is in place for performance-critical data access patterns.
- ☐ A developer portal provides Golden Paths for creating new services.

With these architectural foundations in place, you're ready to optimize for resilience, efficiency, and scale in Stage 4.

Chapter 6:

Stage 4—Optimize



Elastic cost efficiency with predictable performance at scale	54
Organization maturity at this stage	54
Application patterns in depth:	55
Service mesh	55
Event-driven autoscaled applications	56
Circuit breaker and retry logic	57
CQRS and event sourcing	57
Security at this stage	58
Common mistakes to avoid	59
Red Hat OpenShift capabilities that accelerate progression	59
Customer spotlights: Ortec Finance, Atruvia	61
The optimization journey continues	62



Elastic cost efficiency with predictable performance at scale

Stage 4 is where cloud-native maturity delivers its highest return. With applications containerized, the platform operationalized, and architecture modernized into microservices, the focus shifts to building resilience that prevents cascading failures, scaling precisely to demand through event-driven architecture, and managing the complexity of large distributed systems.

Organizations at this stage are **platform innovators**. They operate mature microservices architectures in production where high availability and resilience are business requirements. The goal is delivering the velocity achieved in Stage 3 without proportionally increasing infrastructure costs. Event-driven architecture scales precisely to demand. Service mesh provides resilience and security at the platform level. Advanced data patterns handle the read/write complexity that comes with truly distributed systems.

Organization maturity at this stage

Maturity at this stage consists of:

- ▶ **Mature microservices architecture.** A large number of services in production with established patterns for development, deployment, and operations.
- ▶ **High availability and resilience.** The platform must prevent cascading failures, handle traffic spikes gracefully, and recover automatically from component failures.
- ▶ **Large scale requiring advanced traffic management.** Load balancing, traffic shaping, canary releases, and blue-green deployments are managed with service mesh.
- ▶ **Event-driven design for resource optimization.** Applications react to events rather than poll for changes, allowing efficient resource use and responsive scaling.
- ▶ **Comprehensive self-service developer platform.** Self-service capabilities include service creation, observability dashboards, deployment pipelines, and security scanning.

Application teams at this stage are asking: How do we prevent cascading service failures? How do we secure traffic between services? How do we scale based on application events rather than resource use? How do we handle complex read/write data requirements?

Application patterns in depth

Service mesh

A [service mesh](#) provides a dedicated infrastructure layer for managing service-to-service communication. Instead of embedding networking logic in application code, the mesh handles these concerns through sidecar proxies deployed alongside each service.

For traffic management, the mesh load-balances across service instances using advanced algorithms, supports traffic splitting for canary deployments and progressive rollouts, injects faults to verify resilience under production-like conditions, and mirrors live traffic to shadow services for safe testing. For security, it establishes zero trust networking, mTLS encrypts and authenticates all communication automatically, authorization policies control which services can reach which endpoints, and request authentication validates tokens before they reach backend services. For observability, the mesh captures request-level metrics (latency, error rates, and throughput), distributed traces, and real-time service topology visualizations for every service-to-service call, without application instrumentation code.

All of this happens without code changes. The sidecar proxy handles networking concerns while the mesh control plane manages configuration declaratively, letting developers focus on business logic while the platform handles resilience, security, and observability uniformly. The service mesh builds directly on the sidecar pattern introduced in [Chapter 5](#), elevating it from an individual container concern to a system-wide infrastructure layer.

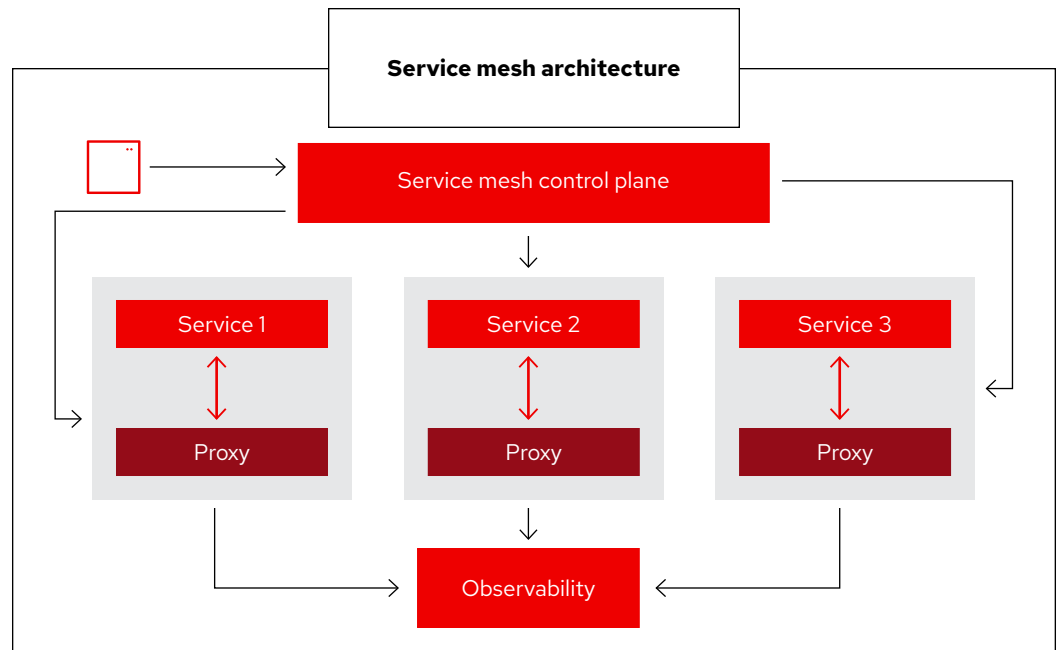


Figure 20. Service mesh architecture.

Event-driven autoscaled applications

[Event-driven architecture](#) decouples producers from consumers through an event broker, which supports asynchronous processing, better resource use, and more responsive scaling. Events are produced when something meaningful happens, like an order placed, a payment processed, a sensor reading captured. An event broker provides durable, ordered, and replayable streams. Multiple services consume these events independently. A single “order placed” event might trigger inventory updates, payment processing, and notifications, all handled by separate services in parallel.

Event-driven autoscaling scales consumer services based on event-source metrics rather than CPU or memory, the natural evolution of the application-metrics-based autoscaling introduced in [Chapter 4](#). When a message queue has growing consumer lag, the platform scales up consumer pods. When the queue is empty, it scales down to 0. This precision means resources are used when there is work to do, reducing infrastructure costs.

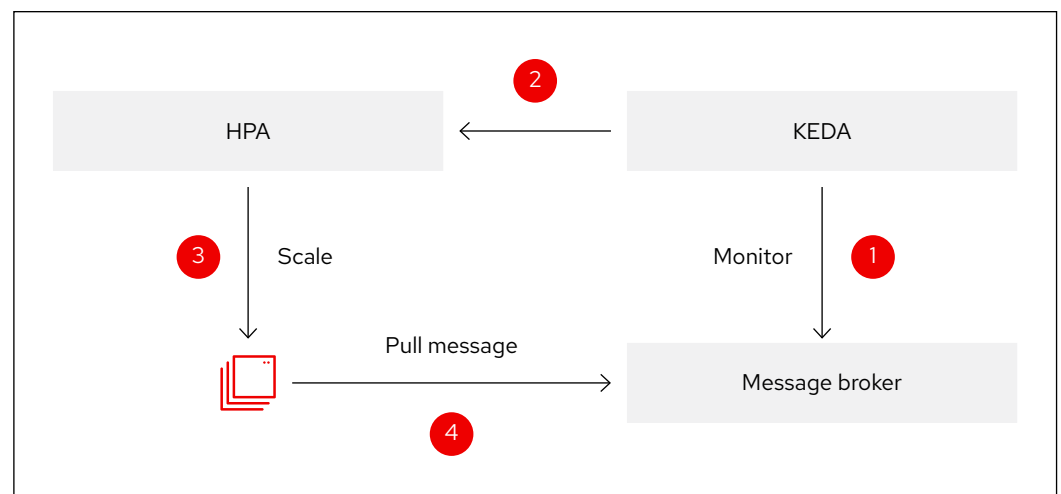


Figure 21. Event-driven autoscaled applications.

Circuit breaker and retry logic

In a microservices architecture, a single slow service can cascade into a system-wide outage as upstream services exhaust their thread pools waiting for responses. The circuit breaker pattern prevents this issue by monitoring error rates to a downstream service and “tripping” the circuit when failures exceed a threshold. In its closed state, requests flow through normally. When failures exceed the threshold, the circuit opens, and requests immediately return a fallback response instead of waiting, protecting upstream services from exhausting resources. After a timeout period, the circuit enters a half-open state, allowing limited test requests through. If they succeed, the circuit closes; if they fail, it stays open.

Implementing circuit breakers at the platform level through a service mesh, rather than in application code, means services get consistent behavior without each team maintaining their own library. The mesh control plane provides a system-wide view of circuit breaker state. Retry logic complements this by automatically retrying failed requests with specific error codes (such as 503 Service Unavailable), with jitter to prevent thundering herds and retry limits to prevent amplification. All are configured declaratively through the mesh.

CQRS and event sourcing

[Command Query Responsibility Segregation \(CQRS\)](#) separates the data model for reads from the data model for writes. The write model enforces business rules and stores normalized data. The read model is optimized for queries—denormalized, precomputed, or cached. An event stream typically keeps the read model current. CQRS is valuable when read and write workloads have dramatically different scaling needs, when queries require expensive joins against a write-optimized schema, or when different consumers need different views of the same data.

Event sourcing complements CQRS by storing all state changes as a sequence of immutable events rather than overwriting the current state, which is derived by replaying events. This provides a complete audit trail, supports temporal queries, and supports rebuilding read models by replaying events. However, the trade-offs are substantial including eventual consistency between read and write models, careful schema versioning as event formats evolve, and snapshotting to manage storage growth. These patterns suit domains with complex business rules, strict audit requirements, or highly asymmetric read/write workloads—not general-purpose applications.

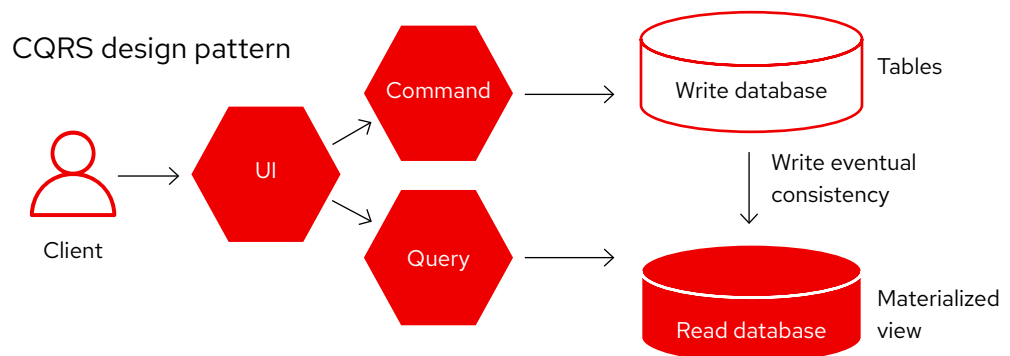


Figure 22. CQRS and event sourcing.

Security at this stage

At the optimization stage, the attack surface spans hundreds of services, each with its own deployment pipeline, dependencies, and network interactions. Manual security review cannot keep pace. The supply chain integrity from [Chapter 5](#) (signed images, verified provenance, tracked dependencies) now becomes the input to fully automated, policy-based enforcement at the platform.

Zero trust networking

[Red Hat OpenShift Service Mesh](#) enforces mTLS for all service-to-service communication, encrypting traffic and verifying both caller and receiver on every request. Authorization policies define which services can communicate with which endpoints, enforcing least privilege at the proxy level. Authorization policies are designed so that services can reach only the endpoints explicitly permitted by policy. For organizations operating across multiple clusters, clouds, and infrastructure types, [zero trust workload identity manager](#), a Red Hat OpenShift operator, provides a universal identity plane built on the [Secure Production Identity Framework For Everyone \(SPIFFE\) and SPIFFE Runtime Environment \(SPIRE\)](#) framework. It issues short-lived, automatically rotated workload identities verified through hardware-rooted attestation, replacing static API keys and service account tokens with cryptographic identities that work consistently across containers, VMs, datacenters, and edge locations. This is the full realization of the service-to-service security introduced in [Chapter 5](#), extended across the entire hybrid cloud architecture.

Automated policy enforcement

With hundreds of services deploying independently, security policies must be enforced programmatically. [Red Hat Advanced Cluster Security for Kubernetes](#) validates deployments against organizational standards, including image provenance, resource limits, security contexts, and network policies, and rejects noncompliant workloads before they reach the cluster. [Red Hat Quay](#) enforces image verification at the registry level, ensuring only signed and scanned images are available. Pipeline gates in [Red Hat OpenShift Pipelines](#) extend enforcement earlier, blocking builds that fail security scans or lack required attestations. Together, these layers are designed to block noncompliant code from being built, packaged, or deployed.

Runtime security monitoring

[Red Hat Advanced Cluster Security](#) monitors running workloads for unexpected process execution, network connections to unauthorized hosts, privilege escalation attempts, and file system modifications outside expected paths. When a threat is detected, it can automatically alert security teams, quarantine the affected pod, or enforce kill policies, all managed from a single console across the cluster fleet. At this stage, where services scale dynamically and pods are ephemeral, runtime monitoring provides the final layer of defense, catching threats that bypassed earlier controls and detecting compromises in real time.

Common mistakes to avoid

Using CQRS where CRUD is sufficient

CQRS adds complexity, including separate data models, synchronization mechanisms, and eventual consistency handling, that must be justified by genuinely different read and write requirements. Before adopting it, confirm that you have a concrete problem it solves: dramatically asymmetric read/write loads, expensive query patterns against a write-optimized schema, or multiple consumers requiring fundamentally different data views. Without any of these reasons, the complexity slows development without improving outcomes. Most applications work perfectly well with simple create, read, update, and delete (CRUD).

Manually implementing circuit breakers

Library-based circuit breakers require each team to independently implement, configure, tune, and maintain resilience logic, producing inconsistent behavior across services: different thresholds, timeout values, and fallback strategies. A service mesh provides circuit breakers consistently across all services with a unified control plane view, without a line of application code.

Adopting event sourcing without consistency considerations

Read models can lag seconds or minutes behind the latest write. Applications built on strong-consistency assumptions will produce incorrect results: displaying stale data, allowing duplicate actions, or making decisions based on outdated state. Design for eventual consistency from the start by building UIs that acknowledge processing delays, implementing idempotent event handlers that tolerate duplicate delivery, and testing edge cases where read and write models diverge. Retrofitting is harder than designing for it upfront.

Accepting AI-generated event sourcing implementations without expert review

AI agents can generate event sourcing scaffolding quickly and with syntactically correct code, but they can miss subtle consistency requirements, produce incorrect replay logic, or generate projections that silently drop events under certain ordering conditions. The failure modes at this stage only surface under load or during partial failures, exactly the scenarios that are hardest to catch in code review. Expert review combined with integration tests simulating real-world failure conditions is essential.

Red Hat OpenShift capabilities that accelerate progression

Red Hat OpenShift provides specific capabilities that help organizations at this stage to move faster and avoid common pitfalls, including:

Red Hat OpenShift Service Mesh

[Red Hat OpenShift Service Mesh](#) provides traffic management, mTLS, circuit breakers, retry logic, fault injection, and traffic mirroring—all configured declaratively without application code changes. At this stage, where hundreds of services interact across the network, the mesh delivers consistent resilience and security that would be extremely difficult to achieve through individual application-level implementations. Deep observability, such as request-level metrics, distributed traces, and service

topology visualizations, gives operations teams a unified view of system health without requiring instrumentation code in every service.

Zero trust workload identity manager

[Zero trust workload identity manager](#) supports zero trust from build through runtime across hybrid cloud environments, built on the [SPIFFE/SPIRE](#) standards. It delivers a consistent identity plane across containers, VMs, clouds, datacenters, and edge locations, replacing fragmented per-cloud identity configurations and static, long-lived credentials. Node and workload attestation verifies infrastructure integrity before issuing identities, establishing a hardware root of trust. Identities are short-lived and automatically rotated. At this stage, where workloads span multiple clusters and cloud boundaries, the zero trust workload identity manager provides the foundational identity layer used by OpenShift Service Mesh, Red Hat Trusted Artifact Signer, and external secret stores, which supports true end-to-end [zero trust](#) from build through runtime across the entire hybrid cloud environment.

Red Hat OpenShift Serverless and Customer Metrics Autoscaler

Based on KEDA, [Red Hat OpenShift Serverless](#) provides scale-to-zero event-driven application capabilities, automatically scaling from 0 to thousands based on incoming volume and application metrics. Intermittent workloads (batch processing, webhook handlers, scheduled jobs, event consumers) consume no resources when idle, aligning infrastructure costs directly with actual use. Teams deploy functions without managing scaling policies or replica counts. The platform handles the operational complexity of matching capacity to demand.

Red Hat AMQ and Streams for Apache Kafka

[Red Hat AMQ](#) provides enterprise messaging and [Streams](#) delivers Apache Kafka on OpenShift with fully automated operations, including cluster provisioning, topic management, partition rebalancing, and rolling upgrades managed by the operator. It provides the durable, ordered, and replayable event streams that decouple producers from consumers and support the CQRS and event sourcing patterns described in this chapter. Streams for Apache Kafka also integrates with KEDA for event-driven autoscaling, scaling consumer pods based on topic lag rather than CPU or memory.

Power monitoring and observability

[Power monitoring for Red Hat OpenShift](#), based on Kepler, tracks energy consumption at the pod and namespace level, giving organizations visibility into the carbon footprint of individual workloads and services. Combined with end-to-end [observability](#), it supports power-aware scheduling that shifts nonurgent workloads to times or regions with lower carbon intensity. At this stage, where compute-intensive tasks like financial simulations or machine learning inference run across large clusters, energy-efficient scheduling reduces both the carbon footprint and the infrastructure bill, which is increasingly important as sustainability reporting requirements grow.

Red Hat Advanced Cluster Management for Kubernetes

When your cluster fleet spans multiple regions, cloud providers, and infrastructure types, governance requirements must grow to match. [Red Hat Advanced Cluster Management for Kubernetes](#), introduced in [Chapter 4](#) for cluster lifecycle and multicluster observability, now becomes the enforcement engine for fleet-wide security and compliance at scale. Its policy framework evaluates each cluster against organizational standards and automatically remediates drift, helping keep security configurations, required Operators, network policies, and resource constraints consistent without manual intervention. It integrates with [Red Hat Advanced Cluster Security](#) to extend vulnerability and compliance policies across all clusters, and with [zero trust workload identity manager](#) to deliver consistent workload identity across the hybrid cloud environment. For multicluster application placement, it distributes workloads based on region, compliance requirements, available capacity, and cluster health, supporting sophisticated deployment strategies like active-active across regions or cross-cluster failover.

Customer spotlight: Ortec Finance

“Azure Red Hat OpenShift gives us all the flexibility and self-service of a vendor-neutral cloud-native platform, together with the support and firepower of a managed service.”

Joris Cramwinckel

Technologist, Ortec Finance⁹

Ortec Finance, a global wealth management and risk analytics provider serving more than 600 clients across 20+ countries, needed a scalable, cloud-native platform to accelerate feature delivery, ensure regulatory compliance across multiple regions, and reduce the carbon footprint of compute-intensive financial simulations. It built the ORCA Platform on Microsoft Azure Red Hat OpenShift, with GitOps-based delivery via Tekton and Argo CD, and power-aware autoscaling using Knative, KEDA, and Kepler. This platform cut software delivery lead times from days to minutes, achieved on-demand regional deployment for compliance, and reduced their environmental impact with power-aware scheduling.⁹

Customer spotlight: Atruvia

“Red Hat OpenShift Service Mesh automatically integrates communications and authorization policies so the development team does not have to, helping them be more productive.”

Marius Capris

Security Engineer, Atruvia¹⁰

Atruvia, the IT backbone for more than 800 cooperative banks across Germany, was manually managing authorization policies across 5,000 pods, time that could have been spent building banking features. By deploying Red Hat OpenShift Service Mesh, Atruvia automated mTLS encryption and authorization across its entire microservices architecture, introduced circuit breakers and traffic management for greater resilience, and gained real-time visibility into potential security risks.¹⁰

⁹ Red Hat press release. [“Red Hat Helps Ortec Finance Accelerate Growth and Time to Market.”](#) May 2024.

¹⁰ Red Hat case study. [“Atruvia strengthens security with Red Hat OpenShift Service Mesh.”](#) March 2025.

The optimization journey continues

The maturity model isn't a destination; it's a direction. The organizations that succeed are those that continue to invest in their platform, their patterns, and their people.

Stage 4 is where the platform delivers its highest return on investment. Event-driven architecture reduces costs by scaling precisely to demand, service mesh provides zero trust security and resilience without burdening application teams, and advanced data patterns handle the complexity of truly distributed systems.

But optimization is ongoing. New patterns emerge, new capabilities become available, and your application portfolio continues to evolve. The maturity model isn't a destination; it's a direction. The organizations that succeed are those that continue to invest in their platform, their patterns, and their people.

[Chapter 8](#) brings the entire journey together with practical guidance for navigating across all 4 stages.

Chapter 7:

Building your platform engineering practice



What platform engineering is, and why it matters 64

From infrastructure operators to product owners 64

AI agents as platform consumers 65

How the platform team evolves across maturity stages 65

Measuring platform success 67

Building your platform team 68



What platform engineering is, and why it matters

Cloud-native adoption doesn't succeed on technology alone. Behind every successful Red Hat OpenShift deployment is a [platform engineering team](#) that designs, builds, maintains, and continuously improves the platform to serve its users.

Platform engineering is the discipline of building and operating an [internal developer platform](#) (IDP) that provides self-service capabilities to application teams. Rather than requiring every team to independently assemble and manage its own infrastructure, CI/CD pipelines, and operational tooling, the platform team creates a curated, opinionated set of tools, workflows, and [Golden Paths](#) that all teams share.

This distinction matters because the alternative does not scale. When 10 development teams each build their own CI/CD pipeline, the organization ends up with 10 different deployment processes, 10 sets of security gaps, and 10 times the maintenance burden. A platform engineering team that consolidates this into a single well-maintained platform amplifies its impact across the organization, accelerating every team it serves.

Organizations that invest in platform engineering today will have a competitive advantage as cloud-native complexity continues to grow.

From infrastructure operators to product owners

The most important shift in platform engineering is a change in mindset. Traditional infrastructure teams operate in a service-request model: development teams submit tickets, and the infrastructure team provisions resources. This approach creates bottlenecks, delays, and frustration on both sides.

Platform engineering replaces the ticket-based model with a product model. The platform team treats developers as customers, and the platform itself as the product, designed with the user in mind, delivered iteratively, and continuously improved based on feedback.

This means platform teams must:

- ▶ **Understand their users.** What do developers actually need to ship software, and where do they lose time? Teams that build capabilities nobody asked for waste effort; teams that listen build tools that get adopted organically.
- ▶ **Provide self-service.** The goal is to empower developers to do things themselves, safely and within organizational guardrails. A developer should be able to create a service, provision a database, set up a CI/CD pipeline, and deploy to production without filing a ticket.
- ▶ **Measure adoption and satisfaction.** A platform that developers avoid or work around is a failed product. Track adoption rates, satisfaction scores, onboarding time, and support request volume.
- ▶ **Iterate and improve.** The platform is never finished. The roadmap is based on developer needs and organizational priorities, not by technology trends alone.

AI agents as platform consumers

The platform has always existed to serve developers, but who counts as a “developer” is changing. Today, the platform has 2 types of consumers: human developers using IDEs, command line interfaces (CLIs), and dashboards, and AI agents generating code, Dockerfiles, Helm charts, pipeline definitions, and pull requests, often at a velocity that exceeds human output. This dual-consumer reality changes what the platform team must provide.

Golden Paths must be machine readable

Software templates are not just for humans clicking through. AI agents need to discover and follow the same standards. Golden Paths must exist as structured templates, API-based scaffolding, and machine-readable documentation, not wiki pages or institutional knowledge. When an AI agent generates a new microservice, it should be able to discover approved base images, standard product structures, and required pipeline configurations through the same platform humans use. Otherwise, AI agents generate code that does not conform to organizational standards, creating a new category of technical debt.

Automated validation becomes nonnegotiable

When a human writes code, a peer reviews it. When an AI agent generates a Helm chart or a Dockerfile, automated policy checks, security scans, and integration tests are the review because the volume and velocity of AI-generated changes exceeds what human review alone can handle. This requires investment in automated security scanning to catch insecure base image choices, missing security contexts, and vulnerable dependencies before code reaches production; Policy as Code (PaC) to automatically validate manifests against organizational standards; and automated testing to verify AI-generated configurations work across environments before merging.

Guardrails shift from guidance to enforcement

With human developers, training and code review can carry some of the load. AI agents follow instructions literally without exercising judgment, so guardrails must be codified. An AI agent told to “use the smallest base image possible” might choose an alpine image lacking required security libraries. An AI agent generating a Helm chart might produce valid YAML that passes linting but ignores your parameterization conventions. The platform must catch these issues automatically.

Organizations that prepare their platforms for AI agent consumption today will be better positioned to benefit from AI-assisted development at scale. Those that don’t will face a growing gap between the speed of AI-generated code and their ability to validate and deploy it safely.

How the platform team evolves across maturity stages

The platform team’s responsibilities grow as the organization progresses through the 4 stages of cloud-native adoption. Understanding this evolution helps you staff appropriately, set realistic expectations, and plan for the skills your platform team will need next.

Stage 1: Establishing the foundation

The team focuses on getting the basics right, including:

- ▶ **Cluster provisioning and configuration.** Setting up Red Hat OpenShift clusters, configuring networking, storage, and authentication.
- ▶ **Developer onboarding.** Helping development teams containerize their initial applications by writing example Dockerfiles, creating deployment manifests, and providing hands-on guidance.
- ▶ **Basic operational procedures.** Monitoring, alerting, backup, and disaster recovery processes for the platform itself.
- ▶ **Security baseline.** Configuring RBAC, network policies, and image pull policies to establish a foundation hardened for security.

The team is typically small, often 2-4 people, and they wear many hats. They are simultaneously platform administrators, developers, trainers, and support staff.

Stage 2: Building self-service

As more teams adopt the platform, the ticket-based model breaks down, and the platform team shifts toward self-service, including:

- ▶ **Standardized CI/CD pipelines.** Creating reusable [Tekton pipelines](#) and [Argo CD](#) templates.
- ▶ **Developer Hub deployment.** Standing up [Red Hat Developer Hub](#) as a self-service portal with software templates, documentation, and service catalogs. Golden Paths are codified at this stage.
- ▶ **Namespace and project provisioning automation.** Creating new projects, configuring resource quotas, and setting up environments without platform team involvement.
- ▶ **Observability platform.** Deploying and configuring the [observability](#) tooling that development teams use for their applications.

The team grows and begins to specialize, with members focusing on pipeline engineering, platform reliability, and developer experience.

Stage 3: Supporting service governance

The shift to microservices introduces new demands:

- ▶ **Service mesh deployment and management.** Operating the [service mesh](#) that provides traffic management, security, and observability for inter-service communication.
- ▶ **API management.** Providing API gateway capabilities and enforcing versioning and documentation standards.
- ▶ **Developer portal maturation.** Maturing Developer Hub into the primary interface with service catalogs, dependency graphs, and full-lifecycle self-service capabilities.
- ▶ **Migration support.** Assisting with monolith decomposition with tooling such as the [migration toolkit for applications](#).

The team is now operating a sophisticated product, often organizing into sub-teams focused on platform reliability, developer experience, and security.

Stage 4: Allowing advanced capabilities

At the highest maturity level, the team provides advanced capabilities that support event-driven architecture, sophisticated scaling, and platform-level resilience:

- ▶ **Event-driven infrastructure.** Deploying and managing event brokers (Streams for Apache Kafka), event-driven autoscaling (KEDA), and serverless functions (Knative).
- ▶ **Advanced traffic management.** Configuring service mesh for traffic routing, fault injection testing, and progressive delivery.
- ▶ **Platform analytics and optimization.** Using resource optimization tools to minimize infrastructure costs while maintaining service-level agreements (SLAs).
- ▶ **Advanced self-service.** Providing comprehensive capabilities with minimal day-to-day intervention required.

Measuring platform success

A platform engineering team that can't measure its impact can't improve, but the right metrics depend on your organization's maturity stage, team structure, and business priorities. Rather than prescribing a fixed dashboard, consider 3 categories as a starting point.

1. Developer experience.

The platform exists to serve development teams, so the most important signals come from the people using it. How quickly can a new team member go from onboarding to 1st deployment? Can a team spin up a new service without filing tickets? Are developers satisfied, or are they building workarounds? These questions matter more than any single number; answer them through lightweight surveys and direct observation of how teams actually use the platform.

2. Delivery performance.

The platform should make software delivery faster, safer, and more predictable over time. Industry frameworks such as the DevOps Research and Assessment (DORA) metrics (deployment frequency, lead time for changes, change failure rate, and mean time to recovery (MTTR)) offer a well-established starting point. A maturing platform should show improving trends across these dimensions.

3. Platform operational health.

The platform itself is a product with its own reliability, efficiency, and security posture. How available is the platform to its users? How efficiently is infrastructure being used? What percentage of workloads comply with organizational security policies? These indicators help the platform team manage its own service quality and justify continued investment to leadership.

Start with a small number of indicators that reflect your most pressing priorities, make them visible to both the platform team and the teams it serves, and refine the approach as the platform matures. Measurement transparency builds the trust that sustains long-term platform adoption.

Building your platform team

This developer-first mindset is the single most important factor in building a platform engineering practice that produces real cloud-native adoption.

The platform engineering practice does not need to start large. Begin with a small, cross-functional team combining infrastructure operations expertise with software development skills. A platform engineer who can write code is more effective than the team member who can only configure tools. Grow the team to match the increasing scope of responsibilities outlined in the maturity stages above.

Organizations newly establishing a platform engineering practice should focus on 3 priorities:

1. **Listen and learn.**

Interview development teams to understand current workflows, pain points, and tooling. Identify the biggest time sinks and the most common support requests, and map existing deployment processes to surface inconsistencies. This research forms the foundation of your platform roadmap.

2. **Deliver a quick win.**

Choose a single capability that addresses a widely shared pain point, such as standardized CI/CD pipelines, automated namespace provisioning, or a centralized observability dashboard. Now deliver it. The goal is demonstration, not perfection. Show development teams the platform team exists to make their lives easier, not to add bureaucracy.

3. **Establish feedback loops.**

Set up regular channels for developer feedback, such as office hours, Slack channels, surveys, or retrospectives. Use this feedback to prioritize the next round of platform investments. Publish a lightweight roadmap communicating what the platform team is working on and why.

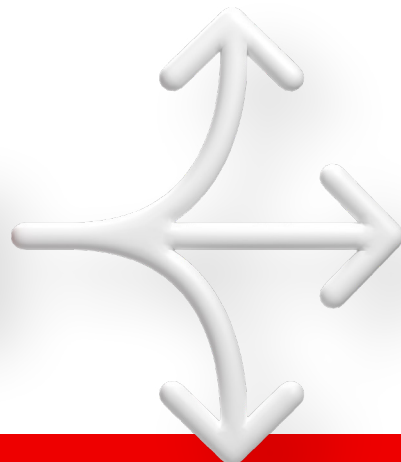
The most successful platform teams share a common trait: they care deeply about developer experience, measuring success not by the sophistication of the tools they deploy but by the productivity and satisfaction of the developers they serve. This developer-first mindset is the single most important factor in building a platform engineering practice that produces real cloud-native adoption.

Chapter 8:

Navigating your journey



How to identify your current stage	70
Building a cross-functional adoption team	72
Measuring progress at each stage	73
Recommended training and certification path	74
Planning your adoption roadmap	75
Avoiding common organizational pitfalls	76
Getting started	76



How to identify your current stage

Most organizations have a portfolio of applications at different maturity levels. A recently containerized internal tool sits at Stage 1 while a customer-facing platform with microservices operates at Stage 3. This is normal. The maturity model applies to workloads and teams, not to the organization as a single unit.

Honest assessment matters more than ambition. Teams that skip stages or misidentify their current maturity adopt patterns they're not yet equipped to operate, leading to fragile architectures and frustrated engineers. A team that hasn't mastered health checks, externalized configuration, and structured logging is not ready for service mesh traffic management. Starting from the right place means building on proven capabilities rather than an aspirational target.

Work through the following diagnostic questions for each major workload or application team.



Stage 1: Containerize

- ☐ Are applications being packaged into containers for the 1st time?
- ☐ Are developers still learning Dockerfiles, pod lifecycles, and Kubernetes basics?
- ☐ Is the deployment manual or script-based?
- ☐ Are health checks, externalized configuration, and structured logging not yet in place?
- ☐ Is the platform team focused primarily on cluster setup and basic operations?

Focus on Chapter 3 patterns: containerized monolith, health checks, log forwarding, externalized configuration, and base images hardened for security.



Stage 2: Operationalize

- ☐ Are multiple applications running in production?
- ☐ Do CI/CD pipelines exist but vary across teams?
- ☐ Is observability in place but alerts not consistently useful?
- ☐ Is scaling manual or CPU and memory-only?
- ☐ Is the platform team building self-service capabilities?

Focus on Chapter 4 patterns: observability instrumentation, application metric autoscaling, graceful shutdown, Helm/Kustomize standardization, and the Operator pattern.



Stage 3: Modernize

- ☐ Are monolithic applications being decomposed into microservices?
- ☐ Are multiple teams deploying independent services?
- ☐ Is service discovery and API management an active concern?
- ☐ Are deployment strategies evolving beyond rolling updates?
- ☐ Is the platform team operating a developer portal with Golden Paths?

Focus on Chapter 5 patterns: strangler fig decomposition, cloud-native frameworks, API gateway, sidecar pattern, library charts, and supply chain security.



Stage 4: Optimize

- ☐ Is a service mesh managing traffic, security, and observability?
- ☐ Are applications scaling based on events or business metrics?
- ☐ Are circuit breakers implemented at the platform level?
- ☐ Are advanced data patterns like CQRS or event sourcing in use?
- ☐ Does the platform provide comprehensive self-service for the entire development lifecycle?

Focus on Chapter 6 patterns: service mesh traffic management, event-driven autoscaling, circuit breakers, and CQRS event sourcing.

Building a cross-functional adoption team

Cloud-native adoption requires coordinated effort across platform engineering, application development, security, and architecture. The most successful organizations build a cross-functional adoption team with representatives from each discipline.

This team is not the same as the platform team. Platform engineers can deliver self-service capabilities, CI/CD templates, and observability tooling, but without input from application teams, security, and architecture, those capabilities might not match what workloads actually need. Where the platform team asks “what should we build?” the cross-functional team answers “what do our workloads need next, and how do we get there together?” This coordination prevents the most common failure mode: a capable platform that teams don’t fully use because it was built without their input.

Key roles

- ▶ **Platform engineers** build and operate the internal developer platform. They provision clusters, create CI/CD pipeline templates, deploy and manage the observability tooling, and provide self-service capabilities through the developer portal.
- ▶ **Application architects** define target architecture for workloads progressing through the maturity stages. They identify bounded contexts for monolith decomposition, define API contracts, and validate AI-generated architectural scaffolding.
- ▶ **Site reliability engineers (SREs)** bridge the gap between development and operations. They define SLOs, build monitoring dashboards, respond to incidents, and conduct project debriefs, shifting from reactive response to proactive reliability engineering as the platform matures.
- ▶ **Security engineers** embed security into every stage of the adoption journey. They define security policies, configure admission controllers, manage the vulnerability scanning pipeline, and ensure compliance with organizational and regulatory requirements.
- ▶ **Engineering leadership** provides organizational mandate, funding, and support. They set priorities, resolve cross-team conflicts, and communicate the value to business stakeholders.

The most common dysfunction we see in cross-functional teams is misaligned priorities. The platform team wants to build new capabilities; security wants stricter policies; development teams want fewer constraints; leadership wants faster delivery.

The solution is a shared adoption roadmap with clearly defined, stage-specific goals the entire team has agreed to. When everyone understands that the current priority is “advance our top 5 customer-facing applications from Stage 2 to Stage 3 by Q3,” each discipline knows exactly what to prioritize.

Measuring progress at each stage

There is no universal scorecard that applies to every organization. The right metrics depend on your business goals, workload characteristics, and what your teams are trying to achieve. A financial services company optimizing for regulatory compliance will measure different things than a Software-as-a-Service startup optimizing for time to market. The examples below are starting points, not requirements.

Principles for effective measurement

Measure outcomes, not activity. The number of containers deployed tells you about adoption breadth but nothing about quality. An organization running 500 containers that crash daily is worse off than an organization running 50 containers with zero-downtime deployments.

Start with a few metrics and refine. Teams tracking 20 key performance indicators (KPIs) from Day 1 end up tracking none of them well. Pick 2 or 3 per stage that align with your most pressing goals, instrument them properly, and act on the results.

Benchmark against your own trajectory. A “good” deployment frequency for a mobile banking application is different from an internal HR tool. The question is whether you’re improving, not whether you match industry averages.

Stage 1 (Containerize): Foundation metrics

How quickly can teams containerize and deploy an application? How long does developer onboarding take? Do container images consistently pass security scans?

Stage 2 (Operationalize): Operational maturity metrics

How does your deployment frequency, change failure rate, and MTTR look? (These metrics determine if your CI/CD pipelines, observability tooling, and operational practices work together effectively.) How consistently do teams adopt standardized pipelines? Does autoscaling respond accurately to demand?

Stage 3 (Modernize): Architectural transformation metrics

Are services truly deployable on their own? Are API contracts well-defined and respected? How quickly can a team spin up a new service using the developer portal? What percentage of images are signed and verified?

Stage 4 (Optimize): Optimization metrics

Examine your service mesh adoption, resource use efficiency, frequency of cascade failure incidents, and infrastructure cost trends. How do these compare to business throughput?

DORA metrics as a baseline

Regardless of stage, the 4 [DORA](#) metrics offer a useful baseline for tracking software delivery performance over time:

- ▶ **Deployment frequency:** How often does the organization deploy to production?
- ▶ **Lead time for changes:** How long from code commit to production deployment?
- ▶ **Change failure rate:** What percentage of deployments cause production incidents?
- ▶ **Mean time to recovery (MTTR):** How quickly does the organization recover from production incidents?

Their strength is consistency: because they apply at every maturity stage, they let you track improvement over the entire adoption journey without switching indicators as workloads advance. Focus on the trend rather than the absolute numbers. A team cutting lead time from 2 weeks to 2 days has made a meaningful leap, even if it hasn't reached the "elite" benchmarks published in the annual [State of DevOps Report](#).¹¹

As your measurement practice matures, consider complementing DORA with the [SPACE framework](#) (satisfaction, performance, activity, communication, efficiency), which captures the human and collaborative dimensions that influence delivery outcomes. This approach can provide a more complete picture of effectiveness, not just speed.

Recommended training and certification path

Building cloud-native skills requires a structured approach. Red Hat offers [certifications](#) that map to each maturity stage.

Foundation certifications

- ▶ **[Red Hat Certified System Administrator \(RHCSA®\)](#):** Linux fundamentals underpinning container operations, essential for troubleshooting and security.
- ▶ **[Red Hat Certified Developer in Cloud-native Applications](#):** Container fundamentals including Podman, Buildah, and Skopeo. Validates Stage 1 skills.

Platform certifications

- ▶ **[Red Hat Certified System Administrator in OpenShift \(EX280\)](#):** Cluster administration including installation, networking, storage, and security configuration. Essential for platform engineers at Stages 1-2.
- ▶ **[Red Hat Certified Advanced Developer in Cloud-native Applications \(EX288\)](#):** Application deployment, configuration, and management on Red Hat OpenShift. Validates developer skills for Stages 1-2.
- ▶ **[Red Hat Certified Advanced System Administrator in OpenShift](#):** Advanced CI/CD, GitOps, and automation skills. Maps to Stages 2-3.

¹¹ Perforce Software. "[The State of DevOps Report 2026](#)," 24 Feb. 2026.

Looking for more Red Hat OpenShift training? Visit our [training and certification page](#) or explore [Red Hat OpenShift skills paths](#).

Advanced certifications

- ▶ **Red Hat Certified Specialist:** Targeted certifications for specific Red Hat technologies, tailored to organizational needs. Maps to Stages 3–4.
- ▶ **Red Hat Certified Architect (RHCA®):** The highest level of Red Hat certification, requiring multiple specialist certifications. Equips architects to guide organizations through Stages 3–4.

Continuous learning

Beyond formal certification, encourage teams to participate in:

- ▶ **Red Hat OpenShift Commons:** A community sharing real-world experience, patterns, and best practices—the source of the customer case studies in this e-book.
- ▶ **Red Hat Developer program:** Free access to developer tools, tutorials, and sandbox environments for hands-on learning.
- ▶ **Cloud-Native Architectures Workshops:** Assessment-based sessions that help organizations evaluate their current state, define target architectures, and build migration plans.

Planning your adoption roadmap

- 1 Assess your portfolio.** Classify applications by current maturity stage, business criticality, and change frequency. High-change, high-criticality applications benefit most from advancement. Map each major application to its current stage, target stage, and business justification.
- 2 Identify pilot applications.** Select 2–3 applications to advance to the next stage. Choose applications with engaged teams, clear business value, and manageable complexity. Avoid your most critical, highest-risk application as the 1st pilot; choose an application important enough to demonstrate value, but not so critical that failure is unacceptable.
- 3 Set stage-specific goals.** Define measurable, time-bound success criteria for each pilot. For example, a Stage 1 to Stage 2 goal might be: “Achieve zero-downtime deployments with automated rollback and reduce MTTR to less than 1 hour within 3 months.”
- 4 Invest in the platform ahead of applications.** There’s no point pushing applications to Stage 3 if the platform doesn’t yet provide service discovery, API management, or a developer portal. Platform investment should lead application advancement by at least 1 quarter.
- 5 Share learnings.** Document what worked, what didn’t, and what you would do differently. Use these learnings to create repeatable playbooks for the next wave of applications. Internal tech talks, brown bag sessions, and written project debriefs are all effective formats for knowledge sharing.
- 6 Expand deliberately.** Apply proven patterns to the next wave of applications. Each wave should be faster than the last as tooling, documentation, and organizational knowledge improve.

Avoiding common organizational pitfalls

Trying to do everything at once.

Organizations that attempt to containerize, operationalize, and modernize simultaneously do none well. Each stage builds the skills needed for the next. Patience with the progression pays dividends.

Underinvesting in the platform team.

A platform team of 2 people cannot serve 50 development teams. As adoption grows, the platform team must grow proportionally. Understaffed platform teams become bottlenecks, and developers route around bottlenecks, creating the fragmentation the platform was supposed to prevent.

Ignoring the developer experience.

If the platform is harder to use than the tools it replaced, developers will resist adoption. Invest in documentation, onboarding, and feedback channels. Survey developers regularly, and act on the results.

Treating cloud-native as a single migration.

Cloud-native adoption has no end date. Technologies evolve, patterns improve, and organizational needs change. Operate with a continuous improvement mindset, regularly reassessing the roadmap based on real-world experience.

Cloud-native adoption is a journey measured in quarters and years, not days and weeks. The organizations that succeed are those that commit to sustained, incremental progress, building capabilities stage by stage, application by application, and team by team.

Getting started

Red Hat OpenShift provides the consistent foundation for this entire journey, from the 1st containerized workload to a fully optimized, event-driven microservices architecture. Whether you're running on bare metal, private cloud, public cloud, or at edge locations, Red Hat OpenShift provides a unified platform for building, delivering, and scaling applications.

Assess where your applications stand today. Identify pilot workloads to advance to the next stage. Invest in your platform team. Build a stage at a time toward the velocity, resilience, and efficiency that cloud-native architecture delivers.

To start exploring the technologies discussed, [watch an interactive demo](#), [get hands-on with a lab or learning path](#), or [try Red Hat OpenShift on Developer Sandbox](#).

Reference



Cloud-native adoption playbook for Red Hat OpenShift

Maturity stages

	 Stage 1: Containerize	 Stage 2: Operationalize	 Stage 3: Modernize	 Stage 4: Optimize
Goal	Lift workloads into containers and establish platform fundamentals	Standardize delivery pipelines, observability, and operational practices	Decompose monoliths, adopt microservices, and support team independence	Maximize resilience, efficiency, and advanced traffic management
Application patterns	▶ Containerized monolith ▶ Log to stdout ▶ Health check endpoints ▶ Init containers ▶ Externalized configuration ▶ High-performance local volumes ▶ External Database-as-a-Service	▶ Observability instrumentation ▶ Autoscaling on application metrics ▶ Graceful shutdown ▶ Kustomize overlays ▶ Helm chart value overlays ▶ Operator pattern	▶ Strangler fig ▶ Cloud-native frameworks ▶ API gateway ▶ Service discovery ▶ Sidecar pattern ▶ Library/Helm umbrella chart ▶ External cache	▶ Red Hat OpenShift Service Mesh ▶ Event-driven autoscaling ▶ Circuit breaker and retry logic ▶ CQRS and event sourcing
Security focus	▶ Trusted base images ▶ Run as nonroot ▶ Avoid Kubernetes Secrets	▶ Pipeline security ▶ Continuous image scanning ▶ RBAC maturation ▶ Centralized identity ▶ Network segmentation	▶ Supply chain security ▶ Software composition analysis ▶ Centralized identity access management (IAM) ▶ Service-to-service encryption	▶ Zero trust networking ▶ Automated policy enforcement ▶ Runtime security monitoring
OpenShift capabilities	▶ Red Hat Hardened Images ▶ Red Hat build of Podman Desktop ▶ Red Hat OpenShift web console ▶ Builds for Red Hat OpenShift ▶ Migration toolkit for applications ▶ External Secrets Operator ▶ Red Hat OpenShift Data Foundation	▶ Red Hat OpenShift Pipelines ▶ Red Hat OpenShift GitOps ▶ Red Hat Developer Hub ▶ Custom Metrics Autoscaler ▶ Distributed tracing and OpenTelemetry ▶ Red Hat Advanced Cluster Security for Kubernetes ▶ Red Hat Quay	▶ Red Hat build of Quarkus ▶ Migration toolkit for applications ▶ Red Hat OpenShift Dev Spaces ▶ Red Hat OpenShift Service Mesh ▶ Red Hat Connectivity Link ▶ Red Hat Data Grid ▶ Red Hat Trusted Artifact Signer ▶ Red Hat Trusted Profile Analyzer	▶ Red Hat OpenShift Service Mesh ▶ Zero trust workload identity manager ▶ Red Hat OpenShift Serverless ▶ Streams for Apache Kafka ▶ Power monitoring for Red Hat OpenShift ▶ Red Hat Advanced Cluster Management for Kubernetes