DigitalOcean

Kubernetes adoption journey for startups and SMBs

Table of contents

The **decision to adopt Kubernetes** shouldn't be taken lightly. It's a complex system that requires hard work and dedication. In fact, when it comes to using Kubernetes, the best value comes from adopting a DevOps culture. Kubernetes isn't just installing something on a virtual machine (VM), it's a transformative way of viewing IT and enabling your teams. Automation is a direct path to becoming more productive. Kubernetes can make every aspect of your applications and tools accessible to automation, including role-based access controls, ports, databases, storage, networking, container builds, security scanning, and more. Kubernetes gives teams a well-documented and proven system that they can use to automate so that they can focus on outcomes.

Startups and small-to-medium-sized businesses (SMBs) are uniquely positioned in their journeys. Unlike enterprises, they typically don't have dedicated departments for operations or security, and they usually run lean development teams. At the same time, SMBs are ultra-efficient, sometimes going from discovery phases to production in a matter of two or three weeks. When startups, SMBs, and individual developers consider adopting Kubernetes, they need to retain their ability to be flexible and pivot quickly while benefiting from the scalability and automation provided by Kubernetes. To achieve both without large and ongoing maintenance overhead, SMBs may consider the hands-free operations provided by Managed Kubernetes offerings.

In this guide, you'll find recommendations for every phase of the Kubernetes adoption journey. From Discovery and Development through Staging, Production, and ultimately Scale, we'll offer a simple yet comprehensive approach to getting the most out of Kubernetes. **We'll walk you through some best practices for any Kubernetes deployment, plus recommendations for tools and how to get started using DigitalOcean Kubernetes, a managed Kubernetes offering.**

If you would like to experience hands-on tutorials from day-1 to production, please follow the **Kubernetes Adoption Journey**. It provides detailed step-by-step guidance on how to adopt Kubernetes, irrespective of the phase of your development cycle.

If you are new to Kubernetes, we highly recommend the Kubernetes starter kit for DigitalOcean. It provides a comprehensive tutorial and automation stack for getting to production-ready kubernetes.

Breaking down the phases of adoption

Discovery

Explore setups, resources, and options for implementing Kubernetes for your business.

Development

Deploy and test iteratively in a Kubernetes environment without spending time in operational activities.

CI/CD

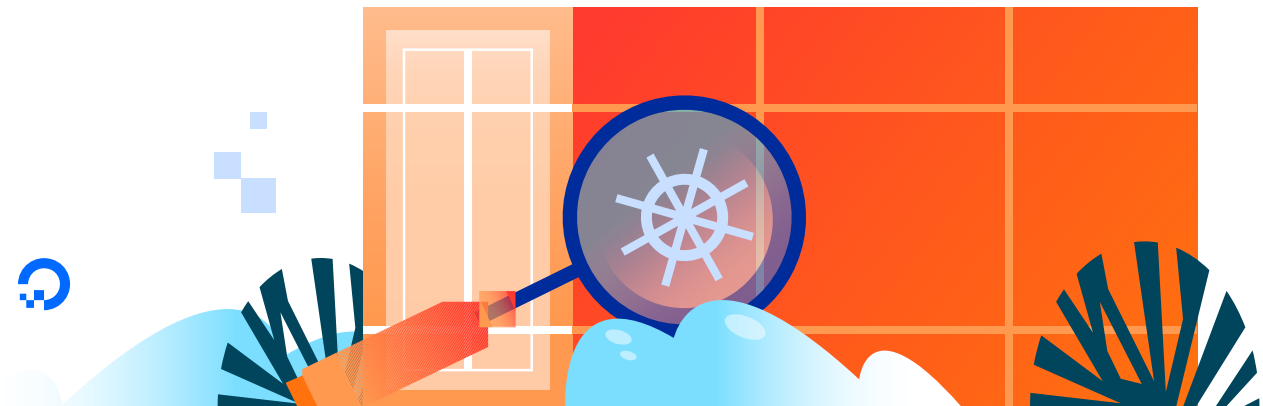
With continuous integration and continuous delivery, set up an automated pipeline for building and deploying application code to Kubernetes.

Production

Create a production-ready Kubernetes cluster.

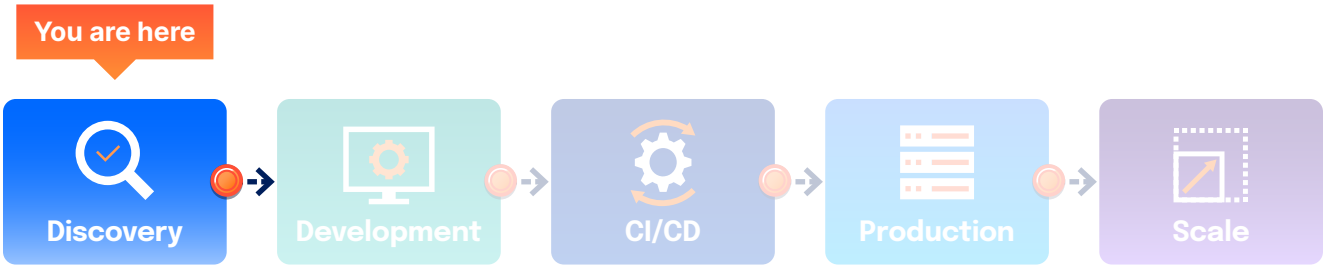
Scale

Utilize more sophisticated tools to optimize Kubernetes deployments for continued reliability and scale.

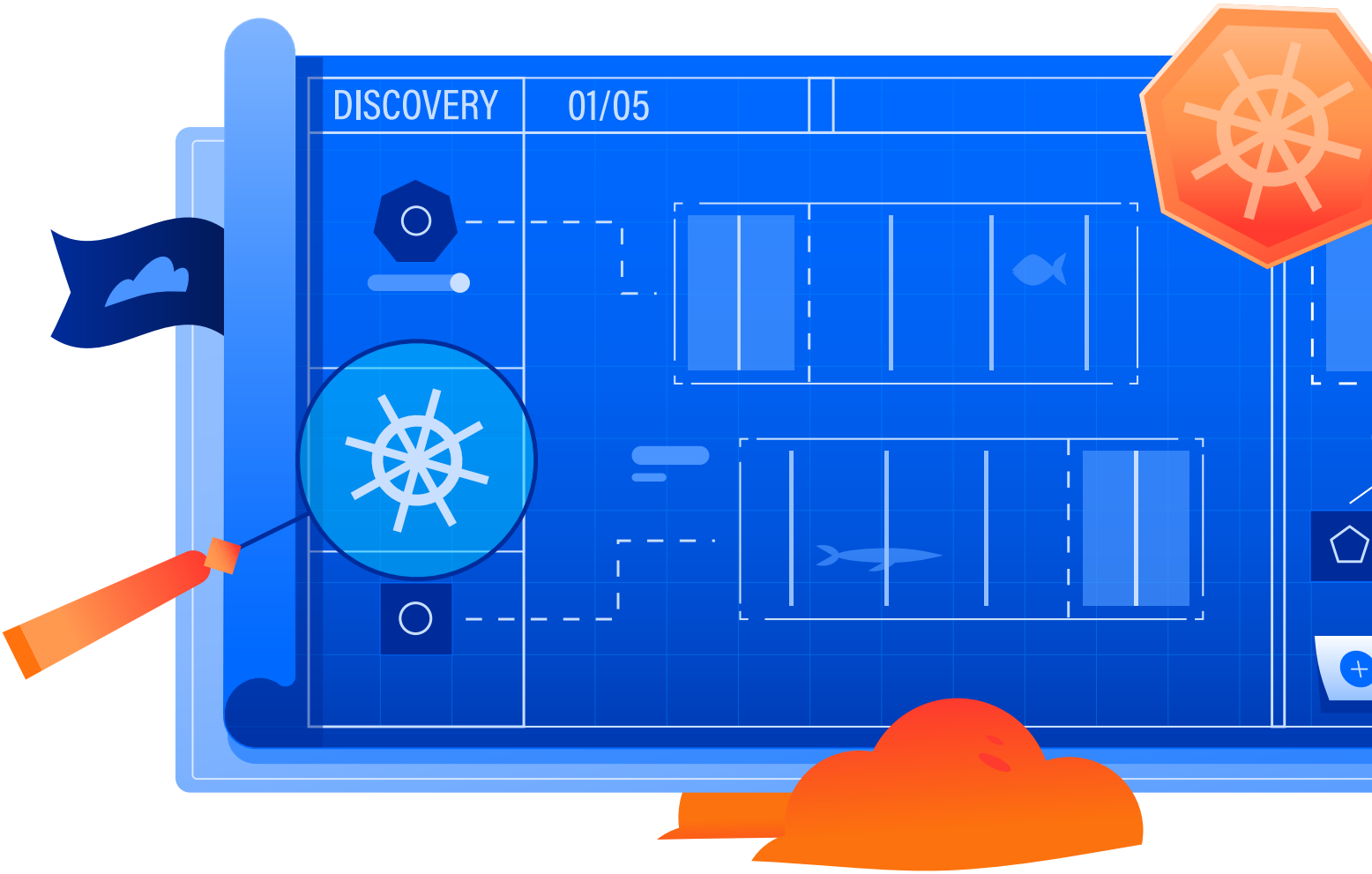


Discovery

Discover what you need to be successful with Kubernetes



During the Discovery phase of the Kubernetes adoption journey, teams will explore their options for using Kubernetes and begin to plan their infrastructure on a block level or by grouping together one or more infrastructure resources. The Discovery phase of the Kubernetes adoption journey is unique to you and your business, and you'll decide how your application will best map to a Kubernetes infrastructure. With effective planning, you can move from Discovery to Production and Scale in as little as 4-6 weeks.



Plan infrastructure components

As your team begins to plan infrastructure inclusive of Kubernetes, find out what tech stack you need to be most successful, both now and in the future. Understanding the reason you want to use Kubernetes is key to deciding what types of components you'll need to evaluate. Whether you're a developer interested in hassle-free Kubernetes management, a startup needing a faster time to market, or an SMB that needs dynamic scaling capabilities, managed Kubernetes offerings like DigitalOcean Kubernetes can help you achieve your goals.

As you architect your infrastructure, consider the following:

Kubernetes installation

Consider if your team will use a **managed or self-hosted** version of Kubernetes. While there are times when it makes sense to self-host Kubernetes, maintaining self-hosted clusters is time and resource intensive and requires a level of expertise that many small businesses lack. Managed Kubernetes like **DOKS** bears the burden of managing the control plane, ensuring high availability with the latest security patches and release updates, providing autoscaling capabilities, and more. Businesses leveraging Managed Kubernetes often have a faster time to market.

> **"DigitalOcean Managed Kubernetes has totally changed the way I think about deploying applications. In an ad push, we'd have to know ahead of time to spin up servers and it could take a whole afternoon. Now, I can click a button and it's done."**
>
> Joshua Verdehem, Co-Founder, Loot.tv

Workload distribution

Do all workloads run on Kubernetes or are they distributed across Kubernetes, VMs, and other compute platforms? For example, you may want to keep frontend running on Droplets while moving backend jobs to Kubernetes and adding a load balancer and a managed MongoDB.

An opportunity for multi-cloud

Open source software like Kubernetes allows startups to avoid vendor lock-in by facilitating migrating projects to other clouds or spreading workloads across multiple clouds, while Infrastructure as Code (IaC) tools like **Terraform** easily provision environments. With Kubernetes, teams can pick up their whole infrastructure and run it the same way in different environments, easily moving networking, applications, databases, and even things like secrets. Using Kubernetes, developers can mix and match services from different providers to best fit their solution.

Kubernetes provides a consistently proven system across clouds that can lessen the technical debt of multi-cloud environments. If you include Infrastructure as Code (IaC), tools like Terraform, Ansible, Pulumi, and many more have further simplified provisioning across clouds, all of which can be fully automated through DevOps workflows.

Sizing your cluster

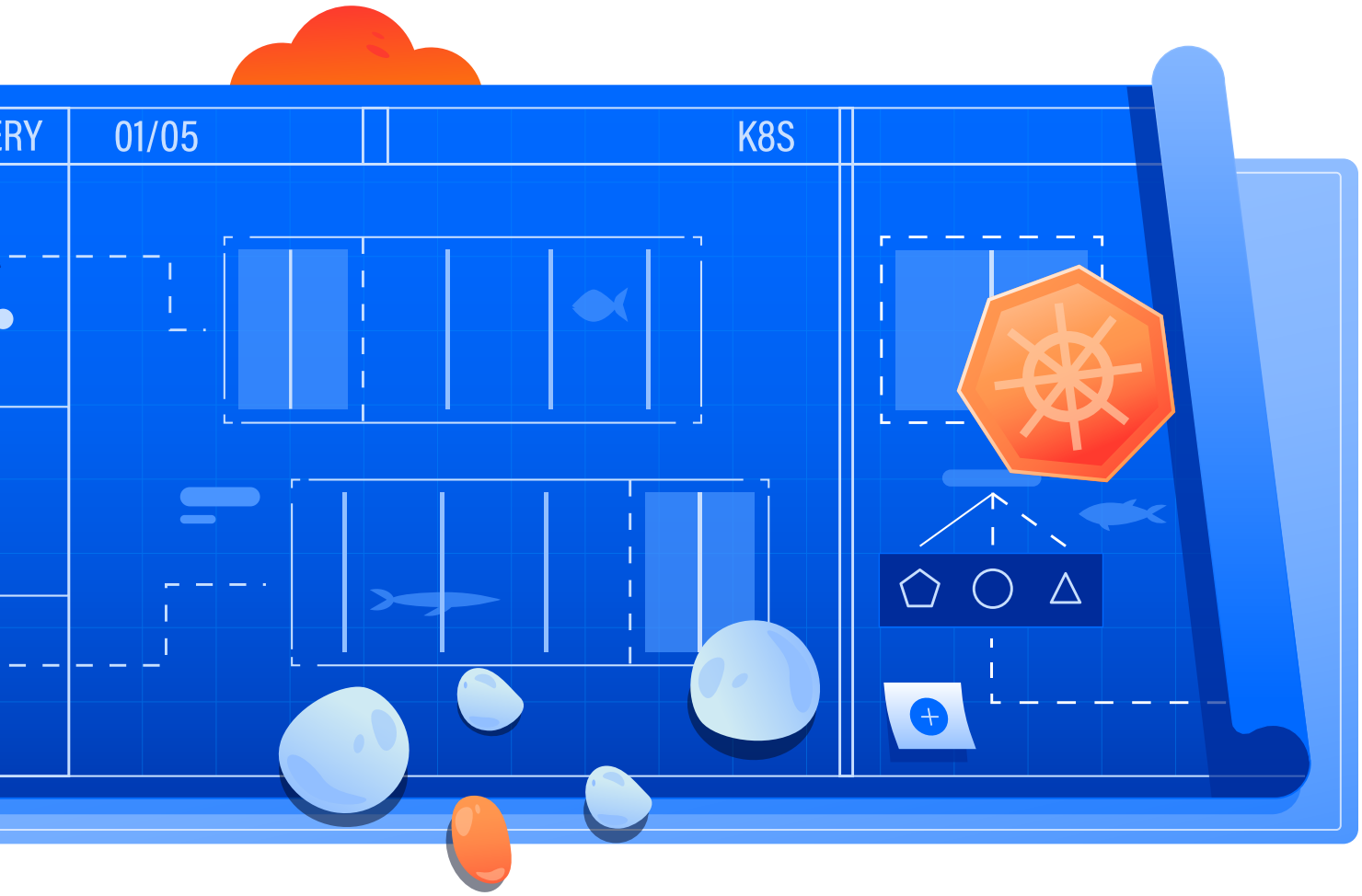
An oversized cluster is inefficient with its resources and costs more, but an undersized cluster running at full CPU or memory suffers performance issues and can crash under stress. Look at your **node size and count** to determine the overall CPU, RAM, and storage of your cluster. **Larger nodes** are easier to manage, in many cases are more cost-efficient, and can run more demanding applications; however, they require more pod management and cause a larger impact if they fail.

Managed Databases

While a large business can usually afford to hire employees with a deep knowledge of databases, smaller teams usually have fewer resources available. This makes tasks like replication, migrations, and backups more difficult and time-consuming. Managed Databases handle common database administration tasks such as setup, backups, and updates so you can focus more on your app and less on the database.

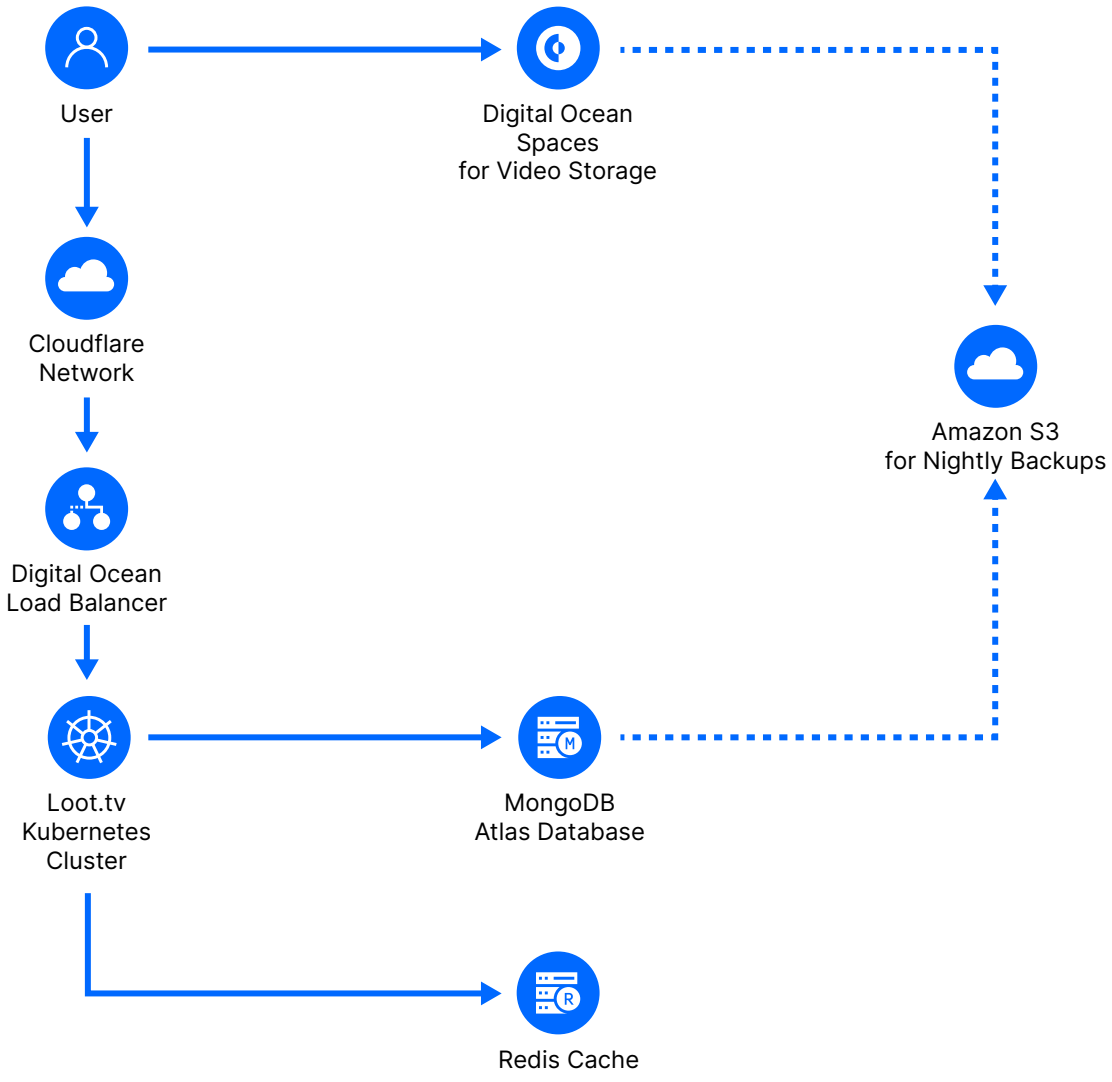
1-Click applications

Certain cloud providers have 1-click applications. For example, **DigitalOcean Marketplace** provides a variety of apps and stacks that you can install to run pre-configured container images on Kubernetes clusters. Most **Kubernetes 1-Click apps** are Helm charts under the hood and require Helm 3 package manager to run.



Loot.tv

Loot.tv is an engaging video platform where creators and communities can earn rewards for uploading and interacting with their favorite content. DigitalOcean's low bandwidth costs and SLA guarantees allow Loot.tv to effectively serve its customers while paying a fraction of the cost for bandwidth that they would pay on hyperscalers. DigitalOcean Managed Kubernetes allows the Loot.tv team to focus on building their application rather than maintaining infrastructure. DigitalOcean Kubernetes also provides them with reliable scaling to match demand. Month to month Loot.tv can see three petabytes of data transfer and average 8,000 viewers at any one time.



Development

Testing and deploying applications iteratively in a Kubernetes environment

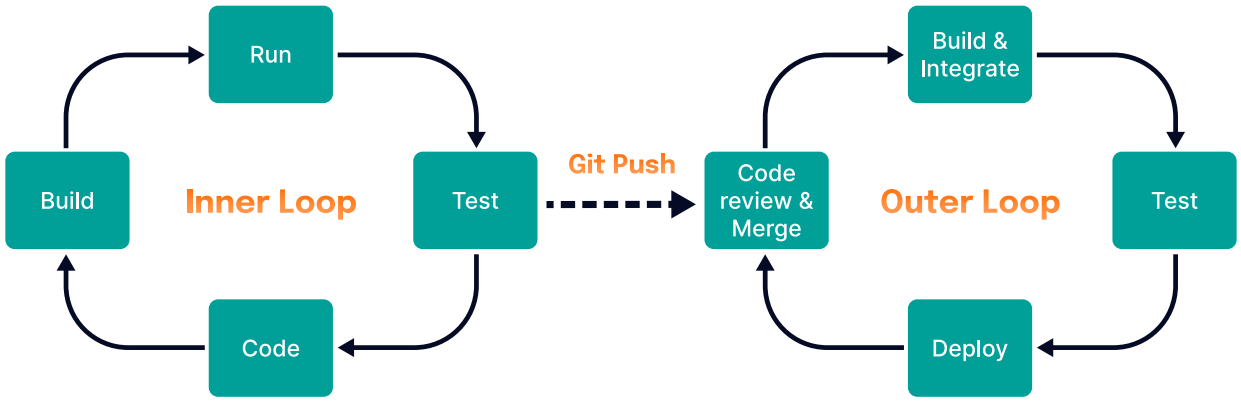
Prerequisites: Discovery is complete. You have identified the resources, tools and setup needed to incorporate kubernetes for your application deployment.

Goals: Create a kubernetes cluster for development work. Install and configure necessary tools for developer efficiency.

The Development phase of the Kubernetes adoption journey focuses on developer efficiency. Significant operational complexities come into play from the time the developer commits their code until it's deployed in a Kubernetes cluster. Understanding an optimal developer workflow can help teams build, test, and deploy faster, ultimately enabling the business to increase productivity and innovate faster. In order to move into the Development phase, your team needs to have the basic tooling for Kubernetes in place, and the application should be running in a development environment.

Development workflow in Kubernetes

A typical developer workflow consists of several cycles of an inner development loop and an outer development loop. An Inner Loop includes coding, building, running, and testing the application. It takes place on the developer's machine before committing the code to the source code repository. It's the phase before the developer shares the changes done locally with others. An Outer Loop is made up of several cycles, including code merge, code review, build artifact, test execution, and deployment in a shared dev/staging/production environment. Everything in this loop should be automated as much as possible. The developer's "commit and push" to the source code repository from the inner development loop triggers the outer development loop.



As a developer, if you code six hours per day (360 minutes) in a traditional local iterative development loop of eight minutes, you can expect to make about 45 iterations of your code per day. When you bring "containerization" in, it takes additional four minutes to

- Package code into container images.
- Write a YAML-based Kubernetes manifest.
- Push the image to the container registry.
- Deploy containers in the Kubernetes cluster.

This, in turn, reduces the iterations to about 30 per day.

Building and developing containers can slow down this inner development loop that impacts the cycle time, a "hidden tax" in the container-native inner loop of application development.

The default inner loop in Kubernetes takes a long time and impacts developer productivity. Due to this, the CI/CD pipeline should not be an inner loop for development. The primary focus of the development phase is to create a development setup that makes the inner loop faster, just like you are building an application and running it on your laptop.

In order to achieve the inner loop of application development, teams need to focus on three areas. First, creating the development cluster, including local and remote setup. Next, installing basic tools for working efficiently with the cluster, and finally, setting up development tooling and workflow for rapid application development.

Create a development cluster

Teams can use either a local or remote development cluster. There's no hard and fast rule, and Kubernetes users can be successful either way. Consider your use case and resources when deciding which is right for your application.

Options/Capabilities	Remote Cluster	Local Cluster
When to use	• Need resources on demand • Minimal need for K8s knowledge • Need for microservice team	• POC and Experimentation • Small team of experienced K8s practitioners • Low computational requirements
Advantages	• Free from control plane management • Realistic environment • No compute limitations • Controlled access to the control plane API	• No operational costs • Reduced blast radius due to environmental isolation • Full cluster access with complete freedom
Disadvantages	• Operational compute cost • Access management gets difficult at scale	• Hardware constraint • K8s knowledge required • Potential environmental disparity to a cloud setup

Local development cluster

There are several options to choose from to create local Kubernetes clusters for application development, such as **Docker Desktop**, **Kind**, **MicroK8s**, **K3d**, **Minikube**, and **Rancher Desktop**.

If you're already using a local cluster of your choice and it's working well, don't change it. If that's not the case, use **Docker desktop** for entry-level developers on Kubernetes as it is one of the easiest, fastest, and most secure ways to build containerized applications. It's a native application designed by Docker for both Windows and MAC users. Built on top of the Docker engine, it's a perfect tool for beginner-level developers who are looking to leverage the Docker ecosystem to increase development velocity and delivery of modern container applications. Docker Desktop packages Docker Engine, Docker CLI client, Docker Compose, Kubernetes, and Credential Helper into one single big component.

Key capabilities like simple graphical UI, easily mapped Storage Volumes, managed kernel patches and security updates of the linuxkit VM, ready-to-operate single node Kubernetes cluster, and tailored support of multi-architecture image building with fairly large community support make it a strong differentiator amongst others.

Remote development cluster

While there is an indispensable benefit in application development in the local Kubernetes cluster, it is still limited by the CPU, Memory, and disk constraints of the local environment. You would benefit from a remote Kubernetes cluster for your application development for two main reasons

- Remotely distributed teams can contribute to the same project to the same Kubernetes cluster
- On-the-fly changes are possible that do not hinge on a local machine's capabilities

Digital Ocean's **Managed Kubernetes** provides easy-to-run Kubernetes in the cloud without the pain of setting up and maintaining the control plane. It takes away the management overhead of configuring, maintaining, and scaling the cluster so that software teams can focus on optimizing their software applications. The control plane is also free, which helps keep the cost lower.

> **"The fact that it's so easy to configure, administrate and scale with DigitalOcean is something which I love. I've worked with Kubernetes before, hands-on, self-hosted, but the DigitalOcean integration and provision of Kubernetes has been the most seamless that I've experienced so far."**
>
> James Hooker, CTO, Hack The Box

If you need multiple developers sharing a single remote development cluster, we recommend telepresence. See the tooling section below for details on telepresence.

Learn Step-by-Step on how to incorporate DigitalOcean Kubernetes as a **development cluster**.

Basic tool installation

Kubernetes comes with prepackaged tools that are suitable for basic operations. We recommend the following tools from the Kubernetes community for your day-to-day work with Kubernetes.

- **kubectl**: To communicate with the Kubernetes Control Plane, i.e., API server using the K8s API.
- **Kubectx & kubens**: Kubectx to switch context between multiple Kubernetes clusters & Kubens to switch between namespaces.
- **Stern**: To tail multiple pods on Kubernetes and various containers within the pod color coded for quicker debugging.
- **k9s**: Terminal-based UI to navigate, observe and manage your deployed applications. If you like vi, you will love k9s. Alternatively, you can use lens or Kubernetes dashboard.
- **k8syaml**: Web-based UI to help developers build YAML files for Kubernetes cluster deployments.
- **Kustomize**: Kustomize introduces a template-free way to customize application configuration that simplifies the use of off-the-shelf applications.
- **Helm**: An Omnipresent Kubernetes package manager to install and uninstall off-the-shelf apps.

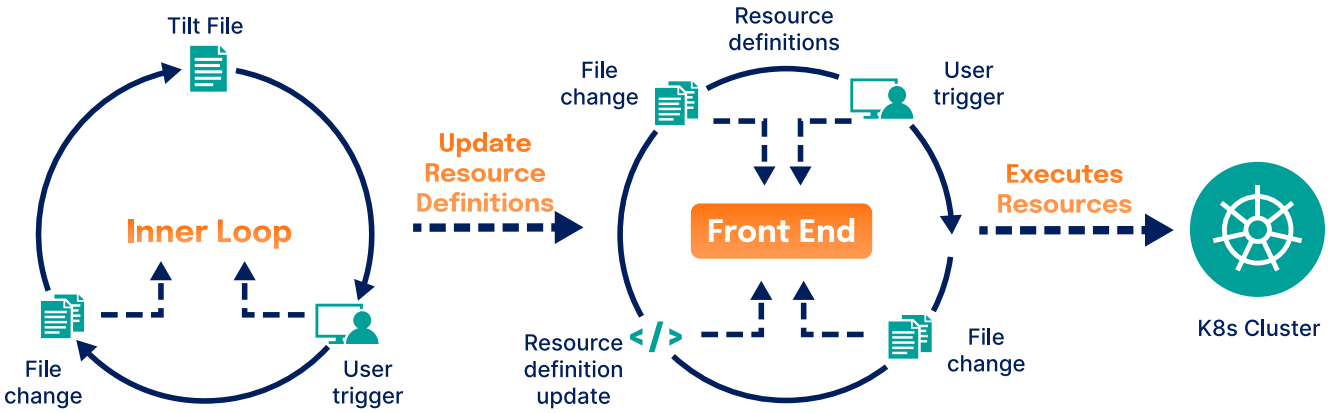
Development tooling for Inner Loop

There are several popular Kubernetes development tools for automatic build, push and deploy.

- **Tilt**
- **Skaffold**
- **DevSpace**
- **Telepresence**

If you're already using a tool, such as **Okteto**, **Gitpod**, or another choice, keep using it. If not, we have the following recommendations for local vs. remote clusters.

We recommend Tilt for teams with little-to-no Kubernetes experience. Tilt simplifies the Kubernetes workflow by automating and abstracting away the process of building and deploying container images in a Kubernetes Cluster.

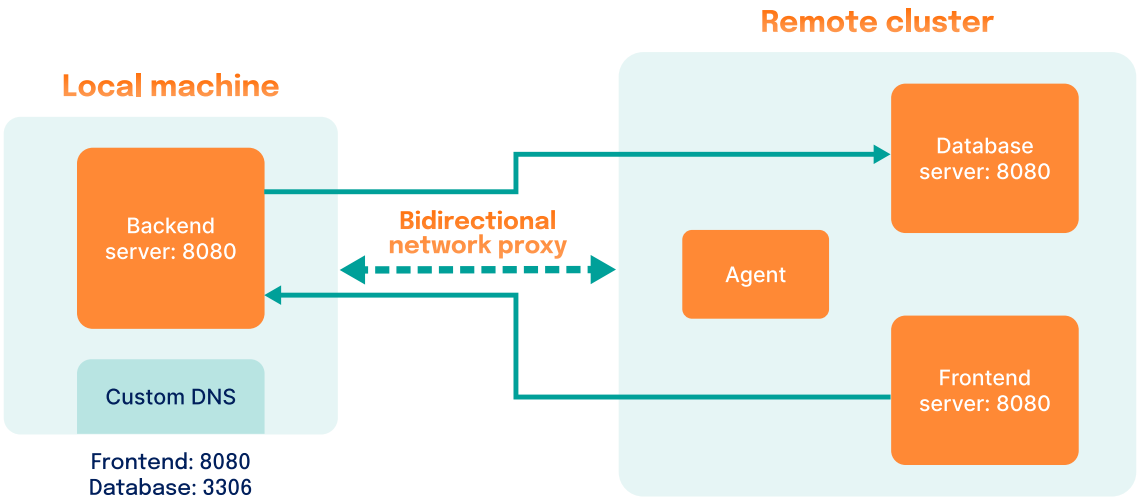


Tilt helps in creating continuous delivery pipelines for both your local and remote development environments. It keeps track of the changes made to the dockerfile and automatically builds, pushes, and deploys them to the live containers in the Kubernetes clusters in real-time. The Heads-up display in the terminal and the Web-UI are the two key features that differentiate Tilt from its competitors, which provide in-depth information about your service, health status, build and run-time logs, and more.

The main benefits of Tilt are

1. Easy onboarding capability with tilt up command
2. Service health status monitoring
3. Fast feedback loop
4. Quick visibility into errors via UI that enables developing interrelated microservices for any application
5. Highly versatile as it works well with Helm charts, Kustomize and Docker-Compose, enabling faster deployment of multiple microservices

We recommend **Telepresence** for multiple developers sharing a single remote DOKS cluster. Telepresence is an open source CNCF project that lets you set up a local development environment that connects to a remote Kubernetes cluster. You can make code changes locally and immediately see the impact of those changes in the remote Kubernetes cluster.



The key feature of Telepresence is the bidirectional network proxy that is created between the process that runs in the development machine and the pod in the remote Kubernetes cluster. This way, it magically transforms your laptop into a pod that runs in a remote Kubernetes cluster, making it look like your local environment is inside a pod. You can launch your application locally and tell Telepresence to intercept and route all the traffic to your local port.

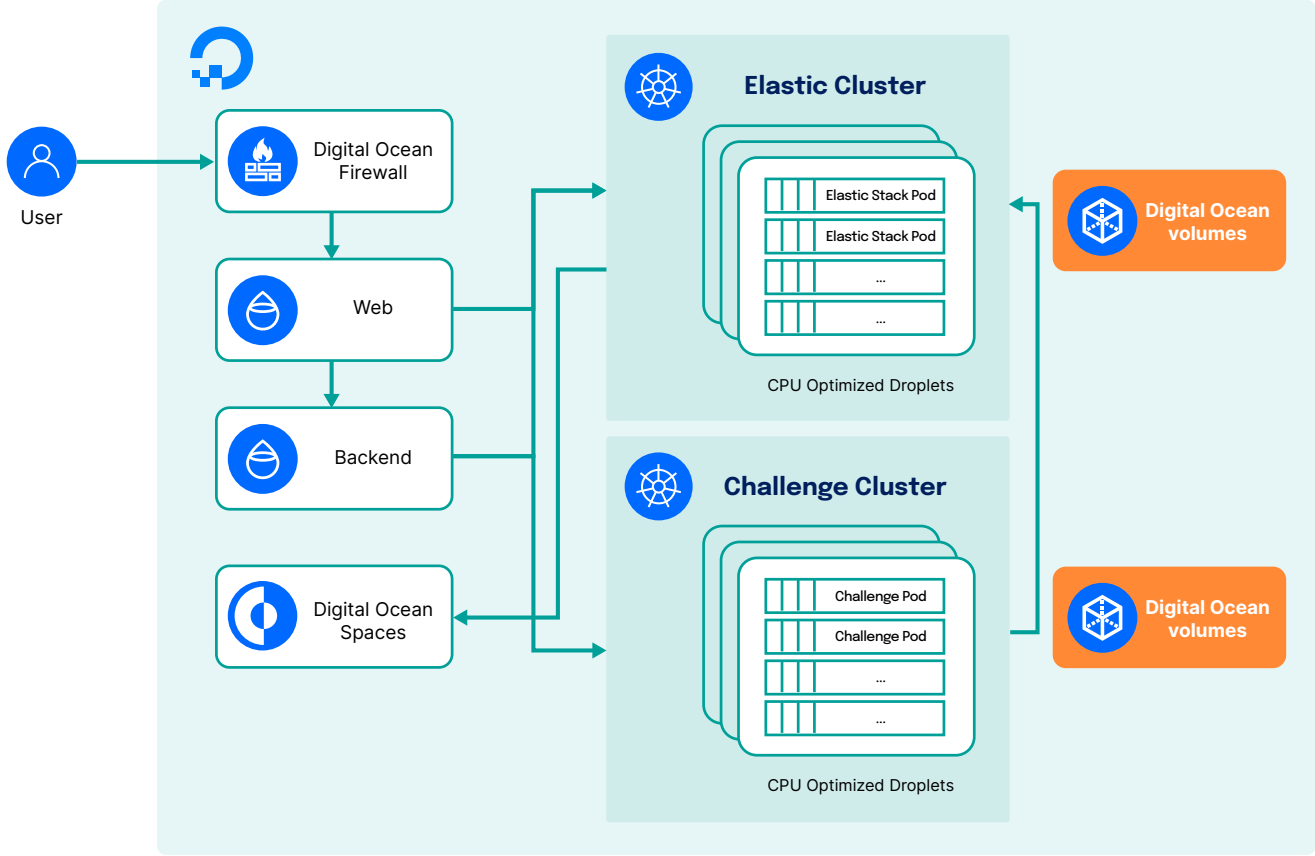
The main benefits of using Telepresence are

1. Quick debugging and analysis of several microservice-based applications at once
2. Quick development without a local Kubernetes cluster
3. Ability to preview the created environment with the team
4. No need for code-sync, redeploys, repackage of docker images, and hot-reloads

Hack the Box

Hack The Box Team transitioned to Kubernetes, completely replacing the Docker and Docker Swarm environments with DigitalOcean's Managed Kubernetes offering. Hack The Box Challenge environments need to work quickly and also provide strict network policies for users, enabling them to access parts of the machine required to complete challenges but preventing access to outside networks. With DigitalOcean's managed Kubernetes, the team can use Kubernetes to handle the application and required network policies while scaling to meet traffic needs.

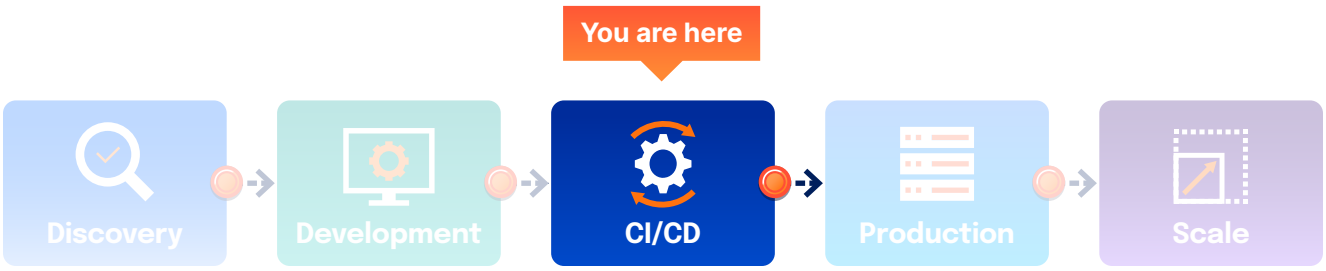
Utilizing Kubernetes has enabled the Hack The Box team to tremendously scale Capture The Flag, a platform that allows users to compete in challenges with other members. Recently, Hack The Box hosted Cyber Apocalypse, their first-ever global community Capture The Flag event. That event saw 9,900 players sign up, compared to their typical few hundred users. Even with thousands of users and 61 challenges to choose from, DigitalOcean Kubernetes seamlessly scaled up backend clusters and supported containers needed to host the event.



CI/CD:
The code-to-delivery pipeline

Set up an automated pipeline for building and deploying application code to Kubernetes.

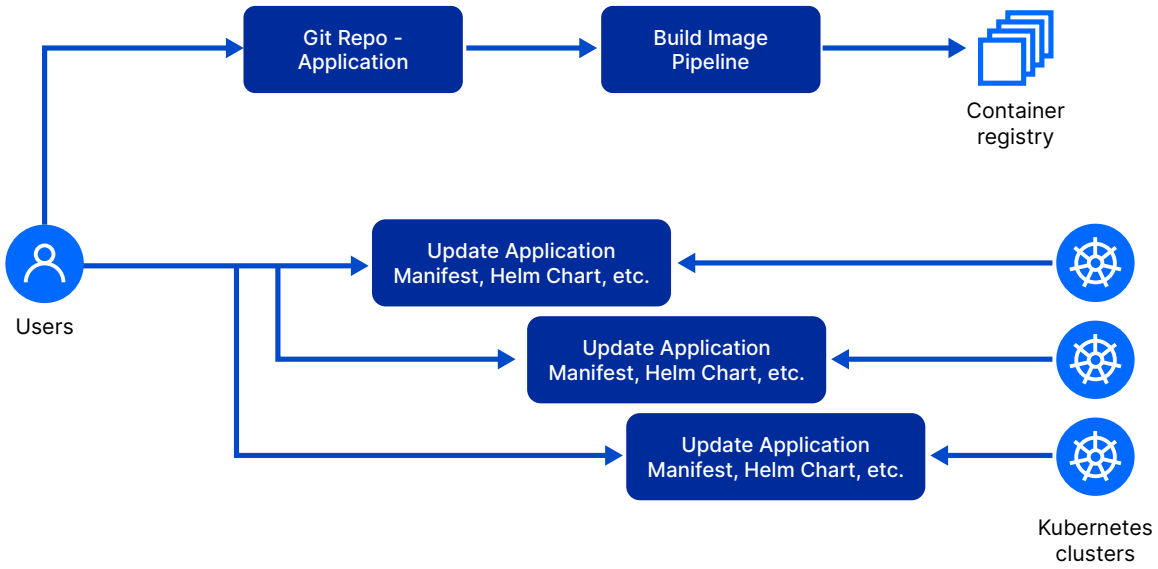
What you need to get started: You have already created a development setup and the necessary client tools. Your application is running in a development environment.



Now that the application is running in a development environment, you're ready to plan for the production deployment. CI/CD (Continuous Integration and Deployment) readiness is a journey that's more about DevOps and automation practices than the specific tooling. Teams need to have good test coverage, for example, before they can automate releases. A good CI/CD process should enable teams to recreate their applications anywhere (minus the user data) in a matter of minutes. Consider the needs of your team as you explore CI/CD options. Depending on the nature of your application and shipping frequency, you may not even need a full-fledged CI/CD.

CI/CD Overview

Let us go over the high-level overview of the code-to-delivery pipeline and the typical challenges you will encounter. Refer to the diagram below.



There are broadly four stages in going from code to delivery.

1

Git Repository: Committing application code to the git release branch kickstarts the pipeline. The most important decision you will make is your branching strategy. There are various types of **branching options**.

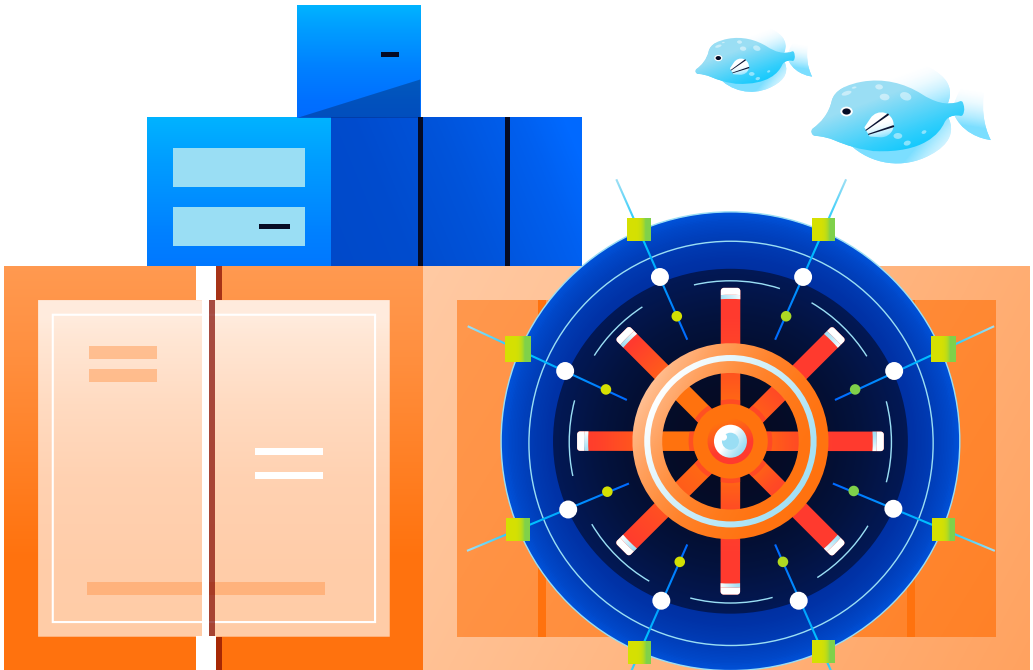
- **GitHub flow:** Use GitHub flow to create a branch for your feature work. When your feature is ready, submit a pull request (PR) to the main branch. GitHub flow works great for small teams.
- **GitLab flow:** With GitLab flow, you can maintain separate branches for different environments (pre-prod, prod, dev) and promote code as needed. This works well when you want to control when you deploy code to a certain environment.
- **Trunk-based:** Developers using trunk-based branching are continuously committing code to the main branch. No branching, no PR. This works well if you have seasoned developers and mature testing and rollback processes.

If you already have a preferred branching structure, stick to that. Otherwise, we recommend GitHub flow. It is well-understood and simple to start. Pull Requests are the source of truth for the application history, so it's important to keep the PR well-scoped and self-explanatory.

2

Build Pipeline: At any time, you have one or more application repositories with multiple branches and ongoing development. As you continue to update your code, you need to build new images for testing. This is done using a build pipeline. You can trigger an image build for every PR or decide to do it once a week or based on any other trigger. It comes down to your business needs. The issue with delaying the build is that you will discover issues much later, and it will be hard to troubleshoot. So it is a recommended practice to build and publish a new image for a reasonable set of code changes (PR). A well-scoped PR makes it much easier.

For building images, there are many tools (managed or self-hosted). We recommend using a managed provider for the image build pipeline. For the image registry, use **DigitalOcean container registry** (DOCR) unless you already have a working registry like Docker Hub.



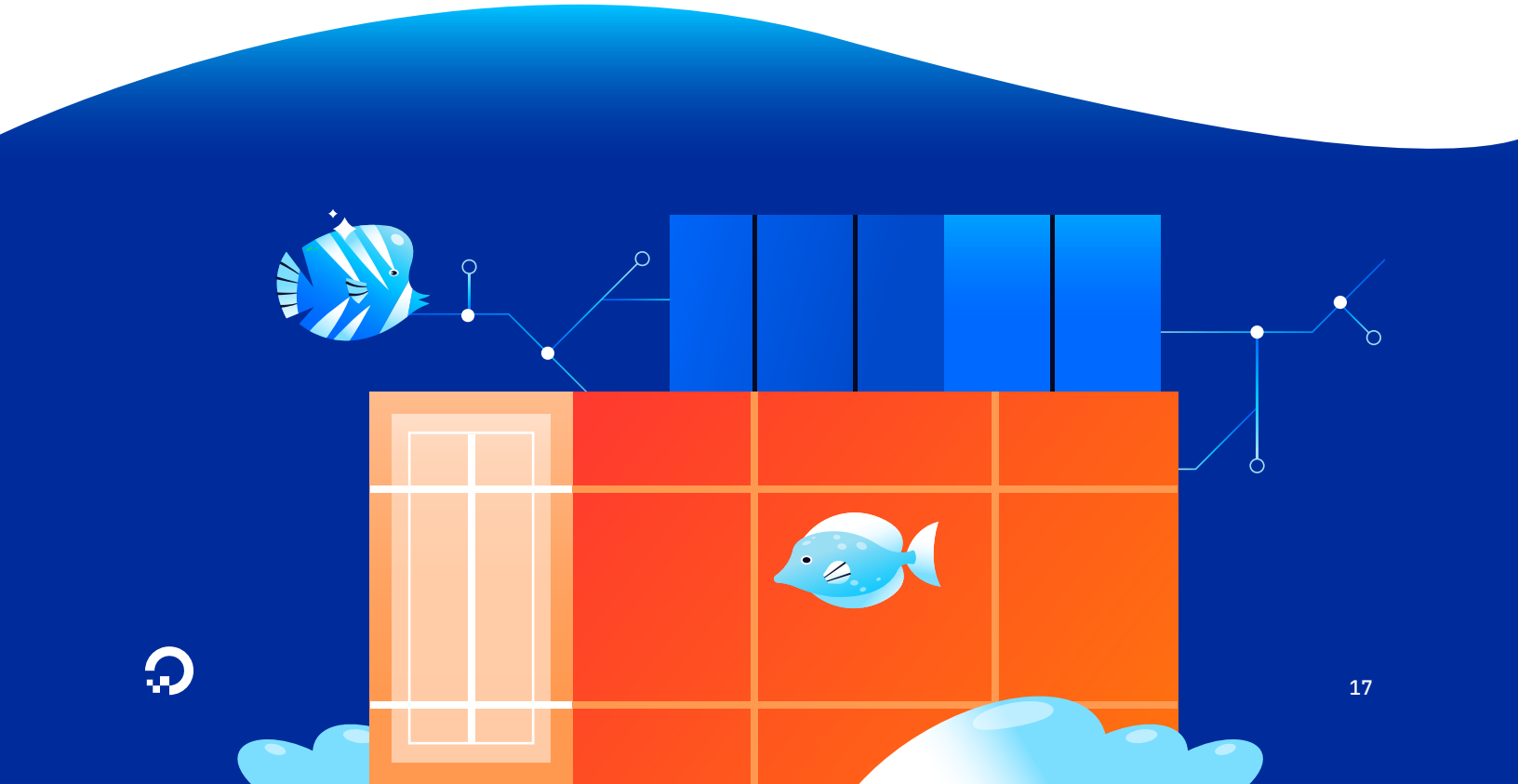
3

Application Manifests: The CI pipeline has pushed the new application image (with a new tag) to the image registry. Now, you need the application image to be deployed onto the cluster. A Kubernetes cluster needs application metadata (YAML file, Helm chart, etc.). You must store and version the application metadata in the git repository. You may decide to have the same repo for both applications and metadata or use separate repositories. We recommend using the same repo for both applications and configuration.

Another important aspect is promoting your code changes to the staging and production environments. When you have a new image available, you must first deploy it to the staging environment. Deployment to the staging environment should be automated for every image push. So the continuous integration (CI) pipeline should update the metadata for the staging environment. For production, we recommend manually approving changes, meaning an individual manually submits a PR to update the application manifest for the production environment.

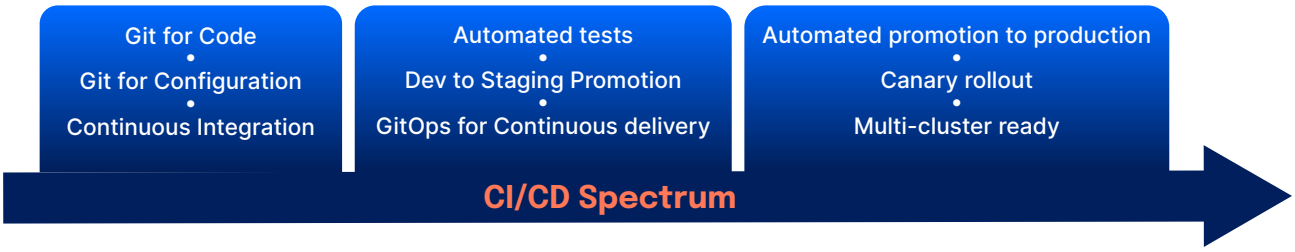
4

Rollout: The issue with Kubernetes deployment kubectl apply -f is that you're just overwriting the existing application, and all the users will have to interact with the new application. In case of an issue, you will have to do additional work to roll it back. That's why the general practice is to roll out the new version of the application slowly. It can be one replica at a time, cluster at a time, blue-green deployment (rollout a copy of the new replica first), canary deployment (rollout one replica and redirect a subset of the production traffic), or a mix of these. The whole idea is to reduce the surface area and roll back quickly if the new version does not work well in production. We don't recommend this at the start. Your first focus should be to get the build pipeline working, enable proper automated testing in the CI pipeline, and set up automated deployment to the staging environment.



Challenges in CI/CD

CI/CD readiness is a journey. DevOps and automation practices make a larger impact on success than the tooling that's chosen. To achieve your goals, identify and implement the right level of automation needed at the current stage of your business. Always start by deciding your objective for the CI/CD. This should be driven by your current business needs and skillset. As you build the appropriate level of automation for your needs, challenges may arise with structuring your environments, test coverage, and handling Drift, rollbacks, permissions, and secrets.



Structuring your development, staging, and production environments.

It's necessary to have at least one staging environment in addition to production clusters. For these environment(s), you will need to worry about the cost and setup.

- If you start allocating one remote dev cluster for each developer, it will quickly increase costs. Refer to the Development stage for setting up the dev clusters.
- Your application likely uses load balancers, block storage, object storage, managed databases, external SAAS APIs, and other tools. You will need to customize the development manifests for different environments. This is a one-time effort.
- You may likely have many production clusters, meaning you're looking at many copies of the application manifest with different levels of customizations. Structuring this right is very important. This will avoid manifest sprawl and improve maintainability.

Test coverage.

This is the hard part of a good pipeline. This is no different for Kubernetes than any other application. Teams need to invest the time in writing down unit tests and any other tests to reduce manual testing time for every iteration of the application.

Drift in the cluster vs. Git repository.

By maintaining the source of truth of your cluster in the Git repository, you can manage your application via the version control system. But how do you figure out if the cluster state has drifted because of manual or external changes? Traditionally CI/CD pipelines employed a push model for deploying applications. With Kubernetes, GitOps models, popularized by Flux and Argo, have become sought-after. In the GitOps model, a GitOps controller runs on the cluster and is responsible for synchronizing the state of the cluster with the specified Git location. See **this guide** for more information if you are new to GitOps. We recommend using a GitOps tool (ArgoCD or Flux) along with the CI pipeline.

Rollback.

When you keep your application metadata in the Git repository, rollback to a prior version of the application may be tricky sometimes. While it sounds pretty straightforward to bring the Git folder to a prior version and let the cluster synchronize the metadata, complications can occur.

As an example, consider a team using WordPress version 1.18 via GitOps that upgrades the WordPress manifests to 1.19 and lets the cluster synchronize. If they realize something broke along the way, they'd be required to switch back to 1.18. This can easily be done by changing the Git folder manifests, but what if the application fails while applying 1.18 manifests? Scenarios like these are why it's important to test properly before deploying to production and why it's important to keep the PRs atomic.

Permissions.

Some businesses may demand different permissions structures. For example, it may be necessary to separate access rights for different employees for infrastructure, applications, and more. In this case, teams must start with the business need and then build the repo structures, with the goal of avoiding manifest code duplication and ensuring that they can quickly roll out new clusters with the same applications and configurations.

Secrets.

Cloud credentials, third-party SaaS credentials, DigitalOcean Spaces tokens, and private keys are all examples of secrets. Secrets are normally referenced in a YAML file in Kubernetes, which is a base64 encoded text, as is plain text. Secrets shouldn't be exposed outside the cluster, particularly in a Git repository, creating a challenge because the manifests are kept in the Git repository. There are many tools available to encrypt the secrets outside the cluster—we recommend one of the following:

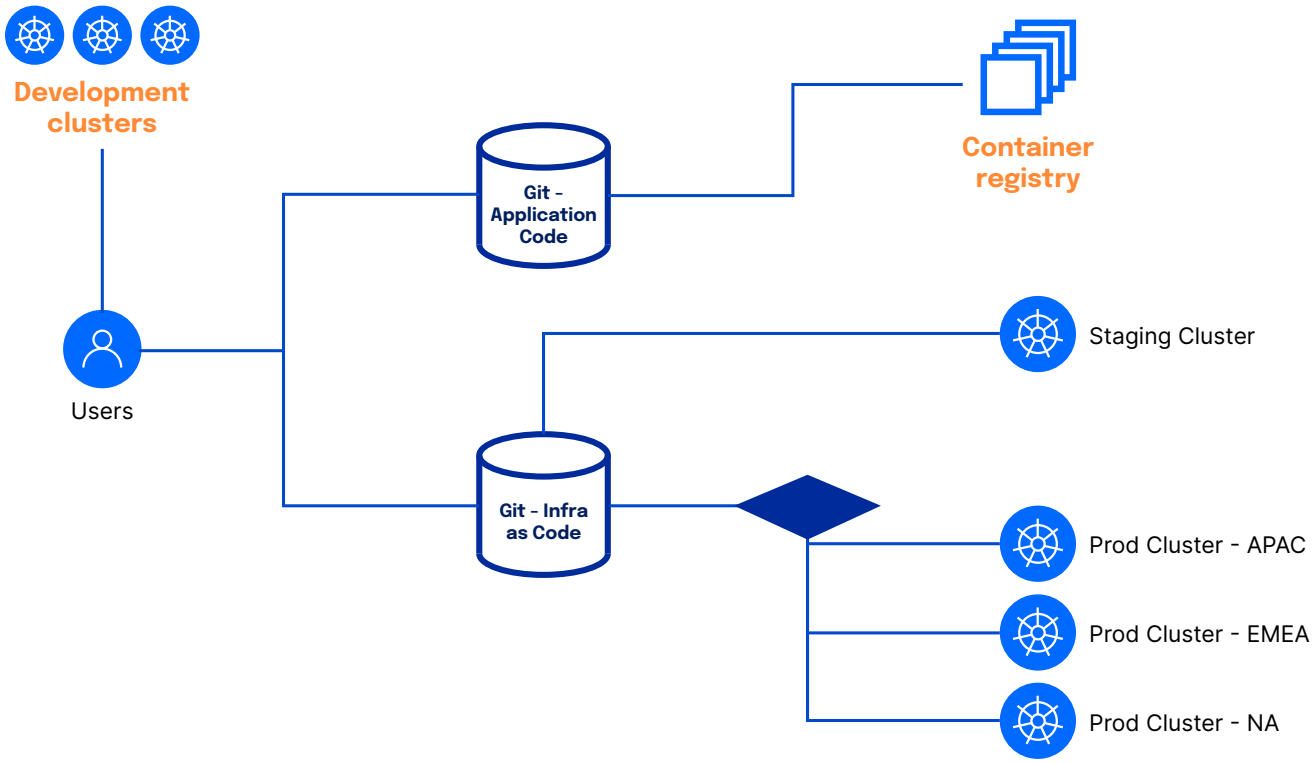
- An external secrets manager using a Kubernetes CRD (External Secrets operator) to interface. If you're already using something like Vault, AWS/GCP/Azure secrets manager, or 1password, then setting up an external secrets operator on your cluster is all that's required.
- A sealed secrets controller to enable encryption of secrets outside the cluster. This is popular in the GitOps world.

Our recommendation is to choose the external secrets operator as the first option because it has better security and less management overhead. See this **section** of the tutorial for the external secrets operator setup with Vault. If you prefer to use sealed secrets, then refer to this **tutorial** from the starter kit.

Recommended steps

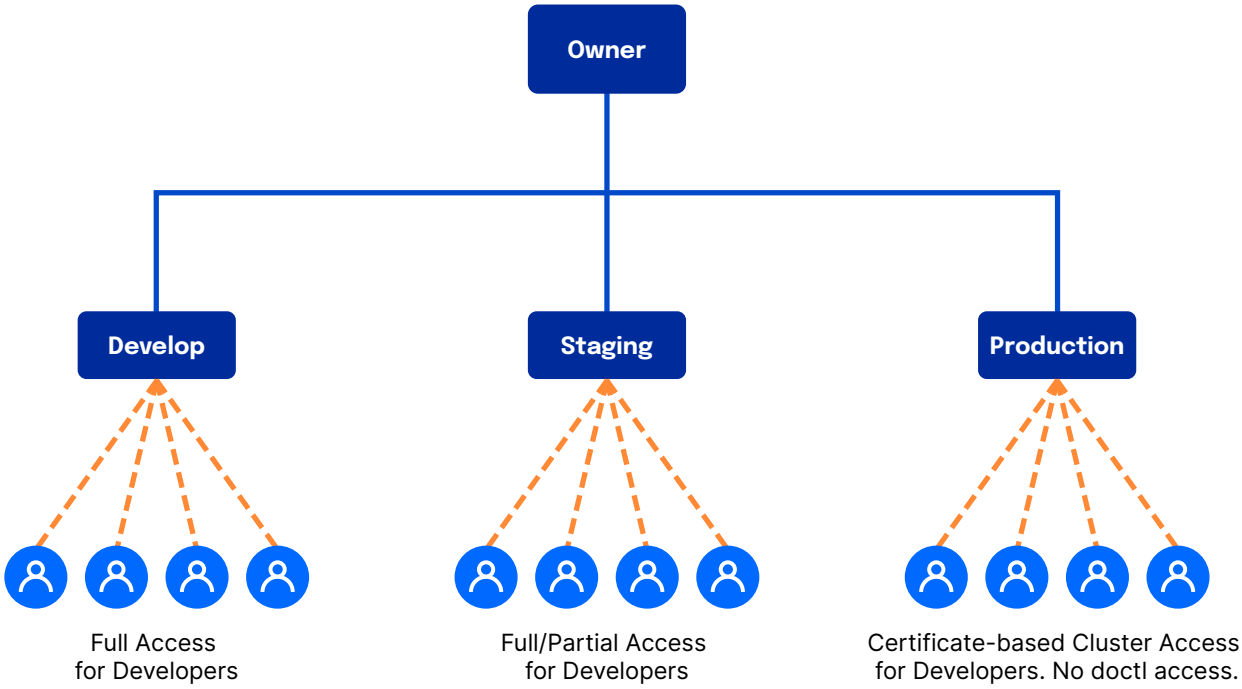
There are several approaches to setting up CI/CD, and which approach you choose will depend on the needs of your business and the expertise of your team. We recommend starting with the following steps.

- **Keep your infrastructure configuration as code (IaC) in the same Git repository.** Every manifest applied to Kubernetes is a code (YAML file). This includes application metadata and even the infrastructure components like prometheus stack, velero, nginx ingress, and others. As you're setting up the Git repository, CI pipeline, and GitOps, we recommend keeping IaC in the same Git repository. This will allow you to roll out a new Kubernetes infrastructure in minutes.

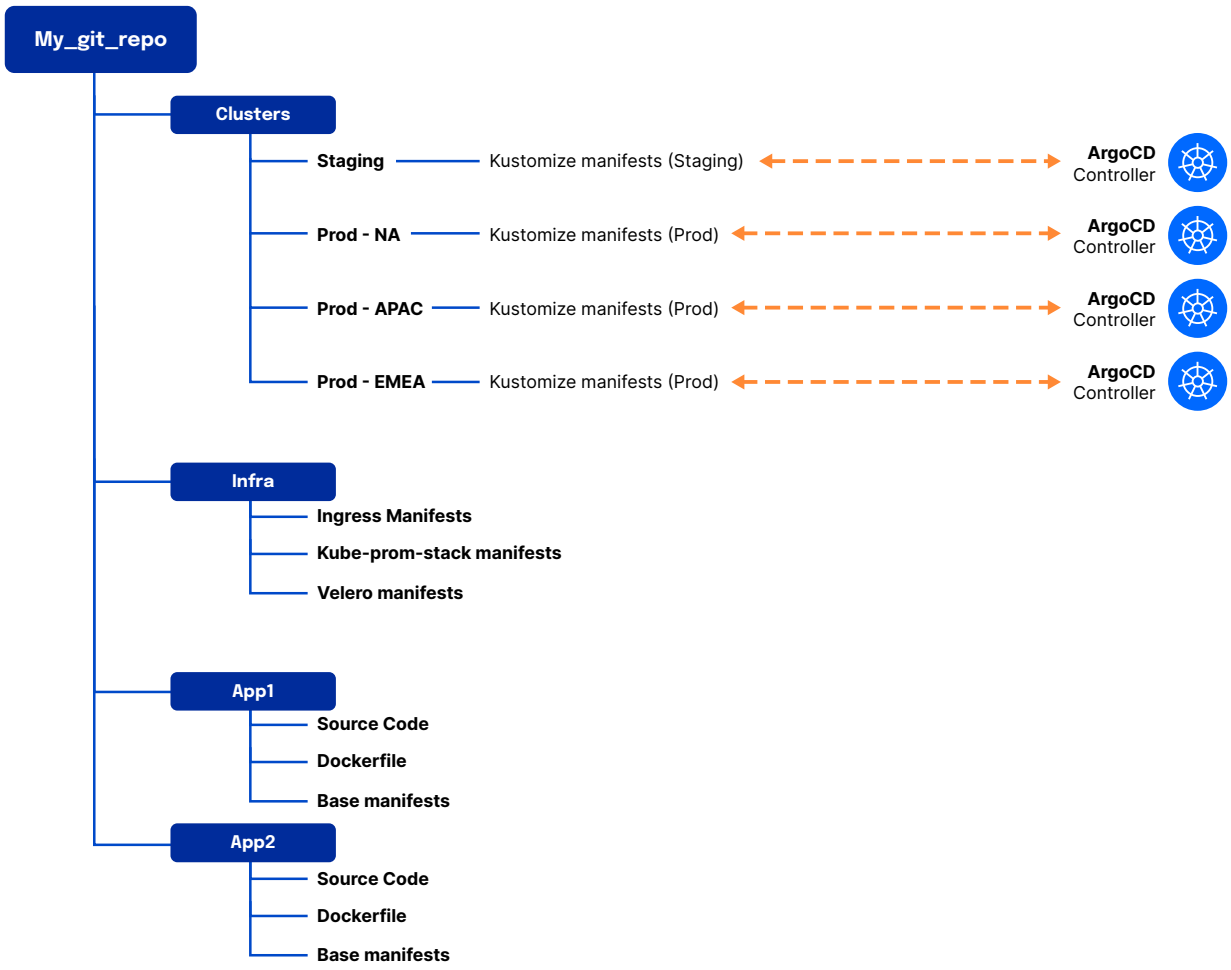


- **Use any CI (continuous integration) tool.** For this **tutorial**, we used Github and Github actions.
- **Use GitOps.** For our guide, we'll use ArgoCD. Flux is equally popular—feel free to pick whatever you are comfortable with.

- **Structure your environments.** Structure your environments and keep separate accounts for development, staging, and production. You can use the **team structure** to hold different accounts. Give employees access to different teams as needed.



- **Lock down your production cluster.** The default kubeconfig for a Kubernetes cluster has admin rights. While it makes sense for a development cluster (and sometimes staging), the production cluster should be locked down. It means no doctl access to production accounts, and you grant limited access to Kubernetes clusters using **client certificates**. If you have a small team, feel free to skip this step for now and maybe limit production account access to a few developers.
- **Plan the staging environment.** We recommend a 24×7 staging environment.
- **Choose a Git folder layout.** This is an important decision you will make. Here's a recommended practice to structure your git folder layout.



In the above diagram, kustomize manifests are the ones that get synced to the clusters. We have four clusters (staging, prod-NA, prod-APAC, prod-EMEA). Kustomize manifests refer to the base manifests in the same repo or another repo. Likewise, you may directly refer to the upstream helm chart for monitoring stack (eg. kube-prom-stack) and customize it directly. Kustomize is built into Kubernetes tooling and offers a lot of flexibility.

While the above example shows all apps and infrastructure configurations in a single repo, you may separate those into different repositories.

- **Write your CI program to trigger on PR to the application source code folder.** This CI program should run necessary verifications, build the docker image and publish it to the registry with an appropriate tag. If successful, it should update the kustomize manifest of the staging cluster with the image tag as the newly created one. That will automatically trigger the Argo CD controller running on the staging cluster. Argo CD will deploy the new image on the staging cluster.
- **Set up manual promotion between environments.** While deployment to staging should be automatic, we do not recommend automatic deployment to production clusters. Promoting the production should be done manually by someone updating the kustomize manifests for the production clusters and creating the PR. This allows you to control when to ship a feature to production.

Production

Create a robust, production-ready Kubernetes cluster that's future-proof and ready to serve end-users.

What you need to get started: A ready staging environment, CI/CD tooling in place, and a promotion strategy.

Getting a Kubernetes cluster production-ready means more than just serving traffic. Commercial-ready clusters should be future-proof—prepared for scale, optimization, security, and disaster recovery. The below section focuses on features available in DigitalOcean Kubernetes, but many of these steps will also be applicable to other managed Kubernetes providers or to a self-hosted Kubernetes. In order to run microservices on a Kubernetes stack in production, you must:

- Choose the right node types
- Enable High Availability for the Control Plane
- Ensure cluster scalability
- Set up Ingress
- Set up observability
- Have a disaster recovery plan
- Practice security Hygiene

Choose the right node types

Different applications need different compute/memory footprint. Depending on your application type, it is important to start with the **right node type** and size for your production clusters. These decisions will ultimately guide you in **choosing the right Kubernetes plan**.

Enable High Availability

If using a Managed Kubernetes instance, explore if they have high availability. The Control Plane (master node) is the brain of the Kubernetes cluster. The components, including the API Server, Controller Manager, Scheduler, and more, in the control plane control and manage the worker nodes by efficiently scheduling resources and distributing workloads. The DigitalOcean Kubernetes control plane configuration is non-High Availability by default. For business-critical production usage, **enable High Availability** for the control plane.

Ensure cluster scalability

The production environment of Kubernetes should be built with scalability in mind so that teams don't need to worry about application load and cluster performance with increased demand. The following auto-scalers are available in DigitalOcean Kubernetes:

- **Cluster Autoscaler** → Deploy more nodes to the existing Kubernetes cluster.
- **Horizontal Pod Autoscaler (HPA)** → Deploy more pods if the demand grows. We recommend using HPA along with cluster autoscaler at this stage.
- **Vertical Pod Autoscaler** → Deploy additional resources (CPU or Memory) to existing pods. We recommend this in the future - when you get to the scale/optimization stage.

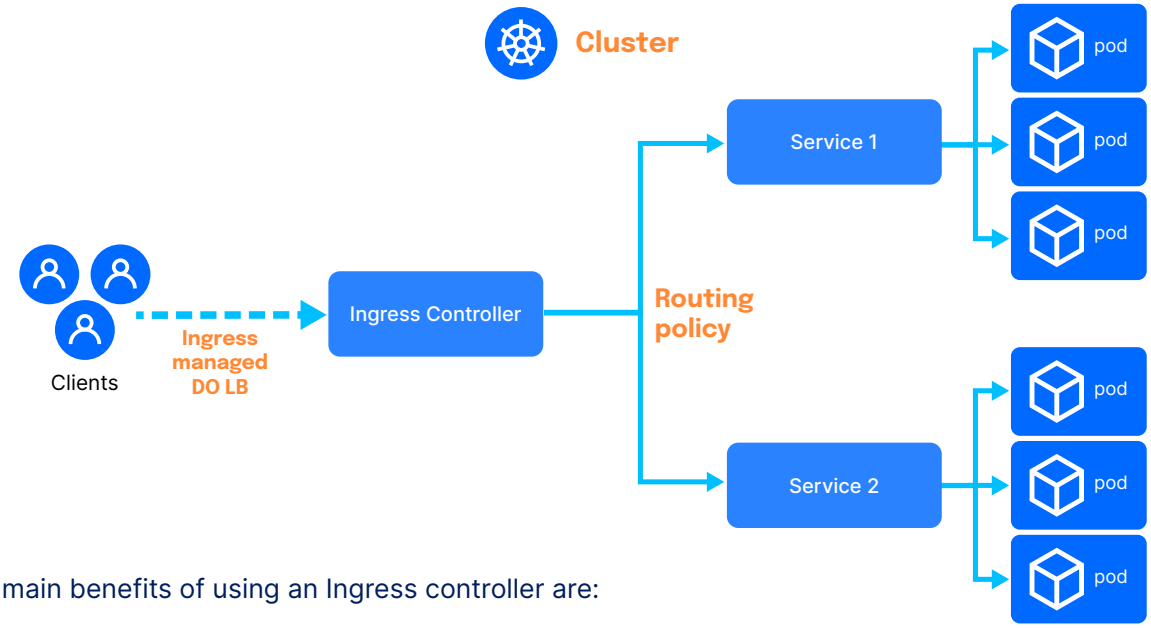
DigitalOcean offers **Horizontal Pod Autoscaler** (HPA) in conjunction with **Cluster auto-scaler** (CA) to autoscale your pods and worker nodes, respectively. When HPA decides to increase the pod limit to maintain resource utilization thresholds, CA will add more worker nodes to the cluster to maintain the cluster size needed for those pods.

Check out the **Scale** section to learn more about when to use auto-scaling.

Set up ingress

While building web applications on microservices is a popular solution, exposing and managing external access to multiple software components can be hard to manage. Traditionally, teams employ load balancers provided by the cloud provider to fulfill these needs, but load balancers can be rigid in nature and get more expensive as the application grows. This is where an ingress comes in. The main difference between ingress and load balancers is that load balancers can route only to a single service staying external to the cluster while Ingresses are native objects inside the cluster that can route to multiple services.

An ingress Kubernetes object along with its associated controller fundamentally lets you to route the web traffic traffic from outside the Kubernetes cluster to one or more services inside the cluster acting as a single point of entry for all the incoming traffic. This makes it the best option for the production environments. It can also handle much more than just load balancing web applications, and can scale up easily.



The main benefits of using an Ingress controller are:

- HTTP/HTTPS traffic load balancing.
- TLS encryption for web applications.
- Support for virtual host and path based routing for backend services.
- Reduces complexity and costs by eliminating the need to create multiple external load balancers.
- Saves network bandwidth by using compression. HTTP is a text based protocol, and text has a very high compression ratio.
- Offloads backend applications when it comes to encryption, compression, etc.

There are many ingress implementations available for Kubernetes, including NGINX ingress controller, Traefik Kubernetes Ingress provider, Ambassador API Gateway, Contour, an Envoy based ingress controller, Istio ingress, and HAProxy ingress.

While we see growing adoption of all of the above, we recommend NGINX ingress for users who do not have an existing preference. NGINX is well established and has a massive user base.

Getting started with NGINX ingress controller

The primary reason developers set up an ingress is to receive external traffic to the services running on Kubernetes clusters. Ideally, a DNS would be mapped to the ingress controller load balancer IP. This can be done via a DNS provider with the domain name you own. The load balancer receives traffic on HTTP/HTTPS ports 80/443 and forwards it to the Ingress Controller Pod. The Ingress Controller will then route the traffic to the appropriate backend Service.

Here's one way to set up the NGINX ingress controller on DigitalOcean Kubernetes. You may need to modify the installation process based on your unique requirements. For a hands-on walkthrough of the below steps **check out this tutorial**.

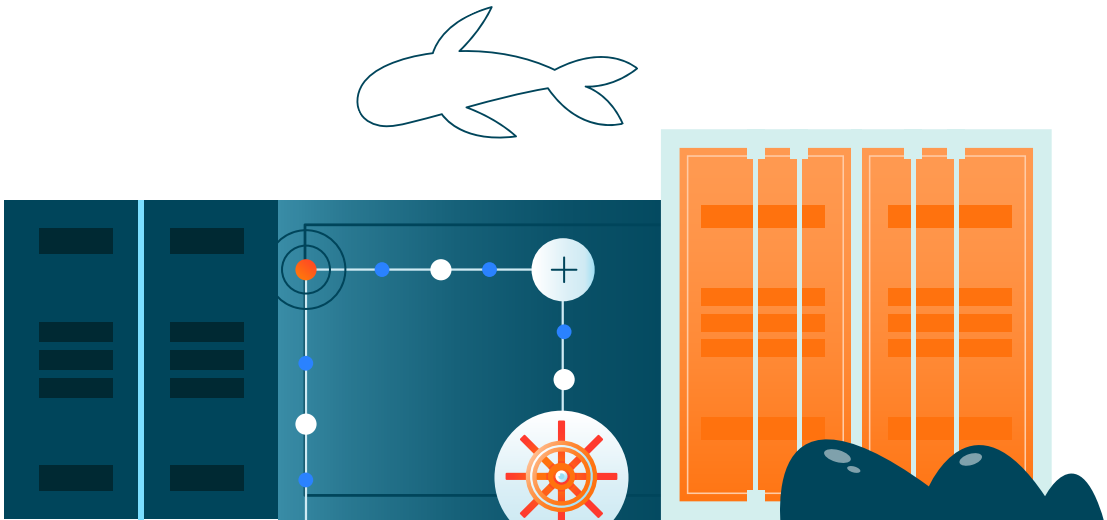
1 Install Nginx ingress controller

There are 2 variations (community and vendor maintained) of NGINX ingress controllers. We recommend using Kubernetes maintained **NGINX Ingress Controller** because it's open source and has wider community support. The NGINX Ingress Controller is an NGINX web server running as a replica set in your Kubernetes cluster. It continuously monitors the control plane for new and updated ingress resource objects.

The ingress controller will create a **LoadBalancer Service** that provisions a **DigitalOcean Load Balancer** to which all external traffic will be directed. For higher capacity, teams can configure the load balancer node counts using annotations. There are a number of customizations available. Always customize the load balancer from Kubernetes NGINX service definition and never directly from places like cloud console or cli. Sometimes, you may need to use the existing load balancer in your cloud account. This is a fairly **straightforward process**.

2 Set up Domain names

Now that the ingress pod is running, set up a domain name for the application you're hosting in the Kubernetes cluster. Developers can purchase domain names rom registrars like **GoDaddy** or **Google Domains** and then manage the DNS records on the DigitalOcean platform. You can choose to purchase a single domain name (e.g. www. sample.com) or a wildcard domain name(e.g *.k8shark.com) depending on your business needs.



3 Set up Domain Name Mapping

Developers can choose to map a single domain name (www.sample.com) to the A record of the Load balancer or map a wildcard domain (*.sample.com) to the **A record of the load balancer**. Through Single DNS Mapping, developers can map only one domain to the ingress controller and use multiple path-based routing to route to the backend services. For example:

- http://www.k8shark.com → <Load Balancer External IP>
- http://www.k8shark.com/app1

If opting for wild card DNS Mapping, you can have multiple **dynamic subdomains** through the single ingress controller and each sub-domain can have its own **path-based routing**.
For example:

- *.k8shark.com → <Load Balancer External IP>
- *.app.k8shark.com → <Load Balancer External IP>
- http://echo.k8shark.com/api/v1 → subdomain with path-based routing

The main advantage of using wildcard DNS is that you can have **dynamic DNS endpoints** through ingress. Once you add the wildcard entry in the DNS records, you need to mention the required DNS in the ingress object and the ingress controller will take care of routing it to the required service endpoint.

4 Choose single or multiple load balancers

A single DigitalOcean load balancer can point to any one DigitalOcean Kubernetes cluster. You can't have a single load balancer across multiple clusters. Developers can have multiple load balancers in a single cluster. This is needed when segregating the networks for completely different applications, like app A, and app B, or even for different use cases like staging vs. production.

Some use cases require multiple replicas of applications running across different clusters. For these use cases, use DNS load balancing. This creates multiple A records for the same domain name that point to different load balancers to distribute the application load, avoiding cluster downtimes. DigitalOcean allows developers to **set up DNS load balancing**. By doing this, developers can expose multiple endpoints to the outside world to serve their applications. This set up protects through cluster failovers and increases overall throughput, giving you the ability to scale for high availability and high performance.

5 Setup backend services

Create the backend services where you'll route the external traffic using the ingress. Define the appropriate service manifest which routes the traffic to the pods and the appropriate deployment manifest that keeps the group of identical pods running with a common configuration. Validate if the service is UP by confirming it has the cluster IP which is the internal IP on which the service is exposed.

6 Terminate TLS on DigitalOcean Load Balancer or NGINX Ingress

If you have just one service running on DigitalOcean Kubernetes and you need to expose the service outside the cluster, there's no need to use NGINX. Developers can directly expose the service via a DigitalOcean Load Balancer. In these situations, terminate the TLS on DigitalOcean Load Balancer. You can scale this model (1:1 mapping between LB and DOKS service) for a few services. After that, cost will play a role.

If you have multiple backend services that need to be exposed to outside, you definitely need an Ingress. For example, if you have x.app.com and y.app.com subdomains pointing to different services, then you need "app.com" to point to DigitalOcean Load Balancer, which then directs the connection to NGINX ingress. We recommend terminating TLS on NGINX whenever you use one. This is done using the "TLS passthrough" option on DigitalOcean Load Balancer.

Finally, note that DigitalOcean load balancers do not support path-based routing, so if you have multiple services that share a subdomain—like app.com/x and app.com/y are handled by different Kubernetes services—you must use an ingress.

7 Preserve client source IP

Once the Load balancer is up, you can notice that the access logs capture the IP address of your load balancer instead of the IP address of clients. When you terminate TLS on the Ingress, you can enable proxy protocol in both the DigitalOcean load balancer and Ingress-Nginx service to get visibility to the client IPs through annotations. This requires the receiving application or ingress provider to be able to parse the PROXY protocol header.

When you terminate TLS on the DO LB, use the X-Forwarded-For HTTP header. DigitalOcean load balancers automatically add this header. This option works only when the entry and target protocols are HTTP or HTTP/2.

8 Setup TLS certificate

Setting up your application accessible over HTTPS is an essential step for Kubernetes clusters serving in production as it ensures that traffic from the internet into your cluster is encrypted. For that, you need to enable Ingress-NGINX to use production-ready TLS certificates. Use **cert-manager**, a powerful and extensible X.509 certificate controller for Kubernetes which obtains certificates issued by Issuers like **Let's Encrypt**, **HashiCorp Vault**, and **Venafi**. This tool ensures the certificates are valid and up-to-date, and auto-renews certificates before expiration.

If you're using sub-domains, use **wildcard certificates** from cert-manager. Keep in mind that deleting/re-creating NGINX resources triggers new certificate negotiation. Since Letsencrypt has rate limits, if not monitored, you may get blocked on the application development. When you hit the rate limit, there's no way to reset it. You will need to wait until the rate limit expires after a week. They have a rate-limiting form that can be used to request a higher rate limit, but it is no faster than its natural resetting time. You can opt for paid issuers to save yourself from these rate limits.

9 Optimize for performance

Now that the production setup is ready, you can look for the below tuning parameters to optimize for the maximum performance

- **Keep-alive-requests:** Keepalive connections on the client side
- **Upstream-keepalive-request:** Keepalive connections on the upstream backend services
- **Worker-processes:** Number of Nginx worker processes, each handling a large number of simultaneous connections.
- **Max-worker-connections:** Maximum number of connections each worker process can handle simultaneously
- **Enable Compression:** Enabling compression improves the performance of slow connection clients, and reduces network transfers thus saving bandwidth
- **Enable Caching:** Enabling caching reduces disk I/O and load on backend servers

For high concurrency environments, set the following values:

- Keep-alive-requests: 10,000
- Upstream-keepalive-requests: 1000
- Worker-processes: defaults to auto
- Max-worker-connections: defaults to 512; 65535 on high concurrency

Use **wrk2**, a modern HTTP benchmarking tool to measure the NGINX performance that can produce a constant throughput load and accurate latency details to the high 9s.

Set up observability

When running a production Kubernetes cluster, it's important to have complete visibility into the health of the Kubernetes cluster and application workloads. Observability helps teams understand what's happening across environments and technologies so they can detect and resolve issues and keep systems efficient and reliable. Without effective observability, any downtime in the cluster may result in increased time to recovery, resulting in customer dissatisfaction and a cascading impact on the business.

Observability is made up of the following main areas: monitoring, logging, alerting, tracing, and chaos experiments. Start with monitoring, logging, and alerting.

Enable monitoring and alerting

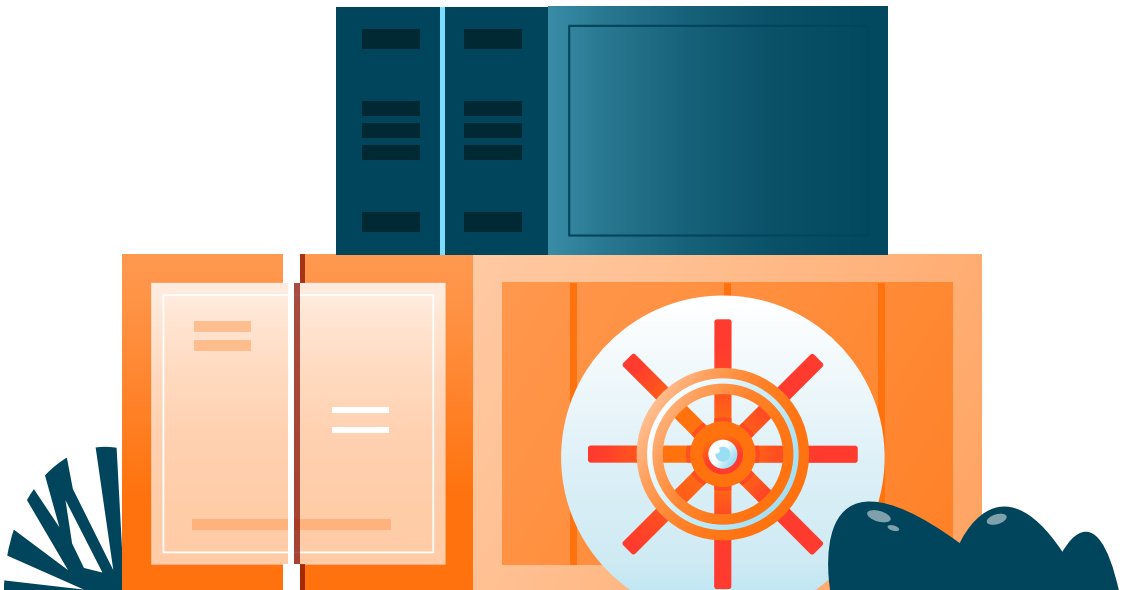
There are three aspects of monitoring you need to be aware of when running microservices in Kubernetes:

1. Cluster monitoring which includes node availability and node resource utilization.
2. Pod Monitoring which includes pod instance monitoring and container resource utilization.
3. Application and service monitoring, which includes service-to-service communication and ingress/egress traffic monitoring.

Maintaining a robust Kubernetes monitoring and alerting solution gives teams the ability to diagnose the issues that occurred and the location they occurred, optimize hardware utilization based on metrics alerting, optimize costs by scraping off underutilized resources, detect malicious ingress and egress traffic, and predict imminent problems.

There are several popular open source Application Performance Monitoring (APM) tools as well as paid enterprise solutions that could fit your needs, including **Datadog**, **Newrelic**, **Splunk**, **Prometheus**, and **Grafana** stack.

If you're already using a SaaS or home-grown solution for Kubernetes monitoring, please continue to use what works for you. If you're not currently using a robust solution, DigitalOcean strongly recommends **Kubernetes Monitoring Stack**, the most popular **Prometheus**, **Grafana**, and **Alertmanager** stack for monitoring Kubernetes clusters, node/pod resource utilizations, and application/service deployments.



The stack is deployed with **kube-state-metrics** and **node-exporter** to expose cluster-level Kubernetes object metrics as well as machine-level metrics like CPU, disk I/O, and memory usage. The kube-prometheus-stack includes six main components:

- **Prometheus Operator** spins up and manages Prometheus instances in your DOKS cluster, managing the lifecycle of Prometheus & Alert Manager.
- **Prometheus** extracts metrics from the exporters and stores them in a time series database.
- **TimeScaleDB** stores metrics for the long term.
- **Node Exporter** exports metrics from the Nodes.
- **Grafana** visualizing layer to visualize metrics in dashboards.
- **Alertmanager** configures alerts and notifications via third-party integrations like Slack/Email/PagerDuty.

Tips for success:

- The Cluster size and applications determine the size of your observability stack.
- Storing the time series data is a huge cost overhead. If you don't plan to look at the logs/metrics in the long term, don't plan to store the data.
- Deploy log collectors as Daemon Sets so that at least one instance of the pod runs on every node in the Kubernetes Cluster.
- Use Service discovery to efficiently capture the container metrics as applications are scheduled dynamically in Kubernetes.
- For multiple clusters, you may like to centralize the observability stack in a single cluster.
- Dedicated node pool for monitoring stack is a must to provide resource isolation and reduce blast radius.
- A decent practice is to bypass the logs for Kubernetes components in a managed cluster as it is taken care of by the managed service provider.
- You can recycle the older logs, saving your storage based on your preferred budget. DigitalOcean's Spaces provides the configuration to do this.

Please follow this **tutorial** to incorporate the monitoring stack to your Kubernetes cluster.

Enable logging

For a production Kubernetes Cluster, creating a continuing record of cluster and application events through Logging and Eventing is essential to debug, trace back and troubleshoot any pod/container performance issue, audit for compliance needs, and identify event patterns for performance/scale forecasting. The Kubernetes environment has grown significantly, and the sheer volume of data generated by the components involved is tremendous, warranting a strong need for centralized Logging and eventing that brings significant value to customers and to your business.

There are four aspects of logging in Kubernetes

1. Cluster logging
2. Node logging
3. Container logging
4. Application / Service logging

Popular open source solutions for logging include **Graylog**, ELK Stack (ElasticSearch, Logstash, and Kibana), EFK Stack (ElasticSearch, Fluentd, and Kibana), **Grafana Loki**, **Kubewatch**, and **Zebrium**.

If you're already using a SaaS or home-grown solution for Kubernetes logging, please continue to the solution that works for you, if you're not currently using a working solution, DigitalOcean recommends **Grafana-Loki**-based logging stack, a multi-tenant log aggregation system. The stack has three components:

- **Promtail**: The agent responsible to collect worker node and application logs to send to Loki
- **Loki**: The main server that stores logs and processes queries
- **Grafana**: The visualization platform to query and display the logs

This stack uses DigitalOcean block storage as Persistent Volumes starting from 5 GB for Loki Time Series database.

Tips for success:

- When aggregating logs at scale, use a logging agent that collects logs at node level instead of running as a sidecar container as the latter causes resource drain.
- Control access to logs using RBAC. DigitalOcean provides step-by-step instructions to configure **RBAC** to secure your Kubernetes cluster.
- Set resource limits on log collector daemons so that you can optimize the log collection based on resource availability.
- Follow a consistent format of logs for easy processing by log aggregators and reduced latency during log analysis.

Follow this **tutorial** to incorporate the logging stack into your Kubernetes cluster.

Enable event logging

In a production environment that operates at scale, you will observe several node/pod resource reallocations, rescheduling, and state transitions triggering Kubernetes events. These **events** are entirely different from the **logs** generated by the Kubernetes components of the control plane. Kubernetes events are not regular objects like logs. They are transient and persist for less than one hour. There is no built-in mechanism in Kubernetes to store events for a long period of time.

The key benefits of event logging are that it's

- Easy to Setup alerting by routing specific events to tools like slack and pagerDuty
- Easy to Log all the events in Elastic Search or Syslog or Splunk
- Easier to conduct post-mortem analysis after any event

DigitalOcean strongly recommends **Kubernetes Events Explorer**, an open-source event logging tool that exports all these events for observability of alerting purposes.

Set up a Disaster Recovery plan

Teams should be able to recover application(s) or the entire cluster in the event of an unexpected occurrence like a data center outage, an application crashing and not recoverable, human error resulting in loss of data or configuration, or migrating a cluster to the latest version of Kubernetes.

Teams using GitOps (explained in CI/CD) are already in a position to bootstrap a new cluster and get the applications running with just a few commands. For disaster recovery, consider the following challenges:

You have a working setup with a DO load balancer and external DNS names. When recovering the cluster, it's necessary to preserve the load balancer and associated DNS records. Otherwise, it will take days for the DNS information to be synchronized among your users, impacting the end-user experience.

You have applications running on your cluster, which may be using persistent volumes (PV). When an application crashes and needs restored, teams need a last working volume copy of that application. Otherwise, the newly created application will start in an empty state. There are multiple tools including velero/trilio, volumesnapshot, and backup/restore offered by the application itself. It's important to go with the tested and recommended path for your specific application.

You're using many certificates for various domains and an issuer like letsencrypt. Depending on your usage, you may run into certificate rate limit issues when you are trying to restore too many applications. Under the hood, the restore process will trigger new certificate creation. Solving this (to backup and restore the certs/keys) is a bit tricky, and you must test it beforehand.

You're encrypting the secrets outside the cluster. In a new cluster, you will need to recover the secrets. Otherwise, GitOps will work fine, but you won't have the secrets you need in the new cluster.

The following sequence is recommended for restoring any application or the entire cluster.

If you are restoring an application in an existing cluster:

1. Delete the application.
   Exception- If the service type is a load balancer, then make sure to detach the **load balancer** first! Otherwise, you will lose the external IP and will need to recreate DNS mapping.
2. Delete the associated PV.
3. Create the new cluster, if needed.
4. Restore the PVCs from the volume snapshot.
5. If it is a new cluster:
   - Restore Argo CD.
   - Restore the secrets.
6. Bring up the applications via Argo CD.

Give preference to the recommended guidelines for your specific application (eg. elasticsearch, kafka, etc.), which may be different from the above.

Disaster recovery is hard and needs practice. We recommend testing your procedure from time to time by deleting apps or the entire cluster.

Refer to this **section** of the hands-on tutorial for an example of disaster recovery.

Security Hygiene

Security for Kubernetes is a fairly broad topic, and security practices for a large enterprise can be substantially different from security best practices for a small organization. In a large organization, there are dedicated security teams. In an SMB or startup the developers often do everything.

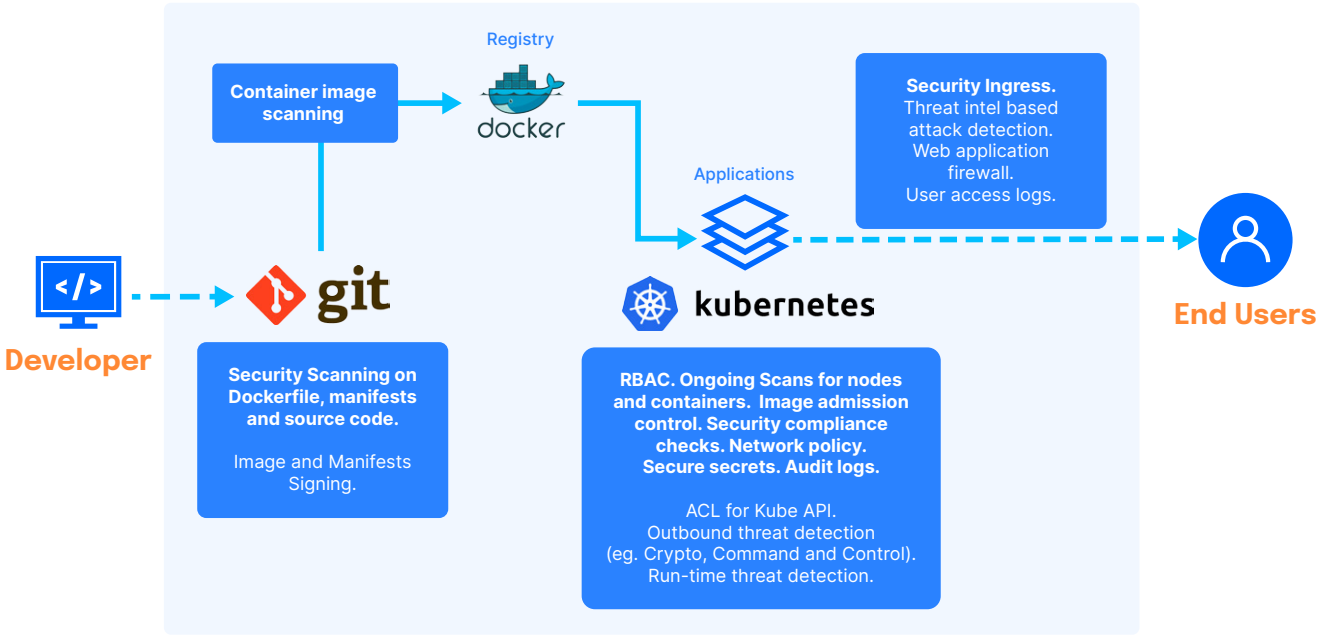
The big picture

Good security practices should be an integral part of the entire developer lifecycle. Four questions to consider as you implement security practices are

1. Are the components in your kubernetes clusters up-to-date, and hardened based in industry best practices (eg. NSA/CISA Kubernetes hardening, CIS Kubernetes and Docker controls)?
2. Do you know who is accessing your cluster? Have you enforced the right role based access control (RBAC) for different users/entities?
3. How are you ensuring that safe images with reasonably safe configurations are deployed to your cluster?
4. How are you protecting against run-time attacks like web application attacks, malicious container deployment, and more.

Keeping Kubernetes clusters up-to-date should be covered by the managed Kubernetes provider. Knowing who is accessing your cluster and ensuring safe images are deployed to your cluster provide maximum ROI for your team and are comparatively easier to automate. Protecting against run-time attacks is hard, especially for ongoing maintenance.

The following diagram shows a big-picture view of kubernetes security.

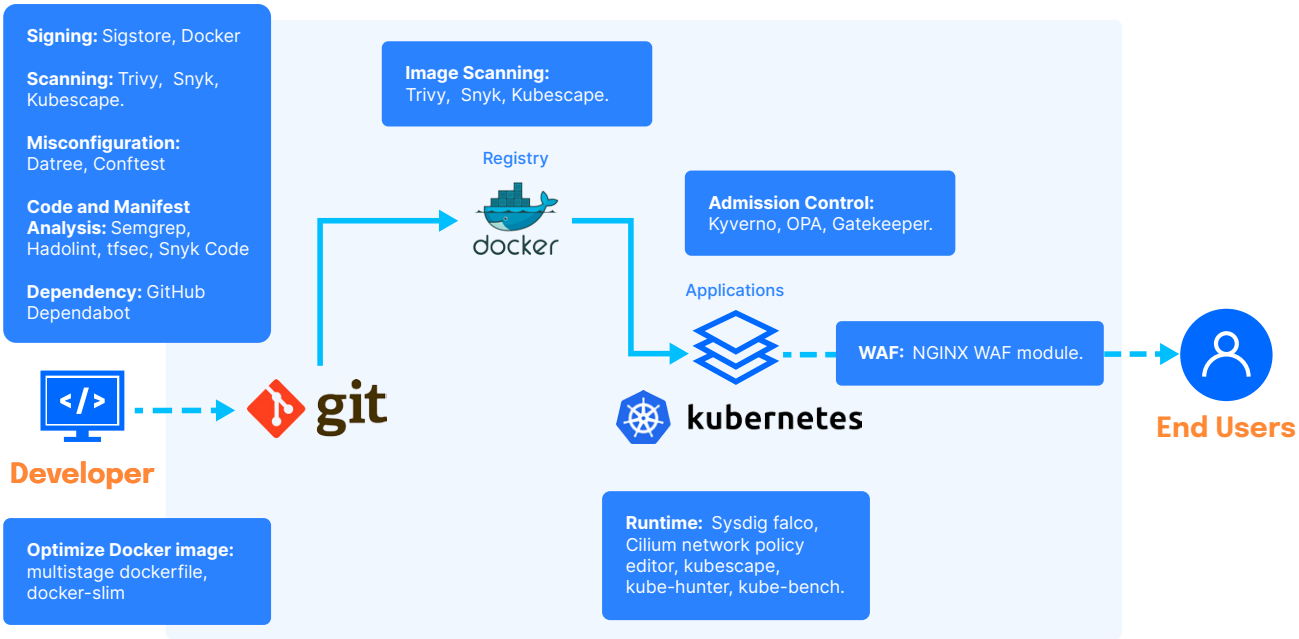


The Kubernetes and containers developer workflow is ideal for bringing security into the development process. As part of the development workflow, developers should build container images, set up configuration files and keep those in registry and git repositories. Given that they have to automate this process, it makes sense to incorporate necessary security controls along the way.

Developers should use an IDE like VScode for development. Using a plugin for code/library/configuration scanning for vulnerabilities can help in fixing the issues at the earliest. Once they commit the code to the git repository the CI pipeline is triggered. At this stage, it's recommended to incorporate the necessary security checks. This includes vulnerability scanning, configuration checks, and more.

Next, the the image is built and stored in the registry and continuous scanning is necessary because new vulnerabilities are detected over time. Now, the application is run on the cluster. Assuming you've done the necessary hardening of the cluster, some tasks like image admission control, network policies, and others, can be performed with less effort. This becomes important if you have a larger team and less visibility. Once end users have access to the application, it's important to secure the ingress with appropriate configuration.

For security tools, the challenge is not creation, but ongoing update of the tool. There are many tools available and in addition to considering the ongoing updates and maintenance of the tool, consider the target persona of the tool and use products builtw ith developers in mind.



Most of these tools have an open-source and free-tier component. It is important to adopt a few tools and implement them well.

For **supply-chain security** consider the following recommendations:

- Try **Snyk or Kubescape**, both have an open source CLI tool that can be integrated into the CI pipeline.
- Reduce Docker image file size, which improves security, performance, and productivity while reducing cost.
- Remember to follow the recommendations from your source-code repositories and invest in one or more good tools to identify vulnerabilities and insecure configurations upfront and resolve those prior to deployment.
- Lock down the deployment to accept infrastructure-as-code (IaC) using GitOps.
- Use a managed build pipeline, eg. GitHub, GitLab, CircleCI etc.
- Implement periodic scanning as additional vulnerability information is learned every week.
- If you have a reasonable team size with a continuously used supply chain, lock down authorizations, sign images and artifacts, and verify those in deployment.

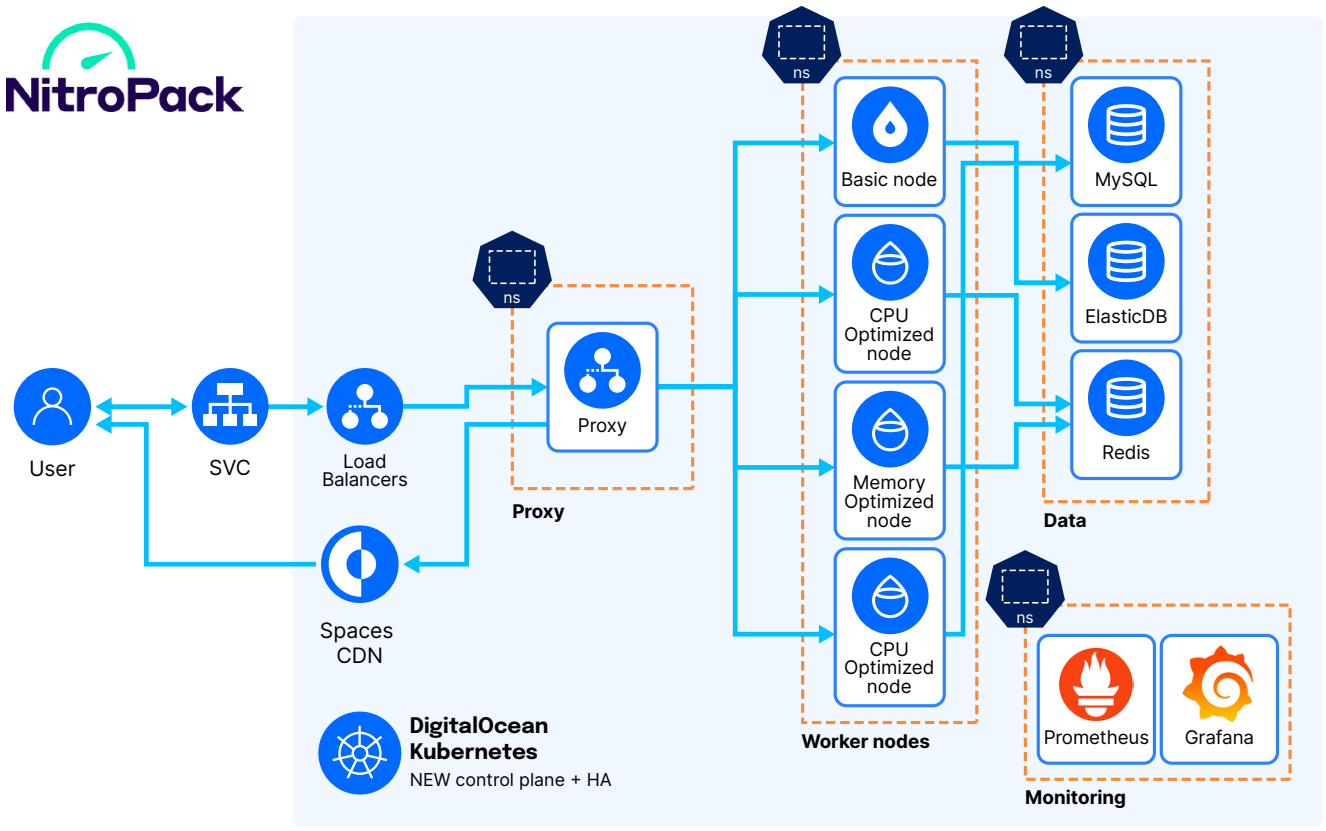
For **run-time security** consider the following recommendations:

- Secure the ingress configuration.
- Secure access to the cluster.
- Implement ongoing scanning.
- Get timely notifications about security issues found in your cluster or Docker registry by enabling email and/or Slack notifications (both Snyk and Kubescape support this feature). This extra step also helps to identify possible issues over time, and take the necessary steps to remediate the situation.

NitroPack

NitroPack's product scans a webpage and applies techniques to solve performance problems using its own programs and logic. The application needs to be able to handle hundreds of requests per second and needs to be able to scale up and down to meet demand seamlessly. Originally, the NitroPack team was managing servers by hand. As a small startup with a lean team, they wanted a solution that would allow them to spend more time developing their product and less time managing infrastructure while still supporting their need to scale. **DigitalOcean Kubernetes** provides the automated infrastructure they need.

NitroPack set up one cluster using a local environment for development featuring over 150 nodes. They use a microservices architecture that automates the delivery of over 120,000 sites with over five million pages. They enabled High Availabilty (HA) for their DigitalOcean Kubernetes clusters to ensure peak performance and reliability. Using the High Availability feature Nitropack doesn't worry about losing the control plane under stress like their rapid growth.



Scale

Utilize sophisticated tools to optimize Kubernetes deployments for continued reliability and scale.

What you need to get started: You'll need an application running in production and experiencing growth.

As an application in production begins to experience growth, developers need to scale the existing cluster or multiple clusters/regions, enforce appropriate policies, and optimize for resource consumption. Optimizing Kubernetes and applications that run in Kubernetes is similar to scaling applications outside of Kubernetes. Optimization starts by developing a thorough understanding of the performance characteristics of an application and the environment in which the processes are run. Measure memory, CPU, and i/o. Focus measurements and alerts on the **four golden signals**: latency, traffic, errors, and saturation.

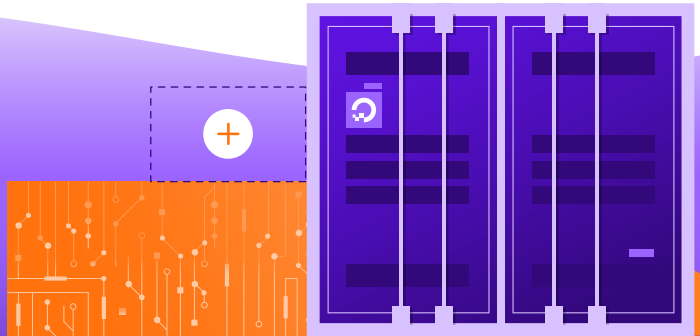
Access Control

If you're using DigitalOcean Managed Kubernetes, all users in the DigitalOcean team have administrative access to the clusters provisioned in the team. Provision clusters in a team account for the Kubernetes administrators and **use certificates for regular Kubernetes users** and **RBAC** for users that need to operate applications running in the cluster. Be sure that only the users that should have administrative access are in the team that hosts the Kubernetes cluster.

Measure, observe, and practice

Build out robust observability tools and set up actionable alerts. Alerts should be accompanied by runbooks that provide instructions to remediate problems that require manual intervention. Over time, automate the runbooks as much as possible to facilitate greater scaling by asking how you would handle 10x load. What processes, tools, and changes will your system need to handle 10x load?

Develop and refine the runbooks for your applications and infrastructure by holding disaster recovery drills periodically.



Limits

Optimizing Kubernetes deployments for scaling requires considering failure modes, the account, resilience, network, and CPU load, storage needs, dynamic runtime conditions, and deployment best practices.

If you're using DigitalOcean Kubernetes, all DigitalOcean accounts have some level of limits; newer accounts have tighter limits than established accounts. Kubernetes users should consider Droplet, load balancer, firewall, firewall rule, and Volume limits.

Teams can review their current Droplet limits on the respective team page and can request the limits to be increased there as well. Limits on other resources are not exposed but can be adjusted by support. As Kubernetes customers enable autoscaling, they may need to have their account limits adjusted to allow their cluster to scale to their maximum expected size. When considering account limits, consider:

- Given the Droplet account limits and autoscaling configuration, will your nodes be able to scale out enough to handle the expected maximum load? Consider the max_nodes setting of each node pool and the account's Droplet limit.
- There is a total volumes limit and a hard limit of four volumes per Droplet: can your applications scale to their maximum respective limits given the total volumes limit and the hard volume limit per Droplet? Pay careful attention to the affinity and anti-affinity rules in statefulsets that have persistent volume claim templates.
- Load balancer limits may need to be adjusted to allow a Kubernetes service of type LoadBalancer to be created.
- Two firewalls per Kubernetes cluster are provisioned for all clusters. Services of type NodePort **add firewall rules by default**.

Probes

Add useful **readiness probes** to your pods. Readiness probes should signal ready only when the pod is ready to accept traffic. When a container is not yet ready for traffic (e.g., not fully initialized) or should not receive additional traffic (e.g., traffic is blocked by a synchronous process that needs to complete before handling more requests), it is desirable to direct traffic elsewhere. In these cases, the readiness check should return a failure, causing traffic to be directed elsewhere.

Most applications do not need liveness probes. Liveness probes cause a container to be restarted without causing the pod to be rescheduled/replaced). In general, liveness probes should be used sparingly. A typical use case for liveness probes is for a temporary mutation of a detectable condition that can be handled via a restart (e.g. memory corruption or a memory leak) until the root cause of the problem can be found.

Graceful Shutdown

Cluster upgrades is the most common reason for node shutdown. DigitalOcean Kubernetes performs a surge upgrade by default by creating up to ten worker nodes in a pool at a time and recycling existing nodes. This can mean disruption to the workloads. Make sure to implement graceful shutdown for your pods. For stateful workloads that need time to drain from a node, DigitalOcean Kubernetes waits for up to 30 minutes for a given node.

Affinity

Add **pod anti-affinity preferences** to pod templates to cause pod replicas to be spread across nodes for resilience to node failures.

Resource requests and limits

Measure application needs and set resource requests and limits deliberately.

Set accurate CPU requests and never set CPU limits. CPU requests guarantee that processes will receive at least the amount of CPU requested. By omitting CPU limits, processes will be able to use any additional CPU capacity available on the node. When CPU limits are set, applications may be CPU throttled even when the node has otherwise idle CPU capacity.

Set memory requests and limits.

refs:

- **https://twitter.com/thockin/status/1134193838841401345**
- **https://home.robusta.dev/blog/stop-using-cpu-limits/**

Node Pools

Use node pools and **node affinity rules** on pods to cause workloads to be distributed to nodes of the appropriate type given the work of the pod.

All nodes in a node pool will have two labels by default that can be used for scheduling:

- doks.digitalocean.com/node-pool=<node pool name>
- doks.digitalocean.com/node-pool-id=<node pool uuid>

Additional labels, as well as taints, can be configured via the **DigitalOcean public API**. Note that any labels or taints applied directly to a node via the Kubernetes API will not be persisted across node recycles or upgrades.

DNS Load

DNS load can be mitigated by using **NodeLocalDNSCache** daemonsets. Using NodeLocalDNSCache will require that pods have dnsConfig.nameserver set to the address to which the NodeLocalDNSCache binds. A 169.254.0.0/16 address is a good choice for the address. The NodeLocalDNSCache can be configured to use TCP for upstream requests while listening on UDP for client requests.

Network Load Testing

Execute realistic load tests to exercise the network paths to understand how your application will need to horizontally scale to handle expected and peak load.

Resilience to API Latency and Failure

The Kubernetes control plane is a distributed system with many failure modes. Anything that interacts with kubernetes via requests to kube-apiserver (e.g. operators or controllers) should employ best practices: retries, backoffs, and circuit breaking. Handling all the distributed computing concerns can be difficult at first. Start with retries and implement backoffs and circuit breaking later.

Be prepared for HTTP failure responses. Typical examples include 5xx and 4xx errors, especially when kube-apiserver requests to etcd fail or are rejected.

NFS for Shared Volumes

Use Kubernetes Persistent Volume (PV) if you want persistent storage that is independent of the life cycle of the pod. PV is an abstracted storage object that allows pods to access persistent storage on a storage device, defined via a Kubernetes StorageClass. DOKS includes do-block-storage as the default StorageClass, which points to DigitalOcean block storage.

The other type of Persistent Volume required in Kubernetes is NFS Volume, a shared file system that can be accessed by multiple pods at the same time. We recommend **OpenEBS Dynamic PV NFS provisioner** (follow **this link for helm chart install**) which can be used to dynamically provision NFS Volumes using the block storage available on the Kubernetes nodes. The NFS provisioner runs an NFS server pod for each shared storage volume. It uses DigitalOcean block storage. Below are the high-level set-up recommendations.

- Get the openebs-nfs-provisioner pod up and running.
- Create the read-write storage class. Storage classes allow users to control the type and configuration when provisioning volumes. Use separate StorageClass based on the type of provisioner and mount parameters you need.
- Use the appropriate manifest to create a PersistentVolumeClaim-based storage class and set the access mode.
- Use an appropriate deployment manifest to create the required number of shared volume replicas and mount them on all the application pods to consume.
- Finally, make sure to take snapshots of the NFS volume.

Note: The NFS Server pod is a single point of failure. It takes about 90 seconds for the NFS server to be back up if the pod goes down. It takes about 7 minutes for the volume to be fully functional if the node hosting the NFS server pod goes down. So, create your application deployment minding these downtimes.

Check out this step-by-step **tutorial** to incorporate the NFS Volume in your Kubernetes infrastructure.

Egress Gateway

While deploying containerized applications in production, it is a common developer interest to have some sort of gateway so that multiple nodes can use the same IP address for the egress traffic for whitelisting purposes. It is also a common security requirement to route all cluster traffic through a specific set of dedicated nodes. If you're using DigitalOcean Kubernetes, it does not have a home-grown egress resource to manage the outgoing traffic. This is handled in a few ways:

- Network Level Setup via a choice of CNI plugin in the cluster.
- Egress gateway via a service mesh.
- Explicit routing via an external NAT gateway.

DigitalOcean does not provide a managed NAT gateway. We recommend setting up a droplet as a NAT gateway in the VPC. Refer to this **guide** for setting up an egress gateway in two steps. This way, the egress gateway translates the source address (i.e SNATs) of the outgoing connections such that for the external services outside the cluster, the traffic appears to be coming from the single IP address of the egress gateway. Note that the NAT gateway in this case is a single point of failure for the routed egress traffic.

Please check out this **guide** to incorporate egress gateway in your infrastructure.

Droplets and Kubernetes in a VPC

Regular Droplets, managed Databases, and Kubernetes clusters can exist in the same VPC. Use the same VPC for Droplets and Kubernetes clusters that are part of the same system. Prefer to communicate over the VPC when possible (e.g., use the VPC address of a managed Database rather than the public address).

Canary

Leverage **canary testing** to validate changes to a limited set of users and/or a small percentage of requests. Canaries can take a couple of different forms. Kubernetes services can direct traffic to pods from multiple deployments. By leveraging multiple deployments whose pods match the selector in a Kubernetes service, Kubernetes users can direct connections to multiple variations of a handler and thereby control the proportion of requests that are handled by each variant.

Security

All users in the DigitalOcean team will have administrative access to the clusters provisioned in the team. Therefore, provision clusters in a team account for the Kubernetes administrators and **use certificates for regular Kubernetes users** and **RBAC** for users that need to operate applications running in the cluster. Be sure that only the users that should have administrative access are in the team that hosts the Kubernetes cluster.

DigitalOcean

About DigitalOcean

DigitalOcean simplifies cloud computing so developers and businesses can spend more time building software that changes the world. With its mission-critical infrastructure and fully managed offerings, DigitalOcean helps developers, startups, and small- and medium-sized businesses (SMBs) rapidly build, deploy, and scale applications to accelerate innovation and increase productivity and agility. DigitalOcean combines the power of simplicity, community, open source, and customer support so customers can spendless time managing their infrastructure and more time building innovative applications that drive business growth.

> To get started, **sign up for an account at DigitalOcean.com.** For more information or help migrating your infrastructure to DigitalOcean, **speak to a sales representative**.