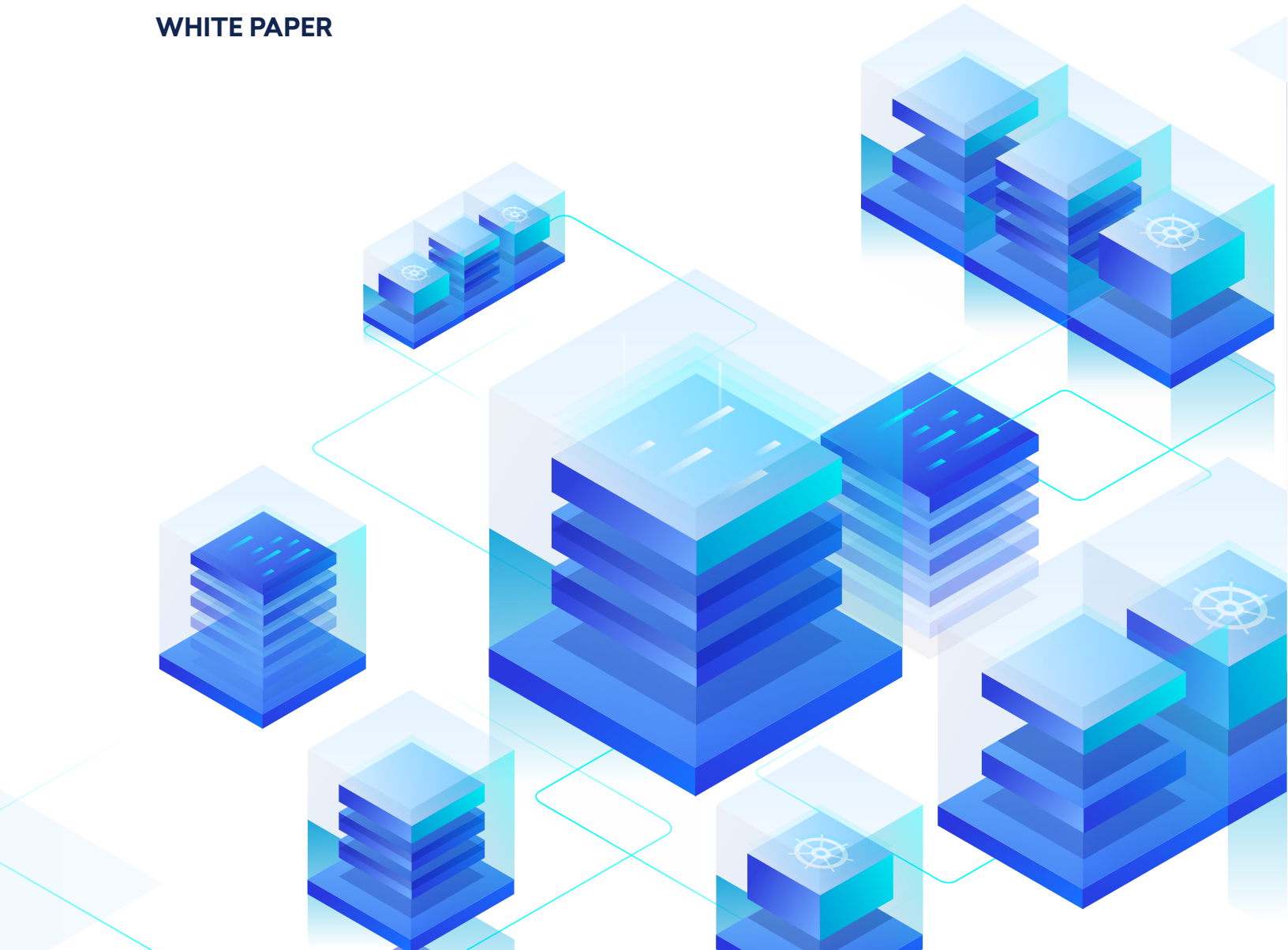
DigitalOcean

Running Cloud Native Applications on DigitalOcean Kubernetes

WHITE PAPER



Simplicity at Scale

Developer teams experiencing rapid growth and success know there are obstacles they must face when scaling to meet the demands of users. It can be especially challenging to maintain performance and reliability while engineering teams expand and users are rapidly acquired. Containerized, microservices-oriented applications that leverage the cloud's extensible functionality can support teams as they scale, mitigating the adversities that arise from accelerated growth. However, Cloud Native delivery pipelines come with their own challenges and require investments in DevOps processes and systems that can often pose significant hurdles. In offering a managed platform for running containerized applications, DigitalOcean Kubernetes empowers development teams to spend less time worrying about provisioning and operating cloud infrastructure and more time building powerful, scalable, and resilient applications.

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit **digitalocean.com** or follow **@digitalocean** on Twitter.

DigitalOcean

Executive Summary

In today's fast-moving software landscape, advances in operations technologies have fostered the dramatic reduction of application release cycles. Traditionally, software releases follow a time-based schedule, but it has become increasingly common to see applications and services continuously delivered and deployed to users throughout the day. This truncating of the traditional software release cycle has its roots both in technological developments — such as the explosive growth of cloud platforms, containers, and microservices-oriented architectures — as well as cultural developments — with tech-savvy and mobile-enabled users increasingly expecting new features, fast bug fixes, and a responsive and continuously developing product.

This symbiotic relationship between end users and developers has become increasingly linked. Shifting organizational structures and application architectures allow developers to quickly incorporate feedback and react to user demands. This accelerated development cadence often accompanies the packaging of applications into containers, and the use of systems that automate their deployment and orchestration, like Docker Swarm, Marathon, and Kubernetes. These open-source platforms, now stable enough for large-scale production deployments, allow service owners to launch and scale applications themselves, effortlessly managing hundreds of running containers.

Kubernetes and DigitalOcean Kubernetes

Kubernetes, initially open-sourced by Google in 2014, has today grown to become one of the highest velocity projects on GitHub, with over 11,300 contributing developers and 75,000 commits.[1] The growth of its thriving open-source community mirrors its popularity in the private sector, with over 50% of Fortune 100 companies relying on Kubernetes every day to rapidly deploy new features and bug fixes to users.[2]

DigitalOcean Kubernetes enables development teams both small and large to quickly take advantage of this market-leading container orchestration platform without the lead time required

[1] Cloud Native Computing Foundation. "Kubernetes Is First CNCF Project To Graduate." Cloud Native Computing Foundation Blog, Mar. 2018.

[2] RedMonk. "Cloud Native Technologies in the Fortune 100." RedMonk Charting Stacks, Sept. 2017.

to provision, install, and operate a cluster. With its simplicity and developer-friendly interfaces, DigitalOcean Kubernetes empowers developers to launch their containerized applications into a managed, production-ready cluster without having to maintain and configure the underlying infrastructure. Seamlessly integrating with the rest of the DigitalOcean suite — including Load Balancers, Firewalls, Object Storage Spaces, and Block Storage Volumes — and with built-in support for public and private image registries like Docker Hub and Quay.io, developers can now run and scale container-based workloads with ease on the DigitalOcean platform.

With full programmatic control of their cluster using the exposed Kubernetes REST API, developers can benefit from the rich ecosystem of open-source tools while still reaping the convenience of managed infrastructure. Teams can flexibly deploy and scale their Cloud Native applications. A Certified Kubernetes conformant platform, DigitalOcean Kubernetes helps developers launch their application containers and bring their Kubernetes workloads into the DigitalOcean cloud with minimal configuration and operations overhead.

Trends in Modern Application Development

As Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) offerings have matured, software development and architecture patterns have shifted to meet new infrastructure paradigms. Cloud providers abstract the underlying hardware away, and applications must correspondingly be designed to handle failure and changes in this commodity computing infrastructure. Exposing application endpoints to publish health and metric data (with the expectation that these endpoints will be regularly polled and the data will be acted upon), as well as packaging applications in smaller, self-contained disposable pieces has become the new norm in developing resilient cloud-based applications.

Designing applications that will be rapidly and continuously deployed into cloud environments has led to the development of new software methodologies like "Cloud Native" and "Twelve Factor." These high-level frameworks address common challenges in running scalable applications on cloud platforms and serve to guide software developers and architects in the design of resilient and highly observable applications. Such frameworks build on recent developments in software engineering like containers, microservices-oriented architectures, continuous integration and deployment, and automated orchestration.

Twelve Factor

I. **Codebase**
II. **Dependencies**
III. **Config**
IV. **Backing services**
V. **Build, release, run**
VI. **Processes**
VII. **Port binding**
VIII. **Concurrency**
IX. **Disposability**
X. **Dev/prod parity**
XI. **Logs**
XII. **Admin processes**

Synthesizing extensive experience developing and deploying apps onto their cloud PaaS, Heroku constructed a framework for building modern applications consisting of 12 development guidelines, conceived to increase developers' productivity and improve the maintainability of applications.

As PaaS providers abstract away all layers beneath the application, it is important to adapt the packaging, monitoring, and scaling of apps to this new level of abstraction. The Twelve Factors allow a move towards declarative, self-contained, and disposable services. When effectively leveraged, they form a unified methodology for building and maintaining apps that are both scalable and easily deployable, fully utilizing managed cloud infrastructure.

Cloud Native

As cloud platforms, infrastructure, and tooling have evolved and matured since the original Twelve Factors were published, and large-scale cloud migrations and deployments informed the software development community, an extended and broader methodology called *Cloud Native* has emerged. At a high level, Cloud Native apps are containerized, segmented into microservices, and are designed to be dynamically deployed and efficiently run by orchestration systems like Kubernetes.

What makes an application Cloud Native?

To effectively deploy, run, and manage Cloud Native apps, the application must implement several Cloud Native best practices. For example, a Cloud Native app should:

- Expose a health check endpoint so that container orchestration systems can probe application state and react accordingly
- Continuously publish logging and telemetry data, to be stored and analyzed by systems like Elasticsearch and Prometheus for logs and metrics, respectively
- Degrade gracefully and cleanly handle failure so that orchestrators can recover by restarting or replacing it with a fresh copy
- Not require human intervention to start and run

To drive the adoption of Cloud Native best practices and support and promote the growing **Cloud Native Landscape**, the Cloud Native Computing Foundation (CNCF) was created under the umbrella of the Linux Foundation to foster the growth and development of high-quality projects like Kubernetes. Examples of other CNCF projects include Prometheus, a monitoring system and time-series database often rolled out alongside Kubernetes; and FluentD, a data and log collector often used to implement distributed logging in large clusters.

In its current charter, the CNCF defines three core properties that underpin Cloud Native applications:

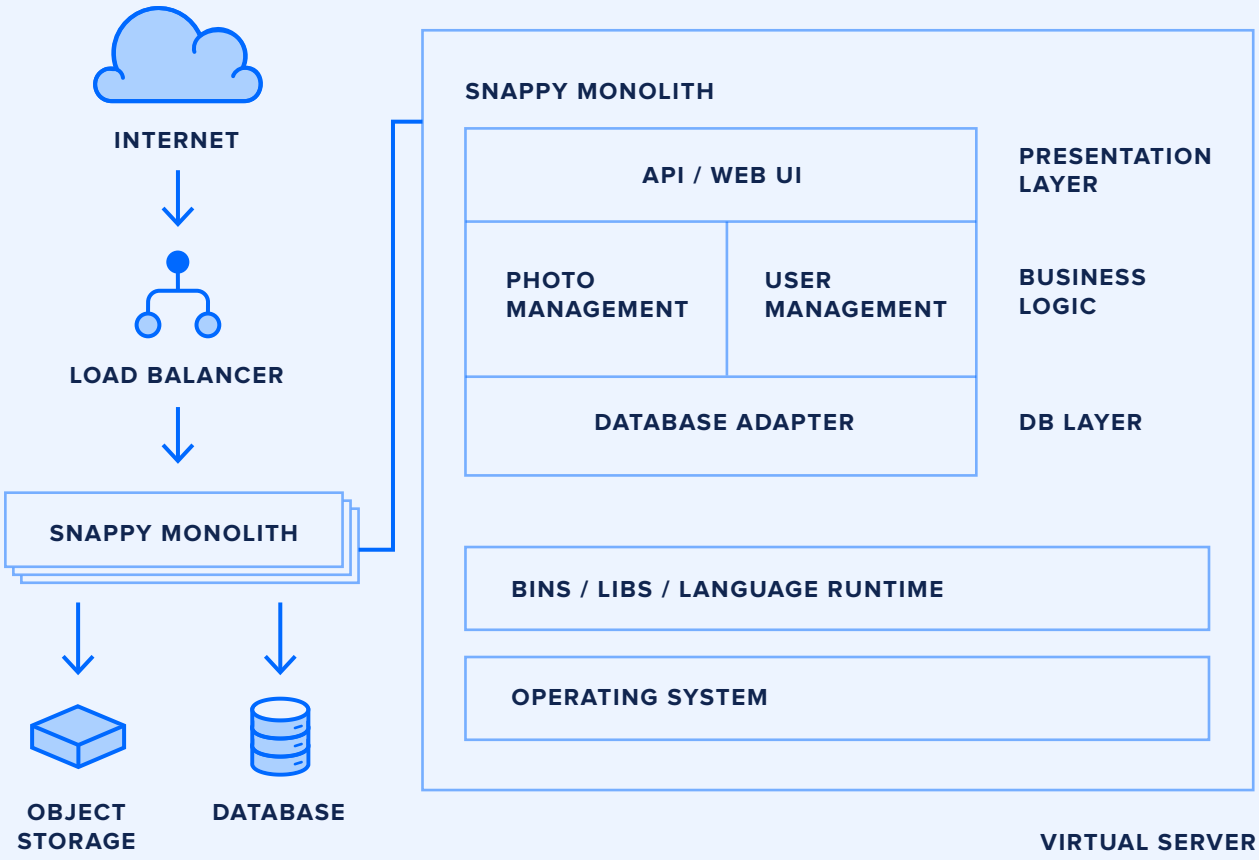
- ☐ Packaging applications into containers: "**containerizing**"
- ☐ Dynamic scheduling of these containers: "**container orchestration**"
- ☐ Software architectures that consist of several smaller loosely-coupled and independently deployable services: "**microservices**"

Scaling with Kubernetes Case Study: The Snappy Monolith

To demonstrate the value of implementing Cloud Native best practices including containerization along with a microservices architecture, we'll use a running example throughout this paper: a photo sharing app called Snappy that provides basic photo upload and sharing functionality between users through a web interface.

Throughout this paper, we'll modernize Snappy by:

- Decomposing the app's business functions into microservices
- Containerizing the various components into portable and discretely deployable pieces
- Using a DigitalOcean Kubernetes cluster to scale and continuously deploy these stateless microservices



With our photo-sharing monolith app Snappy, we observe that at first the web UI, photo management, and user management business functions are combined in a single codebase where these separate components invoke each other via function calls. This codebase is then built, tested, and deployed as a single unit, which can be scaled either horizontally or vertically.

As Snappy acquires more users — and subsequently scales and iterates on the product — the codebase gradually becomes extremely large and complex. Although code changes and bug fixes may be minor, the entire monolith app needs to be rebuilt and re-deployed, forcing a slower iteration cycle. Furthermore, if any individual subcomponent becomes a bottleneck in the application, the entire monolith must be scaled as a whole.

Microservices

Microservices is a software architecture style that advocates for many granular services that each perform a single business function. Each microservice is a self-contained, independently deployable piece of a larger application that interacts with other components, typically via well-defined REST APIs.

Microservices evolved as a hybrid of several software development trends and ideas. Some, like Service-Oriented Architectures (SOA), DevOps, and containerization are more recent, while others, like the Unix philosophy of “Do One Thing and Do It Well,” evoke principles developed decades ago. Initially pioneered by large, web-scale internet companies like Google, Amazon, and Netflix, microservices architectures have now become commonplace in applications of all sizes.

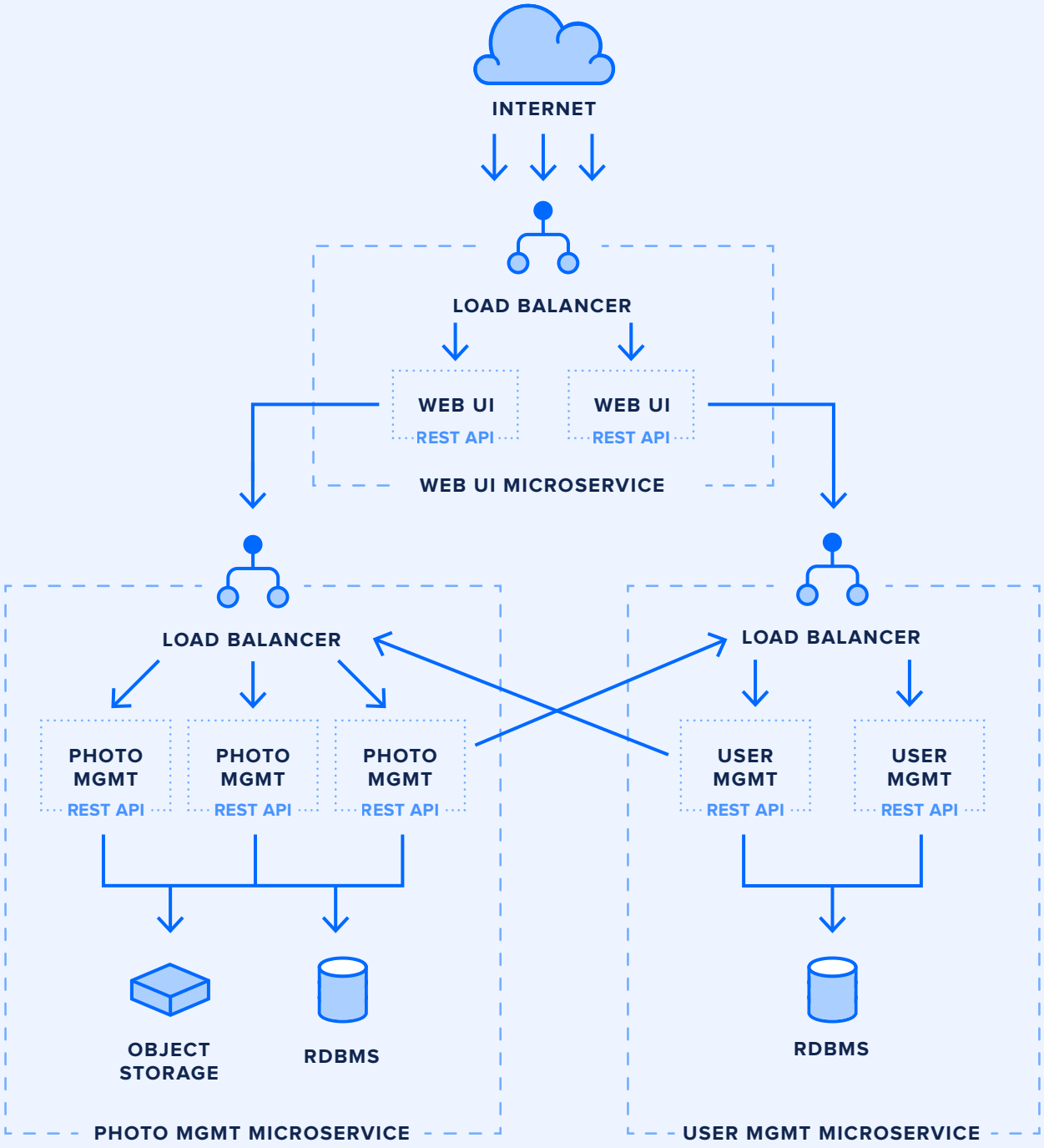
Monolithic vs. Microservices Architectures

Large, multi-tier software monoliths must be scaled as a cohesive whole, slowing down development cycles. In contrast, microservices provide several advantages over this inflexible model:

- They can be scaled individually and on demand
- They can be developed, built, tested and deployed independently
- Each service team can use its own set of tools and languages to implement features, and grow at its own rate
- Any individual microservice can treat others as black boxes, motivating strongly communicated and well-defined contracts between service teams
- Each microservice can use its own data store which frees teams from a single overarching database schema

Breaking the Monolith: Snappy Microservices Architecture

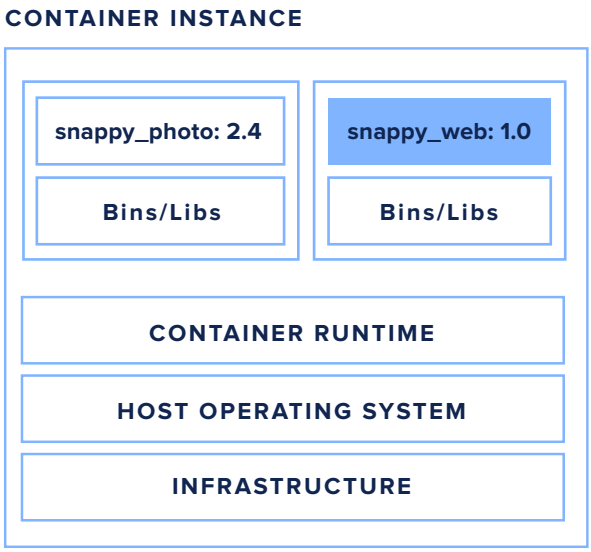
Looking at Snappy through the lens of a microservices-oriented architecture, we see that individual business functions such as photo management and user administration have now been broken out into separate microservices that communicate via REST APIs. Each microservice can use the data store most appropriate for the type of data it will be handling. For example, the photo management microservice can use a cloud object store for images along with a traditional relational database management system (RDBMS) for metadata. The service can then be developed, tested, deployed, and scaled independently from other services. Allowing for greater development agility, teams leveraging microservices can more efficiently use cloud infrastructure.


INTERNET
LOAD BALANCER
WEB UI
REST API
WEB UI
REST API
WEB UI MICROSERVICE
LOAD BALANCER
LOAD BALANCER
PHOTO MGMT
REST API
PHOTO MGMT
REST API
PHOTO MGMT
REST API
USER MGMT
REST API
USER MGMT
REST API
OBJECT STORAGE
RDBMS
RDBMS
PHOTO MGMT MICROSERVICE
USER MGMT MICROSERVICE

A microservices architecture lies at the heart of Cloud Native application development as these apps consist of portable, containerized microservices that are scalable and disposable. By breaking up a monolithic application into many fine-grained, self-contained services — each of which can be independently scaled, distributed across regions, and deployed across cloud platforms — microservices enable faster development iteration, and increased flexibility in running applications on cloud infrastructure.

Containers

Thus far we've discussed high-level methodologies and architectures that underpin Cloud Native applications. We'll now discuss a core technology that allows developers to implement these broader architectural designs: **containers.**



What are Containers?

Containers are a way of packaging applications with all of their required dependencies and libraries in a portable and easily deployable format. Once launched, these packages provide a consistent and predictable runtime environment for the containerized application. Taking advantage of Linux kernel isolation features such as `cgroups` and `namespaces`, container implementations — or runtimes — provide a sandboxed and resource-controlled running environment for applications.

Containers vs. Virtual Machines

Compared to virtual machines, containers are more lightweight and require fewer resources because they encapsulate fewer layers of the operating system stack. Both provide resource-limited environments for applications and all their software dependencies to run, but since containers share the host's OS kernel and do not require separate operating systems, they boot in a fraction of the time and are much smaller in size.

Container Runtimes

Though multiple container runtimes are available, Docker is the most mature, widely supported, and common format, embedded into most container orchestration systems. More recently, the Open Container Initiative (OCI), a Linux Foundation project, has worked to standardize container formats and runtimes, leading to the development and integration of lightweight and stable OCI-compliant runtimes such as containerd into orchestration systems. In mid-2018, The Kubernetes project announced general availability for the more minimal `containerd` runtime, embedding it into Kubernetes versions 1.10 and above.

DOCKERFILE

```
FROM      node: 10.2.1
...
copy      package.json ./
run       npm install
expose    8080
cmd       ["npm", "start"]
...
```

Build and Ship: Dockerizing an Application

Containerizing an application using Docker first involves writing a container image manifest called a *Dockerfile*. This file describes how to build a container image by defining the starting source image and then outlining the steps required to install any dependencies (such as the language runtime and libraries), copy in the application code, and configure the environment of the resulting image.

Developers or build servers then use the Docker container runtime to build these dependencies, libraries, and application sources into a binary package called a Docker image. Docker images are built in ordered layers, are composable, and can be reused as bases for new images. Once built, these images can be used to start containers on any host with a Docker container runtime.
Containers are central to running portable Cloud Native applications because using them naturally guides development towards the implementation of several core Twelve Factor and Cloud Native principles. As needed, a given Docker container:

- Implements some narrow piece of business or support logic
- Explicitly declares all of its software dependencies in a Dockerfile
- Is extremely portable across cloud providers as long as it has the requisite resources and a
- Docker runtime
- Deploys quickly to replace a failed running container of the same type
- Replicates easily to accommodate the additional load on a heavily requested business function by launching additional container instances

Once your team's application has been neatly packaged into a set of microservice containers, each performing some unit of business functionality, you should consider the following questions:

- How do you then deploy and manage all of these running containers?
- How do these containers communicate with one another, and what happens if a given container fails and becomes unresponsive?
- If one of your microservices begins experiencing heavy load, how do you scale the number of running containers in response, assigning them to hosts with resources available?

This is where Kubernetes and container orchestration come into play.

Building a Snappy Microservice: Web UI Container Lifecycle

To demonstrate one possible container-based development workflow, we'll zoom in on Snappy's Web UI microservice, an Express and Node.js based web server.

Develop & Push

Since this Express app depends on Node.js and its related libraries to run, the `snappy_web` Dockerfile first sources a Node.js image from a Docker registry like Docker Hub. Further Dockerfile steps include app configuration, copying in source code files, and finally telling Docker which command to run when launching the application container.

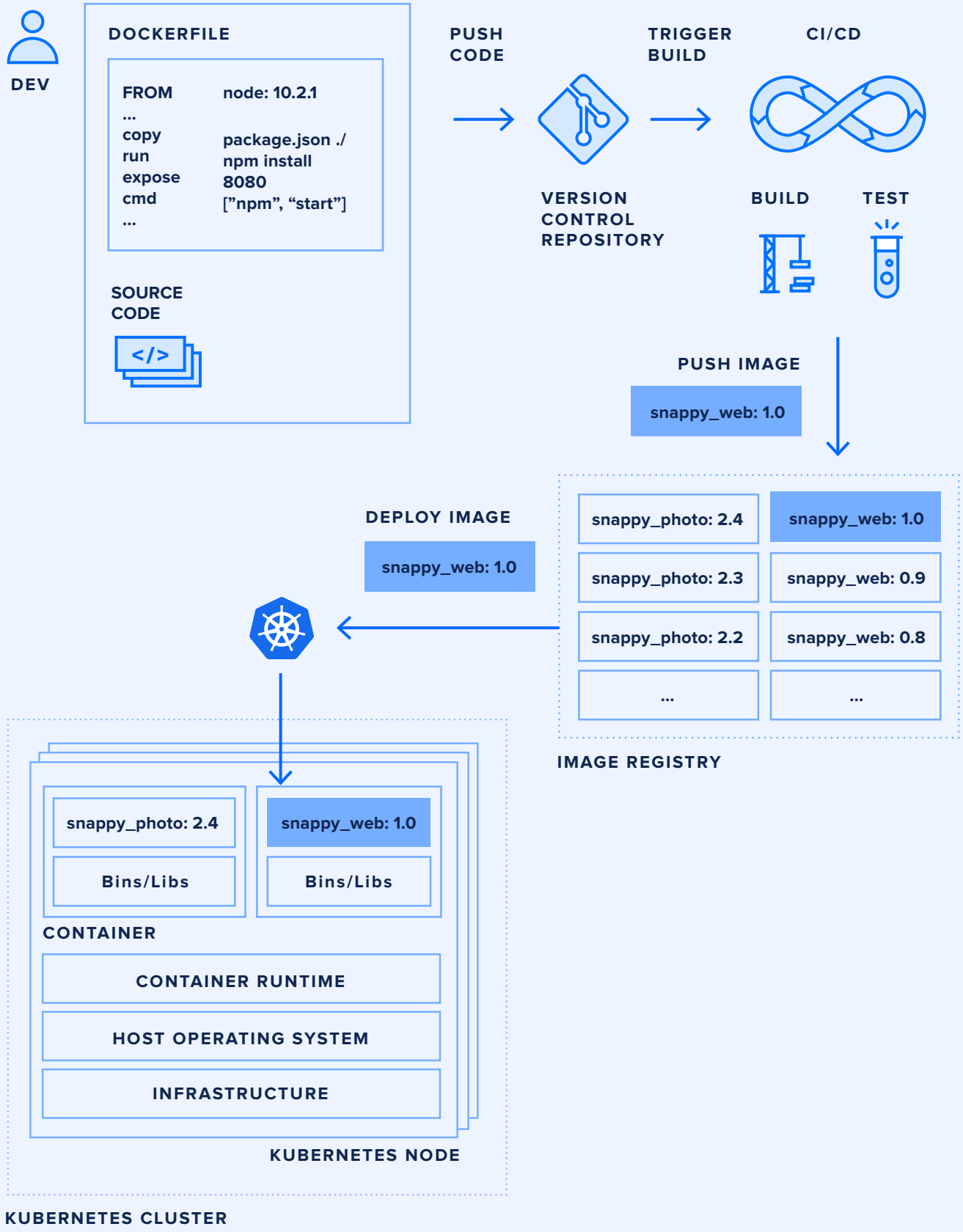
A Snappy developer on the web UI team begins work on a small UI bug fix. The developer tests code changes locally by rebuilding the Express Docker image and running the web server container on their laptop. The developer, satisfied with the changes, then pushes the code to a branch in the team's source code repository.

Build & Test

This push triggers a continuous integration pipeline build. The continuous integration server builds the Node.js dependencies, libraries, and modified sources into a Docker image. Once built, the pipeline runs this image as a container and proceeds to execute unit and integration tests.
If all the tests pass, this validated Docker image is then pushed to a container image registry — an abstraction layer typically built on top of object storage — for container image tagging, organization, and efficient reuse of existing image layers. Private, organization-specific container image registries allow teams to share and publish images to a central, secure location, whereas public registries allow popular open-source projects like Node.js to make the latest versions of their software available to developers in prebuilt container image form.

Deploy & Run

Now that the Snappy web UI container image containing the bug fix has been approved and pushed to a registry, a container orchestration system such as Kubernetes can then "pull" this image and deploy it onto any host with a Docker runtime. This running container is given a narrow view of available resources specified by the developer, and is run in isolation from other containers. A single host with a container runtime installed can run several containers simultaneously, depending on available resources.


DEV
DOCKERFILE
FROM node: 10.2.1
...
copy package.json ./
run npm install
expose 8080
cmd ["npm", "start"]
...
SOURCE CODE
</>
PUSH CODE
VERSION CONTROL REPOSITORY
TRIGGER BUILD
CI/CD
BUILD
TEST
PUSH IMAGE
snappy_web: 1.0
DEPLOY IMAGE
snappy_web: 1.0
snappy_photo: 2.4
snappy_web: 1.0
snappy_photo: 2.3
snappy_web: 0.9
snappy_photo: 2.2
snappy_web: 0.8
...
...
IMAGE REGISTRY
snappy_photo: 2.4
Bins/Libs
snappy_web: 1.0
Bins/Libs
CONTAINER
CONTAINER RUNTIME
HOST OPERATING SYSTEM
INFRASTRUCTURE
KUBERNETES NODE
KUBERNETES CLUSTER

Clusters

Adopting a microservices architecture consisting of containerized applications paves the way for more efficient use of infrastructure, close control of application runtime environments, and the ability to automatically scale. However, one of the major tradeoffs in moving to a microservices-oriented architecture are the added complexities (e.g. in business logic, observability, and incident management) in managing a constantly evolving distributed system. Container orchestration systems were designed to reduce some of the operations overhead by abstracting away the underlying infrastructure and automating the deployment and scaling of containerized applications. Systems such as Kubernetes, Marathon and Apache Mesos, and Swarm simplify the task of deploying and managing fleets of running containers by implementing some or all of the following core functionality:

- Container Scheduling
- Load Balancing
- Service Discovery
- Cluster Networking
- Health Checking and State Management
- Autoscaling
- Rolling Deployments
- Declarative Configuration

Let's briefly take a look at each of these features:

Container Scheduling

When deploying a container or sets of identical containers, a **scheduler** manages allocating the desired resources (like CPU and memory) and assigns the containers to cluster member nodes with these resources available. In addition, a scheduler may implement more advanced functionality like container prioritization, as well as balancing out sets of identical containers across different members and regions for high availability.

Load Balancing

Once deployed into a cluster, sets of running containers need some load balancing component to manage the distribution of requests from both internal and external sources. This can be accomplished using a combination of cloud provider load balancers, as well as load balancers internal to the container orchestration system.

Service Discovery

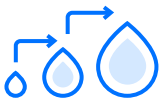
Running containers and applications need some way of finding other apps deployed to the cluster. Service discovery exposes apps to one another and external clients in a clean and organized fashion using either DNS or some other mechanism, such as local environment variables..

Cluster Networking

Clusters also need to connect running applications and containers to one another across machines, managing IP addresses and assignment of network addresses to cluster members and containers. Networking implementations vary across container cluster projects; some like Docker Swarm bake a set of networking features directly into the cluster, whereas others like Kubernetes impose a minimal set of requirements for any networking implementation, allowing administrators to roll out their own custom overlay network solution.

Health Checking and State Management

A core feature implemented by Cloud Native applications is health reporting, usually via a REST endpoint. This allows orchestrators to reliably check the state of running applications and only direct traffic towards those that are healthy. Also using this endpoint, orchestrators repeatedly probe running apps and containers for “liveness” and self-heal by restarting those that are unresponsive.



Autoscaling

As load increases on a given application, more containers should be deployed to match this growth in demand. Container orchestrators handle scaling applications by monitoring standard metrics such as CPU or memory use, as well as user-defined telemetry data. The orchestrator then increases or decreases the number of running containers accordingly. Some orchestration systems also provide features for scaling the cluster and adding additional cluster members should the number of scheduled containers exceed the amount of available resources. These systems can also monitor utilization of these members and scale the cluster down accordingly, rescheduling running containers onto other cluster members.

Rolling Deployments

Container orchestration systems also implement functionality to perform zero-downtime deploys. Systems can roll out a newer version of an application container incrementally, deploying a container at a time, monitoring its health using the probing features described above, and then killing the old one. They also can perform blue-green deploys, where two versions of the application run simultaneously and traffic is cut over to the new version once it has stabilized. This also allows for quick and painless rollbacks, as well as pausing and resuming deployments as they are carried out.

Declarative Configuration

Another core feature of some container orchestration systems is deployment via declarative configuration files. The user “declares” which desired state they would like for a given application (for example, four running containers of an NGINX web server), and the system takes care of achieving that state by launching containers on the appropriate members, or killing running containers. This declarative model enables the review, testing, and version control of deployment and infrastructure changes. In addition, rolling back applications version can be as simple as deploying the previous configuration file. In contrast, imperative configuration requires developers

to explicitly define and manually execute a series of actions to bring about the desired cluster state, which can be error-prone, making rollbacks difficult.

Depending on the container orchestration project, some of these features may be implemented with more or less maturity and granularity. Whether it's Apache's Mesos and Marathon, Docker's Swarm, or Kubernetes, a container orchestration system greatly facilitates running scalable and resilient microservices on cloud platforms by abstracting away the underlying infrastructure and reducing operational complexity.

Rolling out cluster software to manage your applications often comes with the cost of provisioning, configuring, and maintaining the cluster. Managed container services like DigitalOcean Kubernetes can minimize this cost by operating the cluster Control Plane and simplifying common cluster administration tasks like scaling machines and performing cluster-wide upgrades.

As open-source container clusters and their managed equivalents have evolved and gradually taken on large-scale production workloads, Kubernetes and its expanding ecosystem of Cloud Native projects have become the platform of choice for managing and scheduling containers. By implementing all of the features described above, Kubernetes empowers developers to scale alongside their success, and managed Kubernetes offerings provide them with even greater flexibility while minimizing DevOps administration time and software operations costs.

Kubernetes

The Kubernetes container orchestration system was born and initially designed at Google by several engineers who architected and developed Google's internal cluster manager Borg. These engineers sought to build and open-source a container management system that integrated many of the lessons learned from developing this internal container platform. Since its July 2015 v1.0 release, Kubernetes has rapidly matured and been rolled out in large production deployments by organizations such as Bloomberg, Uber, eBay, and also here at DigitalOcean. In March 2018, Kubernetes became the first project to graduate from the Cloud Native Computing Foundation, indicating that it has become mature and stable enough to handle large-scale production deployments and has achieved a high level of code quality and security.

Beyond implementing all of the container cluster features listed above (and many more), Kubernetes benefits from a thriving open-source community, which actively develops new features and provides constant feedback and bug reporting across a variety of deployments and use cases. In addition to Kubernetes features, this growing developer community continuously builds tools that simplify the process of setting up, configuring, and managing Kubernetes clusters.

Before discussing DigitalOcean Kubernetes and how it enables developers to rapidly deploy their containerized Cloud Native applications into a managed cluster, we'll first dive into the design and architecture of the Kubernetes system and discuss some of the core features that simplify deploying and scaling applications.

Kubernetes Design Overview

Kubernetes is a container cluster, a dynamic system that manages the deployment, management, and interconnection of containers on a fleet of worker servers. These worker servers where containers run are called **Nodes** and the servers that oversee and manage these running containers are called the Kubernetes **Control Plane**.

Containers and Pods

It's important to note here that the smallest deployable unit in a Kubernetes cluster is not a container but a **Pod.** A Pod typically consists of an application container (like a Dockerized Express/Node.js web app), or an app container and any "sidecar" containers that perform some helper function like monitoring or logging. Containers in a Pod share storage resources, a network namespace, and port space. A Pod can be thought of as a group of containers that work together to perform a given function. They allow developers to ensure that these sets of containers are always scheduled onto Nodes together.

Scaling, Updating and Rolling Back: Kubernetes Deployments

Pods are typically rolled out using **Deployments,** which are objects defined by YAML files that declare a particular desired state. For example, an application state could be running three replicas of the Express/Node.js web app container and exposing Pod port `8080`. Once created, a **controller** on the Control Plane gradually brings the actual state of the cluster to match the desired state

declared in the Deployment by scheduling containers onto Nodes as required. Using Deployments, a service owner can easily scale a set of Pod replicas horizontally or perform a zero-downtime rolling update to a new container image version by simply editing a YAML file and performing an API call (e.g. by using the command line client `kubectl`). Deployments can quickly be rolled back, paused, and resumed.

Exposing your Application: Kubernetes Services

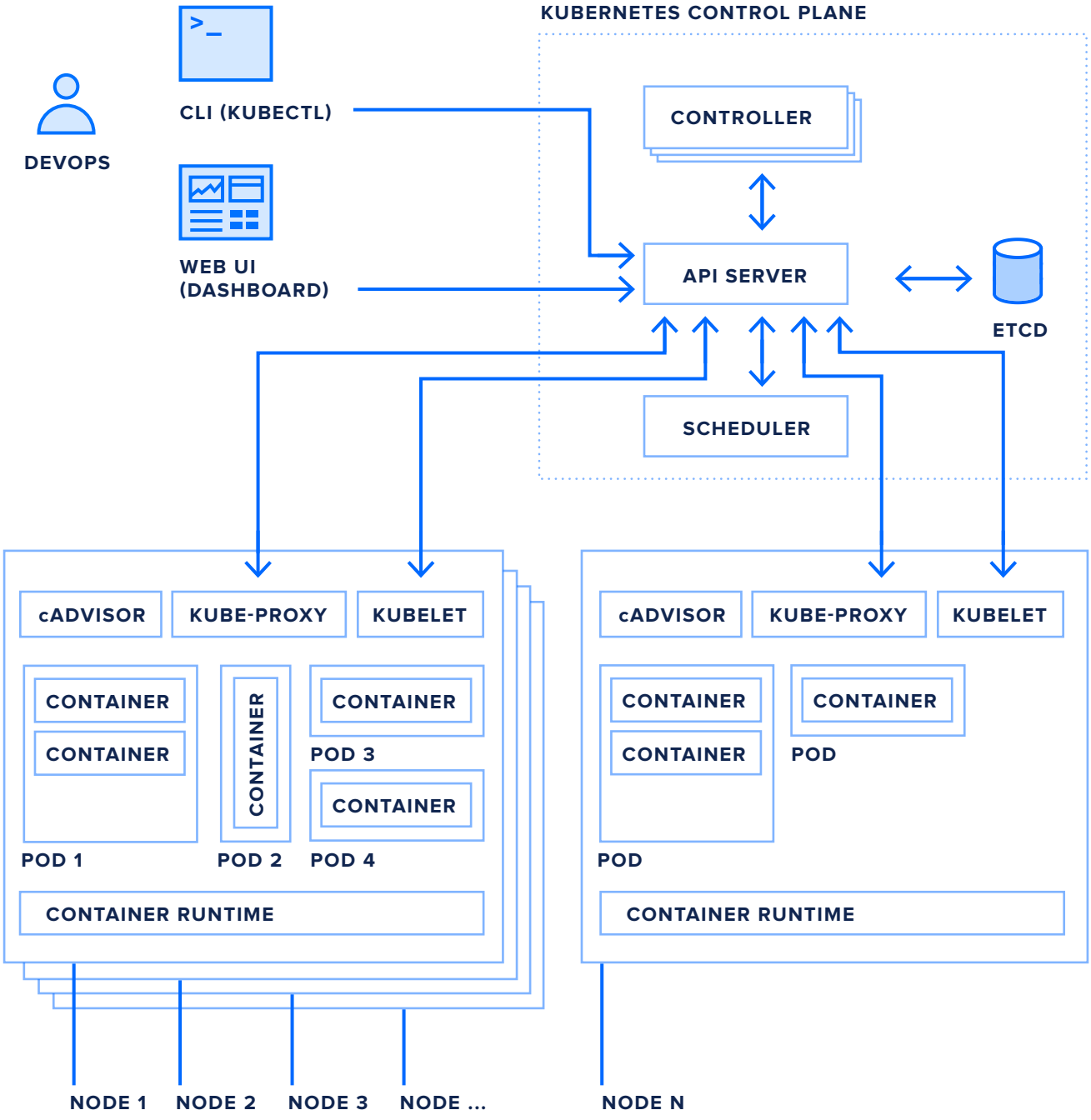
Once deployed, a **Service** can be created to allow groups of similar deployed Pods to receive traffic (Services can also be created simultaneously with Deployments). Services are used to grant a set of Pod replicas a static IP and configure load balancing between them using either cloud provider load balancers or user-specified custom load balancers. Services also allow users to leverage cloud provider firewalls to lock down external access.

Pod Management: Kubernetes Node Agents

To start and manage Pods and their containers on worker machines, Nodes run an agent process called **kubelet** which communicates with a **kube-apiserver** on the Control Plane. Using a **container runtime** like Docker, also running on Nodes, these scheduled containers are first pulled as images from either a private or public image registry, and then created and launched. Nodes also run a kube-proxy, which manages network rules on the host.

Control and Schedule: Kubernetes Control Plane

The Kubernetes Control Plane oversees the Nodes and manages their scheduling and maintains their workloads. It consists of the **kube-apiserver** front-end, backed by the key-value store **etcd** to store all the cluster data. Finally, a **kube-scheduler** schedules Pods to Nodes, and a set of **controllers** continuously observe the state of the cluster and drive its actual state towards the desired state.


KUBERNETES CONTROL PLANE
DEVOPS
CLI (KUBECTL)
WEB UI
(DASHBOARD)
CONTROLLER
API SERVER
ETCD
SCHEDULER
cADVISOR
KUBE-PROXY
KUBELET
CONTAINER
CONTAINER
CONTAINER
POD 1
POD 2
CONTAINER
CONTAINER
POD 3
CONTAINER
POD 4
CONTAINER RUNTIME
cADVISOR
KUBE-PROXY
KUBELET
CONTAINER
CONTAINER
CONTAINER
POD
POD
CONTAINER RUNTIME
NODE 1
NODE 2
NODE 3
NODE ...
NODE N

This brief high-level architectural overview demonstrates that Kubernetes provides advanced and extensible functionality for deploying and running containerized applications on cloud infrastructure. However, provisioning, configuring, and managing a Kubernetes cluster, and integrating existing CI/CD pipelines and automation servers often requires non-trivial lead and maintenance time for operators and DevOps engineers. Managed Kubernetes solutions like DigitalOcean Kubernetes allow development teams to quickly provision a fully featured Kubernetes cluster on which they can launch their containerized workloads. This means that software teams can spend more time building applications and less time managing integration, deployment, and the infrastructure that apps run on.

DigitalOcean Kubernetes

Features described in this section will be ready for use when DigitalOcean Kubernetes is generally available.

DigitalOcean Kubernetes provides a simple and cost-effective solution for developers seeking to deploy and run their containerized applications on a Kubernetes cluster. With DigitalOcean, developers can quickly launch their containerized workloads into a managed, self-healing Kubernetes environment without having to provision, install, and manage a cluster from scratch.

Rolling your own production-ready Kubernetes cluster often involves several time-consuming and costly steps: provisioning the underlying compute, storage, and networking infrastructure for the Control Plane and Nodes; bootstrapping and configuring Kubernetes components like the etcd cluster and Pod networking; and thoroughly testing the cluster for resiliency towards infrastructure failures. Once set up, Kubernetes clusters need to be managed and monitored by DevOps teams, while routine maintenance tasks like upgrading the cluster or the underlying infrastructure requires manual intervention by engineers.

Completing the Shift to Cloud Native: Snappy on DigitalOcean Kubernetes

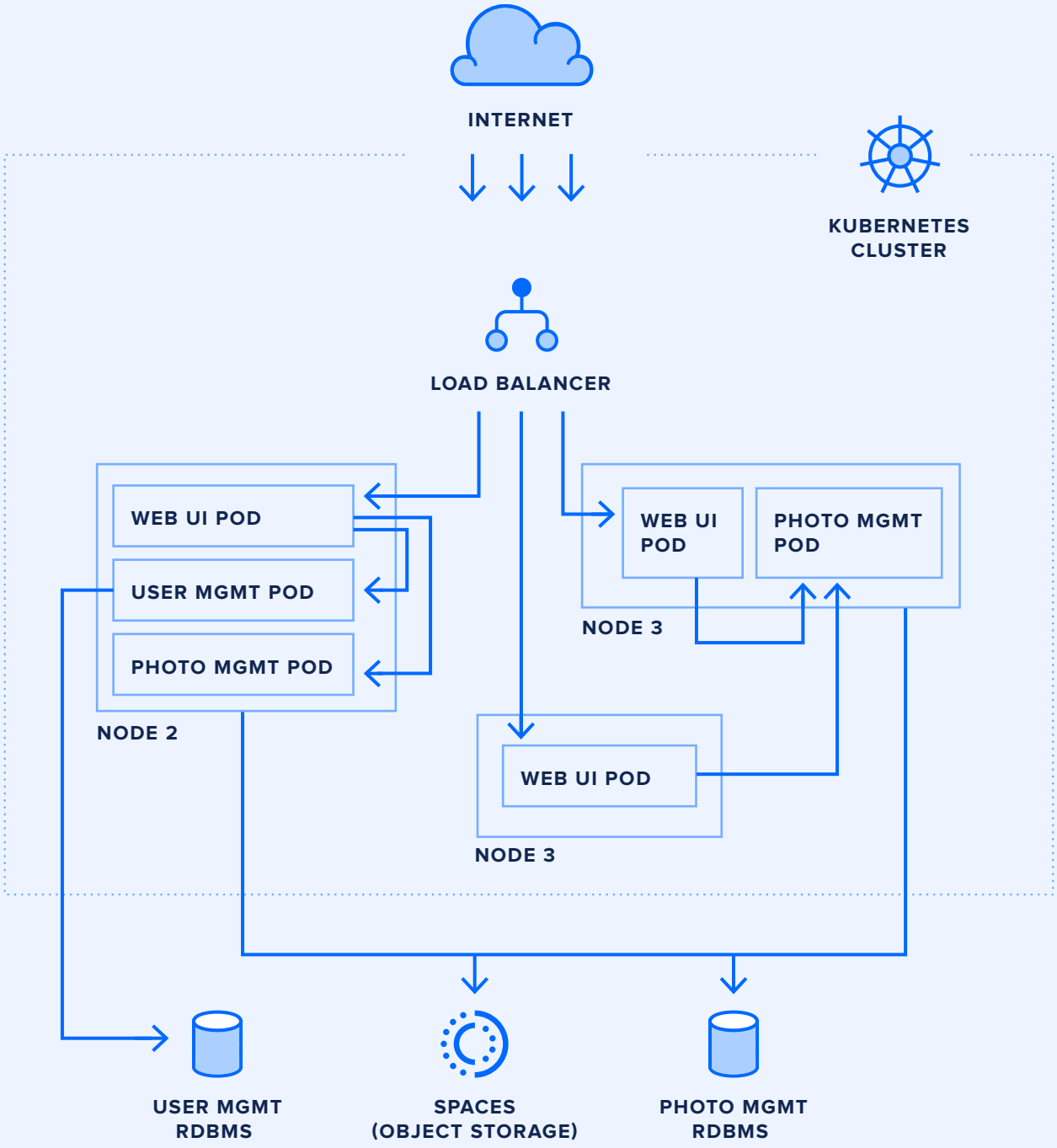
The Snappy team has evaluated DigitalOcean Kubernetes and decided they would like to roll out their microservices-based photo sharing application onto the platform. Running on Kubernetes, Snappy consists of the following three Pods (it's helpful to think of each as a microservice):

- Web UI
- Photo Management
- User Management

Once the cluster has been provisioned with the desired number of Nodes, Snappy's developers create Kubernetes Services and Deployments for each microservice component, referencing Docker images from their private registry. Using Kubernetes Secrets, they can store database credentials and other sensitive information that they would like Pods to have access to in their environment.

Subsequently, they could then use kubectl or their preferred Kubernetes management tool to launch the Services into their cluster. Once launched, DigitalOcean Kubernetes will automatically provision and configure remaining cloud infrastructure such as Block Storage, Load Balancers and Firewalls, as declared by the developer.

Service owners can then independently scale their services by first increasing the number of replica Pods in the Deployment configuration file, and then launching the deployment using the Kubernetes API. This same flow can be automated and integrated with a CI/CD pipeline to continuously roll out new versions of the application.


INTERNET
KUBERNETES CLUSTER
LOAD BALANCER
WEB UI POD
USER MGMT POD
PHOTO MGMT POD
NODE 2
WEB UI POD
PHOTO MGMT POD
NODE 3
WEB UI POD
NODE 3
USER MGMT RDBMS
SPACES (OBJECT STORAGE)
PHOTO MGMT RDBMS

Simple, Flexible Scaling

Development teams using DigitalOcean Kubernetes can quickly create managed clusters for workloads of all sizes — from a single Node, single Pod web server to a massive set of Nodes with continuously churning compute-intensive processing pods. Regardless of the size or number of running applications, cluster creation and operation remains simple via a streamlined web interface and REST API, allowing developers to quickly launch and scale a managed Kubernetes cluster. Users can define **custom configurations** of standard and compute-optimized Droplets to handle variable workloads, optimizing price-to-performance and maximizing the use of underlying resources.

Minimizing Day 2 Cost: Managed Operations and Maintenance

Automated cluster upgrades and **Control Plane backup and recovery** further reduce operations and day-to-day management overhead. Teams can quickly integrate the latest security and performance improvements from the Kubernetes community while keeping their clusters available to run workloads and remaining resilient to failure. DigitalOcean Kubernetes clusters **self-heal automatically** — Control Plane and Node health are continuously monitored, and recovery and Pod rescheduling occurs in the background, preventing unnecessary and disruptive off-hours alerts.

Kubernetes in the DigitalOcean Cloud

DigitalOcean Kubernetes **integrates seamlessly with other DigitalOcean infrastructure products**, bringing in all the cloud primitives needed to handle scaling and securing your team's applications. When creating a Service to expose your app, DigitalOcean Kubernetes can **automatically provision a Load Balancer** and route traffic to the appropriate Pods. Additionally, you'll be able to **set up a DigitalOcean Firewall** to lock down and restrict web traffic to your running applications. Finally, DigitalOcean **Block Storage Volumes** can be used as `PersistentVolumes` to provide non-ephemeral and highly available shared storage between containers.

Harness the Kubernetes Ecosystem

By fully exposing the Kubernetes Control Plane API, developers have **complete control** over workload deployment, scaling, and monitoring. Container images can be pulled directly from public and private registries like Docker Hub and Quay.io, granting teams complete flexibility in designing and implementing continuous integration and deployment pipelines. With this exposed API, developers can also benefit from the rich ecosystem of third-party Kubernetes tools, as well as

build their own implementation-specific tooling should no existing solution meet their needs. In addition, operations teams can quickly leverage the Kubernetes Dashboard UI as well as Prometheus — built-in to DigitalOcean Kubernetes — for out-of-the-box cluster monitoring and troubleshooting.

Simple, Transparent Pricing

Using DigitalOcean Kubernetes, software teams can maximize their cloud resource utilization and accurately forecast spend with transparent, predictable pricing. Paying only for running Nodes, teams have their bandwidth pooled at the account level, making DigitalOcean Kubernetes a market leading price-to-performance container platform.

DigitalOcean Kubernetes supports development teams in quickly reaping the benefits provided by Cloud Native computing patterns. Helping teams as they build containerized, microservices-oriented applications that are designed to run on the cloud, DigitalOcean Kubernetes reduces time to market and facilitates the scaling of products and iteration of features. This streamlined, fully-managed, CNCF-conformant platform runs all kinds of production container workloads. Your team will experience increased development velocity while fully leveraging the flexibility offered by cloud infrastructure. DigitalOcean Kubernetes provides you with **simplicity as you scale.**

DigitalOcean

Hanif Jetha
Lisa Tagliaferri
DigitalOcean
June 2018