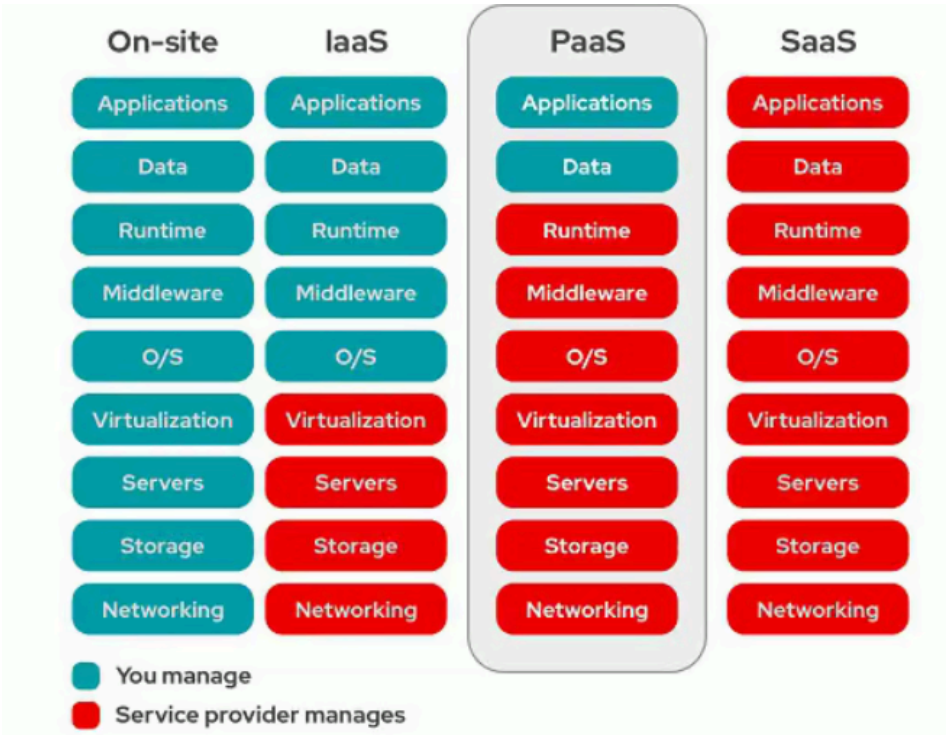
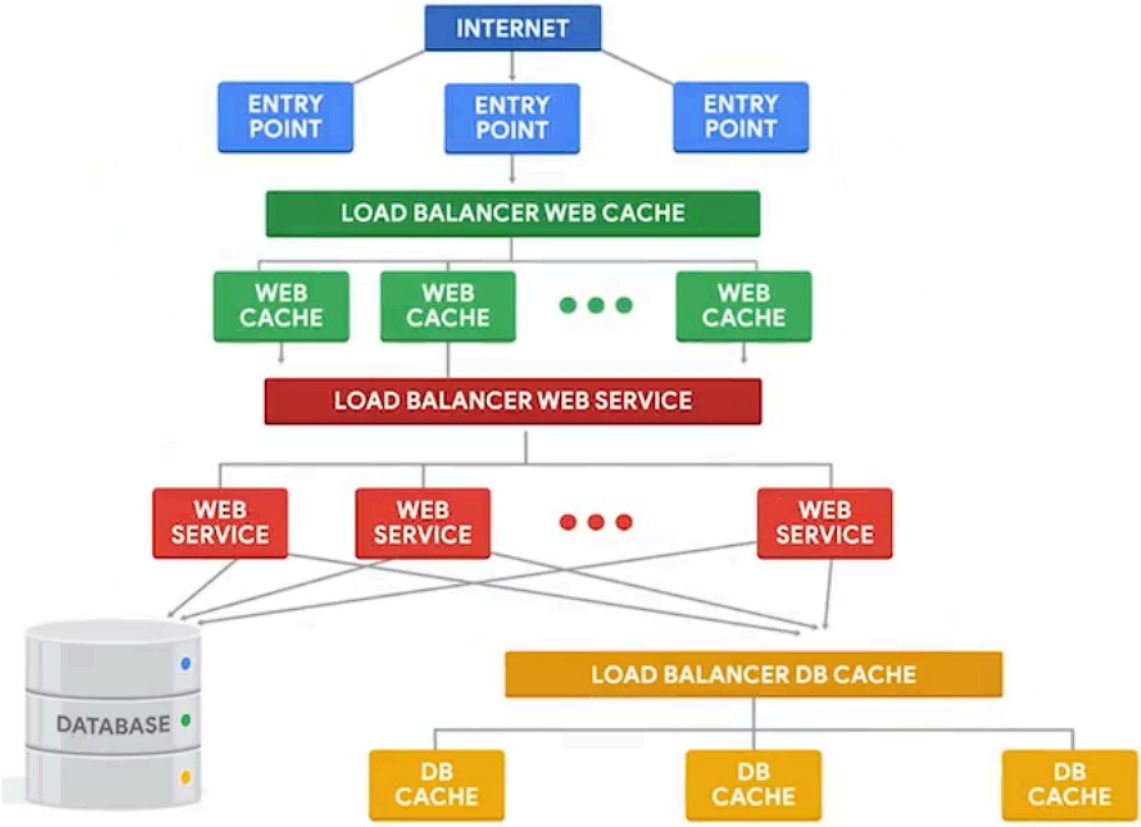
- X
- 

- **Creating Virtual Machines (VMs):** You'll need to define parameters like:
    - **Instance Name:** For identification.
    - **Region and Zone:** Choose based on user location for optimal performance.
    - **Machine Type:** Balance processing power (CPU, memory) with cost considerations.
    - **Boot Disk:** Select the operating system and storage space.
- **Web UI vs. Command Line Interface (CLI):**
    - **Web UI:** Great for visualization and experimentation.
    - **CLI:** Ideal for automation and managing multiple VMs.
- Snapshot: Full copy of disk
- Image: Lets us create a template based on snapshot

Web Application Architecture

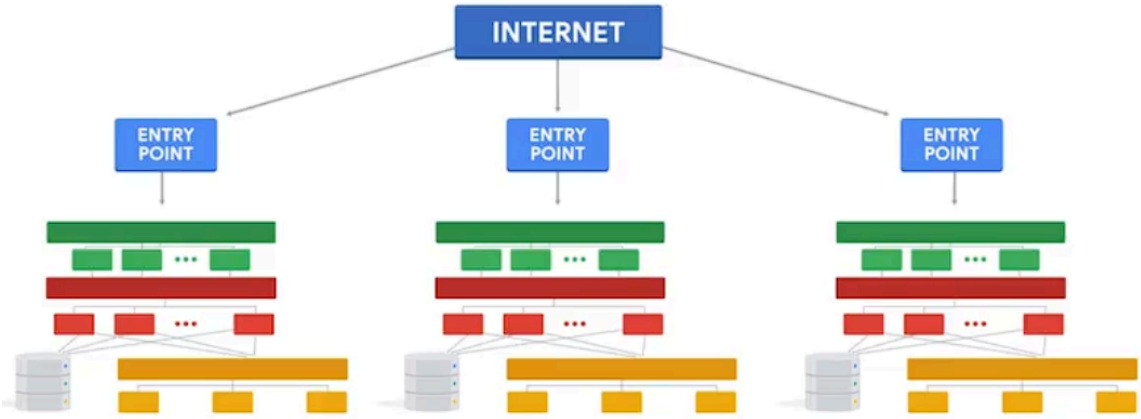
- A typical large-scale web application uses multiple layers of caching (like Varnish or Nginx) to speed up content delivery.
- These layers, along with load balancers and potentially geographically distributed entry points, help ensure a fast and reliable user experience.



-
    - ^Set up monitoring and alerting
- Popular DB caching: MemcacheD and Redis
- Popular web caching servers: Varnish and Nginx
    - Use a load balancer across multiple web caching servers
    - Cloudflare and fastly also do web caching

What is Orchestration?

- Orchestration goes beyond simply automating individual tasks; it involves automating the configuration of entire systems, ensuring that different components interact seamlessly.
- Can use orchestration to ensure monitoring and alerting are standard across all deployments



-

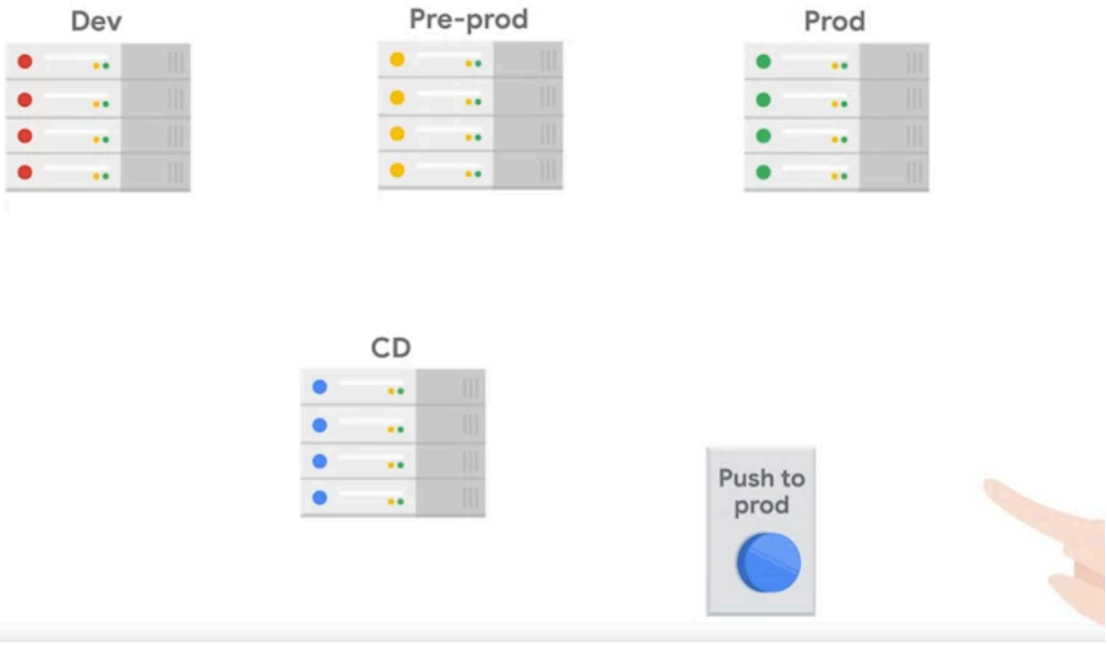
- In order to enable `hello_cloud.py` to run on boot, copy the file `hello_cloud.py` to the `/usr/local/bin/` location.
- `sudo cp hello_cloud.py /usr/local/bin/`
- Copied!
- content_copy
- Also copy `hello_cloud.service` to the `/etc/systemd/system/` location.
- `sudo cp hello_cloud.service /etc/systemd/system`
- Copied!
- content_copy
- Now, use the systemctl command to enable the service **hello_cloud**.
- `sudo systemctl enable hello_cloud.service`

- Gcloud auth login
- Gcloud config set project “ProjectID”
- gcloud compute instances create --zone us-central1-a --source-instance-template vm1-template vm2 vm3 vm4 vm5 vm6 vm7 vm8
- gcloud compute instances list

- **Dedicated Load Balancers:** Offer more control by acting as a proxy between clients and servers. They can direct traffic based on various rules, monitor server health, and enable sticky sessions for stateful applications.

Advanced Load Balancing Techniques

- **GeoDNS and GeoIP:** Route traffic to the geographically closest server, reducing latency and improving response times for users worldwide.
- **Content Delivery Networks (CDNs):** Cache content on servers distributed globally, ensuring users receive data from the closest possible source for faster loading times.
- Many cloud providers also offer continuous integration as a service. Once the change has committed, the CI system will build and test the resulting code. Now you can use continuous deployment, or CD, to automatically deploy the results of the build or build artifacts. Continuous deployment lets you control the deployment with rules. For example, we usually configure our CD system to deploy new builds only when all of the tests have passed successfully.



- When you think a new prod is ready maybe try A/B testing at 1% then slowing ramp up if it works

Cloud Limitations

- **Rate and Utilization Limits:** Cloud providers often set limits on the number of operations you can perform within a specific timeframe to prevent system overload and control resource allocation.
- **Handling Limits:** To manage these limits, you can optimize your application to batch operations, switch to alternative services, or request quota increases from your provider.

Module 1 review

This has been a big module! We've covered a lot of ground related to cloud computing. We started by learning about clouds — not the ones that float through the sky, but computer services that run in a data center or remote servers that we access over the Internet. These clouds have different service types, from a bare bones environment that gives you the computing power to run your own software with your own configuration settings, to those that deliver a whole application or program to a user, such as easy to use email solutions like Gmail, storage solutions like Google Cloud, or productivity suites like Google Workspace.

Next we learned how to deploy a virtual machine (VM), made sure the VM was set up to serve our web app, and that it would stay updated via Puppet. A single VM can be useful for small operations with lower technical requirements, but as technical requirements increase, an organization will need to deploy more and larger cloud solutions. Luckily, it's easy to scale in the cloud. So we turned that single VM into a customized VM template that could be used to clone that VM as many times as we want, so scaling our cloud deployments would be easy.

When we had a VM template, we looked at different ways to interact with the platform. We saw how both the web interface and the command line tool can be used to create VMs in the cloud, modify

their configuration, and control other things, using tools which are very effective at a small or medium scale.
At a large scale, you'll need to automate cloud deployments even further using orchestration. We looked at how tools like Terraform allow us to define our cloud infrastructure as code, which can give us a lot of control over things like how the infrastructure is managed, and how changes are applied. Orchestration lets us combine the power of infrastructure as code with the flexibility of cloud resources.
Then it was time to look at some of the options for building software for the cloud. FIrst, we looked into the different types of storage available, and what to consider when deciding which storage solution to use. These considerations include how we want to request data from storage, and how fast we want to be able to access the requested data.
Maybe the number one reason for using cloud computing is how much computing power is available across many servers. But in order to distribute our service evenly across however many instances we have available, we use load balancing. We talked about the different methods load balancers use to distribute the workload, and how they can monitor server health to avoid sending requests to unhealthy servers.
There is no doubt that computing methods are constantly changing, and this happens pretty fast. We talked about how change management allows us to make changes in a safe and controlled way. We looked at how continuous integration (CI) can build and test code every time there is a change, and how continuous deployment (CD) can automatically control the deployment of new code to a specified set of rules. And we talked about different environments where new code could be tested before being pushed to production.
And we talked about limitations. Limitations you may come across when running services in the cloud are not the same as limitations when running services on physical machines, and you should take time to understand the limitations of any cloud solution you may choose.
Basically, you just learned a lot about cloud computing in a short period of time. And there's more to come. First, you've got a graded assessment that covers the material from this past module. Then it's time to start learning about deploying applications to the cloud.

Glossary terms from course 5, module 1

Terms and definitions from Course 5, Module 1

A/B testing: A way to compare two versions of something to find out which version performs better
Automatic scaling: This service uses metrics to automatically increase or decrease the capacity of the system
Autoscaling: Allows the service to increase or reduce capacity as needed, while the service owner only pays for the cost of the machines that are in use at any given time
Capacity: How much the service can deliver
Cold data: Accessed infrequently and stored in cold storage
Containers: Applications that are packaged together with their configuration and dependencies
Content Delivery Networks (CDN): A network of physical hosts that are geographically located as close to the end users as possible
Disk image: A snapshot of a virtual machine's disk at a given point in time

Ephemeral storage: Storage used for instances that are temporary and only need to keep local data while they're running
Hot data: Accessed frequently and stored in hot storage
Hybrid cloud: A mixture of both public and private clouds
Input/Output Operations Per Second (IOPS): Measures how many reads or writes you can do in one second, no matter how much data you're accessing
Infrastructure as a Service (or IaaS): When a Cloud provider supplies only the bare-bones computing experience
Load balancer: Ensures that each node receives a balanced number of requests
Manual scaling: Changes are controlled by humans instead of software
Multi-cloud: A mixture of public and/or private clouds across vendors
Object storage: Storage where objects are placed and retrieved into a storage bucket
Orchestration: The automated configuration and coordination of complex IT systems and services
Persistent storage: Storage used for instances that are long lived and need to keep data across reboots and upgrades
Platform as a Service (or PaaS): When a Cloud provider offers a preconfigured platform to the customer
Private cloud: When your company owns the services and the rest of your infrastructure
Public cloud: The cloud services provided to you by a third party
Rate limits: Prevent one service from overloading the whole system
Reference images: Store the contents of a machine in a reusable format
Software as a Service (or SaaS): When a Cloud provider delivers an entire application or program to the customer
Sticky sessions: All requests from the same client always go to the same backend server
Templating: The process of capturing all of the system configuration to let us create VMs in a repeatable way
Throughput: The amount of data that you can read and write in a given amount of time
Utilization limits: Cap the total amount of a certain resource that you can provision

- Containers eliminate the need for recipients to download all libraries and dependencies
- They act as a bridge between virtual machines (VMs) and Python virtual environments
- A hypervisor, also known as a virtual machine monitor or virtualizer, is a type of computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine

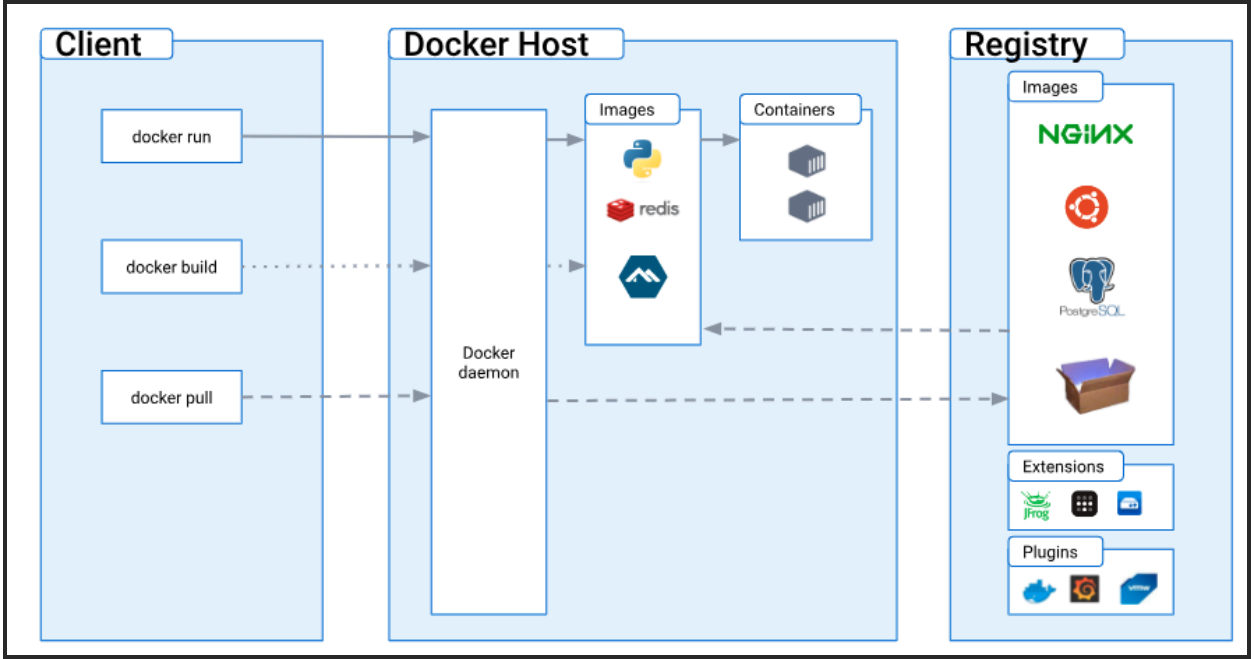
arts of Docker

The Docker ecosystem consists of the following parts:

- Docker daemon. This manages running containers on a host machine called the Docker Host.
- Docker CLI (command-line interface). This command-line tool interacts with Docker Daemon.

- Docker Desktop. This graphical user interface (GUI) tool interacts with the daemon.
- Docker Hub. This is the central repository for downloading containers.

You might hear the host machine referred to as Docker Host. Docker uses a client-server architecture as outlined by the image below. Docker supports running the client tools and daemon on different machines. This is an advantage of Docker as it allows you to manage containers on a remote server as easily as if they're on your own workstation.



Docker images

Docker images are the building blocks of Docker containers. They are lightweight, immutable, and composed of multiple layers. A Docker image contains the application code, data files, configuration files, libraries, and other dependencies needed to run an application.

In this reading, you will learn more about Docker images and their layers. You'll also learn how to build a Docker image, and you'll review an example of a Dockerfile.

Docker images and image layers

You can think of a Docker image as a template from which Docker containers are created and executed. Each Docker image is composed of multiple layers—adding or removing files from the previous layer. Each layer represents a specific set of changes made to the image and is composed based on the instructions in a Dockerfile. The instructions in a Dockerfile define how the image should be built.

Note: It's not uncommon for an image to be composed of a dozen or more layers.

The purpose of having multiple layers is to keep the final images as small as possible—you do this by reusing layers in multiple images—and to speed up the process of building containers, as Docker has to rebuild only the layers that have changed.

How to build a Docker image

The key to packaging your own application as a Docker image is to have a Dockerfile. The Dockerfile acts as your source of truth or instruction manual: It specifies how Docker should build the image and contains a series of commands to build the image. Each command builds a new layer that becomes part of the final image. A common process is to start with a base image such as Debian Linux or Python 3.10, install the libraries your application requires, then copy the application and any related files into the image. Let's take a look at a simple Dockerfile for a Python application example:

```
FROM python:3.9
```

This line of code says that you're starting from the Python 3.9 base image.

```
COPY *.py setup.cfg LICENSE README.md requirements.txt /app/
WORKDIR /app
```

This command says to copy all of the application's files to a folder inside the container named `/app` and make it the current working directory.

```
RUN pip install -r requirements.txt
RUN python setup.py install
```

These two lines of code run the Python commands to install the libraries required by the app. When that step is complete, build and install the app inside the container.

```
EXPOSE 8000
CMD [ "/usr/local/bin/my-application" ]
```

This command tells Docker what executable should run when the container starts and that the container will listen for network connections on port 8000.

Pro tip: To build this image from the Dockerfile, use the command: `docker build`. If the build is successful, Docker outputs the ID of the new image, which you can then use to start a container. Refer to the Dockerfile reference for a full list of commands that can appear in a Dockerfile.

Image names, tags, and IDs

You use tags and IDs to identify and reference Docker images. Their unique names provide a way to differentiate between specific versions of Docker images. The ID is a random string of numbers and letters, which most of the time are way too complicated to remember. But there's good news! You can assign any number of tags to the image, in addition to the ID. Tags are alphanumeric labels that help users find the correct image. Most images are tagged with the author's Github username, the name of the application, and a version number.

Pro tip: Tag the most recent version of an image with `latest` in addition to a version number. This makes it easy for people to find the current version of your application.

Let's look at an example:

```
csmith/my-docker-image:1.0
csmith/my-docker-image:latest
sha256:abc123def456
```

`csmith` is the name of the author, `my-docker-image` is the image name, `1.0` is the version number (and it's the latest version), and `sha256:abc123def456` represents the image ID.

How to manage images

A great thing about Docker is that it caches images on a disk. Therefore, you don't need to go grab them or rebuild them every time you need them. This saves you so much time! Some of the Docker CLI (command line interface) commands you can use include:

- `docker image ls` – This command lists the images cached locally.
- `docker image tag` – This command applies tags to a local image.
- `docker image pull` – This command fetches an image from a remote repository.
- `docker image push` – This command sends a local image to a remote repository.
- `docker image rm` – This command removes an image from the cache.
- `docker image prune` – This command removes all unused images to reclaim disk space.

Key takeaways

Docker images—including tags and IDs—are essential for programmers to package, distribute, and deploy applications more efficiently, reducing issues and improving the stages of the software workflow. Remember, in order to have a Docker image, you must have a Dockerfile. These components work hand-in-hand; you can't have one without the other.

Using multiple containers

Imagine you are developing a web-based platform that allows users to browse products, add items to their cart, pay for items, and ship items to different addresses. This application requires multiple components to execute properly because it relies on a number of microservices. The idea behind microservices is to take a large application and break it up into smaller, more tangible, independent parts of the application that are self-contained. This allows for each part of the application to be better maintained. Because these microservices are independent of each other, you use multiple containers to test the entirety of the application to ensure everything runs smoothly. It's no surprise that in the programming world, programmers and developers work with multiple containers at a time.

In this reading, you will learn more about the use of multiple containers, commands for working with multiple containers, how related services find each other, and how to install Docker Compose and view an example.

Starting multiple containers

To start multiple containers, you need to run multiple `docker run` commands. A `docker run` command creates a container and starts it. Let's look at an example of how to create and start two containers that work together once they find each other by name.

As a programmer, you've been asked to set up a WordPress blog. You know WordPress requires a database to store its content. You create and start two containers, `wordpress` and `db`, using the following command:

```
$ docker run -d --name db --restart always \
    -v db_data:/var/lib/mysql -p 3306 -p 33060 \
    -e MYSQL_ROOT_PASSWORD=somewordpress \
    -e MYSQL_DATABASE=wordpress \
    -e MYSQL_USER=wordpress \
```

```
    -e MYSQL_PASSWORD=wordpress \
    mariadb:10
```
This command starts the `mariadb` database, determines a storage volume, and sets the initial password for the WordPress user. It declares two network ports open to other containers, but it is not shown on the host machine.
Now, start the WordPress container using the following command:
```
$ docker run -d --name wordpress --restart always \
    -v wp_data:/var/www/html -p 80:80 \
    -e WORDPRESS_DB_HOST=db \
    -e WORDPRESS_DB_USER=wordpress \
    -e WORDPRESS_DB_PASSWORD=wordpress \
    -e WORDPRESS_DB_NAME=wordpress \
    wordpress:latest
```
Note: The environment variable `WORDPRESS_DB_HOST` is set to `db` on the third line. This line of code is needed to refer to another container. Docker provides domain name system (DNS) services that allow containers to find each other by their name.

Networking with multiple containers

Imagine you have several customers using the same application. For security reasons, you have isolated the application and created multiple containers, one for each customer. Docker allows you to create private networks for a container or groups of containers. These private containers are able to discover each other, but no other networks will be able to find the private containers you've started. Let's look at an example: modifying the `wordpress` and `db` containers by putting them on a private network.
First, stop and delete both containers:
```
$ docker stop wordpress && docker rm wordpress
$ docker stop db && docker rm db
```
Then, create a private network for both containers to use:
```
$ docker network create myblog
0f6abeb9d85a7063298cd70082ac5e5a2f0d1624bae06619fd14dbaa0942b0e2
```
Once the containers are on private networks, restart them with the additional option `-network myblog`. This appears on the second to last line for both container commands.
```
$ docker run -d --name db --restart always \
    -v db_data:/var/lib/mysql -p 3306 -p 33060 \
    -e MYSQL_ROOT_PASSWORD=somewordpress \
    -e MYSQL_DATABASE=wordpress \
    -e MYSQL_USER=wordpress \
    -e MYSQL_PASSWORD=wordpress \
    --network myblog \
    mariadb:10
$ docker run -d --name wordpress --restart always \
    -v wp_data:/var/www/html -p 80:80 \
    -e WORDPRESS_DB_HOST=db \
    -e WORDPRESS_DB_USER=wordpress \
```

```
    -e WORDPRESS_DB_PASSWORD=wordpress \
    -e WORDPRESS_DB_NAME=wordpress \
    --network myblog \
    wordpress:latest
```

It's good practice to verify that containers on other networks can't access the private networks you created. To check this, start a new container and attempt to find the private containers you created.

```
$ docker run -it debian:latest
root@7240f1e3ddab:/# ping db.myblog
ping: db.myblog: Name or service not known
```

Docker Compose

Docker Compose is an optional tool, provided by Docker, that makes using multiple containers easy. In most instances, Docker Compose is automatically installed during the installation process of Docker Desktop. If not, follow the instructions in Scenario two: Install the Compose plugin to install Docker Compose on your platform.

Docker Compose allows you to define a multiple-container setup in a single YAML format, called a Compose file. (YAML is a format for configuration files that's designed to be both human- and computer-readable.) The Compose file communicates with Docker and identifies the containers you need and how you should configure them. The containers in a Compose file are called services. Let's look at how you can use Compose to recreate the private networks from the `wordpress` and `db` example. Run the following on your machine.

1. Create an empty folder and save the file below as `docker-compose.yml`.

```
version: '3.3'
services:
  db:
    image: mariadb:10
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
    networks:
      - myblog
    expose:
      - 3306
      - 33060
  wordpress:
    image: wordpress:latest
    volumes:
```

```
      - wp_data:/var/www/html
    ports:
      - 80:80
    networks:
      - myblog
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
  wp_data:
networks:
  myblog:
```

1. Run the command `docker compose up`. This pulls up the images, creates two empty data volumes, and starts both services. The output from both services will intermingle on your screen.

The Compose file grants you control over how each service is configured, including:

- Choosing the image
- Setting environment variables
- Mounting storage volumes
- Exposing network ports

Pro tip: You can also express any option you pass to the `docker run` command as YAML in a Compose file.
A helpful third-party tool—that's a fan favorite of programmers—that simplifies the process of converting existing Docker run commands into Docker Compose configurations is called Composerize. Refer to Composerize for additional information. You can test the above Docker command in the Composerize textbox. This command defines a `db` service similar to the one presented above. Remember, Composerize is just a tool, and unfortunately sometimes tools come and go. It's best to understand and practice the process of converting an existing Docker run command into a Docker Compose configuration without the help of tools.
For additional information about the options you can put into a Compose file, view the Compose file overview documentation.

Additional Compose commands

Compose has additional commands when working with a single or multiple containers. Let's look at some examples:

- `docker compose pull`: This fetches the latest image for each service.
- `docker compose up`: This creates the containers and starts the service.
- `docker compose down`: This stops the service and deletes the container.

- `docker compose logs`: This displays the console logs from the container.

Key takeaways

Using multiple containers enables the adoption of a microservice architecture for your application. Separating a large application into smaller, independent parts allows for a more manageable approach to building, fixing, maintaining, and deploying each part of the application.

Mark as completed

Like

Dislike

Report an issue

https://www.composerize.com

Docker and GCP

Docker and Google Cloud Platform (GCP) are two types of technologies that complement each other, allowing programmers to build, deploy, and manage containerized applications in the cloud. In this reading, you will learn more about GCP, how to run Docker containers in GCP, and how to use Cloud Run.

Google Cloud Platform

GCP is a composition of all the cloud services provided by Google. These include:

- Virtual machines
- Containers
- Computing
- Hosting
- Storage
- Databases
- Tools
- Identity management

GCP is widely used by businesses, startup companies, developers, and organizations of all sizes across a variety of industries to help their users go digital.

How to run Docker containers in GCP

You can run containers two ways in the cloud using GCP. The first way is to start a virtual machine with Docker installed on it. Use the docker run command to create a container and start it. This is the same process for running Docker on any other host machine.

The second way is to use a service called Cloud Run. This serverless platform is managed by Google and allows you to launch containers without worrying about managing the underlying

infrastructure. Cloud Run is simple and automated, and it's designed to allow programmers to be more productive and move quickly.
An advantage of Cloud Run is that it allows you to deploy code written in any programming language if you can put the code into a container.

Use Cloud Run to deploy containers in GCP

Before you begin, sign into your Google account, or if you do not have one, create an account.

1. Open Cloud Run.
2. Click **Create service** to display the form.

In the form,

1. Select **Deploy one revision from an existing container image**.
2. Below the **Container image URL** text box, select **Test with a sample container**.
3. From the **Region** drop-down menu, select the region in which you want the service located.
4. Below **Authentication**, select **Allow unauthenticated invocations**.
5. Click **Create** to deploy the sample container image to Cloud Run and wait for the deployment to finish.

3. Select the displayed URL link to run the container.

Pro tip: Cloud Run helps keep costs down by only charging you for central processing unit (CPU) time while the container is running. It's unlike running Docker on a virtual machine, for which you must keep the virtual machine on at all times—running up your bill.

Key takeaways

GCP supports Docker containers and provides services to support containerized applications. Integrating GCP and Docker allows developers and programmers to build, deploy, and run containers easily while being able to focus on the application logic.

Build artifact testing

No matter what code you write, you'll need to test it. You want to create a product that is free of errors and bugs. Testing build artifacts and troubleshooting within your tests are great ways to ensure the quality of your work.
In this reading, you will learn more about different types of build artifacts, how to test a Docker container, and how to troubleshoot any issues along the way.

Build artifacts

Build artifacts are items that you create during the build process. Your main artifact is your Docker container, if you're working within a Dockerized application. All other items that you generate during the Docker image build process are also considered build artifacts. Some examples include:

- Libraries
- Documentation
- Static files
- Configuration files
- Scripts

Build artifacts in Docker

Build artifacts in Docker play a crucial role in the software development and deployment lifecycle. No matter what you create with code, you need to test it. You must test your code before deployment to ensure that you catch and correct all issues, defects, and errors. This is true whether your code is built as a Docker container or built the more “classic” way. The process to execute the testing varies based on the application and the programming language it’s written in.

Pro tip: It’s important to check that Docker built the container itself correctly if you are testing your code with a containerized application.

There are several types of software testing that you can execute with Docker containers:

- Unit tests: These are small, granular tests written by the developer to test individual functions in the code. In Docker, unit tests are run directly on your codebase before the Docker image is built, ensuring the code is working as expected before being packaged.
- Integration tests: These refer to testing an application or microservice in conjunction with the other services on which it relies. In a Dockerized environment, integration tests are run after the docker image is built and the container is running, testing how different components operate together inside the Docker container.
- End-to-end (E2E) tests: This type of testing simulates the behavior of a real user (e.g., by opening the browser and navigating through several pages). E2E tests are run against the fully deployed docker container, checking that the entire application stack with its various components and services functions correctly as a whole.
- Performance tests: This type of testing identifies bottlenecks. Performance tests are run against the fully deployed Docker container and test various stresses and loads to ensure the application performs at expectations.

Docker makes it easy to set up and tear down tests in a repeatable and predictable way. Testing Docker containers ensures the reliability, stability, and quality of the application running within them. By testing containers, you can discover bugs and compatibility and performance issues to ensure your application functions as intended.

How to test a Docker container

Automated testing often requires supplying configuration files, data files, and test tools to the application you want to test, which unfortunately increases the size of your container. Instead, you can build a container just for testing, using your output artifact as a base image. Let’s take a look at an example:

Let’s say a Python application uses pytest as a unit testing framework and Sphix to generate documentation. You can reuse your application container and build a new image that includes the tools on top.

```
FROM myapp:latest
RUN pip install pytest pydoc
WORKDIR /opt/myapp
CMD pytest .
```

This part of the code shows that you have a container that has both the application and the test framework in it. Now it’s time for you to run the test according to the framework you chose:

```
docker run -it myapp:test
```

You can mount data files for input or configuration as a volume when you create your test container:

```
docker run -it -v ./testdata:/data myapp:test
```

What should you do if your test fails? Hopefully it won't, but if it does, don't worry! You can troubleshoot it. First, open the shell inside the failed container and see if you can identify the problem. If the container is still running, use the docker exec command and the container ID:

```
docker exec -it c47da2b409a1 /bin/sh
```

If you get an error running the above command, that means the container exited. You can restart the container and then try again:

```
docker start c47da2b409a1
docker exec -it c47da2b409a1 /bin/sh
```

Sometimes the container is built with automatic health checks, and Docker might terminate the container before you can investigate the issue. If this happens, you can disable the health checks in your test container by adding the command `HEALTHCHECK NONE` to the Dockerfile.

If you run into this type of troubleshooting often, there are tools you can add to the Dockerfile so they're always available. A couple of examples include:

- jq: This is for examining JSON files.
- curl, httpie, netcat: These are for testing network services.

If you want to add these tools to your test container, add it using the line below in bold:

```
FROM myapp:latest
RUN apt update && apt install -y jq curl netcat
RUN pip install pytest pydoc
WORKDIR /opt/myapp
CMD pytest .
```

Pro tip: You can never have too many automated tests. At a minimum, a good test suite will include both unit tests and integration tests.

Key takeaways

Running tests for your build artifacts and Docker containers helps ensure the reliability, stability, and quality of your work. You can never run too many tests. Tests are designed to catch bugs, identify compatibility issues, and find performance problems to assist in ensuring the application runs as intended.

Kubernetes principles

Kubernetes is an open-sourced container orchestration platform that automates the deployment, scaling, and management of containerized applications. Kubernetes provides developers with a framework to easily run distributed systems. Kubernetes also provides developers choice and flexibility when building platforms.

In this reading, you will learn more about the principles and design philosophy behind Kubernetes, and you'll learn how declarative configuration enables the desired state reconciliation model. In addition, you'll learn about the key components of the control plane.

Kubernetes principles

Kubernetes—a cloud-native application—follows principles to ensure the containerized application runs properly. These principles take into consideration build time and run time. A container is self-contained, relying only on the Linux kernel. Once the container is built, then additional libraries

can be added. In addition, containerized applications are not meant to change to different environments after they are built.
In terms of run time, each container needs to implement APIs to help the platform manage the application in the most efficient way possible. All APIs must be public, as there should be no hidden or private APIs. In addition, APIs should be declarative, meaning the programmer should be able to communicate their desired end result, allowing Kubernetes to find and implement a solution. Support is available to developers, if needed, to run applications in Kubernetes. Workloads are portable, and the control plane is able to transfer a workload to another node without disrupting the overall program.

Declarative configuration

Declarative configuration is an approach that is commonly used in Kubernetes to achieve a desired state of an application. In this approach, developers specify the desired state, but they do not explicitly define how to achieve or reach the desired state. The approach is more focused on what the desired state should be. The system will determine the most efficient and reliable way to achieve the desired state. These configuration assets are stored in a revision control system and track changes over time.
To use declarative configuration in Kubernetes, create a manifest that describes the desired state of an application. Then, the control plane will determine how to direct nodes in the cluster to achieve the desired state.

The control plane

The Kubernetes control plane is responsible for making decisions about the entire cluster and desired state and for ensuring the cluster's components work together. Components of the control plane include:

- etcd
- API server
- Scheduler
- Controller manager
- Cloud controller manager

etcd is used as Kubernetes backing store for all cluster data as a distributed database. This key-value store is highly available and designed to run on multiple nodes.
The Kubernetes **API server** acts as the front-end for developers and other components interacting with the cluster. It is responsible for ensuring requests to the API are properly authenticated and authorized.
The **scheduler** is a component of the control plane where pods are assigned to run on particular nodes in the cluster.
The **control manager** hosts multiple Kubernetes controllers. Each controller continuously monitors the current state of the cluster and works towards achieving the desired state.
The **cloud controller manager** is a control plane component that embeds cloud-specific control logic. It acts as the interface between Kubernetes and a specific cloud provider, managing the cloud's resources.

Key takeaways

Kubernetes is a portable and extensible platform to assist developers with containerized applications. Kubernetes core principles and key components support developers with starting, stopping, storing, building, and managing containers.

Installing Kubernetes

There are multiple ways to set up and run a Kubernetes cluster. Because Kubernetes acts as a set of containers that manages other containers, Kubenetes is not something you download. You decide on the installation type you need based on your programming requirements.
In this reading, you will learn more about a Kubernetes cluster, including how to set it up and run it using Docker Desktop, and you'll receive step-by-step instructions along the way.

Download Docker

You should already have Docker up and running on your computer or laptop. If you don't, download and install Docker according to your operating system:

- Windows: Install Docker desktop on Windows | Docker documentation
- macOS: Install Docker desktop on Mac | Docker documentation
- Linux: Install Docker desktop on Linux | Docker documentation

Enable Kubernetes

After Docker is installed on your machine, follow the instructions below to run Kubernetes in Docker Desktop.

1. From the Docker Dashboard, select **Settings**.
2. Select **Kubernetes** from the left sidebar.
3. Select the checkbox next to **Enable Kubernetes**.
4. Select **Apply & Restart** to save the settings.
5. Select **Install** to complete the installation process.

The Kubernetes server runs as containers and installs the `/usr/local/bin/kubect1` command on your machine.
And that's it! Setting up Kubernetes on Docker Desktop is typically the most common way that developers use Kubernetes since Docker Desktop has built-in support for it. Below are additional tools that you can use to start a Kubernetes cluster.

- kind
- k3s
- microk8s
- minikube

Key takeaways

Kubernetes is not a replacement for Docker, but rather a tool that developers use while working in Docker. It can run and manage Docker containers, allowing developers to deploy, scale, and manage containerized applications across clusters.

Pods

In Kubernetes, a **container** is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and system tools. Containers are isolated from each other and bundle their own software, libraries, and configuration files, but they share the operating system kernel with other containers. They are designed to be easily portable across different environments, which makes them ideal for consistent deployment across different platforms.

In the context of Kubernetes, containers are the smallest units of deployment that are scheduled and managed. They are encapsulated within **Pods**, which are the fundamental deployment units in a Kubernetes cluster. A Pod can contain one or more containers that need to run together on the same host and share the same network and storage resources, allowing them to communicate with each other using localhost.

Pods serve as an abstraction layer, allowing Kubernetes to schedule and orchestrate containers effectively. When a deployment requires multiple containers to work together on the same node, a Pod is created to ensure they are co-located and can communicate efficiently. This simplifies the deployment and management of containerised applications, making it easier to scale, monitor, and update as needed.

Also note that Pods in Kubernetes are considered to be ephemeral; they can be created, terminated, and replaced dynamically based on the desired state and resource availability in the cluster. As a result, Kubernetes ensures that the desired number of Pods are always running, enabling high availability and fault tolerance for containerised applications.

Pods serve together as a logical host that encapsulates one or more tightly coupled containers within a shared network and storage context. This provides a way to group containers that need to work closely together, allowing them to share the same resources and interact with each other as if they were running on the same physical or virtual machine.

When designing a Pod, consider aspects like resource requests and limits, handling graceful shutdowns, logging and monitoring, and appropriate container images.

Pods as logical host

A Pod can run one or more closely-related containers which share the same network and storage context. This shared context is much like what you would find on a physical or virtual machine, hence the term "logical host."

The key points to understand about a Pod as a logical host are:

- **Tightly coupled containers:** When multiple containers within a Pod are considered tightly coupled, it means they have a strong interdependency and need to communicate with each other over localhost. This allows them to exchange data and information efficiently without the need for complex networking configurations.
- **Shared network namespace:** Containers within the same Pod share the same network namespace. This implies that they have the same IP address and port space, making it easier for them to communicate using standard inter-process communication mechanisms.
- **Shared storage context:** Pods also share the same storage context, which means they can access the same volumes or storage resources. This facilitates data sharing among the containers within the Pod, further enhancing their collaboration.

- **Co-location and co-scheduling:** Kubernetes ensures that all containers within a Pod are scheduled and co-located on the same node. This co-scheduling ensures that the containers can efficiently communicate with each other within the same network and storage context.
- **Ephemeral nature:** Like individual containers, Pods are considered to be ephemeral and can be easily created, terminated, or replaced based on scaling requirements or resource constraints. However, all containers within the Pod are treated as a single unit in terms of scheduling and lifecycle management.

Pods in action

Let's say you're a software developer in charge of a web application that includes a main web server and a helper component for log processing. The web server interacts with the log processor to handle, analyze, and store log data in real-time. These two components need to be tightly integrated and should communicate with each other efficiently.

In this scenario, you would use a Kubernetes Pod to encapsulate both the web server and the log processor containers. Since both containers exist within the same Pod, they share the same network namespace (they can communicate via localhost) and they can share the same storage volumes. This allows the web server to generate logs and the log processor to access and process these logs efficiently.

The Pod ensures that both the web server and log processor are scheduled on the same node (co-location) and managed as a single entity. If the Pod needs to be rescheduled or if it fails, both containers would be dealt with together, maintaining their coupled relationship. The Pod abstracts away the details of the host machine and the underlying infrastructure, allowing you to focus on managing your application.

This setup, where multiple related containers are grouped in a Pod, is known as a multi-container Pod. You'll explore single– and multiple-container pods in more detail below; for now, just know that multiple containers are an ideal way to manage and deploy tightly coupled application components.

Advantages of Pods

From the above example, you can see that Pods offer a number of advantages in facilitating co-location of containers, enabling data sharing, and simplifying inter-container communication:

- **Facilitating co-location:** Pods allow multiple containers to be co-located on the same host machine. This is particularly useful for closely related components that need to work together, such as an application and its helper components (like sidecar containers that handle logging or monitoring). By running these components in the same Pod, they can be scheduled onto the same machine and managed as a single entity.
- **Enabling data sharing:** Containers within a Pod share the same network namespace, which means they share an IP address and port space. They can communicate with each other using localhost and they can also share data through shared volumes. Shared volumes in a Pod enable data to be easily exchanged between containers, and also allow data to persist beyond the life of a single container, which can be useful for applications that require persistent data storage.
- **Simplifying inter-container communication:** The shared network namespace also simplifies inter-container communication. Because all containers in a Pod share a network stack, they can communicate with each other on localhost, without the need for inter-process

communication (IPC) or shared file systems. This simplifies the development of distributed systems, where components often need to communicate with each other.

Single container vs. multiple containers

The difference between single-container and multi-container Pods lies in the number of containers they host.

As the name suggests, **single-container Pods** contain only one container. This container typically represents the primary application or service that the Pod is meant to run. Single-container Pods are straightforward and are commonly used when you have a simple application that requires no additional sidecar containers or closely related helper components. They are also suitable for running standalone applications that do not need to communicate with other containers within the same Pod.

Multi-container Pods, on the other hand, contain multiple containers that are co-located and share the same resources and network namespace. These containers are meant to work together and complement each other's functionalities. Multi-container Pods are appropriate in various scenarios:

- **Sidecar pattern:** The sidecar pattern is a common use case for multi-container Pods. In this pattern, the main container represents the primary application, while additional sidecar containers provide supporting features like logging, monitoring, or authentication. The sidecar containers enhance and extend the capabilities of the main application without modifying its code.
- **Proxy pattern:** Multi-container Pods can use a proxy container that acts as an intermediary between the main application container and the external world. The proxy container handles tasks like load balancing, caching, or SSL termination, offloading these responsibilities from the main application container.
- **Adapter pattern:** Multi-container Pods can employ an adapter container that performs data format conversions or protocol translations. This allows the main container to focus solely on its core functionality without worrying about the intricacies of data exchange formats.
- **Shared data and dependencies:** Containers within a multi-container Pod can share volumes and communicate over localhost, making them suitable for applications that require data sharing or have interdependent components.

Use a single-container Pod when you have a simple application that does not require additional containers, or when you want to isolate different applications or services for easier management and scaling.

Use multi-container Pods when you have closely related components that need to work together, such as those following the sidecar pattern. This is useful for tasks like logging, monitoring, or enhancing the main application's capabilities without modifying its code. Multi-container Pods are also appropriate for scenarios where multiple containers need to share data or dependencies efficiently.

Key terms

Here are some key terms to be familiar with as you're working with Kubernetes.

- **Pod lifecycle:** Pods have specific lifecycle phases, starting from "Pending" when they are being scheduled, to "Running" when all containers are up and running, "Succeeded" when all

containers successfully terminate, and "Failed" if any container within the Pod fails to run. Pods can also be in a "ContainerCreating" state if one or more containers are being created.

- **Pod templates:** Pod templates define the specification for creating new Pods. They are used in higher-level controllers like ReplicaSets, Deployments, and StatefulSets to ensure the desired state of the Pods.
- **Pod affinity and anti-affinity:** Pod affinity and anti-affinity rules define the scheduling preferences and restrictions for Pods. They allow you to influence the co-location or separation of Pods based on labels and other attributes.
- **Pod autoscaling:** Kubernetes provides Horizontal Pod Autoscaler (HPA) functionality that automatically scales the number of replicas (Pods) based on resource usage or custom metrics.
- **Pod security policies:** Pod security policies are used to control the security-related aspects of Pods, such as their access to certain host resources, usage of privileged containers, and more.
- **Init containers:** Init containers are additional containers that run and complete before the main application containers start. They are useful for performing initialization tasks, such as database schema setup or preloading data.
- **Pod eviction and disruption:** Pods can be evicted from nodes due to resource constraints or node failures. Understanding Pod eviction behavior is important for managing application reliability.
- **Pod health probes:** Kubernetes supports different types of health probes (liveness, readiness, and startup probes) to check the health of containers within a Pod. These probes help Kubernetes decide whether a Pod is considered healthy and ready to receive traffic.
- **Taints and tolerations:** Taints are applied to nodes to repel Pods, while tolerations are set on Pods to allow them to be scheduled on tainted nodes.
- **Pod DNS:** Pods are assigned a unique hostname and IP address. They can communicate with each other using their hostname or service names. Kubernetes provides internal DNS resolution for easy communication between Pods.
- **Pod annotations and labels:** Annotations and labels can be attached to Pods to provide metadata or facilitate Pod selection for various purposes like monitoring, logging, or routing.

Pods and Python

To manage Kubernetes pods using Python, you can use the kubernetes library. Here is some example code of how to create, read, update, and delete a Pod using Python.

```
from kubernetes import client, config

# Load the Kubernetes configuration from the default location
config.load_kube_config()

# Alternatively, you can load configuration from a specific file
# config.load_kube_config(config_file="path/to/config")

# Initialize the Kubernetes client
v1 = client.CoreV1Api()
```

```
# Define the Pod details
pod_name = "example-pod"
container_name = "example-container"
image_name = "nginx:latest"
port = 80

# Create a Pod
def create_pod(namespace, name, container_name, image, port):
    container = client.V1Container(
        name=container_name,
        image=image,
        ports=[client.V1ContainerPort(container_port=port)],
    )

    pod_spec = client.V1PodSpec(containers=[container])
    pod_template = client.V1PodTemplateSpec(
        metadata=client.V1ObjectMeta(labels={"app": name}), spec=pod_spec
    )

    pod = client.V1Pod(
        api_version="v1",
        kind="Pod",
        metadata=client.V1ObjectMeta(name=name),
        spec=pod_spec,
    )

    try:
        response = v1.create_namespaced_pod(namespace, pod)
        print("Pod created successfully.")
        return response
    except Exception as e:
        print("Error creating Pod:", e)


# Read a Pod
def get_pod(namespace, name):
    try:
        response = v1.read_namespaced_pod(name, namespace)
        print("Pod details:", response)
```

```
    except Exception as e:
        print("Error getting Pod:", e)


# Update a Pod (e.g., change the container image)
def update_pod(namespace, name, image):
    try:
        response = v1.read_namespaced_pod(name, namespace)
        response.spec.containers[0].image = image

        updated_pod = v1.replace_namespaced_pod(name, namespace, response)
        print("Pod updated successfully.")
        return updated_pod
    except Exception as e:
        print("Error updating Pod:", e)


# Delete a Pod
def delete_pod(namespace, name):
    try:
        response = v1.delete_namespaced_pod(name, namespace)
        print("Pod deleted successfully.")
    except Exception as e:
        print("Error deleting Pod:", e)


if __name__ == "__main__":
    namespace = "default"

    # Create a Pod
    create_pod(namespace, pod_name, container_name, image_name, port)

    # Read a Pod
    get_pod(namespace, pod_name)

    # Update a Pod
    new_image_name = "nginx:1.19"
    update_pod(namespace, pod_name, new_image_name)

    # Read the updated Pod
```

```
    get_pod(namespace, pod_name)

    # Delete the Pod
    delete_pod(namespace, pod_name)
```

Key Takeaways

- Pods are the fundamental deployment units in a Kubernetes cluster.
- A Pod can contain one or more containers that need to run together on the same host and share the same network and storage resources, allowing them to communicate with each other using localhost.
- Pods serve as an abstraction layer, allowing Kubernetes to schedule and orchestrate containers effectively.
- Use a single-container Pod when you have a simple application that does not require additional containers, or when you want to isolate different applications or services for easier management and scaling.
- Use multi-container Pods when you have closely related components that need to work together, such as those following the sidecar pattern.

Resources for more information

Kubernetes documentation: Pods
Official Python client library for Kubernetes

Services

The challenge

Imagine you're developing a Python-based web application deployed in a Kubernetes cluster. This application is composed of multiple components such as a web server, a caching layer, and a database, each running in separate Pods. These components need to communicate with each other to function properly, but there's a wrinkle: Pods have ephemeral life cycles and their IP addresses can change dynamically due to reasons like scaling, rescheduling, or node failures. But this isn't the only challenge you're facing!

- Imagine that your web server, for instance, was directly communicating with the database Pod using its Pod IP address. The server would need constant updates whenever this IP changes—a manual and error-prone process.
- Furthermore, consider if your caching layer is designed to handle high traffic and hence is replicated into multiple Pods for load balancing. Now, your web server needs to distribute requests among all these cache Pods. Maintaining and managing direct communication with every single cache Pod by their individual IP addresses would be a daunting task, and an inefficient use of resources.

- Plus, there's the issue of service discovery. Say your web server needs to connect with a new analytics service you've just launched. It would require an updated list of all the active Pods and their IP addresses for this service—a difficult and dynamic challenge.

What is a Python developer to do in this scenario?

Services to the rescue

Fortunately, services come to the rescue in these scenarios. **Services offer an abstraction layer over Pods**. For starters, they provide a stable virtual IP and a DNS name for each set of related Pods (like your caching layer or database), and these remain constant regardless of the changes in the underlying Pods. So, your web server only needs to know this Service IP or DNS name, saving it from the ordeal of tracking and updating numerous changing Pod IPs.

Furthermore, Services automatically set up load balancing. When your web server sends a request to the caching layer's Service, Kubernetes ensures the request is distributed evenly among all available caching Pods. This automatic load balancing allows for efficient use of resources and improved performance.

In essence, a Service acts like a stable intermediary within the cluster. Instead of applications (like a front-end interface) directly addressing specific Pods, they communicate with the Service. The Service then ensures the request reaches the right backend Pods. This layer of abstraction streamlines intra-cluster communication, making the system more resilient and easier to manage—even as the underlying Pod configurations change dynamically.

Types of Services

Let's imagine that, with the basic challenges addressed, you've expanded your Python web application and it now includes a user interface, an API layer, a database, and an external third-party service. Different components of your application have different networking needs, and Kubernetes services, with their various types, can cater to these needs effectively.

First, you have the **ClusterIP** service. This is the default type and serves as the go-to choice when you need to enable communication between components within the cluster. For example, your API layer and your database might need to communicate frequently, but these exchanges are internal to your application. A ClusterIP service would give you a stable, cluster-internal IP address to facilitate this communication.

Next, you may want to expose your API layer to external clients. You could use a **NodePort** service for this purpose. It makes your API layer available on a specific port across all nodes in your cluster. With this setup, anyone with access to your node's IP address can communicate with your API layer by contacting the specified NodePort.

However, a NodePort might not be enough if your application is hosted in a cloud environment and you need to handle large volumes of incoming traffic. A **LoadBalancer** service might be a better choice in this scenario. It exposes your service using your cloud provider's load balancer, distributing incoming traffic across your nodes, which is ideal for components like your user interface that might experience heavy traffic.

Finally, you might be integrating an external third-party service into your application. Rather than expose this service directly within the cluster, you can use an **ExternalName** service. This gives you an alias for the external service that you can reference using a Kubernetes DNS name.

In summary, Kubernetes provides different types of services tailored to various networking requirements:

- **ClusterIP:** Facilitates internal communication within the cluster
- **NodePort:** Enables external access to services at a static port across nodes
- **LoadBalancer:** Provides external access with load balancing, often used with cloud provider load balancers
- **ExternalName:** Serves as an alias for an external service, represented with a Kubernetes DNS name

Other features

So far we've just scratched the surface of services. There are several features that extend the capabilities of services and can be employed to address specific use cases within your application's networking requirements.

- **Service discovery with DNS:** As your application grows, new services are added and existing ones might move around as they are scheduled onto different nodes. Kubernetes has a built-in DNS service to automatically assign domain names to services. For instance, your web server could reach the database simply by using its service name (e.g., `database-service.default.svc.cluster.local`), rather than hard-coding IP addresses.
- **Headless services:** Let's say you want to implement a distributed database that requires direct peer-to-peer communication. You can use a headless service for this. Unlike a standard service, a headless service doesn't provide load-balancing or a stable IP, but instead returns the IP addresses of its associated pods, enabling direct pod-to-pod communication.
- **Service topology:** Suppose your application is deployed in a multi-region environment, and you want to minimize latency by ensuring that requests are served by the nearest pods. Service topology comes to the rescue, allowing you to preferentially route traffic based on the network topology, such as the node, zone, or region.
- **External Traffic Policy:** If you want to preserve the client source IP for requests coming into your web server, you can set the External Traffic Policy to "Local". This routes the traffic directly to the Pods running on the node, bypassing the usual load balancing and ensuring the original client IP is preserved.
- **Session affinity (sticky sessions):** Suppose users log into your application, and their session data is stored locally on the server pod handling the request. To maintain this session data, you could enable session affinity on your service, so that all requests from a specific user are directed to the same pod.
- **Service slicing:** Imagine you're rolling out a new feature and want to test it with a subset of your users. Service Slicing enables you to direct traffic to different sets of pods based on custom labels, providing granular control over traffic routing for A/B testing or canary releases.
- **Connecting external databases:** Perhaps your application relies on an external database hosted outside the Kubernetes cluster. You can create a Service with the type `ExternalName` to reference this database. This allows your application to access the database using a DNS name without needing to know its IP address, providing a level of indirection and increasing the flexibility of your application configuration.

Services and Python

Here's an example of some Python code that uses the Kubernetes Python client to create, list, and delete Kubernetes Services in a given namespace.

```
from kubernetes import client, config

def create_service(api_instance, namespace, service_name, target_port,
port, service_type):
    # Define the Service manifest based on the chosen Service type
    service_manifest = {
        "apiVersion": "v1",
        "kind": "Service",
        "metadata": {"name": service_name},
        "spec": {
            "selector": {"app": "your-app-label"},
            "ports": [
                {"protocol": "TCP", "port": port, "targetPort":
target_port}
            ]
        }
    }

    if service_type == "ClusterIP":
        # No additional changes required for ClusterIP, it is the default
type
        pass
    elif service_type == "NodePort":
        # Set the NodePort field to expose the service on a specific port
on each node
        service_manifest["spec"]["type"] = "NodePort"
    elif service_type == "LoadBalancer":
        # Set the LoadBalancer type to get an external load balancer
provisioned
        service_manifest["spec"]["type"] = "LoadBalancer"
    elif service_type == "ExternalName":
        # Set the ExternalName type to create an alias for an external
service
        service_manifest["spec"]["type"] = "ExternalName"
        # Set the externalName field to the DNS name of the external
service
```

```
        service_manifest["spec"]["externalName"] =
"my-external-service.example.com"

    api_response = api_instance.create_namespaced_service(
        body=service_manifest,
        namespace=namespace,
    )
    print(f"Service '{service_name}' created with type '{service_type}'.
Status: {api_response.status}")


def list_services(api_instance, namespace):
    api_response =
api_instance.list_namespaced_service(namespace=namespace)
    print("Existing Services:")
    for service in api_response.items:
        print(f"Service Name: {service.metadata.name}, Type:
{service.spec.type}")


def delete_service(api_instance, namespace, service_name):
    api_response = api_instance.delete_namespaced_service(
        name=service_name,
        namespace=namespace,
    )
    print(f"Service '{service_name}' deleted. Status:
{api_response.status}")


if __name__ == "__main__":
    # Load Kubernetes configuration (if running in-cluster, this might not
be necessary)
    config.load_kube_config()

    # Create an instance of the Kubernetes API client
    v1 = client.CoreV1Api()

    # Define the namespace where the services will be created
    namespace = "default"
```

```
    # Example: Create a ClusterIP Service
    create_service(v1, namespace, "cluster-ip-service", target_port=8080,
port=80, service_type="ClusterIP")

    # Example: Create a NodePort Service
    create_service(v1, namespace, "node-port-service", target_port=8080,
port=30000, service_type="NodePort")

    # Example: Create a LoadBalancer Service (Note: This requires a cloud
provider supporting LoadBalancer)
    create_service(v1, namespace, "load-balancer-service",
target_port=8080, port=80, service_type="LoadBalancer")

    # Example: Create an ExternalName Service
    create_service(v1, namespace, "external-name-service",
target_port=8080, port=80, service_type="ExternalName")

    # List existing Services
    list_services(v1, namespace)

    # Example: Delete a Service
    delete_service(v1, namespace, "external-name-service")
```

Key Takeaways

- Services offer an abstraction layer over Pods.
- Services provide a solution for consistent and reliable networking among ephemeral Pods.
- For Python developers, and developers in general, Kubernetes Services enable the creation of robust and reliable distributed systems, abstracting away the complexities of dynamic Pod networking.
- Services offer flexibility through different types of load balancing and service discovery mechanisms, such as ClusterIP, NodePort, LoadBalancer, and ExternalName. This allows you to choose the mechanism that best aligns with your application's requirements.

Resources for more information

Official Python client library for Kubernetes
DNS for Services and Pods
Create an External Load Balance
External Name – Kubernetes Networking

Deployment

What are deployments?

Let’s continue the example of the Python-based web application running in a Kubernetes cluster, specifically the web server component of the application. As traffic to your application grows, you'll need to scale the number of web server instances to keep up with demand. Also, to ensure high availability, you want to maintain multiple replicas of the web server so that if one instance fails, others can take over. This is where Kubernetes Deployments come in.

In Kubernetes, a Deployment is like your application's manager. It's responsible for keeping your application up and running smoothly, even under heavy load or during updates. It ensures your application, encapsulated in Pods, always has the desired number of instances—or “replicas”—running.

Think of a Deployment as a blueprint for your application's Pods. It contains a **Pod Template Spec**, defining what each Pod of your application should look like, including the container specifications, labels, and other parameters. The Deployment uses this template to create and update Pods.

A Kubernetes Deployment also manages a **ReplicaSet**, a lower-level resource that makes sure the specified number of identical Pods are always running. The Deployment sets the desired state, such as the number of replicas, and the ReplicaSet ensures that the current state matches the desired state. If a Pod fails or is deleted, the ReplicaSet automatically creates new ones. In other words, Deployments configure ReplicaSets, and thus, they are the recommended way to set up replication.

And by default, deployments support **rolling updates and rollbacks**. If you update your web server's code, for example, you can push the new version with a rolling update, gradually replacing old Pods with new ones without downtime. If something goes wrong, you can use the Deployment to rollback to a previous version.

So, in summary, a Kubernetes Deployment consists of several key components:

- **Desired Pod template:** This is the specification that defines the desired state of the Pods managed by the Deployment. It includes details such as container images, container ports, environment variables, labels, and other configurations.
- **Replicas:** This field specifies the desired number of identical copies of the Pod template that should be running. Kubernetes ensures this number of replicas is maintained, automatically scaling up or down as needed.
- **Update strategy:** This defines how the Deployment handles updates. The default is a rolling update strategy, where Kubernetes performs updates by gradually replacing Pods, keeping the application available throughout the process. This strategy can be further customized with additional parameters.

Powerful features

Deployments not only help maintain high availability and scalability, but they also provide several powerful features:

- **Declarative updates:** With a declarative update, you just specify the desired state of your application and the Deployment ensures that this state is achieved. If there are any differences between the current and desired state, Kubernetes automatically reconciles them.

- **Scaling:** You can easily adjust the number of replicas in your Deployment to handle increased or decreased loads. For example, you might want to scale up during peak traffic times and scale down during off-peak hours.
- **History and revision control:** Deployments keep track of changes made to the desired state, providing you with a revision history. This can be useful for debugging, auditing, and rolling back to specific versions.

A Kubernetes Deployment is typically defined using a YAML file that specifies these components. Here is an example YAML manifest.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 3
  selector:
        matchLabels:
        app: example-app
  template:
        metadata:
        labels:
        app: example-app
        spec:
        containers:
        - name: example-container
        image: example-image:latest
        ports:
        - containerPort: 80
```

This Deployment specifies that it should maintain three replicas of the `example-container` Pod template. The Pods are labeled with `app: example-app`, and the container runs an image tagged as `example-image:latest` on port 80. The default rolling update strategy will be used for any updates to this Deployment.

By utilizing Deployments, you can manage your Python web server's life cycle more efficiently, ensuring its high availability, scalability, and smooth updates.

Deployments and Python

The following Python script uses the Kubernetes Python client to create, list, and delete Kubernetes Services in a given namespace.

```
from kubernetes import client, config
```

```
def create_deployment(api_instance, namespace, deployment_name, image, replicas):
    # Define the Deployment manifest with the desired number of replicas and container image.
    deployment_manifest = {
        "apiVersion": "apps/v1",
        "kind": "Deployment",
        "metadata": {"name": deployment_name},
        "spec": {
            "replicas": replicas,
            "selector": {"matchLabels": {"app": deployment_name}},
            "template": {
                "metadata": {"labels": {"app": deployment_name}},
                "spec": {
                    "containers": [
                        {"name": deployment_name, "image": image, "ports": [{"containerPort": 80}]}
                    ]
                },
            },
        },
    }

    # Create the Deployment using the Kubernetes API.
    api_response = api_instance.create_namespaced_deployment(
        body=deployment_manifest,
        namespace=namespace,
    )
    print(f"Deployment '{deployment_name}' created. Status: {api_response.status}")

def update_deployment_image(api_instance, namespace, deployment_name, new_image):
    # Get the existing Deployment.
    deployment = api_instance.read_namespaced_deployment(deployment_name, namespace)

    # Update the container image in the Deployment.
    deployment.spec.template.spec.containers[0].image = new_image
```

```
    # Patch the Deployment with the updated image.
    api_response = api_instance.patch_namespaced_deployment(
        name=deployment_name,
        namespace=namespace,
        body=deployment
    )
    print(f"Deployment '{deployment_name}' updated. Status:
{api_response.status}")

def delete_deployment(api_instance, namespace, deployment_name):
    # Delete the Deployment using the Kubernetes API.
    api_response = api_instance.delete_namespaced_deployment(
        name=deployment_name,
        namespace=namespace,
        body=client.V1DeleteOptions(
            propagation_policy="Foreground",
            grace_period_seconds=5,
        )
    )
    print(f"Deployment '{deployment_name}' deleted. Status:
{api_response.status}")


if __name__ == "__main__":
    # Load Kubernetes configuration (if running in-cluster, this might not
be necessary)
    config.load_kube_config()

    # Create an instance of the Kubernetes API client for Deployments
    v1 = client.AppsV1Api()

    # Define the namespace where the Deployment will be created
    namespace = "default"

    # Example: Create a new Deployment
    create_deployment(v1, namespace, "example-deployment",
image="nginx:latest", replicas=3)

    # Example: Update the image of the Deployment
```

```
    update_deployment_image(v1, namespace, "example-deployment",
new_image="nginx:1.19.10")

    # Example: Delete the Deployment
    delete_deployment(v1, namespace, "example-deployment")
```

Additional learning points

Beyond the fundamental concepts, you should be aware of a few additional features and best practices related to Kubernetes Deployments.

- **A fresh start:** While the default update strategy is rolling updates, Kubernetes also supports a "Recreate" strategy. In the "Recreate" strategy, all existing Pods are terminated before new Pods are created. This strategy may lead to brief periods of downtime during updates but can be useful in specific scenarios where a clean restart is necessary.
- **Don't get stuck:** Deployments have a progressDeadlineSeconds field, which sets the maximum time (in seconds) allowed for a rolling update to make progress. If progress stalls beyond this duration, the update is considered failed. This field helps prevent deployments from getting stuck in a partially updated state. Likewise, the minReadySeconds field specifies the minimum time Kubernetes should wait after a Pod becomes ready before proceeding with the next update. This can help ensure the new Pods are fully functional and ready to handle traffic before more updates are made.
- **Press pause:** Deployments can be paused and resumed to temporarily halt the progress of rolling updates. This feature is helpful when investigating issues or performing maintenance tasks. Pausing a Deployment prevents further updates until it is explicitly resumed.
- **It's alive!:** Deployments can utilize liveness and readiness probes to enhance the health management of Pods. Liveness probes determine if a Pod is still alive and running correctly, while readiness probes determine if a Pod is ready to accept traffic. These probes help Kubernetes decide whether to consider a Pod as healthy or not during rolling updates and scaling operations.

If you want to learn more about Kubernetes Deployments and their components, you can explore the provided resources below.

Key takeaways

From maintaining the desired state of your applications, managing updates and rollbacks, to ensuring high availability, Deployments provide a set of key features that streamline the deployment and management of containerized applications.

- Deployments are crucial resources that manage and scale containerised applications. They automate the deployment and management of Pods and ReplicaSets.

- Deployments use a declarative approach, ensuring that the application's desired state is maintained across the cluster, providing high availability. In case of Pod or node failures, the Deployment replaces the affected Pods automatically.
- Deployments support rolling updates, allowing for smooth transitions during application updates with no downtime. They also offer the capability to roll back to a previous stable version in case of issues with a new update.
- Deployments have several key components, including the desired Pod template (which defines the desired state of the Pods), the number of replicas (indicating the desired level of availability and scalability), and the update strategy (which defines how updates to the Pod template are handled).

Resources for more information

Kubernetes Deployments: Comprehensive documentation on Kubernetes Deployments, their use cases, and operations.
Managing Resources: A guide on managing Deployments in Kubernetes including rolling updates, scaling and rollback.
Kubernetes ReplicaSets: Detailed explanation on ReplicaSets in Kubernetes, their role in maintaining the desired number of Pods.
Declarative Application Management in Kubernetes: Understanding declarative approach in Kubernetes with configuration files.
Configure Liveness, Readiness and Startup Probes: An in-depth guide on liveness and readiness probes, which enhance the health management of Pods.
Rolling Back a Deployment: Documentation on how to perform rollbacks on a Deployment.

Create a Kubernetes cluster on GCP

Introduction

A Kubernetes cluster is a fundamental construct within Kubernetes. The cluster enables the deployment, coordination, and operation of containerised applications at scale. Instead of one incredibly huge, ridiculously fast server positioned in one place to process requests from all around the world, clusters are lots of smaller servers spread out and coordinated to serve everyone close to where they are.
A cluster is a group of machines grouped to work together, but not necessarily all doing the same tasks. In a Kubernetes cluster, virtual machines (VMs) are coordinated to execute all of the functions needed to process requests, such as serving a web application, running a database, or solving big-data problems. Each cluster consists of at least one cluster control plane machine, a server that manages multiple nodes. You submit all of your work to the control plane, and the control plane distributes the work to the node or nodes where it will run. These worker nodes are virtual machine (VM) instances running the Kubernetes processes necessary to make them part of the cluster. They can be in a single zone or spread out all over the world. Depending on the use case, one node might be used for data processing and another for hosting a web server. Each of these nodes is made up

of pods, which are assigned by the control plane. Each pod is made up of one or more containers that work together to execute necessary functions.
Have you ever been to a restaurant where there is a rolling tray of desserts? Well, the tray is like a cluster, holding all the little plates. The plates are like nodes, each working to hold a different dessert. The desserts are like the pods, each performing a different flavor, or function.
In this reading, you'll learn about the steps to creating a Kubernetes cluster on Google Cloud Platform (GCP), and you'll get some pro tips on the process.

Setup

You have already gone through a lot of the prerequisites to creating a Kubernetes cluster on GCP. As a review, and for when you do this work outside of the course, you'll need to do the following:

- Create a valid GCP account and access to the Google Cloud Console. To open a new GCP account, start at the Google Cloud console start page.
- Create a GCP project where you will deploy your Kubernetes cluster.
- Enable billing for your GCP project to use Google Kubernetes Engine (GKE), as this may involve charges for the resources you use.
- Create a service account for GKE with the necessary permissions to manage resources in your GCP project.
- Install the Google Cloud SDK on your local machine. It provides the necessary tools and commands to interact with GCP resources.
- Install kubectl on your local machine. kubectl is a command-line tool used to interact with Kubernetes clusters.
- (Optional) If you're creating a private Kubernetes cluster, set up a Virtual Private Cloud (VPC) network or use an existing one to define the network boundaries for your cluster.
- Configure firewall rules to allow necessary network traffic to and from your cluster.

Note: If you plan to use private container images, set up a container registry like Google Artifact Registry to store your Docker images. This is entirely optional.
Once you have met these prerequisites, you can proceed to create your Kubernetes cluster using Google Kubernetes Engine through the Google Cloud Console or use the gcloud command-line tool. Let's get started!

Creating a GKE Cluster using Google Cloud Console

As you get started working with Kubernetes clusters on Google Cloud Platform (GCP), the first thing to do is create a cluster to work with. Here are the steps to creating a standard cluster:
1. Log in to Google Cloud Console: Go to https://console.cloud.google.com/ and log in with your Google Cloud account.
2. Open Google Kubernetes Engine (GKE). In the left-hand navigation menu, select **Kubernetes Engine**, and then **Clusters**.
3. Click **Create Cluster** to create a new Kubernetes cluster. By default, this will take you to Autopilot cluster. For these instructions, we are setting up a standard cluster, so click **Switch to Standard Cluster**. For more information on the difference between Standard and Autopilot, see Compare GKE Autopilot and Standard.
4. Configure cluster basics. Enter a unique cluster name for your GKE cluster.

5. Choose a Location type. Zonal is for creating a cluster within a single zone. When selecting this, you will also need to select the zone where your cluster's control plane will run.
Regional is for multi-zone deployment. Deployment across a larger area means higher availability to users. When selecting a regional cluster, you'll also need to choose the region. By default, three zones will be selected within the chosen region, or you can manually select the zones if you wish.
6. Configure the node pool. In the Node pool section, specify the desired node count for the initial number of nodes in the default node pool depending on the needs of your application on your Kubernetes cluster. For production clusters, the recommended minimum is usually three nodes.
The maximum number of nodes will depend on the type of application, the expected amount of traffic, and your budget. A maximum of five to ten nodes is a good start while you get a feel for what's needed. As you configure the node pool, there's a cost estimator on the right side of the screen that estimates how much you will pay per month. You can also look over the GKE pricing for Standard mode. If you don't set a maximum that suits your budget, and demand for your application rises sharply, you could get an expensive wake-up call.
Once you have configured the node pool, you can enable Autoscaling by checking the box for Enable cluster autoscaler. Once enabled, Autoscaler will automatically adjust the number of nodes based on resource utilization up to the maximum number of nodes you set for the Node pool.
Finally, choose the machine type for your nodes. There are four machine families:

- General purpose machines are suitable for most workloads. These machines balance performance with price.
- Compute-optimized machines provide high performance for intensive workloads like artificial intelligence and machine learning, entertainment streaming, and game servers
- Memory-optimized machines offer the highest memory configurations. These machines process large data sets in memory in use cases like Big Data analytics.
- Accelerator-optimized machines are for very demanding workloads like machine learning (ML) training and inference, in which a neural network makes deductions about new data based on what it has already learned.

For more details on choosing machine families or specific types, see Choosing the right machine family and type.
7. Choose any optional configurations needed. Based on your projects' specific requirements, you can expand the Node pool section to configure advanced settings including boot disk size, preemptible nodes, node labels, and node locations.
You can also enable networking and security features based on the data governance laws and the level of security you need to maintain for your data. In the Networking section, you can choose the VPC network and subnetwork where your cluster's nodes will be placed. You can also enable Private cluster mode to hide the cluster's master endpoint from the public internet. For pod-level firewall rules, you can define network tags and network policies.
8. Click the **Create** button to start creating the GKE cluster.
9. Wait for cluster creation. GKE will begin creating your cluster based on your specified configuration. The process may take a few minutes.
10. Access and use your cluster. Once the cluster is successfully created, you can click on the cluster name in the GKE dashboard to view its details and manage the cluster.

Congratulations!

That's it! You have now created a Kubernetes cluster on GCP using GKE with the desired configurations, including the specified node count and other settings. You can now start deploying and managing your applications on the GKE cluster.
Pro tip: Using kubectl
To access and manage your cluster from your local machine using kubectl, click the "Connect" button in the cluster details view. This will provide you with the necessary kubectl command to authenticate and connect to your GKE cluster.
kubectl is the Kubernetes command-line tool for interacting with the cluster. Command line is a bit daunting but it allows you to document the steps you take and reuse the commands in the future. Once you get used to working with kubectl, it can streamline your process, allowing you to keep notes and commands all in one place.

Key takeaways

It’s easy to create a Kubernetes cluster on GCP that meets your exact requirements. Kubernetes clusters allow multiple nodes to work together in concert no matter their physical distance.

Types of clusters

Kubernetes clusters

A Kubernetes cluster comprises multiple servers (which Kubernetes calls “nodes”) that work together as a group. These nodes are virtual or physical machines that form the underlying infrastructure of the Kubernetes cluster.
Each node is capable of running containers and hosting workloads. Kubernetes clusters are designed for scalability and high availability. Nodes can be added or removed as needed as workloads vary, so applications can scale up or down seamlessly.
Nodes are interconnected and communicate with each other through the Kubernetes control plane to ensure seamless coordination and collaboration. The control plane is the brain of the Kubernetes cluster. It consists of several components that manage and monitor the cluster's overall state, including:

- An API server
- A controller manager
- A scheduler
- An etcd: This is a reliable data storage that can be accessed by the cluster of machines.

Every Kubernetes cluster has one control plane and at least one control plane node. However, multiple nodes can be tasked with running the control plane components, and these component nodes can be spread out across zones for redundancy.
The standard unit for deployment to a Kubernetes cluster is a container. Containerized applications are software applications packaged along with their dependencies, libraries, and configurations into isolated containers. Containers can be easily duplicated, ensuring easy, consistent deployment across different environments.
Kubernetes is a powerful orchestration platform designed to manage and scale containerised applications. Kubernetes automates the deployment, scaling, and management of containerised applications across the cluster's nodes. Kubernetes also manages resources across the cluster by

optimally allocating CPU, memory, and storage based on application requirements. This ensures that resources are used efficiently, and it minimizes conflicts between applications. And Kubernetes also maintains the health of the cluster by employing features that automatically replace failed or unhealthy containers.
To manage a Kubernetes cluster, users specify the desired state of their applications, and then the cluster handles the actual execution and maintenance of the applications to match that desired state. This is called a “declarative approach.” The declarative approach simplifies management and reduces the need for manual intervention once the initial parameters are set.

Different types of Kubernetes clusters

Selecting the right type of cluster ensures a well-aligned Kubernetes deployment that will meet your specific business needs and objectives. Here are some of the major cluster architectures:

On-premises cluster

An on-premises Kubernetes cluster is deployed within an organization's own data center or on a private infrastructure. Deploying an on-premises cluster involves setting up the control plane and worker nodes on the organization's own hardware, and the organization is responsible for cluster maintenance. This provides complete control over the hardware, networking, and security. This is particularly suitable for situations with specific compliance or data governance requirements.
On-premises clusters are one of the primary types of Kubernetes clusters. Tools like Kubernetes kubeadm and Kubernetes Operations (kOps) are designed for deploying this type of cluster, and there are multiple custom configurations that can be used to create and manage on-premises clusters

Public cloud managed cluster

Another primary type of Kubernetes cluster is a public cloud managed cluster. Public cloud providers offer managed Kubernetes services, handling the underlying infrastructure management so it is easier for users to deploy and manage Kubernetes clusters on the cloud. This is a useful option for teams that prefer to offload cluster management tasks, but who still require the scalability and flexibility of cloud-based deployments. This type of cluster allows the organization to focus on deploying and managing applications without dealing with the complexities of cluster maintenance in the way you would with an on-premise cluster. When you run Kubernetes in the cloud, cluster maintenance is automatically handled by the cloud provider. You don't need to worry about it.
Another advantage of public cloud managed clusters is that they can be spread over zones or even regions. Just by checking a box in your configurations, you can spread your clusters geographically in case a cloud data center goes down or there are network problems. Some examples of managed services by public cloud providers are Amazon Elastic Kubernetes Service (EKS) on AWS, Google Kubernetes Engine (GKE) on Google Cloud, and Azure Kubernetes Service (AKS) on Microsoft Azure.

Private cloud managed cluster

These clusters function similarly to public cloud managed clusters, but private cloud providers manage Kubernetes services for deploying clusters within a private cloud environment. This combines the ease and flexibility of managed services with control over the private infrastructure. Some examples of managed private cloud providers are Nutanix, OpenStack, and Hewlett Packard Enterprises (HPE).

Local development clusters

The third primary type of Kubernetes clusters are local development clusters. These are lightweight and easy-to-set-up Kubernetes environments which facilitate a fast development workflow for individual developers. These are most often set up as local development and testing clusters, and they're primarily used for application development and testing on a developer's local machine. They are often employed during the development phase to enable rapid iteration and debugging of applications before deploying them to production clusters.
Tools commonly used to create local development clusters include Minikube, Docker Desktop (with Kubernetes enabled), and Kubernetes kind (Kubernetes in Docker). Each of these tools allows developers to spin up a single-node Kubernetes cluster locally to develop and validate applications without the need for a full-scale production cluster.

Hybrid cluster

A hybrid Kubernetes cluster coordinates on-premises and cloud environments, allowing workloads to run seamlessly across both locations. This type of cluster is suitable for scenarios in which some applications need to reside on-premises, for instance for security requirements, while others benefit from cloud scalability and services.

Edge cluster

An edge Kubernetes cluster is deployed at the edge of the network, closer to the locations of end-users or Internet of Things (IoT) devices. Edge clusters are designed to support low latency, particularly in regions or zones where power and network connectivity are scarce and expensive.

High-performance computing (HPC) cluster

HPC Kubernetes clusters are tailored for running computationally intensive workloads, such as scientific simulations or large data processing tasks. These clusters optimize performance by leveraging specialized hardware and configurations.

Multi-cluster federation

Multi-cluster federation involves managing multiple Kubernetes clusters as a single logical cluster. This allows centralized management of workloads, which are deployed across clusters similarly to the way a single Kubernetes cluster distributes workloads to multiple nodes. Multi-cluster federation facilitates global-scale deployments like disaster recovery scenarios.

Key takeaways

Kubernetes clusters are nodes which are coordinated to act as singular, cohesive units to provide a flexible and reliable platform. Clusters offer high availability, efficient scaling, and robust security. Depending on the computational purpose and requirements, you can configure a cluster to suit any organization.

Deploying Docker containers on GCP

Docker containers make it simple to deploy applications across different systems and platforms, allowing them to run the same way no matter what environment they are deployed to. This makes it easy to share, test, manage, and deploy applications quickly and reliably.

There are several platforms that allow you to deploy Docker containers, and each has its own set of advantages. Let's look at some of these, and some considerations when choosing where to deploy containers.

Docker containers on Google Cloud Run

If you don't need a lot of flexibility in your configuration, you might find Google Cloud Run to be a good option. Cloud Run is a fully-managed platform that allows you to run containerised applications without having to manage the underlying infrastructure. The platform offers easy deployment, and automatically manages scaling and routing traffic based on incoming requests. It can scale down to zero, which means it will not use any unnecessary resources if there are no requests. The platform is based on containers, so you can write your code in any language and then deploy it through Docker images.

Cloud Run may be the best choice of platform for projects that do not need a high level of monitoring or configuration, providing that these applications are stateless. A stateless application is one which does not read or store information about their states from one run to the next. Kubernetes Deployment is most commonly used for stateless applications, so Cloud Run is often a great option .

Cloud Run Deploying Docker containers on Cloud Run

To deploy Docker containers on Cloud Run:

1. Build your Docker image and push it to any supported container registry.
2. Deploy the container to Cloud Run using the `gcloud run deploy` command or through the Cloud Run Console.

For more about deploying an app to Cloud Run, see Deploy to Cloud Run.

Docker containers on Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) is a container orchestration platform provided by Google Cloud that allows you to automate the process of deploying, managing, and scaling containerised applications. Its streamlined cluster setup process makes deploying Kubernetes clusters fast and straightforward. Many functions are automated as well, including version upgrades and management of SSL certificates.

Although GKE gives you control of all of your configurations, you can choose GKE Autopilot as a fully-managed option. Autopilot uses the Google Cloud Platform (GCP) to automate cluster configuration and to scale the underlying infrastructure based on your parameters and your application's resource requirements. This eliminates the need for manual intervention on your part, and optimizes resources.

GKE is also particularly resilient, continuously monitoring the cluster and its components. Built-in monitoring and logging features provide real-time insights into your application's performance and health. But you don't have to watch constantly, because GKE features self-healing clusters which automatically detect and replace unhealthy nodes or containers, maintaining the desired state and application availability.
GKE is also convenient for integration with all of the Google Cloud ecosystem, making it easy to leverage other services like Cloud Storage and Google Cloud Identity and Access Management (IAM) to enhance security and governance. GCP itself ensures regular security updates and follows industry best practices to protect the underlying infrastructure and containers.
You may not be ready for the following consideration, but GKE also supports running "stateful" applications. A stateful application is one for which a server saves status and session information. Kubernetes Deployment is commonly used for stateless applications, but you can make an application stateful by attaching a persistent volume to it. If you find yourself needing to run a stateful application, GKE is a great option.

Deploying Docker containers on GKE

Now that you've heard about GKE's features, here's how you can deploy Docker containers on GKE:

1. Build your Docker image and push it to any supported container registry, such as Google Container Registry (GCR).
2. Create a Kubernetes Deployment manifest that specifies your configuration settings, including the container image you want to run and the desired number of replicas.
3. Use the GKE Console, or the Kubernetes command line tool kubectl, to deploy the application to your GKE cluster. GKE will handle the orchestration and scaling of the containers for you.

For more about deploying an app to a GKE cluster, see Deploy an app to a GKE cluster.
For more on choosing between GKE and Cloud Run, see Google Kubernetes Engine vs Cloud Run: Which should you use?

Docker containers on Google Compute Engine

Finally, Google Compute Engine is a virtual machine (VM) service that allows you to run your containerized applications on Google's infrastructure. It has lower access time, which tends to translate to faster performance, and offers easy integration with other GCP services.
Perhaps most attractive to its users, Google Compute Engine lets you run code on Google's infrastructure, but grants you more control over the underlying infrastructure of your VM instances than GKE, and far more than Cloud Run. It is particularly suitable for custom environments and applications requiring specific configurations. But with more control comes more responsibility: when using Google Compute Engine, you are responsible for managing the VM instances and scaling. The platform itself is not as simplified as GKE or Cloud Run, and you cannot use all programming languages.

Deploying Docker containers on Google Compute Engine

To deploy Docker containers on Google Compute Engine:

1. Build your Docker image and push it to any supported container registry.
2. Provision a VM instance on Google Compute Engine.
3. Install Docker on the VM.
4. Build your Docker image and copy it to the virtual machine (or pull the image from a container registry).
5. Run the Docker container on the virtual machine using the `docker run` command.

For more about containers on Compute Engine, see Containers on Compute Engine.

Key takeaways

GCP offers several choices for deploying Docker containers, all of which allow you to integrate with other Google services. Cloud Run is the simplest to use, offering a fully managed platform, but with little customization. GKE is a powerful platform that offers more flexibility in configuration coupled with plenty of options for automation. Google Compute Engine lets you control your environments and applications while they run on Google's infrastructure, but requires significantly more technical knowledge than Cloud Run or GKE. The best option for you will be based on your needs.

Kubernetes YAML files

Suppose you're a Python developer working on a web application that's experiencing a surge in user traffic. You've containerized your application using Docker, but manually scaling the application to keep up with demand is becoming cumbersome. Kubernetes to the rescue! Kubernetes can manage and scale your containerized application automatically…if you can tell it what to do! This is where Kubernetes YAML files come in.

Kubernetes YAML files define and configure Kubernetes resources. They serve as a declarative blueprint for your application infrastructure, describing what resources should be created, what images to use, how many replicas of your service should be running, and more.

Structure of Kubernetes YAML files

Every Kubernetes YAML file follows a specific structure with key components: `API version`, `kind`, `metadata`, and `spec`. These components provide Kubernetes with everything it needs to manage your resources as desired.

- `apiVersion`: This field indicates the version of the Kubernetes API you're using to create this particular resource.
- `kind`: This field specifies the type of resource you want to create, such as a Pod, Deployment, or Service.
- `metadata`: This section provides data that helps identify the resource, including the name, namespace, and labels.
- `spec`: This is where you define the desired state for the resource, such as which container image to use, what ports to expose, and so on.

Let's illustrate this with a simple Kubernetes YAML file for creating a Deployment of your Python web application:

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: python-web-app
  labels:
    app: python-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: python-web-app
  template:
    metadata:
      labels:
        app: python-web-app
    spec:
      containers:
      - name: python-web-app
        image: your-docker-repo/python-web-app:latest
        ports:
        - containerPort: 5000
```

In the YAML file above, we're defining a "Deployment" resource for a Kubernetes cluster. A Deployment is a way to tell Kubernetes how to run our application. This YAML file tells Kubernetes to create a Deployment named "python-web-app", which will ensure that three instances of our Python web application are always running.

Key components and fields in Kubernetes YAML files

YAML files can include many other fields, depending on the type of object and your specific needs.

Pods

As you've learned, a Pod is the smallest and simplest unit in the Kubernetes object model. It represents a single instance of a running process in a cluster and can contain one or more containers. Because it is the simplest unit, a Pod's YAML file typically contains the basic key components highlighted above:

- **`apiVersion`**: This is the version of the Kubernetes API you're using to create this object.
- **`kind`**: This is the type of object you want to create. In this case, it's a Pod.
- **`metadata`**: This includes data about the Pod, like its name and namespace.
- **`spec`**: This is where you specify the desired state of the Pod, including the containers that should be running. Specifications include:
    - Containers: An array of container specifications, including "name", "image", "ports", and "env".
    - Volumes: Array of volume mounts to be attached to containers
    - restartPolicy: Defines the Pod's restart policy (e.g., "Always," "OnFailure," "Never")

Deployments

A Deployment is a higher-level concept that manages Pods and ReplicaSets. It allows you to describe the desired state of your application, and the Deployment controller changes the actual state to the desired state at a controlled rate. In addition to the fields mentioned above, a Deployment's YAML file includes:

- `spec.replicas`: This is the number of Pods you want to run.
- `spec.selector`: This is how the Deployment identifies the Pods it should manage.
- `spec.template`: This is the template for the Pods the Deployment creates.

Services

A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Key components in a Service's YAML file include:

- `spec.type`: This defines the type of Service. Common types include ClusterIP, NodePort, and LoadBalancer.
- `spec.ports`: This is where you define the ports the Service should expose.
- `spec.selector`: This is how the Service identifies the Pods it should manage.

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. In addition to the common fields, a ConfigMap's YAML file includes the data field, which is where you define the key-value pairs.

Secrets

A Secret is similar to a ConfigMap, but is used to store sensitive information, like passwords or API keys. A Secret's YAML file includes:

- `type`: The type of Secret. Common types include Opaque (for arbitrary user-defined data), kubernetes.io/service-account-token (for service account tokens), and others.
- `data`: This is where you define the key-value pairs. The values must be base64-encoded.

Each of these resources can be defined and managed using Kubernetes YAML files. For example, a YAML file for a Secret resource might look like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  db_username: dXNlcm5hbWU=  # base64 encoded username
  db_password: cGFzc3dvcmQ=  # base64 encoded password
```

Parameterizing YAML files with Python

As you've seen, YAML files are the backbone of defining and managing resources. However, static YAML files can be limiting, especially when you need to manage different configurations for different environments or deployment scenarios. This is where Python comes in, offering a dynamic and flexible approach to parameterize your YAML files.
For instance, you can customize your rolling update strategy using Python with the following example code.

```
from kubernetes import client, config

def update_deployment_strategy(deployment_name, namespace,
max_unavailable):
    config.load_kube_config()
    apps_v1 = client.AppsV1Api()

    deployment = apps_v1.read_namespaced_deployment(deployment_name,
namespace)
    deployment.spec.strategy.rolling_update.max_unavailable =
max_unavailable
    apps_v1.patch_namespaced_deployment(deployment_name, namespace,
deployment)

if __name__ == "__main__":
    update_deployment_strategy('my-deployment', 'my-namespace', '25%')
```

This is just one example of how Python provides a powerful and flexible way to parameterize your YAML files in Kubernetes.

Key takeaways

Kubernetes YAML files play a crucial role in defining and managing Kubernetes resources, enabling Python developers to manage their applications' infrastructure in a consistent, version-controlled, and automated manner. By understanding the structure of these files and their key components, developers can leverage Kubernetes to its full potential and focus more on writing the application logic rather than managing infrastructure.

Resources for more information

Objects in Kubernetes
Get Started with Kubernetes (using Python)

Scaling containers on GCP

One of the most important factors in cloud computing is scalability. For a moment, let's imagine cloud computing as if it were an actual cloud in the sky. Clouds are made up of water vapor; the more vapor that joins the cloud, the larger it gets. As the cloud releases this water as snow or rain, the cloud gets smaller. That's scalability; the process of expanding or shrinking as necessary.
The conditions in the atmosphere determine the shape of a cloud throughout this scaling process. There is a lot more we could say about sky-clouds, but let's return to computing clouds. Your application's requirements are the conditions that determine the direction and method of scaling in cloud computing.

Horizontal and vertical scaling

Generally, we can talk about scaling on two axes, horizontal, meaning sideways, and vertical, meaning up and down. In horizontal scaling, more containers are added as needed. Horizontal scaling is useful for adding dedicated resources as the number of users increase, and creating fallback containers in case of failure. In vertical scaling, more resources are allocated to each container. Vertical scaling increases performance.

Multidimensional scaling

Multidimensional scaling is a combination of horizontal and vertical scaling. This is sometimes called diagonal scaling, because you are doing some horizontal scaling, adding more containers to add to the number of resources, and some vertical scaling, increasing the performance of existing or added resources.
Imagine it's autumn and someone is burning a pile of leaves, but the fire is a little bigger than they planned. That's pretty dangerous, so they call the fire department. Meanwhile, they fill a bucket and use it to throw water on the fire. Neighbors come over, each with their own bucket, and start throwing more buckets of water. That's horizontal scaling, the addition of more small resources.
The fire truck rolls up and hooks up the big hose, but the pressure isn't very good. In fact it isn't any better than the pressure from the hoses that other neighbors have stretched from their own homes. All these hoses are moving more water than the buckets, though. That's diagonal scaling; more resources with a bit more performance in each.
But then the fire department cranks open the valve on the fire hydrant, and a huge jet of water flies out of the big hose. That's vertical scaling; increasing resources to a single response unit to increase its performance.

Elastic scaling

In cloud computing, you can employ elastic scaling to automatically increase or decrease the number of servers (horizontal) or the resources allocated to existing servers (vertical) or both (multidimensional) based on the current demand.
Containers can scale down to a fraction of a computer, or scale up to use all the resources of multiple computers. It's important to decide on the type or types of scaling your application will require in advance so you can make sure to have the right service-level agreement (SLA). Your SLA is the contract with your platform provider; this dictates what will be furnished to you, and at what cost.
For more details about horizontal and vertical scaling, see Horizontal Vs. Vertical Scaling: How Do They Compare?

For more on the how-to of scaling, see Scaling an application.
For more about horizontal autoscaling, see Horizontal Pod autoscaling and Configuring horizontal Pod autoscaling.
For more on diagonal scaling, see Configure multidimensional Pod autoscaling.
For a primer on using autoscaling in Google Kubernetes Engine (GKE), see Understanding and Combining GKE Autoscaling Strategies.

Scaling containers on GCP

Google Cloud Platform (GCP) has massive amounts of computers and components at the ready, so the platform lets you avoid a delayed response to scaling needs. These resources are shared between GCP users, which means you pay less during off hours when your app requires fewer resources. By comparison, if you were running a Kubernetes cluster on-premises, you would pay for electricity for a device that might always be on, regardless of whether it was idle or busy.
Scaling is also easy to automate using the dashboards in any of GCP's platforms. These dashboards allow you to set policies and limits. When setting your scaling parameters, be sure to pay attention to the pricing involved with scaling. The base price for a Kubernetes cluster may be a lot lower than the price when it operates at scale. If traffic suddenly takes off for your application, you may be thrilled, but the thrill might be gone when you get the bill.
Before you choose settings for your scaling, learn more about the details.
For more about adjusting capacity for demand and resilience, see Patterns for scalable and resilient apps.
For more on instance autoscaling, see About instance autoscaling.
For more about basing your autoscaling needs on metrics, see Autoscale workloads based on metrics.
Last but far from least, learn about pricing as you set limits for your scaling capabilities. For more on this, see Google Kubernetes Engine pricing.

Pro tip

Using `kubectl`, the Kubernetes command line tool, can be intimidating. But once you get used to working with kubectl, it can really streamline your process. It also lets you keep your notes and commands all in one place.
For a tutorial on how to use the command line to scale containers, see the Autoscaling Deployments section in this tutorial on Scaling an application.

GCP networking and load balancing

Why does knowing GCP infrastructure matter?

Imagine that your team develops a recommendations engine that suggests products to customers based on their browsing history. You need a way to **scale**, **secure**, and **optimize** your application. This is where Google Cloud Platform (GCP) comes in.
In this reading, you will explore GCP's networking infrastructure, including virtual private clouds (VPCs), subnets, and load balancing. You'll also learn how these technologies help improve the availability and responsiveness of your recommendation application.

GCP is a high-quality, high-speed, and highly reliable global network that facilitates communication between various resources such as virtual machines (VMs), Kubernetes clusters, and managed databases, regardless of their geographical location.
Google network infrastructure consists of three main types of networks:

- **A data center network**, which connects all the machines in the network together.
- **A software-based private wide area network (WAN)** that connects all data centers together. The software-based private WAN is particularly beneficial for distributed applications that require fast and secure data transfer between different parts of the system, regardless of where they are located.
- **A software defined public WAN** that is designed for user-facing traffic entering the Google network. This network infrastructure is optimized for high performance and low latency, ensuring that users can access your applications quickly and smoothly, no matter where they are in the world.

You can read more about the Google Cloud network structure here.

Pods, clusters, and GCP

Let’s imagine that you want to deploy your recommendation engine using a Kubernetes cluster on GCP. To secure and partition your cluster from the public internet, you must first create a VPC network.
A VPC is a global, private network—partitioned from the broader GCP network—that facilitates communication between the Pods in your cluster, allowing them to interact with each other as if they were on the same local network, even though they might be running on different machines. This is crucial for distributed applications where different components (running in different Pods) need to communicate with each other.
The VPC not only provides an isolated network for the Kubernetes cluster, but it also enables secure communication with other GCP resources. For instance, your Pods might need to interact with a database or a storage service hosted elsewhere on GCP. The VPC ensures that these interactions can occur securely and efficiently over the private network, without exposing the traffic to the public internet.

Key components of a GCP VPC network

Each VPC is divided into subnets, which are regional resources. Each subnet has a specific IP range, and you can have multiple subnets in a single VPC. When you create a Kubernetes cluster in a GCP VPC, you assign each node in the cluster an IP address from the subnet's IP range.
GCP also allows you to define firewall rules at the VPC level. These rules control inbound and outbound traffic to your resources. For example, you might have a rule that allows incoming HTTP traffic to your web servers, or a rule that blocks all outbound traffic to a specific IP range.
A GCP VPC network has several key components:

- **IP ranges:** Each VPC network and its subnets have associated IP ranges. These ranges are used to assign IP addresses to resources within the network and subnets.
- **Routes:** Routes determine the path that network traffic takes to reach an instance. By default, a VPC network has an implied route to the internet default gateway, allowing instances with external IP addresses to reach the internet.

- **Peering:** VPC network peering allows you to connect two VPC networks, potentially across different projects, as if they were one. This is useful for sharing resources across projects or organizations.
- **Firewall rules:** As mentioned earlier, firewall rules control the traffic to and from instances in your VPC network. They are a crucial part of securing your GCP environment.

You can find a complete list of the VPC components on the Google Cloud website.

Other network services

GCP offers several other networking services that are very useful for Python developers using Kubernetes. These include:

- **Cloud Domain Name System (DNS)**: Google Cloud DNS is a scalable, reliable, and managed authoritative Domain Name System (DNS) service that provides high DNS query speeds and low latency for your applications. If your Kubernetes application needs to map domain names to IP addresses (for example, if it's a web application that needs to be accessible via a custom domain), Cloud DNS can be a useful service.
- **Cloud Network Address Translation (NAT)**: Cloud NAT allows instances without a public IP address (like your Kubernetes Pods) to access the internet, while not allowing inbound connections from the internet. This can be useful if your application needs to reach external APIs or services but you don't want to expose your application to incoming internet connections.
- **Cloud Load Balancing**: Google Cloud Load Balancing allows you to distribute traffic across your application instances, which can be located in multiple regions. This can help increase the availability and reduce the latency of your application.

These are just some of the services available to help you. Each of these services can be managed and configured using the Google Cloud Console, the gcloud command-line tool, or the Google Cloud Client Libraries (including the Python library).

A closer look: Load balancing

Imagine you've successfully set up your recommendation engine and it's now running on its own VPC on GCP. During a regular workday, the recommendation engine performs well and handles the site's traffic without any issues. However, during your company's annual summer sale, the site experiences a surge in traffic. The recommendation engine struggles to keep up with the increased load, leading to slower response times and, in some cases, timeouts. You realize that they need a solution to handle these traffic spikes more gracefully. What to do?

This is where Cloud Load Balancing comes into play. GCP provides several load balancing solutions that distribute incoming traffic among multiple instances of your application, helping to ensure that no single instance bears too much load. This can improve the responsiveness and availability of your application, especially during times of high traffic.

Here are a few ways GCP helps with load balancing:

- **Global and regional load balancing**: GCP offers both global and regional load balancing. Global load balancing automatically directs user traffic to the nearest instance of your application, improving latency. Regional load balancing distributes traffic within a specific region.

- **HTTP(S), TCP, and UDP load balancing**: GCP supports load balancing for HTTP(S), TCP, and UDP traffic, allowing you to choose the right option based on your application's needs.
- **Managed Instance Groups**: GCP's managed instance groups work hand-in-hand with its load balancers. They maintain a pool of instances that can automatically scale up or down based on demand, and the load balancer distributes traffic across these instances.
- **Integration with Kubernetes**: GCP's load balancers can be easily integrated with Google Kubernetes Engine (GKE), allowing you to distribute traffic across the Pods in your Kubernetes cluster.

In your case, you do some research and set up a load balancer configured to distribute traffic across the Pods in the Kubernetes cluster. You also configure a managed instance group to automatically scale the number of Pods based on the incoming traffic.
The next day, as the summer sale continues, the recommendation engine performs significantly better. The load balancer efficiently distributes the incoming traffic across multiple Pods, ensuring no single Pod is overwhelmed. Meanwhile, the managed instance group automatically scales up the number of Pods during peak traffic times and scales them back down when the traffic subsides.
This is just one example of how GCP can help provide a consistent and responsive experience for the users of a Python application. How might it help you?

Key takeaways

- A VPC provides the network infrastructure that allows for secure, efficient communication between Pods within the cluster and between the cluster and other GCP resources.
- Leverage GCP's load balancing solutions to provide a consistent and responsive user experience, especially during periods of high traffic.

Resources for more information

Cloud Networking Overview
Subnets
Cloud Firewalls
Cloud Load Balancing
Google Kubernetes Engine

Protect containers on GCP

As Python developers working with Kubernetes on Google Cloud Platform (GCP), understanding how to protect your containers is crucial. This involves a combination of tools and practices provided by GCP, as well as your own responsibilities in configuring and managing these resources.

Security challenges and considerations

The first step in protecting your containers is understanding the security challenges and considerations involved. Containers, while providing a lightweight and portable solution for running your applications, also come with their own set of security challenges. These include the need to secure the container runtime, to protect the underlying host system, and to manage the application dependencies that are packaged within the container.

One approach to addressing this challenge is the Zero Trust model, which involves assuming no trust by default and only granting permissions as necessary. This means starting with a completely locked down system and only opening up what is necessary for your application to function. This approach can help to minimize the potential attack surface and reduce the risk of a security breach. You can read more about the Zero Trust model here.
Also, building on the previous reading about GCP networking, using Virtual Private Clouds (VPCs) and properly firewalled subnets means you can guarantee at the network level—not the software level—that things do not come into contact with other things they shouldn't. It is easier to build a moat than a drawbridge!

Shared Responsibility Model

Another model you can implement for container security is the Shared Responsibility Model. This model outlines the responsibilities of both GCP and you in ensuring the security of the containers. In this model, GCP is responsible for:

- **Infrastructure security:** GCP is responsible for the physical security of data centers, the security of the hardware and software that underpin the service, and the networking infrastructure of the container orchestration service, Google Kubernetes Engine (GKE).
- **Operational security:** GCP is responsible for ensuring that the GKE service is operational and available for customers to use. This includes protecting against threats that could impact the service's availability, such as Distributed Denial of Service (DDoS) attacks.
- **Software supply chain security:** GCP provides tools and features to help secure the software supply chain, such as Binary Authorization for Borg (BAB), which can enforce signature verification checks on container images before they are deployed.

Tools help provide security, but can be configured incorrectly. That's where you come in! To make effective use of the container security tools offered by GCP, you are responsible for:

- **Workload security:** You are responsible for ensuring the security of the workloads (i.e., the applications and data) that you run on GKE. This includes implementing appropriate access controls, protecting sensitive data, and managing cryptographic keys.
- **Network security:** Although GCP provides the underlying network infrastructure, you are responsible for securing the network connections between your workloads. This includes configuring firewalls, managing network policies, and securing network endpoints.
- **Identity and access management:** You are responsible for managing access to your GKE resources. This includes managing user identities, assigning appropriate roles and permissions, and using strong authentication methods.
- **Software supply chain security:** Although GCP provides tools to help secure the software supply chain, you are responsible for using these tools effectively. This includes ensuring that container images are securely built, stored, and signed.

Security Features and Best Practices

As mentioned above, a zero trust approach makes sense. Start with the least possible amount of permissions, open ports, and only change the bare minimum to what is needed to provide service. Security is usually reactive, so this lowers the attack surface for your cloud builds, giving less ways for attackers to get in.

Other security strategies you can implement include:

- **Use minimal base images:** The fewer components and services running in your container, the fewer potential vulnerabilities. Use minimal base images that only contain the essential components needed for your application. This means closing all IP addresses and ports except those that are necessary, closing data containers off to the outside internet so only parts of the application can reach them, and so on. This can also mean only opening up containers on VPCs to other VPCs or members of the same subnet, and having sound firewall rules in place.
- **Regularly update and patch:** This may seem obvious, but regularly update your containers and their dependencies to ensure you have the latest security patches. Outdated software is one way vulnerabilities get in. Sometimes new software is shipped that has new bugs that do not get discovered for a while. Just updating containers is not sufficient to guarantee safety, but it can reduce the risk. Google Cloud provides services like Container-Optimized OS which automatically updates and patches itself.
- **Implement vulnerability scanning:** Use tools like Google's Container Analysis and Container Threat Detection in the Google Cloud console to regularly scan your containers for known vulnerabilities.
- **Use runtime security:** Use a tool like gVisor to provide sandboxing for containers at runtime, isolating them from the host kernel and reducing the risk of a container escape vulnerability.
- **Implement access controls:** Use Identity and Access Management (IAM) to control who can do what with your containers and other resources.
- **Encrypt sensitive data:** Use Google's Key Management Service (KMS) to encrypt sensitive data in your containers.
- **Monitor and log activity:** Use Google's Cloud Audit Logs to keep track of who did what, when, in your Google Cloud environment. Logs are super important to have on from the start. If you wait for an incident to happen to necessitate turning on logs, it may be too late!
- **Use binary authorization:** This is a deploy-time security control that ensures only trusted images are deployed in your environment.

Key takeaways

Container security is a vast topic that could easily be its own course. For right now, however, the key takeaways from this reading are:

- Containers pose some unique security challenges, including securing the container runtime, protecting the host system, and managing application dependencies.
- Adopting a Zero Trust model can help mitigate these challenges. This approach involves assuming no trust by default and only granting permissions as necessary, reducing the potential attack surface.
- Security on GCP is a shared responsibility. GCP is responsible for infrastructure security, operational security, and providing tools for software supply chain security. Developers are responsible for workload security, network security, identity and access management, and effective use of software supply chain security tools.
- GCP provides several security features and best practices for protecting containers, including using minimal base images, regularly updating and patching containers, implementing vulnerability scanning, using runtime security tools like gVisor, implementing

access controls with IAM, encrypting sensitive data with KMS, monitoring and logging activity with Cloud Audit Logs, and using Binary Authorization to ensure only trusted images are deployed.

Resources for more information

Exploring Container Security
Google Cloud Security Command Center
Shared Responsibilities and Shared Fate on Google Cloud
GKE shared responsibility | Google Kubernetes Engine (GKE)

Exemplar: Work with containers on GCP Overview

Docker is an open platform for developing, shipping, and running applications. With Docker, you can separate your applications from your infrastructure and treat your infrastructure like a managed application. Docker helps you ship code faster, test faster, deploy faster, and shorten the cycle between writing code and running code.
Docker does this by combining kernel containerization features with workflows and tooling that helps you manage and deploy your applications.
Docker containers can be directly used in Kubernetes, which allows them to be run in the Kubernetes Engine with ease. After learning the essentials of Docker, you will have the skillset to start developing Kubernetes and containerized applications.
This exemplar is a walkthrough of the previous Qwiklab activity, including detailed instructions and solutions. You may use this exemplar if you were unable to complete the lab and/or you need extra guidance in competing lab tasks. You may also refer to this exemplar to prepare for the graded quiz in this module.

How to start your lab and sign in to the Google Cloud Console

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is the **Lab Details** panel with the following:
    - The **Open Google Console** button
    - Time remaining
    - The temporary credentials that you must use for this lab
    - Other information, if needed, to step through this lab
2. Click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.
   Tip: Arrange the tabs in separate windows, side-by-side.
   Note: If you see the **Choose an account** dialog, click **Use Another Account**.
3. If necessary, copy the **Username** from the **Lab Details** panel and paste it into the **Sign in** dialog. Click **Next**.

Copy the **Password** from the **Lab Details** panel and paste it into the **Welcome** dialog. Click **Next**.

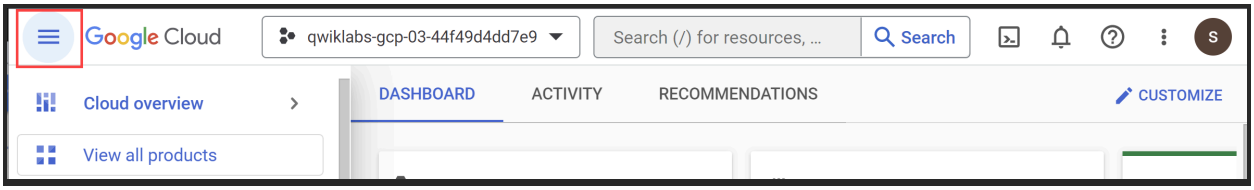
Important: You must use the credentials from the left panel. Do not use your Google Cloud Skills Boost credentials.

Note: Using your own Google Cloud account for this lab may incur extra charges.

4.
5. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

After a few moments, the Cloud Console opens in this tab.
Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.
1. Click **Activate Cloud Shell** at the top of the Google Cloud console.
When you are connected, you are already authenticated, and the project is set to your **PROJECT_ID**. The output contains a line that declares the **PROJECT_ID** for this session:

```
Your Cloud Platform project in this session is set to YOUR_PROJECT_ID
```

gcloud is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.
2. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

3. Click **Authorize**.
4. Your output should now look like this:

Output:

```
ACTIVE: *
ACCOUNT: student-01-xxxxxxxxxxxx@qwiklabs.net

To set the active account, run:
    $ gcloud config set account `ACCOUNT`
```

5. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

Output:

```
[core]
project = <project_ID>
```

Example output:

```
[core]
```

```
project = qwiklabs-gcp-44776a13dea667a6
```

Note: For full documentation of gcloud, in Google Cloud, refer to the gcloud CLI overview guide.

Task 1. Hello world

1. In Cloud Shell enter the following command to run a hello world container to get started:

```
docker run hello-world
```

Output:

```
Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

9db2ca6ccae0: Pull complete

Digest:
sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc

Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!

This message shows that your installation appears to be working correctly.

...
```

This simple container returns **`Hello from Docker!`** to your screen. While the command is simple, notice in the output the number of steps it performed. The Docker daemon searched for the hello-world image, didn't find the image locally, pulled the image from a public registry called Docker Hub, created a container from that image, and ran the container for you.

2. Run the following command to take a look at the container image it pulled from Docker Hub:

```
docker images
```

Output:

```
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE

hello-world   latest    feb5d9fea6a5   14 months ago   13.3kB
```

This is the image pulled from the Docker Hub public registry. The Image ID is in SHA256 hash format—this field specifies the Docker image that's been provisioned. When the Docker daemon can't find an image locally, it will by default search the public registry for the image.

3. Run the container again:

```
docker run hello-world
```

Output:

```
Hello from Docker!

This message shows that your installation appears to be working correctly.

...
```

```
Hello from Docker!

This message shows that your installation appears to be working correctly.



To generate this message, Docker took the following steps:

...
```

Notice the second time you run this, the Docker daemon finds the image in your local registry and runs the container from that image. It doesn't have to pull the image from Docker Hub.

4. Finally, look at the running containers by running the following command:

```
docker ps
```

Output:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

There are no running containers. You already exited the hello-world containers you previously ran.

```
docker ps -a
```

Output:

```
CONTAINER ID    IMAGE         COMMAND    ...   NAMES

6027ecba1c39    hello-world   "/hello"   ...   cranky_meitner
```

```
358d709b8341    hello-world   "/hello"   ...   compassionate_cartwright
```

This shows you the **`Container ID`**, a UUID generated by Docker to identify the container, and more metadata about the run. The container **`Names`** are also randomly generated but can be specified with **`docker run --name [container-name] hello-world`**.

Task 2. Build

In this section, you will build a Docker image that's based on a simple node application.

1. Execute the following command to create and switch into a folder named **`test`**.

```
mkdir test && cd test
```

2. Create a **`Dockerfile`**:

```
```

```
cat > Dockerfile <<EOF
# Use an official Node runtime as the parent image
FROM node:lts

# Set the working directory in the container to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Make the container's port 80 available to the outside world
EXPOSE 80

# Run app.js using node when the container launches
CMD ["node", "app.js"]
EOF
```

This file instructs the Docker daemon on how to build your image.

- The initial line specifies the base parent image, which in this case is the official Docker image for node version long term support (lts).

- In the second, you set the working (current) directory of the container.
- In the third, you add the current directory's contents (indicated by the `"."`) into the container.
- Then expose the container's port so it can accept connections on that port and finally run the node command to start the application.

Note: Spend some time reviewing the Dockerfile command references to understand each line of the `Dockerfile`.

Now you'll write the node application, and after that you'll build the image.

3. Run the following to create the node application:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

```
cat > app.js <<EOF
const http = require('http');

const hostname = '0.0.0.0';
const port = 80;

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World\n');
});

server.listen(port, hostname, () => {
    console.log('Server running at http://%s:%s/', hostname, port);
});
```

```
process.on('SIGINT', function() {

    console.log('Caught interrupt signal and will exit');

    process.exit();

});

EOF
```

This is a simple HTTP server that listens on port 80 and returns "Hello World".
Now build the image.
4. Note again the "`.`", which means current directory so you need to run this command from within the directory that has the Dockerfile:

```
docker build -t node-app:0.1 .
```

It might take a couple of minutes for this command to finish executing. When it does, your output should resemble the following.
Output:

```
 => [internal] load metadata for docker.io/library/node:lts
```

The `-t` is to name and tag an image with the **`name:tag`** syntax. The name of the image is **`node-app`** and the **`tag`** is `0.1`. The tag is highly recommended when building Docker images. If you don't specify a tag, the tag will default to **`latest`** and it becomes more difficult to distinguish newer images from older ones. Also notice how each line in the **`Dockerfile`** above results in intermediate container layers as the image is built.
5. Now, run the following command to look at the images you built:

```
docker images
```

Your output should resemble the following:

```
```

```
REPOSITORY      TAG       IMAGE ID        CREATED             SIZE

node-app        0.1       f166cd2a9f10    25 seconds ago      656.2 MB

hello-world     latest    1815c82652c0    6 days ago          1.84 kB
```

Notice **`node`** is the base image and **`node-app`** is the image you built. The size of the image is relatively small compared to VMs. Other versions of the node image such as **`node:slim`** and **`node:alpine`** can give you even smaller images for easier portability. The topic of slimming down container sizes is further explored in Advanced Topics. You can view all versions in the official repository in node.

Task 3. Run

1. Use this code to run containers based on the image you built:

```
docker run -p 4000:80 --name my-app node-app:0.1
```

Output:

```
Server running at http://0.0.0.0:80/
```

The **`--name`** flag allows you to name the container if you like. The **`-p`** instructs Docker to map the host's port 4000 to the container's port 80. Now you can reach the server at **`http://localhost:4000`**. Without port mapping, you would not be able to reach the container at localhost.

2. Open another terminal (in Cloud Shell, click the **+** icon), and test the server:

```
curl http://localhost:4000
```

Output:

```
Hello World
```

The container will run as long as the initial terminal is running. If you want the container to run in the background (not tied to the terminal's session), you need to specify the **`-d`** flag.

3. Close the initial terminal and then run the following command to stop and remove the container:

```
docker stop my-app && docker rm my-app
```

4. Now run the following command to start the container in the background:

```
docker run -p 4000:80 --name my-app -d node-app:0.1


docker ps
```

Output:

```
CONTAINER ID   IMAGE          COMMAND        CREATED          ...  NAMES
xxxxxxxxxxxx   node-app:0.1   "node app.js"  16 seconds ago   ...  my-app
```

5. Notice the container is running in the output of **`docker ps`**. You can look at the logs by executing **`docker logs [container_id]`**.

Note: You don't have to write the entire container ID, as long as the initial characters uniquely identify the container. For example, you can execute docker logs 17b if the container ID is **`17bcaca6f...`**.

```
docker logs [container_id]
```

Output:

```
Server running at http://0.0.0.0:80/
```

Now modify the application.

1. In your Cloud Shell, open the test directory you created earlier in the lab:

```
cd test
```

2. Edit `app.js` with a text editor of your choice (for example nano or vim) and replace "Hello World" with another string:

```
....
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Welcome to Cloud\n');
});
....
```

3. Build this new image and tag it with `0.2`:

```
docker build -t node-app:0.2 .
```

Output:

```
[+] Building 0.3s (8/8) FINISHED
docker:default

 => [internal] load build definition from Dockerfile
0.0s

 => => transferring dockerfile: 397B
0.0s

 => [internal] load metadata for docker.io/library/node:lts
0.1s
```

```
 => [internal] load .dockerignore
0.0s

 => => transferring context: 2B
0.0s

 => [1/3] FROM
docker.io/library/node:lts@sha256:1ae9ba874435551280e95c8a8e74adf8a48d72b5
64bf9dfe4718231f2144c88f
0.0s

 => [internal] load build context
0.0s

 => => transferring context: 537B
0.0s

 => CACHED [2/3] WORKDIR /app
0.0s

 => [3/3] ADD . /app
0.0s

 => exporting to image
0.0s

 => => exporting layers
0.0s

 => => writing image
sha256:bdbf689be1b01552442eb41a8af56bec3d5a3b0a4c2ab4a62c465b3f12fe0e0f
0.0s

 => => naming to docker.io/library/node-app:0.2
```

Notice in Step 2 that you are using an existing cache layer. From Step 3 and on, the layers are modified because you made a change in **`app.js`**.

4. Run another container with the new image version. Notice how we map the host's port 8080 instead of 80. You can't use host port 4000 because it's already in use.

```
                                                                        1
```

```
docker run -p 8080:80 --name my-app-2 -d node-app:0.2

docker ps
```

Output:

```
CONTAINER ID   IMAGE           COMMAND          CREATED             ...
xxxxxxxxxxxx   node-app:0.2    "node app.js"    53 seconds ago      ...
xxxxxxxxxxxx   node-app:0.1    "node app.js"    About an hour ago   ...
```

5. Test the containers:

```
curl http://localhost:8080
```

Output:

```
Welcome to Cloud
```

6. And now test the first container you made:

```
curl http://localhost:4000
```

Output:

```
Hello World
```

Task 4. Debug

Now that you're familiar with building and running containers, go over some debugging practices.

1. You can look at the logs of a container using **`docker logs [container_id]`**. If you want to follow the log's output as the container is running, use the **`-f`** option.

```
docker logs -f [container_id]
```

Output:

```
Server running at http://0.0.0.0:80/
```

Sometimes you will want to start an interactive Bash session inside the running container.

2. You can use **`docker exec`** to do this. Open another terminal (in Cloud Shell, click the **+** icon) and enter the following command:

```
docker exec -it [container_id] bash
```

The **`-it`** flags let you interact with a container by allocating a pseudo-tty and keeping stdin open. Notice bash ran in the **`WORKDIR`** directory (/app) specified in the **`Dockerfile`**. From here, you have an interactive shell session inside the container to debug.

Output:

```
root@xxxxxxxxxxxx:/app#
```

3. Look at the directory

```
ls
```

Output:

```
Dockerfile  app.js
```

4. Exit the Bash session:

```
exit
```

5. You can examine a container's metadata in Docker by using Docker inspect:

```
docker inspect [container_id]
```

Output:

```
[
    {
        "Id": "xxxxxxxxxxxx....",
        "Created": "2017-08-07T22:57:49.261726726Z",
        "Path": "node",
        "Args": [
            "app.js"
        ],
...
```

6. Use `--format` to inspect specific fields from the returned JSON. For example:

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' [container_id]
```

Example output:

```
192.168.9.3
```

Be sure to check out the following **Docker documentation** resources for more information on debugging:

- Docker inspect reference
- Docker exec reference

Task 5. Publish

Now you're going to push your image to the Google Artifact Registry. After that you'll remove all containers and images to simulate a fresh environment, and then pull and run your containers. This will demonstrate the portability of Docker containers.

To push images to your private registry hosted by Artifact Registry, you need to tag the images with a registry name. The format is `<regional-repository>-docker.pkg.dev/my-project/my-repo/my-image`.

Create the target Docker repository

You must create a repository before you can push any images to it. Pushing an image can't trigger creation of a repository and the Cloud Build service account does not have permissions to create repositories.

1. From the **Navigation Menu**, under CI/CD navigate to **Artifact Registry** > **Repositories**.
2. Click the **+CREATE REPOSITORY** icon next to repositories.
3. Specify `my-repository` as the repository name.
4. Choose **Docker** as the format.
5. Under Location Type, select **Region** and then choose the location: `REGION`.
6. Click **Create**.

Configure authentication

Before you can push or pull images, configure Docker to use the Google Cloud CLI to authenticate requests to Artifact Registry.

1. To set up authentication to Docker repositories in the region `REGION`, run the following command in Cloud Shell:

```
```

```
gcloud auth configure-docker "REGION"-docker.pkg.dev
```

2. Enter `Y`, if prompted.

The command updates your Docker configuration. You can now connect with Artifact Registry in your Google Cloud project to push and pull images.

Push the container to Artifact Registry

1. Run the following commands to set your Project ID and change into the directory with your Dockerfile.

```
export PROJECT_ID=$(gcloud config get-value project)

cd ~/test
```

2. Run the command to tag **`node-app:0.2`**.

```
docker build -t
"REGION"-docker.pkg.dev/$PROJECT_ID/my-repository/node-app:0.2 .
```

3. Run the following command to check your built Docker images.

```
docker images
```

Output:

```
REPOSITORY                      TAG         IMAGE ID          CREATED
```

```
node-app                          0.2          76b3beef845e      22 hours

"REGION"-....node-app:0.2         0.2          76b3beef845e      22 hours

node-app                          0.1          f166cd2a9f10      26 hours

hello-world                       latest       1815c82652c0      7 weeks
```

4. Push this image to Artifact Registry.

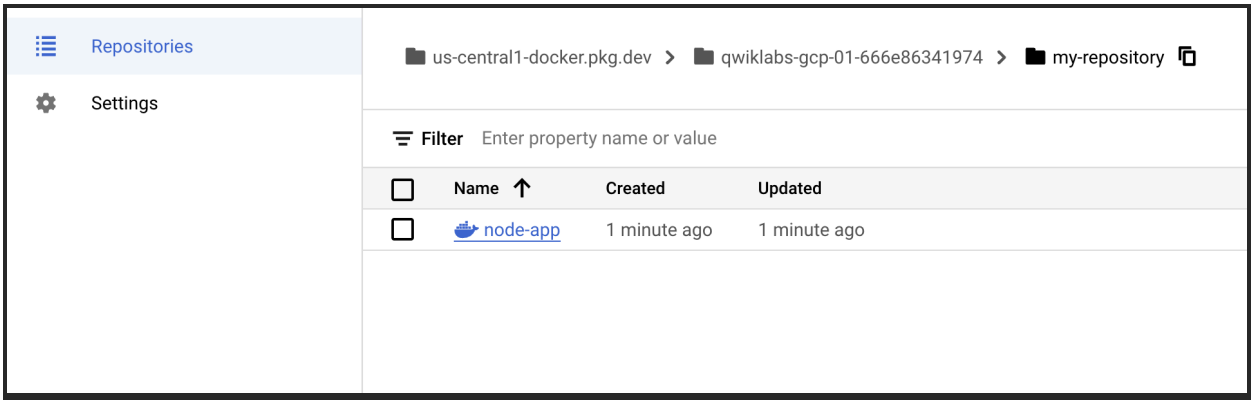
```
1

docker push "REGION"-docker.pkg.dev/$PROJECT_ID/my-repository/node-app:0.2
```

Output (yours may differ):

```
1

2

3

4

5

6

7

8

9

10

11

12

The push refers to a repository
["REGION"-docker.pkg.dev/[project-id]/my-repository/node-app:0.2]
```

```
057029400a4a: Pushed

342f14cb7e2b: Pushed

903087566d45: Pushed

99dac0782a63: Pushed

e6695624484e: Pushed

da59b99bbd3b: Pushed

5616a6292c16: Pushed

f3ed6cb59ab0: Pushed

654f45ecb7e3: Pushed

2c40c66f7667: Pushed

0.2: digest:
sha256:25b8ebd7820515609517ec38dbca9086e1abef3750c0d2aff7f341407c743c46
size: 2419
```

5. After the build finishes, from the **Navigation Menu**, under CI/CD navigate to **Artifact Registry > Repositories**.
6. Click on **my-repository**. You should see your `node-app` Docker container created:



Test the image

You could start a new VM, ssh into that VM, and install gcloud. For simplicity, just remove all containers and images to simulate a fresh environment.

1. Stop and remove all containers:

```
docker stop $(docker ps -q)

docker rm $(docker ps -aq)
```

2. Run the following command to remove all of the Docker images.

```
docker rmi "REGION"-docker.pkg.dev/$PROJECT_ID/my-repository/node-app:0.2

docker rmi -f $(docker images -aq) # remove remaining images

docker images
```

Output:

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
```

At this point you should have a pseudo-fresh environment.

3. Pull the image and run it.

```
docker pull "REGION"-docker.pkg.dev/$PROJECT_ID/my-repository/node-app:0.2

docker run -p 4000:80 -d
"REGION"-docker.pkg.dev/$PROJECT_ID/my-repository/node-app:0.2
```

4. Run a curl against the running container.

```
curl http://localhost:4000
```

Output:

```
Welcome to Cloud
```

Test completed task

Here the portability of containers is showcased. As long as Docker is installed on the host (either on-premise or VM), it can pull images from public or private registries and run containers based on that image. There are no application dependencies that have to be installed on the host except for Docker.

Congratulations!

Congratulations on completing the Introduction to Docker. To recap, you:

- Ran containers based on public images from Docker Hub.
- Built your own container images and pushed them to Google Artifact Registry.
- Learned ways to debug running containers.
- Ran containers based on images pulled from Google Artifact Registry.

Mark as completed
Like
Dislike
Report an issue

Module 2 review

Congratulations on completing Module 2 on Docker & Kubernetes for Course 5. You've learned more about Docker and Kubernetes and about using these platforms to run applications in containers. This module walked you through the steps for installing Docker and Kubernetes to your machine—if they weren't installed already—and explored different attributes of each.
You explored Docker images and their importance in enabling programmers to package, distribute, and deploy applications efficiently. In addition, you discovered why developers use multiple containers with a large application to test the entirety of an application. Speaking of testing, you examined build artifacts and the importance of testing them to troubleshoot any issues discovered to ensure a quality product. Build artifacts are what you build during the development process and include libraries, documentation, and scripts, with the main artifact being your Docker container. Typically, developers use GCP while working with Docker containers as it provides services to support containerized applications.
Kubernetes is another type of container you explored that provides developers with choice and flexibility when building platforms. It's designed to support developers with starting, stopping,

storing, building, and managing containers. Kubernetes includes pods that are the fundamental deployment unit in a cluster, and they contain one or more containers. You learned that services offer an abstract layer over pods. In addition, services provide a stable virtual IP and DNS name for each set of related pods. You examined Kubernetes deployment and how it acts as your application's manager and is responsible for keeping your application up and running smoothly. And lastly, you looked at how to deploy Docker containers and Kubernetes clusters to GCP. Different types of clusters include: on-premises, public and private cloud managed, local, and hybrid clusters. Kubernetes clusters are robust, flexible, and reliable units of multiple nodes that work together. You looked at the benefits of deploying and scaling Docker containers on GCP and how Kubernetes YAML files define and configure Kubernetes resources. Scaling containers allow developers to add containers as needed or allocate additional resources to each container. YAML files allow developers to manage their applications' infrastructure in a consistent and automated manner. You explored the importance of container security and actions you should take to secure your workload and network, and you also explored the actions GCP takes to secure infrastructure and operations. In addition, you learned about the best practices provided by GCP for protecting the containers you work with. Now you're ready to master the Module 2 assessment and then move on to the next module! Good luck!

Glossary terms from course 5, module 2

Terms and definitions from Course 5, Module 2

Artifact: A byproduct of the software development process that can be accessed and used, an item produced during programming
Container registry: A storage location for container images, organized for efficient access
Container repository: A container registry that manages container images
Docker: An open-source tool used to build, deploy, run, update, and manage containers
Pod: A group of one or more containers that are scheduled and run together
Registry: A place where containers or artifacts are stored and organized
Kubernetes: An open-source platform that gives programmers the power to orchestrate containers

IaC options

As you delve into the realm of configuration management, it's essential to explore the diverse array of Infrastructure as Code (IaC) tools at your disposal. While Puppet serves as a robust and well-established solution, several other options bring their unique strengths to the table. Let's take a closer look at these alternatives, including Terraform, Ansible, and Google Cloud Platform (GCP) offerings, and how they compare to using Puppet.

Terraform

Terraform stands out as a potent IaC tool that specializes in provisioning and managing infrastructure resources across various cloud providers. Its declarative syntax allows you to define your desired infrastructure state, and Terraform takes care of translating this into

concrete resources. This approach enables you to codify your infrastructure configurations, fostering version control, collaboration, and reproducibility. Terraform's provider ecosystem empowers you to manage a wide spectrum of resources, from virtual machines to databases, across multiple cloud environments. Its focus on infrastructure provisioning aligns well with cloud-native approaches, making it an excellent choice for orchestrating cloud resources and building scalable, modern applications.

Ansible

Unlike Puppet, which revolves around agent-based communication, Ansible adopts an agentless architecture that relies on SSH or other remote APIs for system management. This lightweight approach simplifies deployment and reduces the overhead of maintaining agents on target nodes. Ansible employs a simple and human-readable YAML syntax to define playbooks, which describe the desired state of systems. These playbooks facilitate a wide range of automation tasks, from configuration management to application deployment. Ansible's versatility extends beyond servers to network devices, making it suitable for managing diverse IT environments. While it may lack the advanced features of Puppet's catalog-based system, Ansible excels in its simplicity, ease of adoption, and suitability for rapid deployment scenarios.

Google Cloud Platform alternatives

Within the realm of Google Cloud Platform (GCP), you can leverage native tools for configuration management and infrastructure orchestration. Google Cloud Deployment Manager enables you to define your infrastructure using YAML or Python templates, offering a declarative approach similar to Terraform. This tool is well-integrated with GCP services and resources, simplifying the orchestration of cloud-specific components like GKE clusters, Cloud Storage Buckets, and load balancers. Additionally, GCP provides a wide range of managed services that abstract away much of the infrastructure management complexity, allowing you to focus more on application development and less on provisioning and configuration.

Comparing to Puppet

While Puppet excels in its ability to manage configuration drift and ensure system consistency through its catalog-based approach, other IaC tools offer unique advantages. Terraform's focus on provisioning cloud resources aligns well with modern, cloud-native development practices. Ansible's agentless architecture simplifies deployment and is well-suited for quick automation tasks across diverse environments. GCP's native tools provide seamless integration within the Google Cloud ecosystem, streamlining infrastructure management for projects hosted on the platform. Ultimately, the choice between these options depends on your specific needs, preferences, and the ecosystem you are operating within.

Key takeaways

Exploring the landscape of IaC options beyond Puppet opens up a world of possibilities for configuration management and infrastructure orchestration. Each tool brings its strengths, from Terraform's cloud provisioning prowess to Ansible's lightweight automation and GCP's native integration. By understanding the nuances of these alternatives, you can make informed decisions that align with your project's goals and requirements, contributing to more efficient, scalable, and reliable infrastructure management.

```
class sudo {

  package { 'sudo':
    ensure => present,
  }
}
class sysctl {
  # Make sure the directory exists, some distros don't have it
  file { '/etc/sysctl.d':
    ensure => directory,
  }
}
class timezone {
      file { '/etc/timezone':
        ensure  => file,
        content => "UTC\n",
        replace => true,
      }
}
class ntp {
  package { 'ntp':
    ensure => latest,
  }
  file { '/etc/ntp.conf':
    source => 'puppet:///modules/ntp/ntp.conf'
    replace => true,
  }
  service { 'ntp':
    enable  => true,
    ensure  => running,
  }
}
vim ntp.pp
```

```
class ntp {
  package { 'ntp':
    ensure => latest,
  }
  file { '/etc/ntp.conf':
    source => '/home/user/ntp.conf',
    replace => true,
    require => Package['ntp'],
    notify  => Service['ntp'],
  }
  service { 'ntp':
    enable  => true,
    ensure  => running,
    require => File['/etc/ntp.conf'],
  }
}
include ntp
```

About this code

View the **`ntp.pp`** manifest file by entering editing mode with the **`vim`** command. This file contains resources related to the NTP configuration: the ntp package, the ntp configuration file, and ntp service.

The relationships between these resources are also included in the code. In Puppet, code that defines a resource begins with a lowercase letter, whereas code that defines a relationship begins with a capital letter. The ntp configuration file requires the ntp package, which is indicated with the code **`require => Package['ntp']`**. The ntp service requires the ntp configuration file, which is indicated with the code **`require => File['/etc/ntp.conf']`**. Additionally, the ntp configuration file resource notifies the ntp service when it is present, which is indicated by the code **`notify  => Service['ntp']`**.

The line **`include ntp`** indicates that the code will include the rules in this file as a class.

```
vim ntp.conf
```

```
driftfile /var/lib/ntp/ntp.drift
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable

server 0.pool.ntp.org
server 1.pool.ntp.org
```

```
server 2.pool.ntp.org
server 3.pool.ntp.org

restrict -4 default kod notrap nomodify nopeer noquery limited
restrict -6 default kod notrap nomodify nopeer noquery limited

restrict 127.0.0.1
restrict ::1

restrict source notrap nomodify noquery
```

Exemplar: Finishing a Puppet deployment

Introduction

In the previous lab, you wanted to automatically manage the computers in your company's fleet, including a number of different machines with different operating systems. You decided to use Puppet to automate the configurations on these machines. Part of the setup was already done, but there were more rules that need to be added, and more operating systems to consider.
This exemplar is a walkthrough of the previous Qwiklab activity, including detailed instructions and solutions. You may use this exemplar if you were unable to complete the lab and/or you need extra guidance in competing lab tasks. You may also refer to this exemplar to prepare for the graded quiz in this module.

Puppet rules

The goal of this exercise is for you to see what Puppet looks like in action. During this lab, you'll be connecting to two different VMs. The VM named **puppet** is the Puppet Master that has the Puppet rules that you'll need to edit. The VM named **linux-instance** is a client VM that you'll use to test that your catalog was applied successfully.
The manifests used for the production environment are located in the directory **`/etc/puppet/code/environments/production/manifests`**, which contains a **site.pp** file with the node definitions that will be used for this deployment. On top of that, the modules directory contains a bunch of modules that are already in use. You'll be extending the code of this deployment to add more functionality to it.

Install packages

There's a module named **packages** on the **Puppet VM instance** that takes care of installing the packages that are needed on the machines in the fleet. Use the command to visit the module:

```
cd /etc/puppet/code/environments/production/modules/packages
```

This module already has a resource entry specifying that **python-requests** is installed on all machines. You can see the **`init.pp`** file using the **cat** command on the **Puppet VM instance.**

```
cat manifests/init.pp
```

Output:

```
class packages {

    package { 'python-requests':

        ensure => installed,

    }



}
```

Now, add an additional resource in the same **`init.pp`** file within the path **`/etc/puppet/code/environments/production/modules/packages`**, ensuring the **golang** package gets installed on all machines that belong to the **`Debian`** family of operating systems (which includes Debian, Ubuntu, LinuxMint, and a bunch of others).
This resource will be very similar to the previous **python-requests** one. Add edit permission to the file before moving forward using:

```
sudo chmod 646 manifests/init.pp
```

To install the package on Debian systems only, you'll need to use the **os family** fact, like this:

```
if $facts[os][family] == "Debian" {

# Resource entry to install golang package

}
```

Now, open the file using nano editor and add the resource entry specifying golang package to be installed on all machines of Debian family after the previous resource entry.
The snippet would now look like this:

```
if $facts[os][family] == "Debian" {

    package { 'golang':

      ensure => installed,

    }

  }
```

The complete **init.pp** file would now look similar to the below file:

```
class packages {
   package { 'python-requests':
       ensure => installed,
   }
   if $facts[os][family] == "Debian" {
     package { 'golang':
       ensure => installed,
     }
  }
}
```

After this, we will also need to ensure that the **nodejs** package is installed on machines that belong to the `RedHat` family. Refer to the below snippet for this.

```
if $facts[os][family] == "RedHat" {

  #Resource entry

}
```

Complete the above snippet just like the previous one.
The complete **init.pp** file should now look like this:

```
class packages {

   package { 'python-requests':
```

```
      ensure => installed,

  }

  if $facts[os][family] == "Debian" {

    package { 'golang':

      ensure => installed,

    }

  }

  if $facts[os][family] == "RedHat" {

    package { 'nodejs':

      ensure => installed,

    }

  }

}
```

Once you've edited the file and added the necessary resources, you'll want to check that the rules work successfully. We can do this by connecting to another machine in the network and verifying that the right packages are installed.

We will be connecting to **linux-instance** using its external IP address. To fetch the external IP address of **linux-instance**, use the following command:

```
gcloud compute instances describe linux-instance --zone="Zone"
--format='get(networkInterfaces[0].accessConfigs[0].natIP)'
```

This command outputs the external IP address of **linux-instance**. Copy the **linux-instance** external IP address, open another terminal and connect to it. Follow the instructions given in the section **`Accessing the virtual machine`** by clicking on **`Accessing the virtual machine`** from the navigation pane at the right side.

Now manually run the Puppet client on your **linux-instance** VM instance terminal:

```
```

```
sudo puppet agent -v --test
```

This command should run successfully and the catalog should be applied.
Output:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
2023-10-17 10:38:21.161341 WARN  puppetlabs.facter - locale environment variables were bad; continuing with LANG=C LC_ALL=C

Info: Using configured environment 'production'

Info: Retrieving pluginfacts

Info: Retrieving plugin

Info: Retrieving locales

Info: Caching catalog for linux-instance.us-central1-a.c.qwiklabs-gcp-03-73433a2333b1.internal

Info: Applying configuration version '1697539102'

Notice: /Stage[main]/Packages/Package[golang]/ensure: created

Notice: /Stage[main]/Machine_info/File[/tmp/machine_info.txt]/content:

--- /tmp/machine_info.txt   2023-10-17 10:33:25.188341331 +0000

+++ /tmp/puppet-file20231017-2844-1f6npxn   2023-10-17 10:39:01.789374002 +0000

@@ -1,6 +1,6 @@

 Machine Information

 -------------------
```

```
 Disks: {"sda"=>{"model"=>"PersistentDisk", "size"=>"10.00 GiB",
"size_bytes"=>10737418240, "vendor"=>"Google"}}

-Memory: {"system"=>{"available"=>"3.63 GiB",
"available_bytes"=>3901550592, "capacity"=>"5.71%", "total"=>"3.85 GiB",
"total_bytes"=>4137762816, "used"=>"225.27 MiB", "used_bytes"=>236212224}}

+Memory: {"system"=>{"available"=>"3.63 GiB",
"available_bytes"=>3897368576, "capacity"=>"5.81%", "total"=>"3.85 GiB",
"total_bytes"=>4137762816, "used"=>"229.26 MiB", "used_bytes"=>240394240}}

 Processors: {"count"=>2, "isa"=>"unknown", "models"=>["Intel(R) Xeon(R)
CPU @ 2.20GHz", "Intel(R) Xeon(R) CPU @ 2.20GHz"], "physicalcount"=>1}

 }



Info: Computing checksum on file /tmp/machine_info.txt

Info: /Stage[main]/Machine_info/File[/tmp/machine_info.txt]: Filebucketed
/tmp/machine_info.txt to puppet with sum d30f80b5fe7b675290df24547d8ec410

Notice: /Stage[main]/Machine_info/File[/tmp/machine_info.txt]/content:
content changed '{md5}d30f80b5fe7b675290df24547d8ec410' to
'{md5}ea6a5de087b843d62eb6a4afe74b61a9'

Notice: Applied catalog in 38.40 seconds
```

Now verify whether the **golang** package was installed on this instance. This being an machine of the Debian family should have golang installed. Use the following command to verify this:

```
1


apt policy golang
```

Output:

```
1

2

3
```

```
golang:
  Installed: 2:1.11~1
  Candidate: 2:1.11~1
  Version table:
     2:1.15~1~bpo10+1 100
        100 http://deb.debian.org/debian buster-backports/main amd64
Packages
 *** 2:1.11~1 500
        500 http://deb.debian.org/debian buster/main amd64 Packages
        100 /var/lib/dpkg/status
```

With this, you've seen how you can use Puppet's facts and package resources to install specific packages on machines within your fleet.

Fetch machine information

It's now time to navigate to the **machine_info** module in our Puppet environment. In the **Puppet VM terminal,** navigate to the module using the following command:

```
cd /etc/puppet/code/environments/production/modules/machine_info
```

The **machine_info** module gathers some information from the machine using **Puppet** facts and then stores it in a file. Currently, the module is always storing this information in **`/tmp/machine_info`**.
Let's check this out:

```
cat manifests/init.pp
```

Output:

```
class machine_info {

    file { '/tmp/machine_info.txt':

        content => template('machine_info/info.erb'),

    }



}
```

You can view the path in the above file. This path doesn't work for Windows machines. So, you need to adapt this rule for Windows.
Add edit permission to the file using the following command before we adapt the rule.

```
sudo chmod 646 manifests/init.pp
```

Now we will be using **`$facts[kernel]`** fact to check if the kernel is "windows". If so, set a **$info_path** variable to **`C:\Windows\Temp\Machine_Info.txt`**, otherwise set it to

`/tmp/machine_info.txt`. To do this, open the file using nano editor and add the below rule after the default path within the class `machine_info`.

```
  if $facts[kernel] == "windows" {
        $info_path = "C:\Windows\Temp\Machine_Info.txt"
  } else {
        $info_path = "/tmp/machine_info.txt"
  }
```

The file should now look similar to:

```
class machine_info {

    file { '/tmp/machine_info.txt':

        content => template('machine_info/info.erb'),

    }

    if $facts[kernel] == "windows" {

        $info_path = "C:\Windows\Temp\Machine_Info.txt"

    } else {

        $info_path = "/tmp/machine_info.txt"

    }

}
```

By default the file resources are stored in the path defined in the name of the resource (the string in the first line) within the class. We can also set different paths, by setting the path attribute.
We will now be renaming the resource to **`machine_info`** and then use the variable in the path attribute. The variable we are using to store the path in the above rule is **$info_path**.
Remove the following part from the file **manifests/init.pp**.

```
1
2
3
    file { '/tmp/machine_info.txt':

        content => template('machine_info/info.erb'),

    }
```

And add the following resource after the rule within the class definition:

```
1
2
```

```
    file { 'machine_info':
            path => $info_path,
            content => template('machine_info/info.erb'),
        }
```

The complete manifests/init.pp file should now look like this:

```
class machine_info {
  if $facts[kernel] == "windows" {
        $info_path = "C:\Windows\Temp\Machine_Info.txt"
```

```
    } else {

        $info_path = "/tmp/machine_info.txt"

    }

 file { 'machine_info':

        path => $info_path,

        content => template('machine_info/info.erb'),

    }

}
```

Puppet Templates

Templates are documents that combine code, data, and literal text to produce a final rendered output. The goal of a template is to manage a complicated piece of text with simple inputs.
In Puppet, you'll usually use templates to manage the content of configuration files (via the content attribute of the file resource type).
Templates are written in a templating language, which is specialized for generating text from data.
Puppet supports two templating languages:

- **Embedded Puppet (EPP)** uses Puppet expressions in special tags. It's easy for any Puppet user to read, but only works with newer Puppet versions. (≥ 4.0, or late 3.x versions with future parser enabled.)
- **Embedded Ruby (ERB)** uses Ruby code in tags. You need to know a small bit of Ruby to read it, but it works with all Puppet versions.

Now, take a look at the template file using the following command.

```
cat templates/info.erb
```

Puppet templates generally use data taken from Puppet variables. Templates are files that are pre-processed, some values gets replaced with variables. In this case, the file currently includes the values of three facts. We will be adding a new fact in this file now.
Add edit permissions to the file using **`templates/info.erb`** using the following command:

```
sudo chmod 646 templates/info.erb
```

Now open the file using nano editor and add the following fact just after the last fact within the file:

```
Network Interfaces: <%= @interfaces %>
```

The template should now look like this:

```
Machine Information
-------------------
Disks: <%= @disks %>
Memory: <%= @memory %>
Processors: <%= @processors %>
Network Interfaces: <%= @interfaces %>
}
```

To check if this worked correctly, return to **linux-instance** VM terminal and manually run the client on that machine using the following command:

```
sudo puppet agent -v --test
```

This command should run successfully and the catalog should be applied.
Now verify that the **machine_info** file has the required information using:

```
cat /tmp/machine_info.txt
```

Output:

```
Machine Information

--------------------

Disks: {"sda"=>{"model"=>"PersistentDisk", "size"=>"10.00 GiB",
"size_bytes"=>10737418240, "vendor"=>"Google"}}

Memory: {"system"=>{"available"=>"3.59 GiB", "available_bytes"=>3853631488,
"capacity"=>"6.87%", "total"=>"3.85 GiB", "total_bytes"=>4137762816,
"used"=>"270.97 MiB", "used_bytes"=>284131328}}

Processors: {"count"=>2, "isa"=>"unknown", "models"=>["Intel(R) Xeon(R) CPU
@ 2.20GHz", "Intel(R) Xeon(R) CPU @ 2.20GHz"], "physicalcount"=>1}

Network Interfaces: ens4,lo

}
```

And with that, you've seen how you can fetch machine information and store it according to the operating system.

Reboot machine

For the last exercise, we will be creating a new module named **reboot**, that checks if a node has been online for more than **30 days**. If so, then reboot the computer.

To do that, you'll start by creating the module directory.
Switch back to **puppet** VM terminal and run the following command:

```
sudo mkdir -p /etc/puppet/code/environments/production/modules/reboot/manifests
```

Go to the **`manifests/`** directory.

```
cd /etc/puppet/code/environments/production/modules/reboot/manifests
```

Create an **init.pp** file for the reboot module in the **`manifests/`** directory.

```
sudo touch init.pp
```

Open **init.pp** with nano editor using **`sudo`**.

```
sudo nano init.pp
```

In this file, you'll start by creating a class called **`reboot`**.
The way to reboot a computer depends on the OS that it's running. So, you'll set a variable that has one of the following reboot commands, based on the kernel fact:

- **shutdown /r** on windows
- **shutdown -r now** on Darwin (macOS)
- **reboot** on Linux.

Hence, add the following snippet in the file **init.pp**:

```
class reboot {
  if $facts[kernel] == "windows" {
    $cmd = "shutdown /r"
  } elsif $facts[kernel] == "Darwin" {
    $cmd = "shutdown -r now"
  } else {
    $cmd = "reboot"
  }
}
```

With this variable defined, we will now define an exec resource that calls the command, but only when the **uptime_days** fact is larger than 30 days.
Add the following snippet after the previous one within the class definition in the file **reboot/manifests/init.pp**:

```
if $facts[uptime_days] > 30 {
        exec { 'reboot':
            command => $cmd,
```

```
        }

    }
```

The complete **reboot/manifests/init.pp** should now look like this:

```
class reboot {

  if $facts[kernel] == "windows" {

    $cmd = "shutdown /r"

  } elsif $facts[kernel] == "Darwin" {
```

```
    $cmd = "shutdown -r now"

  } else {

    $cmd = "reboot"

  }

  if $facts[uptime_days] > 30 {

    exec { 'reboot':

      command => $cmd,

    }

  }

}
```

Finally, to get this module executed, make sure to include it in the **`site.pp`** file.
So, edit **`/etc/puppet/code/environments/production/manifests/site.pp`** using the following command:

```
sudo nano /etc/puppet/code/environments/production/manifests/site.pp
```

Add an extra line that includes the reboot module.
The file **`/etc/puppet/code/environments/production/manifests/site.pp`** should now look like this:

```
node default {
```

```
    class { 'packages': }

    class { 'machine_info': }

    class { 'reboot': }

}
```

Run the client on **linux-instance** VM terminal:

```
sudo puppet agent -v --test
```

Output:

```
Info: Using configured environment 'production'

Info: Retrieving pluginfacts

Info: Retrieving plugin

Info: Caching catalog for
linux-instance.us-central1-a.c.qwiklabs-gcp-03-73433a2333b1.internal

Info: Applying configuration version '1697540321'

Notice: Applied catalog in 0.10 seconds
```

And with that, you've added a whole new module to your deployment!

Congratulations!

Woohoo! You've successfully improved the Puppet deployment, by modifying existing modules and adding new ones!

Troubleshooting commands

- Sudo systemctl status apache2
- Sudo systemctl restart apache2
- Sudo netstat -nlp
- ps -ax | grep python3
- sudo systemctl --type=service | grep jimmy
- sudo systemctl stop jimmytest && sudo systemctl disable jimmytest
- sudo systemctl start apache2

Terms and definitions from Course 5, Module 3

Configuration management: Automation technique that manages the configuration of computers at scale
Domain-Specific Language (DSL): A programming language that's more limited in scope
Facts: Variables that represent the characteristics of the system
Puppet: The current industry standard for configuration management, also known as the client
Puppet master: Known as the Puppet server

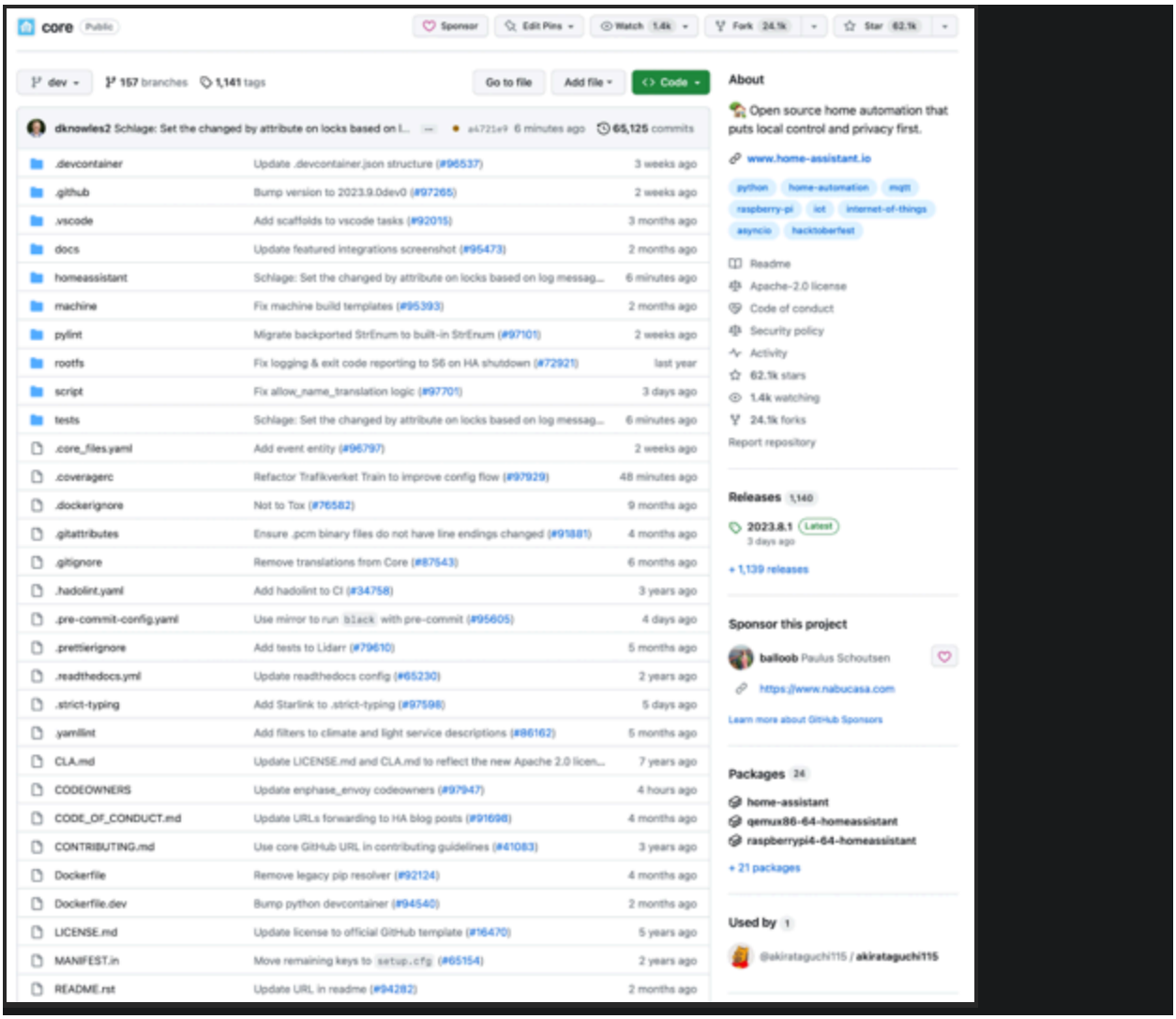
Continuous integration, delivery, and deployment

Continuous integration, delivery, and deployment, or CI/CD, refers to the automation of an entire pipeline of tools that build, test, package, and deploy an application whenever a developer commits a code change to the source control repository.
Continuous integration (CI) automatically builds, tests, and integrates code changes within a shared repository. Then, continuous delivery (CD) automatically delivers code changes to production-ready environments for approval, and continuous deployment (CD) automatically deploys those code changes directly to production.

Continuous integration

CI is a key best practice in DevOps. As the first stage of the automation process in the development lifecycle, developers integrate their code changes early and often to a shared source code repository. This shared repository is the solution to having too many separate iterations of a software or app in development at the same time. It can also:

- Reduce the risk of having multiple pieces—which may be created independently—conflicting with each other
- Save time throughout the development lifecycle by allowing you to identify and address any issues or conflicts as they arise rather than at the end of a phase
- Reduce the amount of time spent on fixing bugs and regression


core
Public
Sponsor
Edit Pins
Watch 1.4k
Fork 24.1k
Star 62.1k
dev
157 branches
1,141 tags
Go to file
Add file
Code
dknowles2 Schlage: Set the changed by attribute on locks based on l...
a4721e9 6 minutes ago
65,125 commits
.devcontainer Update .devcontainer.json structure (#96537) 3 weeks ago
.github Bump version to 2023.9.0dev0 (#97265) 2 weeks ago
.vscode Add scaffolds to vscode tasks (#92015) 3 months ago
docs Update featured integrations screenshot (#95473) 2 months ago
homeassistant Schlage: Set the changed by attribute on locks based on log messag... 6 minutes ago
machine Fix machine build templates (#95393) 2 months ago
pylint Migrate backported StrEnum to built-in StrEnum (#97101) 2 weeks ago
rootfs Fix logging & exit code reporting to S6 on HA shutdown (#72921) last year
script Fix allow_name_translation logic (#97701) 3 days ago
tests Schlage: Set the changed by attribute on locks based on log messag... 6 minutes ago
.core_files.yaml Add event entity (#96797) 2 weeks ago
.coveragerc Refactor Trafikverket Train to improve config flow (#97929) 48 minutes ago
.dockerignore Not to Tox (#76582) 9 months ago
.gitattributes Ensure .pcm binary files do not have line endings changed (#91881) 4 months ago
.gitignore Remove translations from Core (#87543) 6 months ago
.hadolint.yaml Add hadolint to CI (#34758) 3 years ago
.pre-commit-config.yaml Use mirror to run black with pre-commit (#95605) 4 days ago
.prettierignore Add tests to Lidarr (#79610) 5 months ago
.readthedocs.yml Update readthedocs config (#65230) 2 years ago
.strict-typing Add Starlink to .strict-typing (#97596) 5 days ago
.yamllint Add filters to climate and light service descriptions (#86162) 5 months ago
CLA.md Update LICENSE.md and CLA.md to reflect the new Apache 2.0 licen... 7 years ago
CODEOWNERS Update enphase_envoy codeowners (#97947) 4 hours ago
CODE_OF_CONDUCT.md Update URLs forwarding to HA blog posts (#91698) 4 months ago
CONTRIBUTING.md Use core GitHub URL in contributing guidelines (#41083) 3 years ago
Dockerfile Remove legacy pip resolver (#92124) 4 months ago
Dockerfile.dev Bump python devcontainer (#94540) 2 months ago
LICENSE.md Update license to official GitHub template (#16470) 5 years ago
MANIFEST.in Move remaining keys to setup.cfg (#65154) 2 years ago
README.rst Update URL in readme (#94282) 2 months ago
About
Open source home automation that puts local control and privacy first.
www.home-assistant.io
python
home-automation
mqtt
raspberry-pi
iot
internet-of-things
asyncio
hacktoberfest
Readme
Apache-2.0 license
Code of conduct
Security policy
Activity
62.1k stars
1.4k watching
24.1k forks
Report repository
Releases 1,140
2023.8.1 Latest
3 days ago
+ 1,139 releases
Sponsor this project
balloob Paulus Schoutsen
https://www.nabucasa.com
Learn more about GitHub Sponsors
Packages 24
home-assistant
qemux86-64-homeassistant
raspberrypi4-64-homeassistant
+ 21 packages
Used by 1
@akirataguchi115 / akirataguchi115

July 7, 2023 – August 7, 2023

Period: 1 month

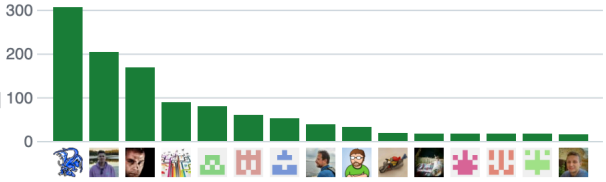
Overview

1,142 Active pull requests

1,286 Active issues

999	143	863	423
Merged pull requests	Open pull requests	Closed issues	New issues

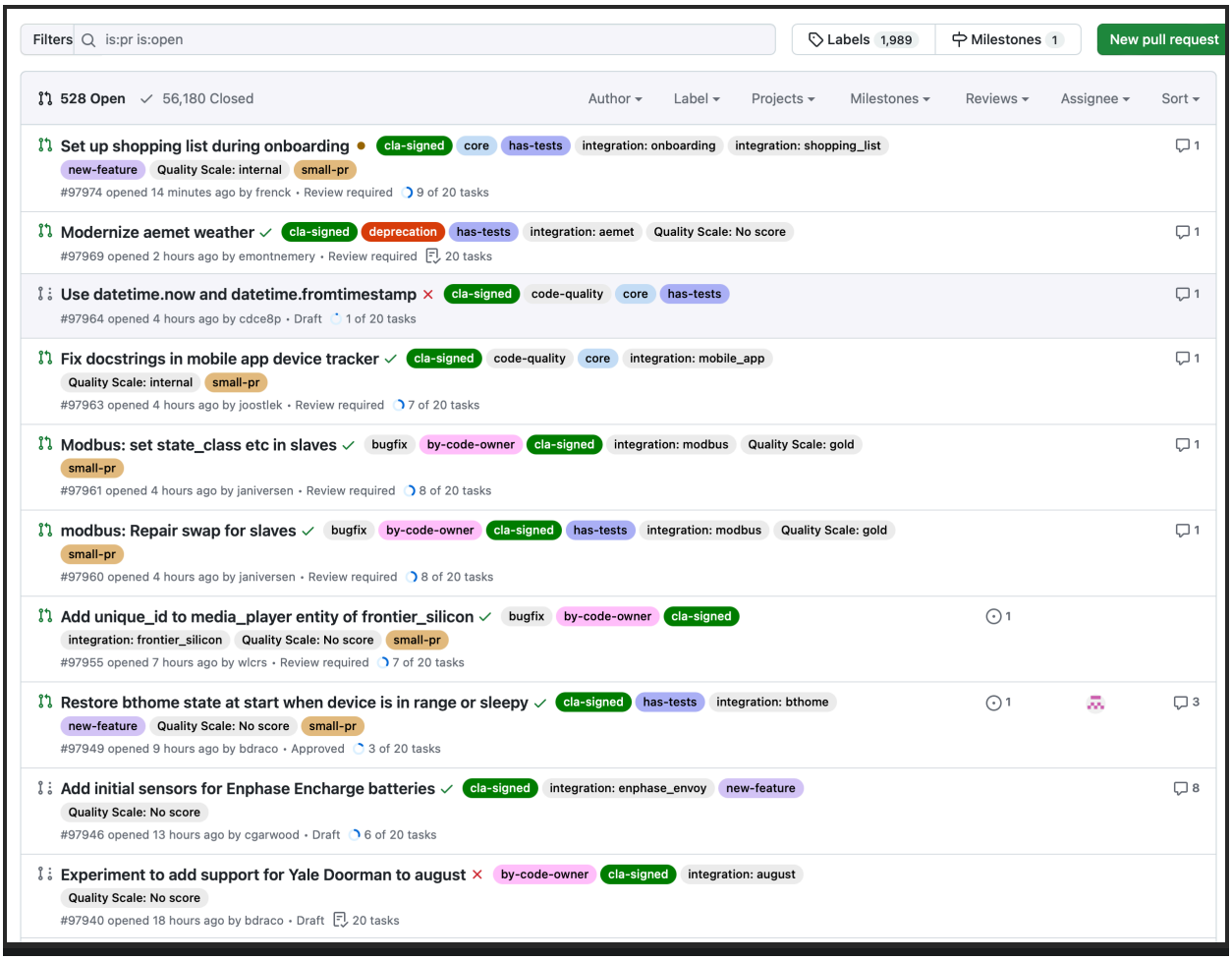
Excluding merges, **139 authors** have pushed **1,131 commits** to dev and **1,358 commits** to all branches. On dev, **2,523 files** have changed and there have been **67,115 additions** and **23,458 deletions**.

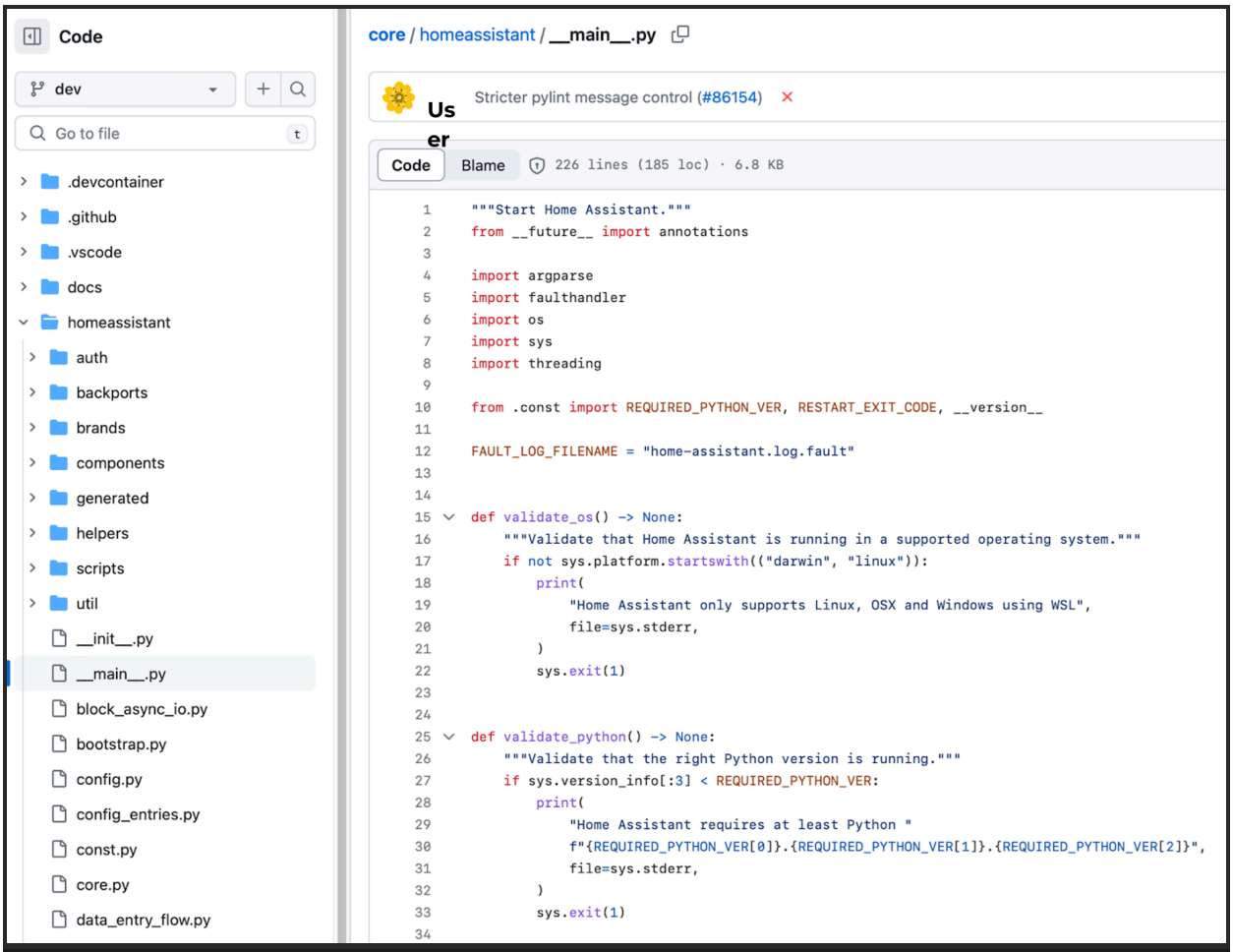


9 Releases published by **2** people

2023.7.2
published last month

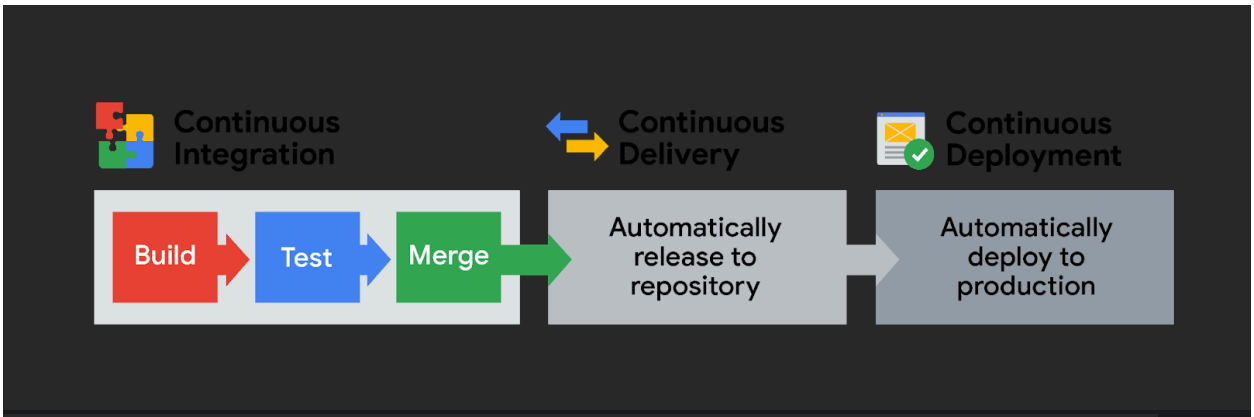
2023.7.3
published 2 weeks ago


Filters
is:pr is:open
Labels 1,989
Milestones 1
New pull request
528 Open
56,180 Closed
Author
Label
Projects
Milestones
Reviews
Assignee
Sort
Set up shopping list during onboarding
cla-signed core has-tests integration: onboarding integration: shopping_list new-feature Quality Scale: internal small-pr
#97974 opened 14 minutes ago by frenck • Review required 9 of 20 tasks
1
Modernize aemet weather
cla-signed deprecation has-tests integration: aemet Quality Scale: No score
#97969 opened 2 hours ago by emontnemery • Review required 20 tasks
1
Use datetime.now and datetime.fromtimestamp
cla-signed code-quality core has-tests
#97964 opened 4 hours ago by cdce8p • Draft 1 of 20 tasks
1
Fix docstrings in mobile app device tracker
cla-signed code-quality core integration: mobile_app Quality Scale: internal small-pr
#97963 opened 4 hours ago by joostlek • Review required 7 of 20 tasks
1
Modbus: set state_class etc in slaves
bugfix by-code-owner cla-signed integration: modbus Quality Scale: gold small-pr
#97961 opened 4 hours ago by janiversen • Review required 8 of 20 tasks
1
modbus: Repair swap for slaves
bugfix by-code-owner cla-signed has-tests integration: modbus Quality Scale: gold small-pr
#97960 opened 4 hours ago by janiversen • Review required 8 of 20 tasks
1
Add unique_id to media_player entity of frontier_silicon
bugfix by-code-owner cla-signed integration: frontier_silicon Quality Scale: No score small-pr
#97955 opened 7 hours ago by wlcrs • Review required 7 of 20 tasks
1
Restore bthome state at start when device is in range or sleepy
cla-signed has-tests integration: bthome new-feature Quality Scale: No score small-pr
#97949 opened 9 hours ago by bdraco • Approved 3 of 20 tasks
1
3
Add initial sensors for Enphase Encharge batteries
cla-signed integration: enphase_envoy new-feature Quality Scale: No score
#97946 opened 13 hours ago by cgarwood • Draft 6 of 20 tasks
8
Experiment to add support for Yale Doorman to august
by-code-owner cla-signed integration: august Quality Scale: No score
#97940 opened 18 hours ago by bdraco • Draft 20 tasks



Continuous delivery and deployment

An important note: Continuous delivery and continuous deployment are related concepts that are sometimes used interchangeably. While they're both about automating later stages of a DevOps pipeline, you can use either to show what is happening during automation. For example, continuous delivery means that any changes a developer makes to an application are automatically released to a repository like GitHub and then deployed by the operations team. This ensures that minimal effort is required to deploy new code. It also includes test and code release automation at every stage, beginning with code changes and ending with the delivery of production-ready builds.
Continuous deployment is an extension of continuous delivery and refers to the automatic deployment of an app or any developer changes from the repository to production. This helps to prevent overloading the operations teams and automates the next stage in the pipeline. However, continuous deployment relies heavily on the success of test automation, so it's important that your testing is written to accommodate the various testing and release stages in the DevOps lifecycle.



steps begin with build, test, and merge, in continuous integration, then move to “automatically release to repository” in continuous delivery, and then end with, “automatically deploy to production” in continuous deployment.

Key takeaways

The idea of CI/CD is to bring efficiency to your software release process so engineers can write code, test, and deploy at any time without impacting your customers. Whenever your team wants to roll out new features either due to business needs, improvement, or hotfixes, as quickly as possible, CI/CD can help you do that in an organized and controlled manner.

xample pipeline

You’ve learned about CI/CD: continuous integration and continuous delivery or continuous deployment, which is the automation of an entire pipeline of tools that build, test, package, and deploy an application whenever a developer commits, or saves, a code change to the source control repository. You’ve also learned about DevOps, the stages of the software development lifecycle, and the DevOps tools that can help you automate the processes in CI/CD. In this reading, you’ll learn about pipelines, which are the steps necessary to build, test, and (optionally) deploy software.

Pipelines

Pipelines are automated processes and sets of tools that developers use during the software development lifecycle. In a pipeline, the steps of a process are carried out in sequential order. This way, if any step fails, the pipeline stops without deploying the changes.
Consider the following example: A pipeline for a Python application is triggered when a pull request is ready to be merged. That pipeline can perform the following steps:

1. Check out the complete branch represented by the pull request.
2. Attempt to build the project by running python `setup.py build`.
3. Run unit tests with `pytest`.
4. Run integration tests against other parts of the application with a framework like `playwright`.
5. Build the documentation and upload it to an internal wiki.
6. Upload the build artifacts to a container registry.
7. Message your team in Slack to let them know the build was successful.

The advantage to automating the CI/CD process is there's much less manual work and coordination required to deploy software, and this saves a tremendous amount of time and effort for your development and operations teams.
Pro tip: The more teams you have working independently, the greater the advantage.

Example pipeline

Now imagine you're part of an organization with many development teams that are working independently but still contributing to the larger application or project. In order to deploy an application successfully, your organization has to:

1. Choose a "release day" when all the code will be merged together.
2. Restrict new code commits until the release is complete, to avoid conflicts.
3. Run integration tests (and maybe performance tests).
4. Prepare the deployment.
5. Notify customers of an upcoming maintenance window.
6. Manually deploy the application and any other updates.

Without automating any (or all) of these steps, the entire process is painful and subject to human error. As a result, many organizations release updates infrequently, e.g., once every few weeks or even months. Imagine the disruption and chaos of "release day" or even "release week" within those organizations! The developers cannot commit any new code, and the operations teams scramble to package and deploy dozens of microservices and other components to meet a deadline.
This is when having a pipeline to automate some or all of the processes during CI/CD comes in handy. By creating a CI pipeline, the process looks very different:

1. Developers commit code to the repository as soon as they're done.
2. The CI server observes the commit and automatically triggers a build pipeline.
3. If the build completes successfully, the CI server runs all the unit tests. If any tests fail, the build stops.
4. The CI server runs integration tests and/or smoke tests, if any.
5. Assuming the previous steps all complete successfully, the CI server signals success. Then, the application is ready to be deployed.
6. If the CD process has also been automated, the code is deployed to production servers.

You can have multiple pipelines run at once, which enables teams across your organization to commit code whenever they're ready. If you've also automated the deployment process (CD), code can be deployed to production without waiting for every other team to prepare for "release day." Some CD pipelines even continue to monitor production for errors, rolling back the deployment if it detects an increase.
Organizations that have high-quality CI/CD pipelines often deploy to production every day, sometimes dozens of times a day, as individual teams release code. The automation of these processes is more efficient, saves you time, and reduces human error. It also provides a better end-product for consumers.
It's important to note that none of this works with any confidence unless your code has high-quality automated tests. If your project is part of a largely distributed application, it's also important to have good integration tests. Otherwise, your changes that aren't caught during the pipeline can cause problems for other components that are then deployed to production unchecked, causing an outage.

Key takeaways

Pipelines are the sets of tools and automated processes that development and operations teams use to build, test, and (optionally) deploy software. The automation of the CI/CD pipeline can reduce the amount of manual work and coordination required to deploy software, save time and effort for your development and operations teams, and allow you and your organization to release any updates more efficiently. In order for your CI/CD pipelines to be successful, make sure you have included high-quality tests for automation, integration, and deployment. Finally, pipelines allow your organization to have multiple teams working independently, accelerate software development and deployment cycles, enhance the quality and stability of your software releases, and create a faster, more efficient review and resolution process, which also reduces the number of errors.

DevOps tools

You've learned that DevOps is a combination of development and operations and represents the process of breaking down silos between the development teams writing code and the operations teams that deploy and maintain it. Historically, these teams often communicated only when it was time to deploy a release or when there was a problem in production that could be traced back to a bug in the code. By breaking down those silos, as well as leveraging technologies like automated builds and testing, organizations can spend less time on manual processes. This enables them to deliver software much more quickly and frequently—which can translate to a competitive advantage in the marketplace.

You've also learned about the DevOps lifecycle, which involves constant collaboration and iteration. DevOps tools help you tackle processes like continuous integration, continuous delivery, automation, and collaboration.

DevOps tools

So, what are DevOps tools? They're any software tools that help to make automated processes possible. DevOps tools are designed to address many of the pain points common to software development organizations everywhere.

While you can certainly build your own tools from scratch, using DevOps tools can save you a lot of time and effort.

There are many different types of DevOps tools including:

- **Source code repositories**, such as GitHub or Bitbucket
- **CI/CD tools**, such as Github Actions, Jenkins, and Google Cloud Deploy
- **Infrastructure as Code (IaC) tools**, such as Terraform or Ansible
- **Container management tools**, such as Docker or Kubernetes
- **Security scanning tools**, such as Snyk or SonarQube
- **Production monitoring tools**, such as DataDog or AppDynamics

But how do you know which tools to use? Let's think about the DevOps lifecycle. This will help you determine which tools are the best fit at each stage.

Discover

During discovery, look for tools that encourage brainstorming to happen asynchronously. This allows everyone on your team to share and comment on anything and will be important throughout the DevOps lifecycle. Examples of tools you can use include Jira Product Discovery, Miro, and Mural.

Plan

When you get to the planning stage, look for tools that provide options for sprint planning and issue tracking, as well as continued collaboration. Examples of tools you can use include Jira Software, Confluence, and Slack.

Build

During the building stage, look for tools that allow you to create individual development environments, reprovision those development environments, monitor versions with version control, continuously integrate and test, and have source control of your code. Also look for tools that allow for continuous integration, delivery, and feedback. Examples of tools you can use include Kubernetes and Docker.

Test

Automation is essential for DevOps, so look for tools that can automate testing and support wallboards, sharing, and commenting on particular builds or deployments. Examples of tools you can use include Veracode and SmartBear Zephyr Squad or Zephyr Scale.

Monitor

Look for tools that can integrate with your group chat clients and send you notifications or alerts when you've automated monitoring your servers and application performance. An example tool you can use is Jira Software.

Operate

Deployment and operation will have similar needs. For deployment, look for tools that can integrate your code repository and deployment tools in a single dashboard. Once your software has deployed, look for tools that can track incidents, changes, problems, and software projects on a single platform. It's easier to find, identify, and resolve issues when they're all in the same place. An example tool you can use is Jira Software.

Continuous feedback

It's important to continuously receive feedback on your builds and gain insights from that feedback. Look for applications that can integrate your chat clients with a survey platform or social media platform. Examples of tools you can use include Slack and GetFeedback.

Popular tools for CI/CD

Jenkins, GitLab, Travis CI, and CircleCI are all tools which can automate the different stages of the software development lifecycle, including building, testing, and deploying. They are often used in DevOps to continuously build and test software, which allows you to continuously integrate your changes into your build.
Tools like Spinnaker, Argo CD, and Harness can be used to automate continuous delivery and deployment and to simplify your DevOps processes.
Pro tip #1: Don't throw out your entire process just because a new tool doesn't fit into your existing pipeline. Choose tools that will fit your workflow and slot into your pipeline easily.

Pro tip #2: Consider that the most popular tools have the broadest base of support and the largest number of practitioners, making it easier for new team members to get up to speed quickly. This also means that there's more online support and forums for discussion available to you.

Key takeaways

There are many different DevOps tools available out there—some will have multiple functions and others will be designed for a single purpose. Some tools will share the same functionalities and can be used in more than one stage of the DevOps lifecycle. It's important to choose the tools that will best fit your workflow and slot into your pipeline easily. Remember, the tools are there to help make DevOps processes like automation and continuous integration, development, and delivery easier for you.

Containers with Docker and Kubernetes

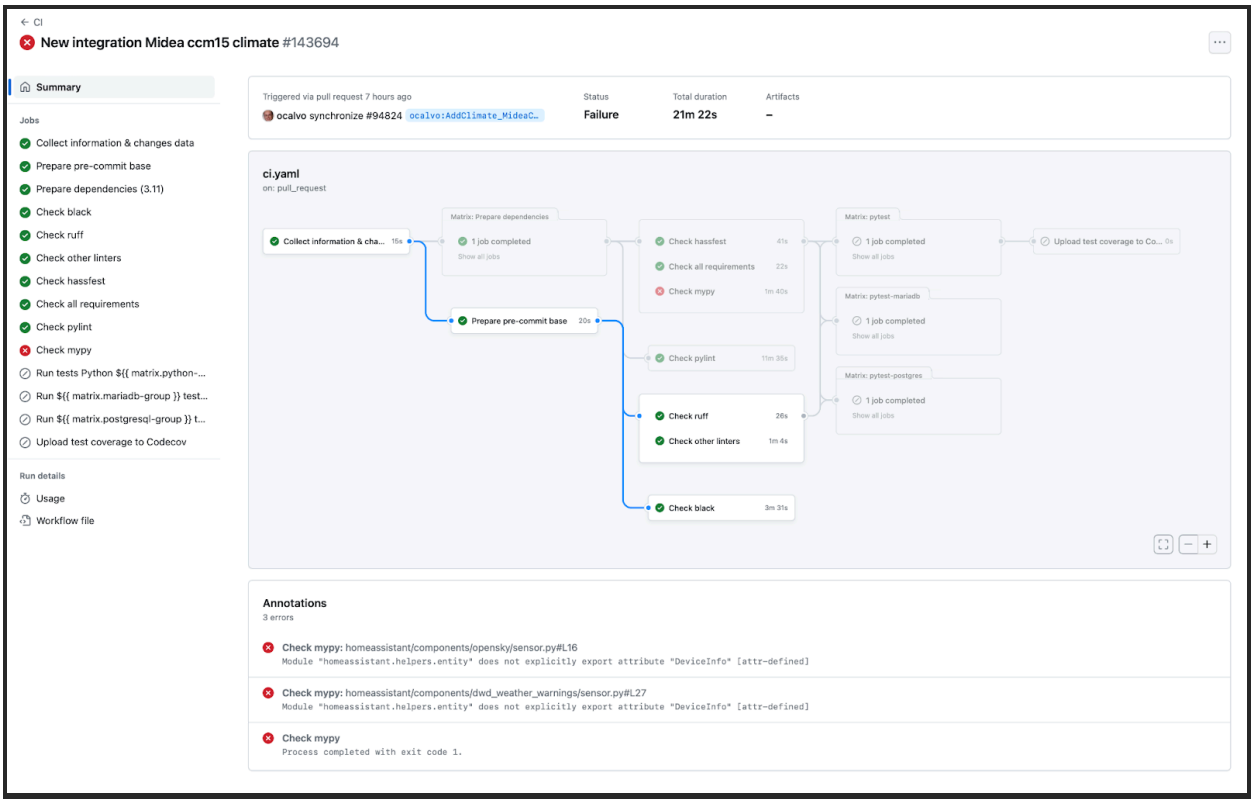
Here's what you should have learned so far:

- **Containers** are applications that are packaged together with their configuration and dependencies. They're used to share applications, post applications to a server for review, test different instances of the same application, and separate a large application into smaller, independent parts, making them easier to manage during the stages of the software development lifecycle.
- **Docker** is the most common way to package and run applications in containers. It can build container images, run containers, and manage container data through volumes and container networks to make the running software talk to the external world.
- **Kubernetes** is a portable and extensible platform to assist developers with containerized applications. It's a tool that developers use while working in Docker to run and manage Docker containers, allowing you to deploy, scale, and manage containerized applications across clusters.

In this reading, you will learn about how containers are used in the CI/CD pipeline.

Containers in the CI/CD pipeline

Continuous integration and continuous delivery/deployment (CI/CD) is the automation of an entire pipeline of tools that build, test, package, and deploy an application whenever developers commit a code change to the source control repository. Feedback can be provided to developers at any given stage of the process. A pipeline is an automated process and set of tools that developers use during the software development lifecycle. In a pipeline, the steps of a process are carried out in sequential order. The reason behind this is that if any step fails, the pipeline can stop without deploying the changes. The pipeline stops executing the steps and marks the job as failed. The following image is what this looks like in Github Actions.



Using containers in the CI/CD pipeline can bring developers additional flexibility, consistency, and benefits to building, testing, packaging, and deploying an application. Because containers are lightweight, they allow for a faster deployment of the application. Containers help eliminate the common “works on my machine” syndrome.

Docker images contain the application code, data files, configuration files, libraries, and other dependencies needed to run an application. Typically, these consist of multiple layers in order to keep the images as small as possible. Container images allow developers to run tests, conduct quality performance checks, and ensure each code change is tested and works as expected before being deployed.

Kubernetes is a tool for organizing, sharing, and managing containers. This powerful tool gives programmers and developers the ability to scale, duplicate, push updates, roll back updates and versions, and operate under version control.

Another advantage of using containers in a CI/CD pipeline is that developers are able to deploy multiple versions of an application at the same time without interfering with one another. It can reduce the number of errors from configuration issues and allow delivery teams to quickly move these containers between different environments, like from build to staging and staging to production. And lastly, using containers in a CI/CD pipeline supports automated scaling, load balancing, and high availability of applications creating robust deployments.

Key takeaways

Using containers in the CI/CD pipeline with both Docker and Kubernetes benefits developers by creating a more seamless process for building, testing, packaging, and deploying an application. Containers provide a more reliable way to work with applications at any stage in the pipeline process.

Continuous testing and continuous improvement

You've learned about CI/CD, the tools you can use to automate your DevOps processes, and the stages of a CI/CD pipeline. In this reading, you'll learn about continuous testing and continuous improvement, why they're important in the CI/CD pipeline, how to measure the effectiveness and efficiency of your CI/CD pipeline, and some tools you can use to detect issues and ensure the quality of your software.

Continuous testing

Continuous testing means running automated test suites every time a change is committed to the source code repository. In practice, this usually means running the tests as part of a CI/CD pipeline, in between the build and deploy stages. Continuous testing is an important part of the CI/CD process. It ensures that all of your code changes are tested early in the development process, preventing them from becoming larger, more difficult, time-intensive, and/or more expensive to fix later on.

There are three types of testing that you'll typically see in the CI/CD pipeline. These include:

- Unit testing
- Integration testing
- System testing

You use unit testing to test an individual unit within your code—a unit can be a function, module, or set of processes. Unit testing checks that everything is working as expected.

Integration testing is part of both continuous delivery and continuous deployment. It allows you to automatically test each change to your code when you commit or merge them to your source code repository. The testing checks for errors and security issues as they arise, again preventing you from having to deal with larger, more difficult, and/or expensive issues later in the process.

System testing does exactly what it sounds like: It simulates active users and runs on the entire system to test for performance. When testing the entire system, testing for performance can include testing how your program, software, or application handles high loads or stress, changes in the configuration; and system security. You can also utilize end-to-end testing, which tests the functionality and performance of your entire application from start to finish by simulating a real user scenario.

Testing frameworks and tools

There are many testing frameworks and tools you can use for automated testing, such as JUnit, Selenium, Cypress, and Postman.

- JUnit is an open-source unit testing framework for the Java programming language and can help you with your unit testing. With JUnit, you can write and execute automated tests and develop reliable, bug-free code. There are similar libraries for other languages such as PyUnit for Python and NUnit for C#.
- Selenium is an open-source, automated testing suite of tools for web application developers. Each tool can be used for different testing needs.
- Cypress is a JavaScript-based testing tool that can automate end-to-end tests. It simulates how users would interact with your web applications. Often used for front-end development

of web-based applications, these kinds of tests will help to ensure that your tests and the users' experiences are the same.
- Postman can be used to automate unit tests, function tests, integration tests, end-to-end tests, regression tests, and more in your CI/CD pipeline.

Continuous improvement

Continuous improvement is a crucial part of the DevOps mindset. It's the idea that every team should constantly be on the lookout for ways to improve their efficiency and reduce errors and bottlenecks. Think of continuous improvement like compounding interest—it starts out small, but slowly increases over time, resulting in a better product for consumers.

Continuous improvement requires investing the time and resources necessary to improve the entire process of developing and deploying software. It also requires commitment from your organization's leadership to allow that investment when they may be considering other priorities.

Key benefits of continuous improvement include:

- Increased productivity and efficiency
- Improved quality of products and services
- Reduced waste
- Competitive products and services
- Increased innovation
- Increased employee engagement
- Reduced employee turnover

The benefits don't stop there! The benefits of continuous improvement can accumulate and improve your organization's processes, efficiency, and profitability overall. It's in your, and your organization's, best interest to invest in continuous improvement.

Key performance indicators (KPIs)

Key performance indicators, or KPIs, are quantifiable measurements of performance. These metrics are data points that can tell you how your DevOps CI/CD pipeline is performing and help you identify if there are any errors or bottlenecks in the process. Metrics can track and measure workflows and the progress of your projects and goals, which can lead to improved software or application quality and performance. There are many metrics you can use in DevOps, so it's important to choose the ones that work best for your project and workflows—just like the tools you use for CI/CD. Popular metrics in DevOps that you can use to measure performance include:

- **Lead time for changes:** This is the length of time it takes for a code change to be committed to a branch and be in a deployable state.
- **Change failure rate:** This is the percentage of code changes that lead to failures and require fixing after they reach production or are released to end-users.
- **Deployment frequency:** This measures the frequency of how often new code is deployed into production.
- **Mean time to recovery:** This measures how long it takes to recover from a partial service interruption or total failure of your product or system.

Key takeaways

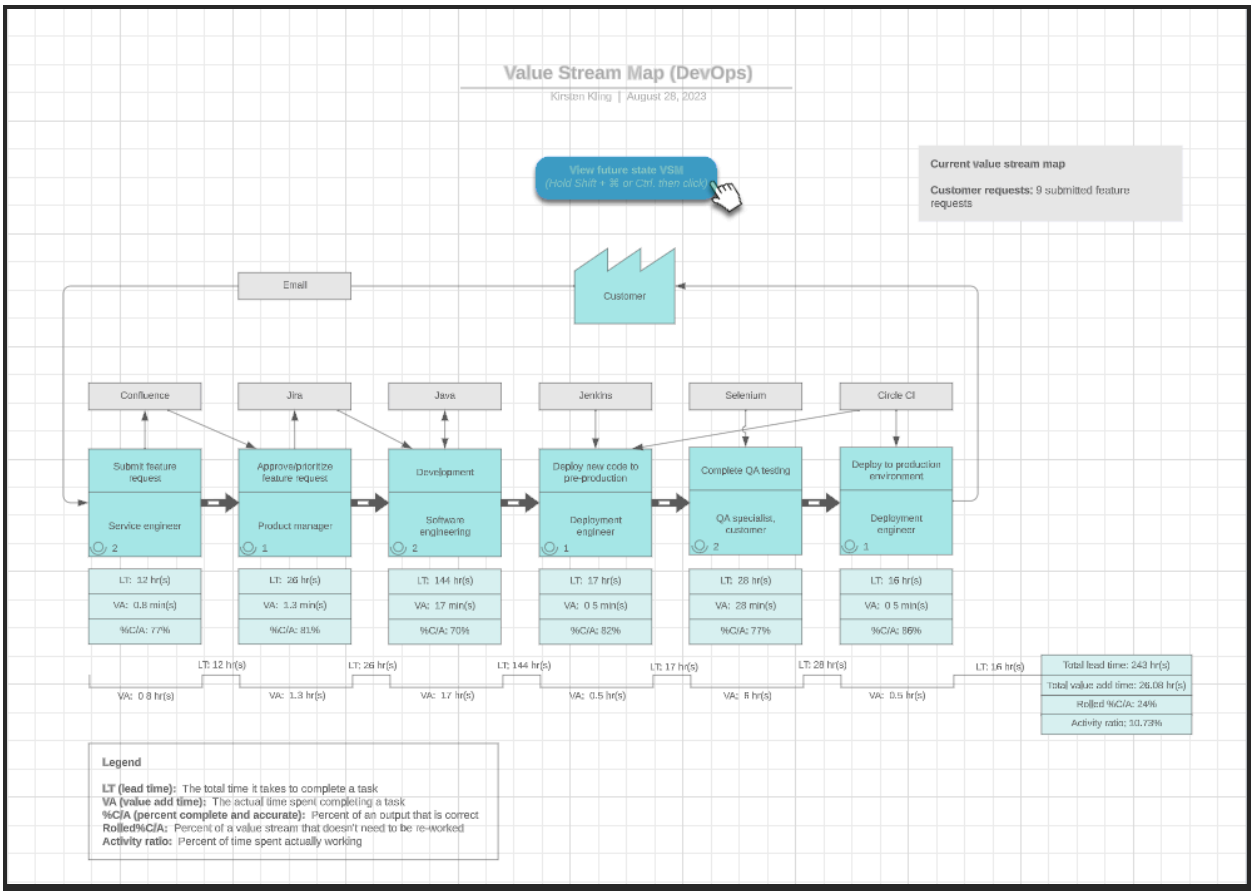
Making high-quality tests part of your CI/CD pipeline is critical to your DevOps success. The more complete your test suite is, the earlier you will be able to catch bugs and squash them. If you're going to deploy your code as soon as it's committed, you want to have the utmost confidence that everything will work together in production. The more tests you are able to run before deployment—unit tests, integration and smoke tests, load tests—the more confident you can be.
Pro tip #1: Take the time to learn how to write good tests. There are tools available that will analyze your code and estimate how complete your test coverage is. You should aim for 100% coverage.
Pro tip #2: Test engineering is a highly skilled and sought-after discipline. Becoming a passionate advocate for DevOps and continuous improvement can make you highly visible and valuable in many organizations. For more information, visit IT automation job opportunities and SRE job opportunities.

Value stream mapping

You've learned that the "CD" in CI/CD can stand for continuous delivery and continuous deployment. They're related concepts that are sometimes used interchangeably. They're both about automating later stages of a DevOps pipeline, but you can use either to show what is happening during automation. Continuous delivery means that any changes a developer makes to an application are automatically released to a repository like GitHub and then deployed by the operations team. Continuous deployment is an extension of continuous delivery and refers to the automatic deployment of an app or any developer changes from the repository to production, which helps to prevent overloading the operations teams and automates the next stage in the pipeline.
In this reading, you'll learn about value stream mapping (VSM), the key components of VSM, the benefits of VSM, and its significance in process improvement and waste reduction.

Value stream mapping (VSM)

Value stream mapping (VSM) is a technique used to analyze, design, and manage the flow of materials and information required to bring a product to a customer. Also known as a material and information flow map, VSM can help you identify bottlenecks in your value stream, inefficiencies in your process, and current areas of improvement. It can also help to reduce the number of steps in your process and help you visualize where handoffs occur. This way, you can identify where wait time is preventing work from moving through your system.
A significant goal of VSM is to reduce waste.
Another goal of a VSM is to increase the efficiency of your processes. To do this, create a detailed map of all the necessary steps involved in your business process with a diagram or a flowchart. Consider the example below:



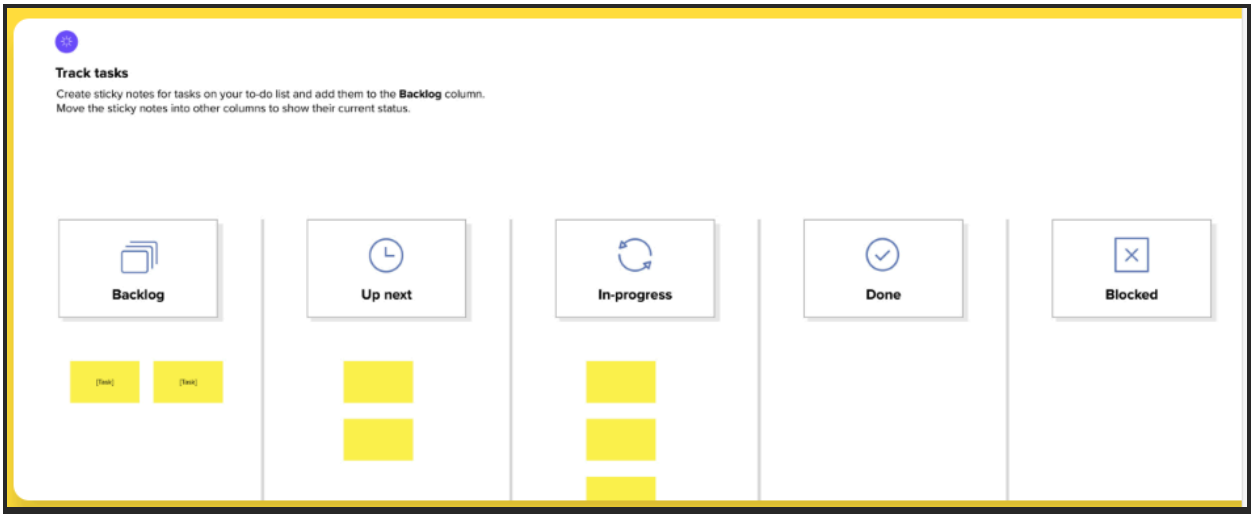
This diagram outlines these steps:

1. **Define the problem.** What are you trying to solve or achieve?
2. **List the steps in your current process.** For each step, make sure to note the amount of time needed, any inputs and outputs, and the resources—both people and materials—necessary to complete each step.
3. **Create and organize the map using the above data.** Your goal is to illustrate the flow of your process, so begin with the start and finish with the end of your process. If you need help organizing the flow, think back to the steps in the software development lifecycle and use that as a guide to organize your steps.
4. **Find areas that can be improved.** Gather information about your current process by answering questions like:
    a. Can some tasks be done in parallel?
    b. Can tasks be reordered to improve efficiency?
    c. Can tasks be automated to reduce the amount of manual labor?
5. **Update the map with your findings.** This will show you where you need to make improvements or change your process.
6. **Implement the new process.** But don't stop here! If this new process works well for your project—great! Keep in mind that coding, software, programs, apps—everything digital—are constantly updating to meet client or business needs. It can be helpful to implement an iterative process—either manual or automated—to make sure that any new

hiccups in your process can be identified and addressed before they become a larger issue.

For more information and an explanation of how value maps benefit DevOps, see the article How to Use Value Stream Mapping in DevOps on the Lucidchart website.

In software development, a kanban board is probably the best and most common way of visualizing VSM. Kanban is the practice of visualizing tasks in a grid. Each column in the grid represents one of the states a task may be in, such as “to do,” “in progress,” or “done.” As a task progresses, it moves from left to right across the grid. Your “in progress” column can be broken down further into the steps of your process. In software development, this can be the steps of the software development lifecycle like plan, design, build, test, and review for feedback.



virtual post-it notes with information about the project.

You can add as many columns as you need to your VSM or kanban board. The more information you have, the more comprehensive your view of the process will be. This will also help you identify waste, inefficiencies, and areas of improvement.

Other common components of a VSM include: lead times, wait times, handoffs, and waste.

- **Lead time** is the length of time between when a code change is committed to the repository and when it is in a deployable state.
- **Wait time** indicates the length of time a product has to wait between teams.
- **Handoffs** are the transfer of information or responsibilities from one party to another.
- **Waste** refers to any time you are not creating value. In software development, there are seven types of waste production.
    - Partially completed work refers to when software is released in an incomplete state. This leads to more waste because additional work is needed to make updates.
    - Extra features refers to creating waste by doing more work than is required. This may be well-intentioned but can signal a disconnect between what the customer wants and what’s being created.
    - Relearning refers to waste generated from a lack of internal documentation. This can be a result of not investigating software errors, failures, or outages when they

occur and having to relearn what to do if they happen again. It also includes having to learn new or unfamiliar technologies, which can create delays or wait times in workflows.
    - Handoff waste can occur in a few places—when project owners change, when roles change, when there is employee turnover, and when there is a breakdown in the communication pipeline between teams.
    - Delays refer to when there are dependencies on coupled parts of the project. A delay in one stage or decision may create a delay in another, which can create a surge in waste.
    - Task switching refers to the waste that is generated when an individual has to jump between tasks, which involves mental context switching. This may result in the individual working more slowly and/or less efficiently.
    - Defects refers to waste that is generated when bugs are released with software. Similar to partially completed work, defects can result in extra time and money down the line, as well as delays and interruptions in workflow due to task switching.

VSM can be implemented with something as simple as sticky notes on a whiteboard, by using project tracking software (like Jira, Asana, Trello, or others), or with special purpose-built tools. Due to its visual nature, VSM can foster collaboration and communication between teams and other stakeholders.

Key takeaways

Value stream mapping, or VSM, is a technique used to analyze, design, and manage the flow of materials and information required to bring a product to a customer. It's important because it can improve the efficiency of your software delivery process, reduce the time needed to deliver updates to your customers, and give you a competitive advantage over other businesses.
Pro tip #1: Don't be surprised if your initial attempt at creating a value stream map omits some tasks that are vital to the process. Documentation isn't perfect and neither is anyone's memory.
Pro tip #2: Involve multiple coworkers in the process. This can give you the benefit of different perspectives.
Pro tip #3: Share the results of your work with your team so they can understand the benefit(s) of making a change in their workflow or the overall process (or both!).
Mark as completed
Like
Dislike
Report an issue

Github and delivery

GitHub can facilitate your efforts in Continuous Integration and Continuous Delivery (CI/CD). Learning about how to utilize this repository will benefit you down the road.

How GitHub supports CI/CD

GitHub supports external CI/CD tools by providing webhooks and APIs that allow those tools to become part of the pull request process. For example, GitHub can be set up to refuse to merge a pull request until several actions are completed:

- The PR is reviewed and signed off by one or more code reviewers.
- The CI build process completes successfully.
- The CI test suites run successfully.
- The PR submitter has acknowledged the project's license, standards, and/or code of conduct.

The feature of GitHub that automates CI/CD is called GitHub Actions.

GitHub Actions

GitHub Actions is a feature of GitHub that allows you to run tasks whenever certain events occur in your code repository. With GitHub Actions, you are able to trigger any part of a CI/CD pipeline off any webhook on GitHub.

To get started with GitHub Actions, go to the **Actions** tab in the GitHub repository. When opening the tab for the first time, you'll find a description of GitHub Actions and some suggested workflows for your repository. These workflows build the code in your repository and run your tests either on GitHub-hosted virtual machines or on your own machines. You'll receive the results of the tests in the pull request.

GitHub Actions has more than 13,000 pre-written and tested CI/CD workflows. If you would like to write your own workflows or customize an existing one, you can do it using YAML files.

Because GitHub Actions is a general-purpose engine that runs tasks in response to events in the repository, you can use it for more than just CI/CD pipelines. Some examples:

- Running a nightly build in the master branch for testing.
- Cleaning up old issues or bug reports.
- Creating a full-fledged bot that responds to comments or commands on a PR or issue. ("Issues" are like bug reports or support tickets.)

Key takeaways

GitHub Actions supports your CI/CD. It offers workflows that you can use right away or customize for your own repository.

Resources for more information

GitHub Actions documentation - GitHub Docs
A beginner's guide to CI/CD and automation on GitHub - The GitHub Blog
GitHub Protips: Tips, tricks, hacks, and secrets from Jason Etcovitch - The GitHub Blog

Configuration management

Image this: You just got a new job as a system engineer to develop a new software product. As you begin to write code for the product, you change and update it continuously to correct any issues

found. You use configuration management to optimize your IT responsibilities, creating a more efficient workflow.

Configuration management is an automated process that ensures your new software project and its assets perform as they should as you update and change your code.

In this reading, you will learn more about configuration management, its role in maintaining consistency and stability, and how to use configuration files to describe desired system states.

Consistency and stability

Configuration management helps you manage your code and all of its components. In addition, it ensures that each component of your code is automatically and properly built, monitored, and updated as needed. It's common to have multiple copies of an application running at the same time to guarantee a highly available, properly functioning system.

If configuration management is not used, a developer will have to manually update and correct any issues or errors on each individual server. Something as simple as a typo could create misconfigured servers, causing errors and unexpected behaviors. The developer would have to correct each typo on each server. This process would be frustrating and unsustainable. That's where configuration management plays a role in maintaining consistency and stability in software systems. It enables you to duplicate server instances, resulting in consistent behavior across all servers.

Configuration files

Configuration files are commonly referred to as a manifest or playbook—depending on the DevOps tool you use. You can think of these as statements on how you want the system to look and perform. Let's take a look at an example. A playbook might say, "I need a server with 32GB of RAM running Debian Linux, with Python 3.9 and Nginx installed." Notice that this playbook statement does not specify the steps to achieve the desired state—it only describes what the desired state should look like. To use configuration files, create a configuration file as the input to your configuration management tool describing the desired state. The configuration management tool works to determine a solution to have the server look and perform as you described in your manifest or playbook.

Pro tip: Store configuration management files alongside the application code in your revision control system, allowing changes to be tracked and audited.

Key takeaways

Configuration management provides developers a solution to simplify and effectively manage changes and updates to codes. It streamlines the process of correcting any issues or errors found on multiple copies of the same application. It operates off of configuration files—manifests or playbooks—to understand what the desired state of the system should be and determine a solution to achieve the desired state.

CD best practices

At this point, we have covered that Continuous Delivery and Continuous Deployment are at the heart of iterative development. The core principles of Continuous Deployment (CD) are:

- Automation: The entire process must be automated so that results are repeatable and consistent.
- Version control: CD configuration and scripts must be stored in your version control system and treated just like code.
- Monitoring: Monitoring the deployed application is essential so that you can confirm the app is working properly after deployment.
- Pairing: CD must be paired with CI and automated testing. Deployment should proceed automatically once tests have completed successfully.

The key of CD is automation. Automation not only reduces errors but also saves you time. As much as you can, automate everything from testing to releasing.

Feature flags

Feature flags are also called feature toggles, release toggles, or feature flippers. The idea behind feature flags is that new features or application behaviors can be hidden and turned on when you're ready to introduce them to users. Feature flags can often be set on a per-user or per-role basis. As you might already know, "ready to be deployed" isn't always the same as "ready for users to see."
For example, when you're deploying a major new feature in an application. You may create a feature flag and turn it off for everyone. In your code, you check the flag before allowing the new feature to be used. Now, you can do the following:

- Allow the new code to be built, tested, and deployed to production. Because the flag is turned off, no one sees the new feature.
- Turn the flag on for a specific group of internal testers.
- If all goes well, you can turn the flag on for a larger group of early adopters or beta testers to gather their feedback.
- Once all the bugs are fixed and feedback has been incorporated, you can turn the flag on for everyone.

Incremental rollout

Incremental rollout means to slowly deploy changes throughout your infrastructure so that you can monitor for trouble and quickly roll back the deployment if there's a problem. The practices that are associated with the incremental rollout are canary releases and blue-green deployments.

Canary releases and blue-green deployments

Both of these are extensions of the concept of incremental rollout. You deploy changes to a portion of your infrastructure or user base. You can then use a variety of monitoring tools to observe any changes in key metrics such as error rate, user engagement, feedback scores, etc. and compare those against the rest of the population.
The difference between the two is that a canary release is often targeted at a specific population of testers or early adopters, whereas a blue-green deployment will send X% of randomly selected users to the blue or green servers.
A canary release can be used to closely observe and gather feedback from a small group of users before unleashing a new release on everyone. If the feedback is negative, or the canary group experiences issues with the new release, you can halt further deployment and fix the issues before deploying everywhere.

With a blue-green deployment, you can use automated monitoring to observe any differences between the blue and green populations. If there is a problem, you can instantly shut off traffic to the blue or green side and direct all user traffic to the other side until the problem is fixed.

Key takeaways

CD must be paired with CI and automated testing. The CD practices of using feature flags, incremental rollout, canary releases, and blue-green deployments will minimize problems that often are caused by human errors.

Resources for more information

- 8 Key Continuous Delivery Principles | Atlassian
- The Fundamentals of Continuous Deployment in DevOps - GitHub Resources
- Continuous Deployment - Scaled Agile Framework
- Feature Flags—What Are Those? Uses, Benefits & Best Practices | LaunchDarkly
- Incremental rollouts with GitLab CI/CD | GitLab
- A Comprehensive Guide to Canary Releases | by Daniel Bryant | Ambassador Labs (getambassador.io)
- What is the Canary Deployment & Release Process? – BMC Software | Blogs
- A Comprehensive Guide to Canary Releases | by Daniel Bryant | Ambassador Labs (getambassador.io)
- Using blue-green deployment to reduce downtime | Cloud Foundry Docs

Release

As a technology user, you may be familiar with alerts on your computing devices for new releases of the software or applications you've downloaded. A release, in the software development world, is a new version of a piece of software. For example, Apple released a new Mac operating system (OS), Ventura, to replace Monterey in 2022. Microsoft released Windows 11 to replace Windows 10 in 2021. There are several parts and processes involved in carrying out a release, and therefore, project management, especially agile, is key to ensure the process goes smoothly. The agile approach has become popular in software development because it focuses on short sprints where incremental changes are implemented, tested, and sent out to end users.

Release management

Release management uses project management methods with CI/CD DevOps tools to produce, document, and release stable new versions of software ("releases"). Release managers, who are in charge of overseeing the release process, coordinate among multiple teams and orchestrate the whole process to make sure the project goes according to plan and meets deadlines. The concepts and processes associated with release management include:

- **Release planning:** Release management begins with detailed release planning. It involves setting the scope of the software build, identifying the features and changes to be included, and setting release goals and timelines.

- **Versioning:** Versioning assigns unique identifiers to different software versions. Each team may have developed their components independently, resulting in various codebases. Release managers use versioning to keep track of the different parts and ensure a coherent and structured integration of the code.
- **Coordinating development and testing:** Release management ensures smooth coordination between the development and testing phases. As the teams complete their individual work, release managers facilitate integration testing to identify and resolve any issues arising from combining the codebases. They also work with testing teams to create test plans and verify that all components function correctly together.
- **Risk assessment and mitigation:** Release managers identify potential risks associated with the software release and plan mitigation strategies. For example, if the release has conflicts or compatibility issues between the modules developed by different teams, the release manager addresses these risks proactively to prevent disruptions during deployment.
- **Communication and stakeholder management:** Release management involves clear and consistent communication with all stakeholders, including the development teams, testing teams, project managers, and end-users. Regular status updates, progress reports, and discussions are essential to keep everyone informed about the release progress and any adjustments to the plan.
- **Release schedule:** Release managers create a detailed release schedule that outlines the steps leading up to the deployment. This includes code integration deadlines, testing periods, and the final release date. The schedule serves as a roadmap for the teams, ensuring that each phase is completed on time and within the defined scope.
- **Change management:** Throughout the development process, changes are inevitable. Release management establishes a structured change management process to evaluate, approve, and track changes effectively. This ensures that only authorized and tested changes are included in the release build.
- **Documentation and release notes:** As part of the coordination process, release managers oversee the creation of comprehensive release notes and documentation. These documents provide crucial information about the software update, including new features, bug fixes, known issues, and any special instructions for end-users.

Types of release

Releases are not created equally. There are four types of release.

Major releases

Major releases are significant updates that introduce substantial changes and new features to the application (e.g., the release of macOS Ventura and Windows 11). These updates often involve major improvements, enhancements, or redesigns that may have a significant impact on the user experience or functionality. Major releases may require extensive testing and development efforts, as they can involve rewriting core components or introducing new infrastructure. Since they bring significant changes, they also have a higher chance of

introducing new bugs or issues that need to be addressed in subsequent releases. For example, SoundGrounds is a coffee-drinker-friendly streaming phone app that makes recommendations on music and drinks. A major release from SoundGrounds is when the app introduces new features like personalized playlists or a complete UI overhaul to enhance the user's streaming experience.

Minor releases

Minor releases are smaller updates that include additional features, improvements, or minor changes. They are often aimed at enhancing the application's existing functionalities without introducing major overhauls. Minor releases are generally less complex than major releases, as they build upon the existing stable version. However, they still require thorough testing to ensure that the new features integrate seamlessly with the current application. An example is the releases of MacOS Ventura 13.4 and 13.5, which involve bug fixes. For SoundGrounds, a minor release may involve adding features like sleep timer functionality or an integration with a new payment gateway for premium features.

Patch releases

Patch releases are focused on fixing bugs, vulnerabilities, and addressing critical issues present in the software. They aim to provide quick solutions to problems that affect the application's stability or security. Patch releases are typically smaller in scope and can be deployed more rapidly than major or minor releases. Since they focus on bug fixes, thorough testing is crucial to ensure that the patches do not introduce new issues.

Hotfix releases

Hotfix releases are immediate updates to address critical and high-priority issues that affect the application's functionality or pose security risks. They are typically rolled out urgently to resolve pressing problems. Hotfix releases undergo minimal testing and are immediately pushed to production to mitigate the impact of urgent problems. For example, Mac OS released hotfixes to resolve issues on bluetooth and other basic tools.

Key takeaways

A release is a new version of a piece of software. For a company to carry out a release, it involves effective release management–from planning the release scope, coordinating between teams, mitigating potential issues, to meeting the deadlines. There are four types of release: major, minor, patch, and hotfix.

End-to-end tests

End-to-end testing or E2E testing involves testing of a complete application environment in a scenario that mimics real-world use, such as interacting with a database, using network

communications, or interacting with other hardware, applications, or systems. Unlike unit testing or integration testing, E2E testing is a comprehensive evaluation.

Scope and objectives

E2E testing covers:

- **User experience:** Ensuring the system behaves as expected and delivers the right user experience from front-end UI interactions to back-end database operations
- **Integration and communication:** Validating the interaction between different system components, such as databases, network protocols, and other interfacing systems
- **Data flow:** Checking the data integrity between different system components. It ensures that data is correctly sent, received, and processed.

For example, imagine your team develops a coffee-and-music app with social media features. A typical user journey involves creating an account, logging in, choosing a coffee, selecting music, interacting with other users, and logging out. To conduct an E2E test, your team would identify all the main user journeys (from account creation to logging out) and create a detailed test case for each journey that includes the expected outcomes.

One of the key objectives of E2E testing is the opportunity to simulate **critical user journeys** or the real-world user scenarios. The journeys usually cover essential user flows like account creation, log-in, placing an order, or making a payment. In addition, the E2E testing allows you to verify if **business processes** are working as expected. For example, if a user places an order, it should trigger a series of events like payment processing, order confirmation, and dispatch details.

Tools for end-to-end tests

You can use testing frameworks such as Selenium and Puppeteer to conduct an E2E testing. Most of these UI testing frameworks target web applications and they tend to be JavaScript-based as web apps almost always use HTML+JavaScript (react.js, angular, vue.js) for their user interface components. For Python developers, when you utilize these frameworks, you will need to use Flask or Django. Flask is commonly used to build web apps and it can handle microservices. Django is used for larger or more complex applications.

Here are some of the testing tools you can use with Flask and Django:

- Selenium is a popular open-source tool that allows for automated testing across various platforms and browsers. It supports multiple programming languages like Java, C#, and Python.
- Puppeteer is a Node library that provides a high-level API to control Chrome or Chromium over the DevTools Protocol. It can be used for testing Chrome extensions and for generating screenshots and PDFs of pages.
- Cypress is a JavaScript-based testing framework that doesn't use Selenium. It allows you to write all types of tests: E2E tests, integration tests, and unit tests. However, Cypress doesn’t support Python.

- Appium is an open-source tool for automating native, mobile web, and hybrid applications on iOS mobile, Android mobile, and Windows desktop platforms. It's widely used for end-to-end testing of mobile applications.
- Protractor is an end-to-end test framework for Angular and AngularJS applications. It runs tests against your application running in a real browser, interacting with it as a user would.

Key takeaways

End-to-end testing is a comprehensive evaluation that includes testing of a complete application environment using real-world user scenarios. Its scope includes the testing of user experience, integration and communication, and data flow. Python users can use testing tools and frameworks such as Selenium. But keep in mind that these tools require Flask or Django.

Mark as completed

Like

Dislike

Report an issue

Postmortem

Imagine this: You are a software developer for a banking company, and your development team has been informed of a security breach that exposed sensitive client data. Your team immediately conducts a postmortem, or incident analysis. A postmortem examines the impact the breach caused, including the vulnerability of the application; determines updates needed to ensure security best practices for protection from future attacks; and determines a process to protect clients' data.

In this reading, you will learn more about postmortem, the role it plays in understanding and learning from system failures and incidents, its key components, and its use in software development and infrastructure operations.

Postmortem

Postmortem is a term typically used in the medical field, but it is also commonly used in the programming world as well. As the name implies, postmortem generally means something bad happened. You might hear other developers call this "putting out a dumpster fire." It's a thorough analysis process used after a major incident has occurred—like a system outage or significant disruption. The analysis includes determining what caused the disruption, establishing steps to resolve the problem, and setting plans to prevent future disruptions. For example, a software developer pushes out poorly written code into production. The software developer failed to have the code undergo a peer review. As a result, the database connection misconfiguration broke, which resulted in outages. A postmortem is written up to explain what went wrong and how software developers can prevent the same issue from occuring in the future.

A postmortem process allows teams to understand and learn from system failures and incidents. The purpose of the postmortem process is to understand what happened, why it happened, and how to avoid it again in the future rather than assigning blame to an individual or team.

Key components

A postmortem report provides clear details of the incident to all team members and promotes a culture of continuous improvement. Key components of a postmortem include:

- Incident timeline: This describes what happened and when it happened.
- Root cause analysis: This includes details of why the incident happened.
- Impact analysis: This includes details of who and what was affected by the incident.
- Mitigation and recovery: This includes the steps taken to correct the incident.
- Action items for improvement: This describes the steps to take to prevent a future incident.

Let's use our banking incident as an example to further explain the components. The incident timeline details the sequence of events from when an application or program was launched to when the disruption occurred. This would be the events that led up to when the security breach occurred. The root cause analysis investigates why the security breach occurred. The impact analysis is an assessment of the number of customers that were impacted by the breach, how their data was exposed, and any potential reputational and financial damages that occurred. Mitigation and recovery are the actions that are taken to stop the breach and to update the code for better security and performance. Action items for improvement explain the strategies the bank will take to prevent similar security breaches from occuring in the future.
A postmortem analysis informs developers which areas of the code need to be updated to create a stronger application. In infrastructure operations, a postmortem analysis is used to learn from a system's disruption. It identifies weaknesses in the infrastructure and guides the measures to improve resilience.

Key takeaways

A postmortem or incident analysis provides developers with insights into an unexpected software incident that has occurred. The analysis provides the necessary details needed to determine the root cause of the disruption, why and how it occurred, and who was impacted. These details also allow developers to create a plan to strengthen the software so a disruption will not happen again in the future.

Module 4 review

Congratulations on completing the last module for *Course 5: Configuration Management and the Cloud*. You're making great progress! You've learned so much about the continuous integration and continuous delivery process, which ensures your software functions as expected and any changes made are tested properly.

You've discovered what a CICD pipeline is and the importance of automating the tools to build, test, package, and deploy an application when code changes are integrated. You learned that DevOps tools are the software tools to help you accomplish the automated process throughout the DevOps lifecycle. The tool you use will be dependent on the stage of the lifecycle you are in. Implementing containers in your CI/CD processes can improve collaboration amongst developers and ensure the delivery of efficient and reliable applications. It's important to run continuous testing as part of a CI/CD pipeline and, depending on your test results, to implement the changes necessary to continually improve efficiency and reduce errors.

In addition, you learned about the continuous integration platform, Github, and the benefits it provides to the coding community. You also explored Cloud Build—a CI/CD service—provided by GCP, its core components, and the benefits of using Cloud Build to automate building, testing, and deploying code changes to various environments. You examined CI best practices including automated building, testing, and version control system integration. And you looked at CI testing to detect and correct bugs or defects in your code to ensure your codebase stays stable and manageable.

You acquired skills in continuous delivery and the iterative development cycle. You learned that CD is essential with CI and automated testing and that the use of feature flags, increment rollout, and blue-green deployments will minimize problems that are commonly caused by human error. In addition, you learned that Github supports CI/CD tools by providing webhooks and APIs and allows those tools to become part of the pull request process. You looked at how value stream mapping optimizes the flow of materials and information required to successfully deliver a product to clients and how configuration management provides a solution to simplify and effectively manage changes to code.

Lastly, you looked at end-to-end tests and how they're a comprehensive evaluation of an application that mimic real-world use. You explored the four different types of releases and the management that must be used to ensure an effective release. You also now know that if a software incident occurs, you can conduct a post-mortem, or incident analysis, to determine what happened, how it happened, and how to create a solution to prevent disruptions in the future.

IT skills in action

Congratulations! You have gained so much knowledge on continuous integration and continuous deployment (CI/CD). There are many technical pieces that are included within CI/CD, but how would you apply the skills you learned in a professional setting?

In this reading, you will review an example of how Docker containers are used in the real world.

The background

Disclaimer: The following scenario is based on a fictitious company.
The Quick Tech Solution Company decided that it is time to rebuild their outdated application into a modern microservice-based architecture with a web frontend. The different development teams knew this was a brilliant—and much needed—idea, so they began to decompose their specific parts of the application into simple microservices in different languages, including Python, Go, and Java. There was very little communication across teams, and each team was unaware of what languages the other teams were using. In addition to using different languages, each team had their own unique process of building and deploying their code to a server. No team worked in the same way.

The nightmare

Soon enough, this became the DevOps team's worst nightmare. There was no consistency in the build process, whatsoever. Because of this, when errors were caught, they had to be sent back to the development team to be corrected. Due to a lack of automation, integration tests would take weeks to complete and releases to production were error prone and infrequent. Nothing was running smoothly. They needed a solution, and they needed it fast.

The solution

The engineering leaders determined that a modern microservice application requires a modern CI/CD solution. Here's what the solution looked like:
First, The DevOps team evaluated various CI/CD tools and chose the ones that best fit their workflow. Then, each development team was required to add a Dockerfile to their repository. This was used to build their component as a container and provided unit tests and integration tests. DevOps made the decision to leverage Kubernetes for orchestrating all the containers and worked with each development team to write deployment manifests in YAML for each component. After the deployment manifests were written, the DevOps team designed a CI/CD pipeline whose purpose was to build containers, deploy them to a fresh—but temporary—Kubernetes cluster using the manifests, and run all of the unit and integration tests.

The aftermath

This gave the DevOps team more confidence in what they built. When these tasks were completed, the DevOps team extended the pipeline so every successful pull request would deploy the containers to the production Kubernetes cluster.