Git, Docker and Kubernetes

Mastering Git, Docker, and Kubernetes empower developers to seamlessly manage a modern application's entire lifecycle, from the first line of code to automated scaling in production. By combining version control, containerization, and orchestration, these tools bridge the gap between development and operations to eliminate the infamous "it works on my machine" dilemma.

What is version control?

Version Control Systems (VCS), also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time. As development environments have accelerated, version control systems help software teams work faster and smarter. They are especially useful for DevOps teams since they help them to reduce development time and increase successful deployments.

Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Besides commonly spread Git, there are some more version control systems:

Software Name	**License Type**	**Focus**
Apache Subversion (SVN)	Open-source	Centralized version control
Mercurial (hg)	Open-source	Distributed version control
CVS	Open-source	Centralized version control
Bazaar (GNU Bazaar)	Open-source	Distributed version control
Unity Version Control (Plastic SCM)	Proprietary	Game development version control
AccuRev	Proprietary	Stream-based version control
Azure DevOps	Proprietary	Integrated DevOps platform
Meta’s Sapling SCM	Proprietary	Large-scale version control
IBM Rational ClearCase	Proprietary	Software configuration management

What is Git?

One of the most popular VCS tools in use today is called Git. Git is a Distributed VCS, a category known as DVCS. Like many of the most popular VCS systems available today, Git is free and open source. Regardless of what they are called, or which system is used, the primary benefits you should expect from version control are as follows.

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

Key Git Concepts

- **Repository:** A folder where Git tracks your project and its history.
- **Clone:** Make a copy of a remote repository on your computer.
- **Stage:** Tell Git which changes you want to save next.
- **Commit:** Save a snapshot of your staged changes.
- **Branch:** Work on different versions or features at the same time.
- **Merge:** Combine changes from different branches.
- **Pull:** Get the latest changes from a remote repository.
- **Push:** Send your changes to a remote repository.

Working with Git

- Initialize Git on a folder, making it a Repository
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered modified
- You select the modified files you want to Stage
- The Staged files are Committed, which prompts Git to store a permanent snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit

Git Commands

Command	Description
git init <directory>	Creates empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository **-q** or **--quiet** Only print error and warning messages; all other output will be suppressed. **--bare** Create a bare repository. If GIT_DIR environment is not set, it is set to the current working directory. **--object-format=<*format*>** Specify the given object **<format>** (hash algorithm) for the repository. **-b <*branch-name*>** **--initial-branch=<*branch-name*>** Use **<branch-name>** for the initial branch in the newly created repository.
git clone <repo>	Clones repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH **-l** or **--local** When the repository to clone from is on a local machine, this flag bypasses the normal "Git aware" transport mechanism and clones the repository by making a copy of HEAD and everything under objects and refs directories. This option does not work with repositories owned by other users for security reasons, and --no-local must be specified for the clone to succeed. **-s** or **--shared** When the repository to clone is on the local machine, instead of using hard links, automatically setup .git/objects/info/alternates to share the objects with the source repository. The resulting repository starts out without any object of its own. **-v** or **--verbose** Run verbosely. Does not affect the reporting of progress status to the standard error stream. **--progress** Report progress status on the standard error stream by default when attached to a terminal, unless --quiet is specified.

git config	Defines configuration to be used for all commits in current repo **--replace-all** Default behavior is to replace at most one line. This replaces all lines matching the key (and optionally --value=**<pattern>**). **--append** Adds a new line to the option without altering any existing values. This is the same as providing **--value=^$** in set. **--comment <message>** Append a comment at the end of new or modified lines. **--global** For writing options: write to global ~/.gitconfig file rather than the repository .git/config, write to $XDG_CONFIG_HOME/git/config file if this file exists and the ~/.gitconfig file doesn't. **--local** For writing options: write to the repository .git/config file. This is the default behavior. **-f <config-file>** **--file <config-file>** For writing options: write to the specified file rather than the repository .git/config.
git add <directory>	Stages all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file **-n** or **--dry-run** Don't actually add the file(s), just show if they exist and/or will be ignored. **-v** or **--verbose** Be verbose. **-f** or **--force** Allow adding otherwise ignored files. **-i** or **--interactive** Add modified contents in the working tree interactively to the index. **-p** or **--patch** Interactively choose hunks of patch between the index and the work tree and add them to the index. **-e** or **--edit** Open the diff vs. the index in an editor and let the user edit it.

git commit	Commits the staged snapshot, but instead of launching a text editor, use <message> as the commit message **-a** or **--all** Automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected. **--amend** Replace the tip of the current branch by creating a new commit. **-C <*commit*>** or **--reuse-message=<*commit*>** Take an existing **<commit>** object, and reuse the log message and the authorship information (including the timestamp) when creating the commit. **-c <*commit*>** or **--reedit-message=<*commit*>** Like -C, but with -c the editor is invoked, so that the user can further edit the commit message. **--fixup=[(amend|reword):]<*commit*>** Create a new commit which "fixes up" **<commit>** when applied with git rebase --autosquash. Plain --fixup=**<commit>** creates a "fixup!" commit which changes the content of **<commit>** but leaves its log message untouched. **-p** or **--patch** Use the interactive patch selection interface to choose which changes to commit. **-m <*msg*>** or **--message=<*msg*>** Use **<msg>** as the commit message. If multiple -m options are given, their values are concatenated as separate paragraphs. The -m option is mutually exclusive with -c, -C, and -F. **--author=<*author*>** Override the commit author. Specify an explicit author using the standard **A U Thor <author@example.com>** format. Otherwise **<author>** is assumed to be a pattern and is used to search for an existing commit by that author (i.e. git rev-list --all -i --author=**<author>**); the commit author is then copied from the first such commit found. **--reset-author** When used with -C/-c/--amend options, or when committing after a conflicting cherry-pick, declare that the authorship of the resulting commit now belongs to the committer.

git status	Lists which files are staged, unstaged and untracked **-b** or **--branch** Show the branch and tracking info even in short-format. **--show-stash** Show the number of entries currently stashed away.
git log	Displays the entire commit history using the default format **--follow** Continue listing the history of a file beyond renames (works only for a single file). **--source** Print out the ref name given on the command line by which each commit was reached. **-<*number*>** or **-n <*number*>** or **--max-count=<*number*>** Limit the output to **<number>** commits. **--skip=<*number*>** Skip **<number>** commits before starting to show the commit output. **--since=<*date*>** **--after=<*date*>** Show commits more recent than **<date>**.
git diff	Shows unstaged changes between your index and working directory **--summary** Output a condensed summary of extended header information such as creations, renames and mode changes. **--output=<*file*>** Output to a specific file instead of stdout. **--raw** Generate the diff in raw format. **--diff-algorithm=(patience|minimal|histogram|myers)** Choose a diff algorithm. The variants are as follows: **myers** The basic greedy diff algorithm, this is the default one. **minimal** Spend extra time to make sure the smallest possible diff is produced. **patience** Use "patience diff" algorithm when generating patches. **histogram** This algorithm extends the patience algorithm to "support low-occurrence common elements".

git revert <commit>	Creates new commit that undoes all of the changes made in <commit>, then apply it to the current branch **-e** or **--edit** With this option, **git revert** will let you edit the commit message prior to committing the revert. **-m parent-number** or **--mainline parent-number** This option specifies the parent number (starting from 1) of the mainline and allows revert to reverse the change relative to the specified parent. **--continue** Continue the operation in progress using the information in .git/sequencer. **--skip** Skip the current commit and continue with the rest of the sequence. **--quit** Forget about the current operation in progress. **--abort** Cancel the operation and return to the pre-sequence state.
git reset <file>	Remove <file> from the staging area but leave the working directory unchanged. This unstages a file without overwriting any changes **-q** or **--quiet** Be quiet, only report errors. **--inter-hunk-context=<*n*>** Show the context between diff hunks, up to the specified **<number>** of lines, thereby fusing hunks that are close to each other. **-U<*n*>** or **--unified=<*n*>** Generate diffs with **<n>** lines of context. The number of context lines defaults to diff.context or 3 if the configuration variable is unset. (-U without **<n>** is silently accepted as a synonym for -p due to a historical accident). **--** Do not interpret any more arguments as options.
git clean	Removes untracked files from the working directory, and is the counterpart to git reset, which only operates on tracked files. **-d** Normally, when no <pathspec> is specified, git clean will not recurse into untracked directories to avoid removing too much. Specify -d to have it recurse into such directories as well.

	-f or **--force** If the Git configuration variable clean.requireForce is not set to false, **git clean** will refuse to delete files or directories unless given -f. **-i** or **--interactive** Show what would be done and clean files interactively. See "Interactive mode" for details. Configuration variable clean.requireForce is ignored, as this mode gives its own safety protection by going interactive. **-n or --dry-run** Don't actually remove anything, just show what would be done. **-X** Remove only files ignored by Git.
git rebase <base>	Rebases the current branch onto <base>, <base> can be a commit ID, branch name, a tag, or a relative reference to HEAD **--onto <newbase>** Starting point at which to create the new commits. **--keep-base** Set the starting point at which to create the new commits to the merge base of **<upstream>** and **<branch>**. Running git rebase --keep-base **<upstream> <branch>** is equivalent to running git rebase --reapply-cherry-picks --no-fork-point -- onto **<upstream>**...**<branch> <upstream> <branch>**. **--apply** Use applying strategies to rebase (calling git-am internally). **--empty=(drop|keep|stop)** How to handle commits that are not empty to start and are not clean cherry-picks of any upstream commit, but which become empty after rebasing (because they contain a subset of already upstream changes): **drop** The commit will be dropped. This is the default behavior. **keep** The commit will be kept. This option is implied when --exec is specified unless -i/--interactive is also specified. **stop** or **ask** The rebase will halt when the commit is applied, allowing you to choose whether to drop it, edit files more, or just commit the empty changes. This option is implied when -i/--interactive is specified. ask is a deprecated synonym of stop.

git reflog	Shows a log of changes to the local repository's HEAD **git reflog show** accepts any of the options accepted by git log. **git reflog delete** accepts options --updateref, --rewrite, -n, --dry-run, and --verbose, with the same meanings as when they are used with expire. **git reflog drop –all** drops the reflogs of all references from all worktrees. **git reflog expire -all** process the reflogs of all references
git branch	Lists all of the branches in your repo **-a** or **--all** List both remote-tracking branches and local branches. **-d** or **--delete** Delete a branch. **-D** Shortcut for --delete --force. **-f** or **--force** Reset **<branch-name>** to **<start-point>**, even if **<branch-name>** exists already. Without -f, git branch refuses to change an existing branch. **-m** or **--move** Move/rename a branch, together with its config and reflog. **-M** Shortcut for --move --force. **-c** or **--copy** Copy a branch, together with its config and reflog. **-C** Shortcut for --copy --force.
git checkout -b <branch>	Creates and checks out a new branch named <branch> **-f** or **--force** When switching branches, proceed even if the index or the working tree differs from HEAD, and even if there are untracked files in the way. This is used to throw away local changes and any untracked files or directories that are in the way. **--ours** **--theirs** When checking out paths from the index, check out stage #2 (ours) or #3 (theirs) for unmerged paths. **-t --track[=(direct|inherit)]** When creating a new branch, set up "upstream" configuration.

git merge <branch>	Merges <branch> into the current branch **--commit** **--no-commit** Perform the merge and commit the result. This option can be used to override --no-commit. **--edit** **-e** **--no-edit** Invoke an editor before committing successful mechanical merge to further edit the auto-generated merge message, so that the user can explain and justify the merge. **--stat** **-n** **--no-stat** Show a diffstat at the end of the merge. **-m <*msg*>** Set the commit message to be used for the merge commit (in case one is created).
git remote add <name> <url>	Create a new connection to a remote repo. After adding a remote, you can use <name> as s shortcut for <url> in other commands
git fetch <remove> <branch>	Fetches a specific <branch> from the repo. Leave off <branch> to fetch all remote refs **--all** **--no-all** Fetch all remotes, except for the ones that has the remote.**<name>**.skipFetchAll configuration variable set. This overrides the configuration variable fetch.all. **-a** or **--append** Append ref names and object names of fetched refs to the existing contents of .git/FETCH_HEAD. Without this option old data in .git/FETCH_HEAD will be overwritten. **--atomic** Use an atomic transaction to update local refs. Either all refs are updated, or on error, no refs are updated. **--dry-run** Show what would be done, without making any changes. **--filter=<*filter-spec*>** Use the partial clone feature and request that the server sends a subset of reachable objects according to a given object filter. When using --filter, the supplied **<filter-spec>** is used for the partial fetch.

git pull <remote>	Fetches the specified remote's copy of current branch and immediately merges it into the local copy Examples of use: Update the remote-tracking branches for the repository you cloned from, then merge one of them into your current branch: git pull git pull origin Merge into the current branch the remote branch next: git pull origin next This leaves a copy of next temporarily in FETCH_HEAD, and updates the remote-tracking branch origin/next. The same can be done by invoking fetch and merge: git fetch origin git merge origin/next
git push <remote> <branch>	Pushes the branch to <remote> along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist Examples of use: **git push** Works like git push **<remote>**, where <remote> is the current branch's remote (or origin, if no remote is configured for the current branch). git push mothership master:satellite/master dev:satellite/dev Use the source ref that matches master (e.g. refs/heads/master) to update the ref that matches satellite/master (most probably refs/remotes/satellite/master) in the mothership repository; do the same for dev and satellite/dev.

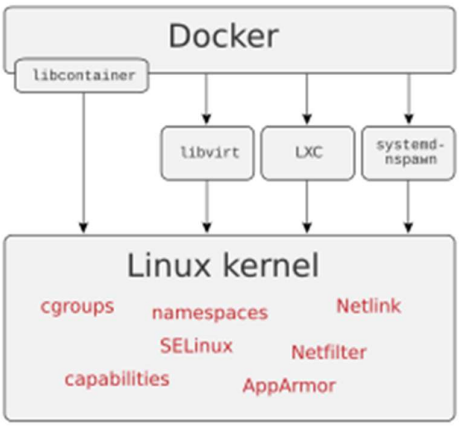
See https://git-scm.com/docs for further references

Concept behind containers

Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. Because all of the containers share the services of a single operating system kernel, they use fewer resources than virtual machines.

Docker can package an application and its dependencies in a virtual container that can, in principle, run on any Linux, Windows, or macOS computer.

What is Docker?



Docker is a set of products that uses operating system-level virtualization to deliver software in packages called containers. Docker automates the deployment of applications within lightweight containers, enabling them to run consistently across different computing environments.

The core software that runs and manages these containers is called Docker Engine. Docker was first released in 2013 and continues to be developed by Docker, Inc. The platform includes both free and paid tiers.

The Docker platform

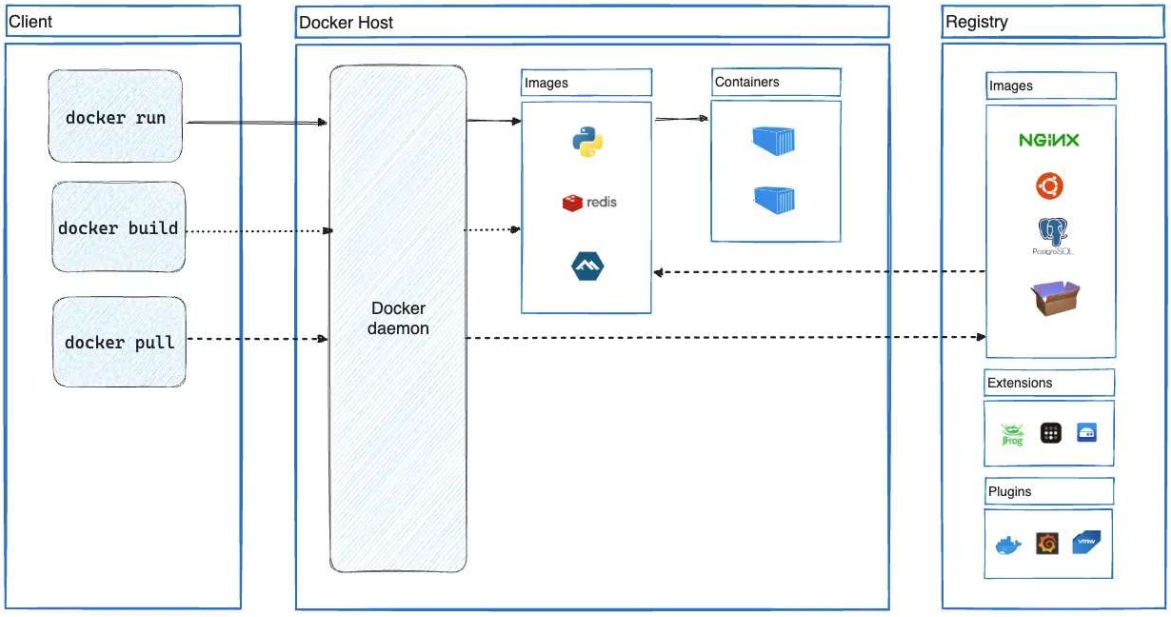
Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security let you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

- Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image that is based on the Ubuntu image but includes the Apache web server and your application, as well as the configuration details needed to make your application run.
You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

- Containers

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.
A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

The underlying technology

Docker is written in the Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality. Docker uses a technology called **namespaces** to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.
These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Instructions

Instruction	Description
ADD	Add local or remote files and directories.
ARG	Use build-time variables.
CMD	Specify default commands.
COPY	Copy files and directories.
ENTRYPOINT	Specify default executable.
ENV	Set environment variables.
EXPOSE	Describe which ports your application is listening on.
FROM	Create a new build stage from a base image.
HEALTHCHECK	Check a container's health on startup.
LABEL	Add metadata to an image.
MAINTAINER	Specify the author of an image.
ONBUILD	Specify instructions for when the image is used in a build.
RUN	Execute build commands.
SHELL	Set the default shell of an image.
STOPSIGNAL	Specify the system call signal for exiting a container.
USER	Set user and group ID.
VOLUME	Create volume mounts.
WORKDIR	Change working directory.

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Docker runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction. This may be after parser directives, comments, and globally scoped ARGs. The FROM instruction specifies the base image from which you are building. FROM may only be preceded by one or more ARG instructions, which declare arguments that are used in FROM lines in the Dockerfile.

BuildKit treats lines that begin with # as a comment, unless the line is a valid parser directive. A # marker anywhere else in a line is treated as an argument. This allows statements like:

```
# Comment
RUN echo 'we are running some # of cool things'
```

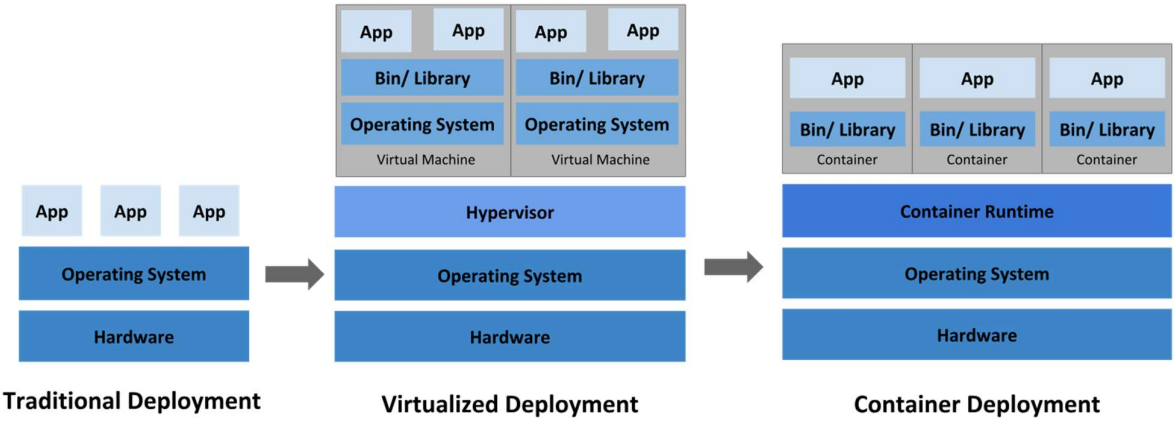
Docker commands

Command	Description	Commonly Used Flags
docker build	Builds a new Docker image from a Dockerfile and a specified build context.	**-t** or **--tag** <name:tag> Applies a repository name and an optional version tag to the image. **-f** or **--file** <path> Points to a specific Dockerfile path if it is not named Dockerfile. **--no-cache** Forces Docker to rebuild all layers from scratch, ignoring cached layers.
docker run	Creates and starts a completely new container instance from a targeted image.	**-d** or **--detach** Runs the container in the background (detached mode) and prints the new container ID. **-p** or **--publish** <host:container> Forwards a port on your local host machine to a port inside the isolated container. **--name** <string> Assigns a memorable, custom name to the container instead of a randomly generated one. **-v** or **--volume** <host:container> Mounts a directory from your host into the container to persist application state or share data. **--rm** Automatically deletes the container's file system and metadata when it exits. **-it** Combines **-i** (interactive) and **-t** (allocate a pseudo-TTY). Essential for running shells like /bin/bash.

docker exec	Runs a brand new command or executable *inside* an already running container instance.	**-it** Keeps stdin open and allocates a terminal, allowing you to interact directly with the process. **-u** or **--user** <username> Specifies the username or UID under which the command should be run inside the container workspace. **-w** or **--workdir** <path> Overrides the default working directory for the execution of this specific command.
docker ps	Queries the daemon to list active container instances on the host system.	**-a** or **--all** Shows all containers managed by the daemon, including those that have exited or stopped. **-q** or **--quiet** Only outputs the alphanumeric container IDs (useful for piping into shell scripts). **--filter** <key=value> Filters output based on conditions like status (status=running), name, or exited error code.
docker images	Lists all read-only images currently cached or built locally on the Docker host machine.	**-a** or **--all** Shows all intermediate image layers generated during multi-stage builds. **-q** or **--quiet** Displays only the distinct 64-character SHA image IDs.
docker pull	Explicitly downloads a remote container image from a configured central network repository.	**-a** or **--all-tags** Downloads all tagged images belonging to a specific repository on the registry. **--disable-content-trust** Skips image verification checks (enabled by default to guarantee image integrity).

docker push	Uploads a locally built or retagged image to a remote central registry.	**--disable-content-trust** Bypasses remote cryptographic signing requirements.
docker rm	Permanently deletes one or more stopped containers from the host filesystem.	**-f** or **--force** Forces the removal of an actively running container (uses a UNIX SIGKILL signal). **-v** or **--volumes** Deletes anonymous volumes attached to the container.
docker rmi	Deletes one or more cached images from the local Docker engine storage.	**-f** or **--force** Forces removal of the image even if it has tags or is referenced by a stopped container.
docker logs	Fetches and displays log outputs generated by a container's primary execution threads (stdout/stderr).	**-f** or **--follow** Streams live log outputs to your console terminal in real-time. **--tail** <number> Specifies how many lines from the end of the log history to display (e.g., --tail 100). **-t** or **--timestamps** Prefixes every logged event line with an exact ISO timestamp.
docker stop	Gracefully halts an active container runtime process environment.	**-t** or **--time** <seconds> Number of seconds to wait for graceful exit before issuing a hard kill signal (10)
docker system prune	Housekeeping command that clears unused data across the local Docker environment.	**-a** or **--all** Deletes all unused images, not just dangling ones (images without a tag). **--volumes** Forces the purge of unused data volumes, which are normally spared to protect database persistence. **-f** or **--force** Skips the manual interactive confirmation prompt.

What is Kubernetes?



Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently.

Kubernetes is an open-source container orchestration engine for automating deployment, scaling, and management of containerized applications. The open-source project is hosted by the Cloud Native Computing Foundation (CNCF).

Kubernetes provides you with:

- **Service discovery and load balancing**
  Kubernetes can expose a container using a DNS name or its own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration**
  Kubernetes allows you to automatically mount a storage system of your choice, such as local storage, public cloud providers, and more.
- **Automated rollouts and rollbacks**
  You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing**
  You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.

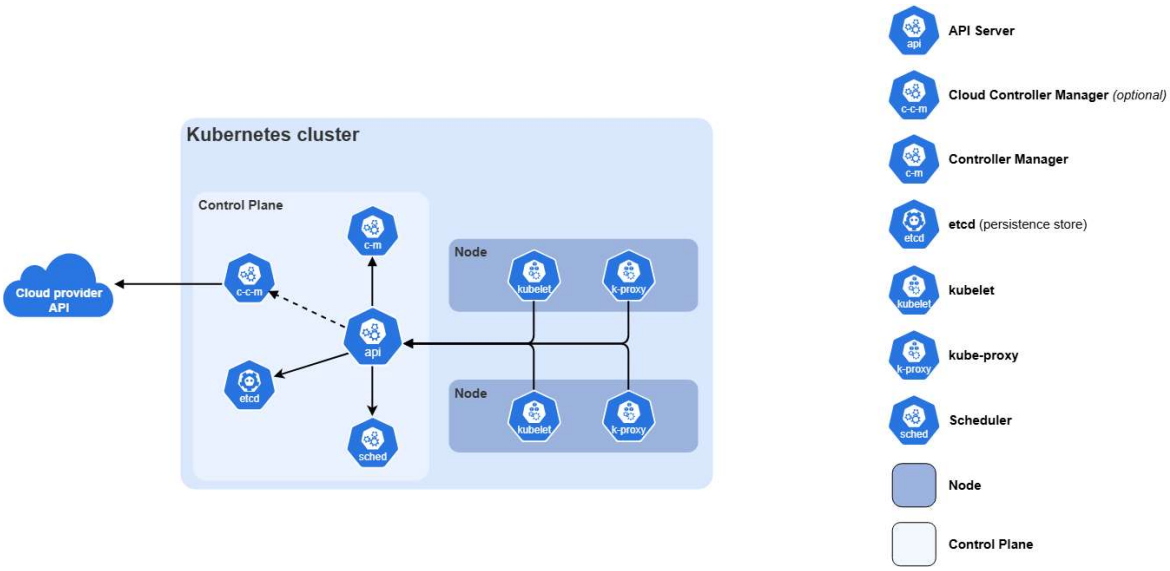
- **Self-healing**
  Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management**
  Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Batch execution**
  Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.
- **Horizontal scaling**
  Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.
- **IPv4/IPv6 dual-stack**
  Allocation of IPv4 and IPv6 addresses to Pods and Services.
- **Designed for extensibility**
  Add features to your Kubernetes cluster without changing upstream source code.

What Kubernetes is not

Kubernetes doesn’t:

- Limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
- Deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
- Provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.
- Dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
- Provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.

Kubernetes Components



A Kubernetes cluster consists of a control plane and one or more worker nodes.

A functional Kubernetes cluster is split cleanly into two structural layers: the Control Plane (which manages the state of the cluster) and one or more Worker Nodes (which host the actual application runtimes).

- **Control Plane Components**

  Manage the overall state of the cluster

  kube-apiserver

  The core component server that exposes the Kubernetes HTTP API.

  etcd

  Consistent and highly-available key value store for all API server data.

  kube-scheduler

  Looks for Pods not yet bound to a node, and assigns each Pod to a suitable node.

  kube-controller-manager

  Runs controllers to implement Kubernetes API behavior.

  cloud-controller-manager (optional)

  Integrates with underlying cloud provider(s).

- **Node Components**

  Run on every node, maintaining running pods and providing the Kubernetes runtime environment:

  kubelet

  Ensures that Pods are running, including their containers.

  kube-proxy (optional)

  Maintains network rules on nodes to implement Services.

  Container runtime

  Software responsible for running containers.

- **Addons**

  Addons extend the functionality of Kubernetes. A few important examples include:

  DNS

  For cluster-wide DNS resolution.

  Web UI (Dashboard)

  For cluster management via a web interface.

  Container Resource Monitoring

  For collecting and storing container metrics.

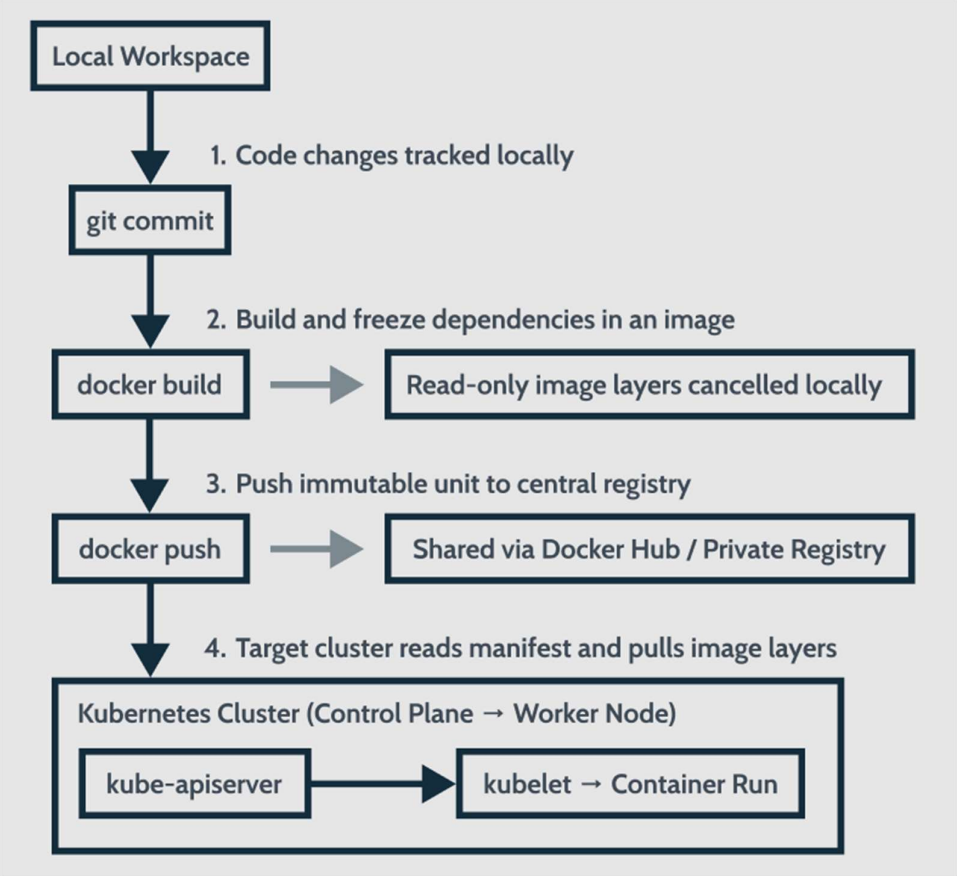
  Cluster-level Logging

  For saving container logs to a central log store.

Main points:

- Kubernetes serves as an open-source container orchestration engine and designed specifically to handle distributed systems resiliently. It automates the deployment, horizontal scaling, and hands-off management of containerized application payloads.
- Kubernetes is not a CI/CD engine**,** Kubernetes does not ingest raw source code, nor does it run compilation or image-building pipelines. Continuous Integration and Delivery workflows are managed by external toolchains like GitHub Actions or GitLab CI; Kubernetes only acts as the final deployment target.
- It does not provide built-in middleware or databases**,** while databases like MySQL, message brokers like Kafka, and caching layers can run beautifully *on top of* Kubernetes, they are not native system components.

From Source Code to Orchestrated Pods



An application flows continuously through these three tools:

- **The Code Layer (Git):** You modify an application file or adjust a configuration schema. Git records the cryptographic snapshot of that precise state. It provides the ultimate safety net, ensuring you can audit, revert, or branch your engineering efforts seamlessly.

- **The Packaging Layer (Docker):** Once your code achieves a functional milestone, Docker takes over. Instead of shipping just your code files, Docker builds an immutable template containing the code, binaries, libraries, and precise configuration adjustments required to run it uniformly anywhere.

- **The Orchestration Layer (Kubernetes):** Finally, when your application must scale globally to handle production volume, it is handed over to Kubernetes. Kubernetes interprets your declarative specification manifests, interacts with the local container runtimes to pull down the standardized Docker layers, and deploys your applications in resilient, self-healing runtime wrappers across a distributed pool of hardware nodes.

Implementing the DevOps Pipeline

Code layer: Local Versioning and Branch Isolation

- Ensure every brand-new service begins with a crisp local execution environment initialized via git init.
- Always perform your feature iterations, patches, or architectural upgrades on isolated feature branches using git checkout -b <branch>. Never commit directly to your primary development or production branches.
- Write atomic, single-focus commits with descriptive flags (e.g., git commit -m "feat: implement database connection retries") so that your version log remains clean and highly readable

Packaging layer: Dockerizing for Environment Parity

- Write a clean, minimalist Dockerfile starting with an official, lightweight base image utilizing the FROM instruction.
- Leverage multi-stage builds by organizing distinct instruction phases within the Dockerfile. Use COPY to bring in dependencies and build-time binaries, then copy only the compiled execution artifacts into the final runtime stage. This heavily minimizes layer sizes and reduces your security attack surface.
- Test your built image locally using detached port redirection: docker run -d -p 8080:80 --name local-test app-repo:v1. Confirm that the container starts up and exposes endpoints correctly without crashing before moving to the registry step.

Orchestration layer: Manifest Drafting and Kubernetes Deployment

- Create your declarative deployment manifests (YAML files) detailing exactly how your application should function at scale.
- Map the configuration details out explicitly: declare your required number of replica sets (Horizontal Scaling), assign your exposed cluster entry points via Services (Load Balancing), and attach external block storage definitions where data persistence is needed.
- Deploy the manifest to your target cluster. The kube-apiserver processes your resource definition, kube-scheduler assigns the target containers to optimized hardware slots , and the local node kubelet pulls down the exact image tag you pushed earlier to spin up live, monitored pods.

Checklist for Production-Ready Deployments

Before declaring an application ready for public client traffic, audit your configuration scripts against the following operational checklist:

- **Audit Trail:**
  Is your current deployment fully accounted for in your Git log? Can any developer on your team trace this production rollout back to a single pull request and specific commit hash?
- **No Host Dependencies:**
  Does your Dockerfile explicitly bundle everything required to run the payload? Ensure that your application runtime never relies on packages, environment paths, or configuration files that only exist natively on your local machine.
- **Active Probes:**
  Have you included explicit liveness and readiness health checks inside your Kubernetes specifications? Without these, Kubernetes cannot orchestrate self-healing updates or determine if a deadlocked application container needs to be killed and rebooted automatically.
- **Credential Separation:**
  Are there any plain-text API keys, passwords, or database string credentials baked into your Docker layers or raw Git commits? All sensitive items must be factored out completely and managed solely through native Kubernetes Secrets.