Product Buildr Team

- Github_Best_Principles
- SOLID Principles of Software EngineeringPage
- Kubernetes
- Docker

Github_Best_Principles

Engineering Standards

Git Standards & Best Practices for Engineering Teams

A practical guide to professional version control — from core concepts and essential commands to team workflows, best practices, and AI-specific considerations.

1. Why Git Matters

In modern software and AI development, code does not live in a single file on one person's laptop. Teams collaborate across branches, timezones, and deployments — and without a robust version control system, that collaboration descends into chaos. Git is the industry-standard distributed version control system that gives every engineer a complete local copy of the project's history, enabling parallel work, safe experimentation, and rapid rollback.

Git vs GitHub

Git is the local command-line tool that tracks changes to files on your machine. GitHub, GitLab, and Azure DevOps are cloud platforms that host Git repositories and add collaboration features: Pull Requests, CI/CD pipelines, code review workflows, and access control.

Why Git is especially critical for AI projects

- **Experiments multiply fast** — Git lets you tag, branch, and compare model iterations.
- **Reproducibility** requires knowing exactly which code produced which result.

- **Model training pipelines** involve many interdependent scripts; broken code needs fast rollback.
- **Team collaboration** on notebooks and data-processing code creates frequent merge scenarios.

2. Core Git Concepts

Before issuing your first command, these seven concepts form the mental model every engineer should internalize:

Concept	What It Means	Analogy
Repository	A folder fully tracked by Git, containing all files and their complete history.	A project filing cabinet with every version of every document.
Commit	A saved snapshot of staged changes with a unique hash (e.g., `a3f9d21`).	A photograph of your project at a specific moment in time.
Branch	An independent line of development; default branch is `main`.	A parallel timeline you can merge back or discard.
Merge	Combines changes from one branch into another.	Merging two document drafts into one final version.
Pull Request	A proposal to merge a branch, used for code review before changes hit `main`.	Submitting a draft for editorial review before publishing.
Remote	The server-hosted copy of the repo (GitHub/GitLab/Azure DevOps).	The shared cloud backup that the whole team reads and writes.
Staging Area	A buffer zone where files wait before being committed (`git add`).	A tray of items to photograph before the shutter fires.

3. Essential Git Commands

Command	Purpose	Example
`git clone`	Copy an existing repository to your local machine.	`git clone https://github.com/org/repo`
`git status`	Check the current status of files and changes.	`git status`

Command	Purpose	Example
`git diff --staged`	Review staged changes before committing.	`git diff --staged`
`git add`	Stage files for the next commit.	`git add src/model.py`
`git commit`	Save staged changes with a commit message.	`git commit -m "feat: add sentiment model"`
`git push`	Upload local commits to the remote repository.	`git push origin feature/login`
`git pull`	Fetch and merge the latest changes from the remote repository.	`git pull origin develop`
`git fetch`	Download updates from the remote repository without merging.	`git fetch origin`
`git branch`	Create, view, or delete branches.	`git branch -d feature/done`
`git switch / checkout`	Switch between branches.	`git switch feature/search`
`git merge`	Combine changes from one branch into another.	`git merge feature/data-pipeline`
`git stash / pop`	Temporarily save and restore uncommitted changes.	`git stash pop`
`git log --oneline`	Display a compact commit history.	`git log --oneline -10`
`git revert`	Safely undo a previously committed change.	`git revert a3f9d21`
`.gitignore`	Exclude files and folders from Git tracking.	`echo "data/" >> .gitignore`

Pro Tip: Commit Messages

Format every commit message as: `type: short imperative description`. Common types include `feat`, `fix`, `refactor`, `docs`, `test`, `chore`, and `perf`. For example, `feat: add FAISS vector search endpoint` or `fix: resolve auth timeout on mobile`. Consistent commit messages make project history easier to read, simplify debugging, and improve collaboration across engineering teams.

4. Standard Team Workflow

Every team follows a branching strategy. The workflow below, a simplified Git Flow, is the standard adopted by most AI engineering teams and scales well from two to two hundred engineers.

Step-by-step explanation

- **Start from develop/main** — Always branch from the latest stable integration point. Pull first.
- **Name your branch clearly** — `feature/`, `bugfix/`, `hotfix/`, or `docs/` prefixes keep the repo readable.
- **Commit small and often** — Each commit should represent one logical change that can be reviewed in isolation.
- **Push early and open a Draft PR** — This signals to teammates what you are working on and enables early feedback.
- **Request review** — At least one peer should approve before merge. Two reviewers for critical paths.
- **Merge and delete** — Once approved, merge and immediately delete the feature branch to keep the namespace clean.

Best Practice: Branch Lifetime

A feature branch that lives longer than 3 days is a risk. Long-lived branches diverge from main, accumulate conflicts, and are harder to review. Keep branches short, focused, and merged fast.

5. Git Best Practices

These are the non-negotiable habits that separate professional engineering teams from chaotic ones:

- Never commit directly to `main` or `master` — always branch and raise a Pull Request.
- Pull before pushing — begin every session with `git pull origin develop` to avoid stale code.
- Write meaningful commit messages — use the `type: description` convention (Section 3).
- Commit small, logical units — one concern per commit, easier to review and revert.
- Review your own diff before opening a PR — be your own first reviewer.
- Keep branches short-lived — aim to merge within 1–3 days.
- Delete merged branches — a clean branch namespace is a healthy repo.
- Resolve conflicts carefully — understand what each side changed before choosing.
- Never force-push to shared branches — it rewrites history and destroys teammates' work.
- Maintain a comprehensive `.gitignore` — exclude secrets, data files, venv, and model weights.

6. Common Mistakes Engineers Make

Every junior engineer (and many seniors) has made these mistakes. Recognising them early saves hours of debugging and awkward team conversations.

Mistake	Why It Hurts	The Fix
Working directly on `main`	Broken code ships immediately; no review gate.	Always branch: `git checkout -b feature/your-task`
Giant commits (100+ files)	Impossible to review; impossible to revert safely.	Commit one logical change at a time.
Vague messages ('fix', 'update')	History becomes unreadable; debugging takes 10x longer.	Use `type: description` format. `feat: add FAISS index`
Forgetting to pull before work	Your branch diverges; conflicts accumulate silently.	`git pull origin develop` at the start of every session.
Committing `.env` or API keys	Secrets are in the public repo forever (even after deletion).	Add `.env` to `.gitignore`; rotate any exposed key immediately.
Pushing large binary files	Bloats the repo permanently; clones become painfully slow.	Add `models/`, `data/` to `.gitignore`; use DVC or Git LFS.

Warning: Exposed Secrets

If you accidentally commit an API key, password, or `.env` file, rotate the credential immediately. Deleting the file in a subsequent commit does not remove it from the repository's history. Once exposed, the credential should be considered permanently compromised and must be invalidated or replaced at its source.

7. Git for AI Engineering Projects

AI repositories have unique challenges that standard software projects do not face: large binary model weights, auto-generated notebook outputs, and massive datasets. Here is how to manage them professionally.

Managing Jupyter Notebooks

- Install `nbstripout` to automatically clear cell outputs before each commit.
- Keep notebook outputs out of Git, as they can bloat diffs and create unnecessary merge conflicts.
- Move finalized and reusable logic from notebooks into Python modules within the `src/` directory.
- Use a consistent naming convention, such as `01_data_prep.ipynb` and `02_feature_engineering.ipynb`, to improve organization and readability.

Handling Datasets and Model Weights

- Use DVC (Data Version Control) to version datasets and model files outside Git.
- Use MLflow or Weights & Biases to track experiment metadata, metrics, and artefacts.
- Store large files in S3, GCS, or Azure Blob Storage — never in the Git repository.
- Commit config files and data manifests (paths, checksums) rather than the data itself.

8. Conclusion

Git is more than just a technical tool; it is the foundation of professional engineering culture. Clean commits create a clear project history, short-lived branches promote focused development, and thorough code reviews encourage collaboration while helping identify issues before they reach production.

For AI engineers, where experiments evolve rapidly and projects involve complex pipelines, Git discipline is essential for maintaining reproducibility, traceability, and efficient teamwork. The practices outlined in this guide are not merely recommendations but widely accepted industry standards. By adopting these practices, engineering teams can improve collaboration, maintain code quality, and build reliable software and AI solutions with confidence.

Git Standards & Best Practices for Engineering Teams

SOLID Principles of Software EngineeringPage

Software Engineering

SOLID Principles of Software Engineering

Five enduring design principles that help teams build maintainable, scalable, and testable systems — from traditional enterprise services to modern AI-powered, agentic applications.

Introduction

Enterprise applications accumulate complexity rapidly. Without clear design principles, teams produce tightly coupled, monolithic classes that become expensive to modify and nearly impossible to test in isolation. SOLID provides a shared language and a practical framework that every engineer can apply from day one — regardless of team size or technology stack.

The SOLID Principles

S Single Responsibility Principle (SRP)

A class should have one, and only one, reason to change. Mixing concerns — e.g., business logic with data persistence — makes changes risky and testing difficult.

```
# X  Mixed concerns
class OrderService:
    def process(self, order): ...
    def send_email(self, order): ...   # unrelated concern

# OK  Separated concerns
class OrderService:
    def process(self, order): ...
class NotificationService:
    def send_email(self, order): ...
```

O Open / Closed Principle (OCP)

Software entities should be open for extension but closed for modification. Use abstraction and polymorphism to add new behaviour without touching existing, tested code.

```
from abc import ABC, abstractmethod
class Discount(ABC):
    @abstractmethod
    def apply(self, price: float) -> float: ...

class SeasonalDiscount(Discount):        # extend by adding new class
    def apply(self, price): return price * 0.9
class LoyaltyDiscount(Discount):
    def apply(self, price): return price * 0.85
```

L Liskov Substitution Principle (LSP)

Objects of a subclass must be replaceable for objects of the superclass without breaking correctness. Violating LSP produces unexpected runtime failures in polymorphic code.

```
class Bird:
    def fly(self): return 'flying'

# X  Ostrich cannot fly — LSP violated
class Ostrich(Bird):
    def fly(self): raise NotImplementedError
```

```
# OK  Redesign hierarchy around capability
class FlyingBird(Bird): ...
class Ostrich(Bird): ...   # omits fly()
```

I Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. Break large interfaces into smaller, role-specific ones to reduce coupling.

```
from abc import ABC, abstractmethod
# OK  Segregated interfaces
class Readable(ABC):
    @abstractmethod
    def read(self) -> str: ...
class Writable(ABC):
    @abstractmethod
    def write(self, data: str): ...
class FileManager(Readable, Writable):   # only what's needed
    def read(self): ...
    def write(self, data): ...
```

D Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. This enables swapping implementations (e.g., databases, APIs) without modifying business logic.

```
from abc import ABC, abstractmethod
class MessageBroker(ABC):               # abstraction
    @abstractmethod
    def publish(self, msg: str): ...

class KafkaBroker(MessageBroker):       # low-level
    def publish(self, msg): ...

class OrderPipeline:                    # high-level
    def __init__(self, broker: MessageBroker):
        self.broker = broker             # injected dependency
```

SOLID Principles — Quick Reference

Principle	Purpose	Key Benefit
S - Single Responsibility	Each class has one reason to change	Easier maintenance & debugging
O - Open/Closed	Open for extension, closed for modification	Safe feature addition without breakage
L - Liskov Substitution	Subtypes must be substitutable for base types	Reliable polymorphism & fewer runtime errors
I - Interface Segregation	Clients depend only on what they use	Reduced coupling & leaner interfaces
D - Dependency Inversion	Depend on abstractions, not concretions	Flexible, testable, mockable components

Practical Benefits in Enterprise Applications

- **Maintainability** — focused classes reduce the blast radius of change.
- **Scalability** — loosely coupled modules scale independently.
- **Testability** — abstractions and DI enable fast, isolated unit tests.
- **Extensibility** — OCP and DIP allow new features without regression risk.
- **Team collaboration** — clear boundaries reduce merge conflicts and ownership ambiguity.
- **Reduced technical debt** — coherent designs resist entropy over time.
- **AI & Agentic systems** — composable, interchangeable components simplify agent orchestration, tool integration, and model swapping.
- **Long-term sustainability** — systems remain comprehensible and adaptable as requirements evolve.

Common Anti-Patterns & Mistakes

Anti-Pattern	Problem	SOLID Violation
God Class	One class owns all business logic, making any change risky	SRP
Tight Coupling	Concrete dependencies hard-coded across layers	DIP
Large Interfaces	Clients forced to implement unused methods	ISP
Hard-Coded Deps	Infrastructure choices baked into business logic	DIP, OCP
Deep Inheritance	Fragile base class; subclasses break on parent changes	LSP, OCP

Common Mistakes to Avoid

- Over-engineering simple solutions with unnecessary abstractions.
- Misusing inheritance where composition is more appropriate.
- Ignoring dependency injection — leads to untestable, tightly coupled code.
- Designing large, bloated interfaces that violate ISP.
- Applying SOLID mechanically rather than purposefully.

Best Practices & Recommendations

- **Design for change** — assume requirements will evolve; build in flexibility.
- **Favour composition over inheritance** for flexibility and reduced coupling.
- **Keep classes focused** — if a class is hard to name, it probably does too much.
- **Program to abstractions** — use interfaces and ABCs at architectural boundaries.
- **Apply dependency injection consistently** — inject dependencies through constructors.
- **Review designs during code reviews** — SOLID violations are often visible at PR time.
- **Refactor continuously** — address violations incrementally rather than in big-bang rewrites.

Conclusion

The SOLID principles are among the most enduring and transferable concepts in software engineering. Adopted as living standards rather than one-time guidelines, they enable teams to confidently evolve complex systems from traditional enterprise services to modern AI-powered, agentic applications. Every engineer, regardless of seniority, should internalise these principles and apply them deliberately in day-to-day design decisions, code reviews, and architectural discussions.

SOLID Principles of Software Engineering

Kubernetes

An Overview of Modern Container Orchestration

Abstract

Modern software systems are expected to be highly available, scalable, and resilient while supporting rapid deployment cycles. Although containerization technologies such as Docker simplify application packaging and portability, they do not address the operational challenges of managing applications at scale. Kubernetes has emerged as the industry-standard container orchestration platform by providing automated deployment, scaling, service discovery, load balancing, and self-healing capabilities. This article presents an overview of Kubernetes, its architecture, core components, deployment workflow, and managed Kubernetes services.

1. Introduction

The rapid adoption of microservices and cloud-native architectures has fundamentally transformed the way software applications are designed and deployed. Modern applications often consist of multiple independent services that communicate over a network and must remain available despite hardware failures, software updates, and changing user demand.

Traditional deployment approaches, where applications are installed directly on physical or virtual machines, introduce several operational challenges:

- Environment inconsistencies between development and production

- Limited scalability
- Manual application deployment and monitoring
- Increased downtime during updates
- Difficulty recovering from infrastructure failures

Containerization solved many of these challenges by packaging applications together with their runtime environment. However, managing hundreds or thousands of containers across multiple machines requires an orchestration platform capable of automating operational tasks. Kubernetes was developed to address these challenges.

2. From Containers to Container Orchestration

A container packages an application together with its libraries, dependencies, and runtime environment into a lightweight, portable execution unit. Unlike virtual machines, containers share the host operating system kernel, making them significantly more resource-efficient.

Docker became the most widely adopted container platform because it allows developers to build an application once and execute it consistently across different environments.

Although containers simplify deployment, organizations still face operational questions:

- How should containers be distributed across multiple servers?
- How can failed containers be restarted automatically?
- How can applications scale during periods of increased demand?
- How can traffic be balanced across multiple application instances?
- How can updates be performed without interrupting users?

Container orchestration platforms automate these operational responsibilities. Kubernetes has become the de facto standard for container orchestration due to its scalability, portability, and extensibility.

3. Kubernetes Architecture

A Kubernetes cluster consists of two major components: the Control Plane and Worker Nodes. The diagram below illustrates how these components interact to manage and execute workloads.

Figure 1: Kubernetes Cluster Architecture — Control Plane and Worker Nodes

Control Plane

The Control Plane is responsible for managing the overall state of the cluster. Rather than executing application workloads, it continuously monitors the cluster and makes scheduling and management decisions. Its primary responsibilities include maintaining the desired cluster state, scheduling workloads onto worker nodes, monitoring cluster health, and handling communication with users and external tools.

Component	Responsibility
API Server	Acts as the primary entry point into the Kubernetes cluster. Every operation performed by users, automation tools, or applications is processed through the API Server.
etcd	A distributed key-value database that stores the cluster's persistent configuration and state, including Deployments, Services, Pods, ConfigMaps, and Secrets.
Scheduler	Determines the most suitable Worker Node for newly created Pods based on resource availability, scheduling policies, and node constraints.

Component	Responsibility
Controller Manager	Continuously compares the desired state with the actual cluster state, and automatically takes corrective actions whenever discrepancies occur.

Worker Nodes

Worker Nodes execute the actual application workloads. Each Worker Node contains the following components:

Component	Responsibility
Kubelet	Communicates with the Control Plane and ensures that Pods assigned to the node are running correctly.
Container Runtime	Responsible for pulling container images, creating containers, and managing their execution. Common runtimes include containerd and CRI-O.
Kube Proxy	Manages network communication within the cluster by routing traffic to the appropriate Pods and providing internal load balancing.

4. Core Kubernetes Objects

Kubernetes defines a set of core API objects that describe how applications are deployed, accessed, and managed.

Pod

A Pod is the smallest deployable unit in Kubernetes. It represents one or more containers that share networking and storage resources while operating as a single execution unit. Although Pods may contain multiple tightly coupled containers, most production workloads deploy one application

container per Pod.

Deployment

A Deployment manages the lifecycle of Pods and maintains the desired application state. Instead of manually creating Pods, developers define a Deployment using a YAML configuration file. Kubernetes continuously ensures that the required number of Pod replicas are available, and provides rolling updates, automatic rollback, horizontal scaling, and self-healing.

Service

Pods are ephemeral resources whose IP addresses change whenever they are recreated. A Service provides a stable network endpoint that allows applications to communicate reliably without depending on individual Pod addresses. Services also provide internal load balancing, service discovery, and stable DNS names.

Ingress

Ingress manages external access to applications running inside the cluster. It routes incoming HTTP and HTTPS traffic to the appropriate Services based on host names or URL paths, while providing centralized traffic management.

5. Declarative Configuration

One of Kubernetes' defining characteristics is its declarative approach to infrastructure management. Instead of writing instructions describing how resources should be created, developers describe what the desired infrastructure should look like using YAML manifests. The Control Plane continuously compares this desired configuration with the actual cluster state and automatically performs corrective actions whenever inconsistencies occur.

Example: Deployment Manifest

The following YAML defines a Deployment that runs three replicas of an nginx web server container, with resource limits and a label selector:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: "64Mi"
              cpu: "250m"
            limits:
              memory: "128Mi"
              cpu: "500m"
```

Example: Service Manifest

The following YAML defines a Service that exposes the Deployment above on port 80, routing traffic to all Pods with the label app: nginx:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Once applied with kubectl apply -f manifest.yaml, Kubernetes takes ownership of ensuring the cluster matches this specification — automatically creating, replacing, or scaling Pods as needed.

6. Deployment Workflow

The deployment lifecycle in Kubernetes follows a well-defined sequence:

1. The developer builds a container image.
2. The image is stored in a container registry.
3. A Deployment manifest is submitted using kubectl apply.
4. The API Server validates and accepts the request.
5. etcd stores the desired cluster configuration.
6. The Controller Manager detects that new Pods must be created.
7. The Scheduler selects suitable Worker Nodes.
8. Kubelet instructs the Container Runtime to pull the required image.
9. The container starts inside a Pod.

10. Services expose the application for internal communication; Ingress manages external access.

This declarative workflow enables Kubernetes to automate deployment, scaling, recovery, and networking without manual intervention.

7. Managed Kubernetes Services

Although Kubernetes can be deployed and managed manually, maintaining the Control Plane requires significant operational expertise. Cloud providers therefore offer managed Kubernetes services that abstract much of the infrastructure complexity.

Cloud Provider	Managed Service
Microsoft Azure	Azure Kubernetes Service (AKS)
Amazon Web Services	Elastic Kubernetes Service (EKS)
Google Cloud	Google Kubernetes Engine (GKE)

In managed environments, the cloud provider operates the Control Plane, performs upgrades, monitors cluster health, and manages infrastructure availability. Developers primarily focus on building applications and deploying workloads.

8. Conclusion

Kubernetes has become the foundation of modern cloud-native application deployment by providing an automated platform for container orchestration. Its architecture enables applications to remain highly available, fault tolerant, and scalable while reducing operational overhead through automation.

By combining containerization with declarative infrastructure management, Kubernetes simplifies the deployment of distributed applications across on-premises and cloud environments. As organizations continue adopting microservices and DevOps practices, Kubernetes remains one of the most important technologies for building resilient and production-ready software systems.

Docker

Introduction

Modern software applications are expected to run consistently across development, testing, and production environments. Traditionally, developers faced a common challenge known as the **"It Works on My Machine"** problem — where an application would function perfectly on one machine but fail on another due to differences in operating systems, libraries, configurations, or software versions.

Docker was created to solve this challenge through a technology known as **containerisation**. Today, Docker has become one of the most widely used platforms for application development, deployment, and cloud-native computing.

What is Docker?

Docker is a containerisation platform that packages an application along with all its dependencies, libraries, runtime environments, and configuration files into a standardised unit called a **container**. This ensures that applications run consistently regardless of the environment in which they are deployed.

In simple terms:

Docker = Application + Dependencies + Runtime + Configuration

What is Containerisation?

Containerisation is the process of packaging an application and everything it needs to run into an isolated environment known as a container. A container typically contains:

- Application Code
- Runtime Environment
- Libraries & Dependencies
- Configuration Files

Containers are lightweight because they share the host operating system kernel instead of running their own operating system — making them faster and far more resource-efficient than traditional virtual machines.

Docker vs Virtual Machines

Understanding the difference between Docker containers and Virtual Machines (VMs) is fundamental to understanding why Docker has become so widely adopted.

Feature	Docker Containers	Virtual Machines
OS Required	Shared Host OS	Separate OS per VM
Startup Time	Seconds	Minutes
Resource Usage	Low	High
Storage	Lightweight	Heavy
Performance	High	Moderate
Portability	Excellent	Limited

Docker Architecture

Docker consists of several key components that work together to build, ship, and run containers.

Docker Client

The Docker Client is the interface through which users interact with Docker. It accepts commands and sends them to the Docker Daemon to execute.

```
docker run
docker build
docker ps
docker logs
```

Docker Daemon

The Docker Daemon is the background service responsible for building images, running containers, managing networks, managing volumes, and pulling images from registries. It performs all the actual work behind Docker operations.

Docker Hub

Docker Hub is Docker's default public image registry. It stores thousands of pre-built images — including Ubuntu, Python, Nginx, MySQL, PostgreSQL, and Redis — so developers can pull them directly without building from scratch.

Docker Image

A Docker Image is a **read-only blueprint** used to create containers. It contains the application code, dependencies, runtime, and configuration. Images are built from a Dockerfile and can be shared via registries like Docker Hub.

Docker Container

A Docker Container is a **running instance** of an image. One image can create multiple independent containers. The relationship flows as:

```
Dockerfile →      Image →      Container(s)
```

Docker Desktop

Docker Desktop is the application installed on a local machine for development. It bundles together the Docker Engine, Docker CLI, Docker Compose, and a graphical user interface — simplifying Docker management for developers on macOS, Windows, and Linux.

Docker Volumes

Containers are ephemeral by nature — if a container is deleted, any data stored inside it is lost. Docker Volumes provide **persistent storage** that lives outside the container's lifecycle, ensuring data remains available even after a container is removed or recreated.

Docker Logs

Logs record application activity inside running containers, including startup messages, database connections, user events, and errors. They are essential for monitoring and troubleshooting.

```
docker logs <container-id>
```

docker Networking

Docker Networking allows containers to communicate with each other securely. Containers reference each other by **service name** rather than IP address, and Docker resolves these names via its internal DNS system.

```
backend:8000
mysql:3306
```

Docker Compose

Docker Compose manages multi-container applications using a single YAML configuration file. Instead of running each service manually, you define all services, their images, ports, and dependencies in one place.

Then start everything with a single command:

```
docker compose up
```

Docker automatically builds images, creates containers, sets up networks, and starts all services in the correct order.

What is a Dockerfile?

A Dockerfile is a plain-text file containing step-by-step instructions for building a Docker image. Each instruction adds a layer to the image.

```
FROM python:3.12
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["uvicorn", "main:app", "--host", "0.0.0.0"]
```

This tells Docker which base image to use, which directory to work in, which files to copy in, which dependencies to install, and which command to run when the container starts.

Docker Installation

Getting Docker up and running on your machine takes just a few minutes.

Step 1 — Install Docker Desktop

Download Docker Desktop from Docker's official website at **docker.com**. The installer includes the Docker Engine, Docker CLI, Docker Compose, and the Docker Desktop GUI.

Step 2 — Verify Installation

Open a terminal and run:

```
docker --version
```

You should see output similar to: *Docker version 26.x.x*

Step 3 — Test Docker

Run the following command to confirm Docker is working correctly:

```
docker run hello-world
```

If Docker is installed properly, you will see a success message confirming the setup is complete.

Deploying an Application with Docker

The following steps walk through the full workflow of containerising and running a Python application with Docker.

Step 1 — Create a Dockerfile

```
FROM python:3.12
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["uvicorn", "main:app"]
```

Step 2 — Build the Image

```
docker build -t myapp .
```

Step 3 — Verify the Image

```
docker images
```

Step 4 — Run the Container

```
docker run -p 8000:8000 myapp
```

Step 5 — Check Running Containers

```
docker ps
```

Step 6 — Access the Application

Open your browser and navigate to:

```
http://localhost:8000
```

Your application is now live inside a Docker container.

Common Docker Commands

Command	Description
docker pull nginx	Download an image from Docker Hub
docker run nginx	Create and start a container
docker ps	List all running containers
docker images	List all local images
docker stop	Stop a running container
docker rm	Remove a stopped container
docker logs	View container logs
docker build -t myapp .	Build an image from a Dockerfile

docker compose up	Start a multi-container application
docker compose down	Stop and remove all Compose services

Conclusion

Docker has transformed the way modern applications are built, packaged, and deployed. By using containers, developers can ensure consistent behaviour across all environments while reducing infrastructure complexity and overhead.

Components such as Docker Images, Containers, Volumes, Networking, and Docker Compose together make Docker an indispensable technology in modern software development, DevOps, cloud computing, and microservices architectures.

Whether you are deploying a simple web application or orchestrating hundreds of microservices with Kubernetes, Docker provides the foundation that modern engineering teams rely on every day.