Elements of Microservices in Cloud

Destiny Anyaiwe, Ph.D

August 16, 2025

Table of Contents

Introduction

This handout serves as a guide to exploring the principles, design, and implementation of microservices architecture in cloud-native applications, while also providing a foundational service layer for the machine learning components of this course. It is intended to be a practical resource as students learn to build, deploy, and manage scalable, distributed systems using industry-standard tools such as Docker, Kubernetes, and API gateways.

CHAPTER 1

Introduction to Microservices

1.1 Monolithic Applications

A monolithic application is a software architecture in which all components of the application are tightly coupled and built as a single, indivisible unit. In practical terms, a monolithic architecture typically consists of one codebase, one repository, and one executable or deployment package. All functionalities —ranging from the user interface to business logic and data access— are developed, tested, and deployed together as a single application.

This design implies that any update, even to a small part of the system, requires redeploying the entire application. To better understand whether an application follows a monolithic architecture, it is important to first identify its core components and assess whether the system is structured as a single-tiered unit. Common components of a monolithic software system may include the User Interface (UI), Authorization and Authentication modules, Presentation Layer, Business Logic Layer, Database Access Layer, Application Integration Services, and auxiliary modules like Notification or Reporting.

Let us consider an Electronic Voting (EV) system as an example. Can you identify and list the key components of such a system alongside their corresponding features? Doing so will help clarify how all functionalities are interconnected within a monolithic architecture and reveal potential limitations or bottlenecks in design.

The EV system can include:

1. DRE (Direct Recording Electronic) machines – touchscreen or button-based terminals where voters cast their ballots.

2. Optical scan voting systems – where paper ballots are scanned and counted electronically.
3. Internet or mobile voting systems – used in limited cases, mostly for absentee or overseas voters.
4. Voter registration and authentication devices – such as biometric verification machines used in some countries (e.g., Nigeria, India).
5. Biometric fingerprint scanners
6. Smart card readers
7. QR code scanners

An example of an EV system is an online voting system (OVS). Based on the requirements, one can identify its key features to include:

- ◇ User Authentication: Ensuring the user's login via username/password, OTP, or biometrics, and voter eligibility verification.
- ◇ Ballot Management: Dynamically generate ballots based on voter category or district. Support for ranked-choice or single-choice voting.
- ◇ End-to-End Encryption: Votes are encrypted during transmission and storage.
- ◇ Vote Anonymity: Votes must remain untraceable to individual voters after submission.
- ◇ Real-Time or Scheduled Tallying: Automated result computation and visualization.
- ◇ Auditability: System must allow for independent verification of vote integrity (e.g., through cryptographic proofs or logs).
- ◇ Access Control & Admin Tools: Admin panel for election setup, voter roll uploads, vote monitoring, and result management.

Monolithic architecture can take various forms, evolving in complexity and modularity over time —namely as **single-process monoliths**, **modular monoliths**, and **distributed monoliths**. A **single-process monolith** is the most basic form, where the entire application —comprising the user interface, business logic, and data access layer— is developed, built, and deployed as one tightly coupled unit within a single process. This structure is straightforward to manage in small applications, but quickly becomes rigid and difficult to maintain as the codebase grows. To address this, developers often move toward a **modular monolith**, where the codebase is logically divided into distinct modules or components, each responsible for a specific concern, such as authentication, voting, or report generation in an online voting system, for example. These modules are well-organized and may even enforce boundaries using techniques like clean architecture or domain-driven design; however, they are still compiled and deployed as a single application. While modular monoliths improve code maintainability and testability, they do not solve the problem of tight runtime coupling, as modules often rely on direct method calls and shared memory. In an attempt to scale further or adopt modern cloud paradigms, some teams break the application into separate deployable services, inadvertently creating a **distributed monolith**. In this case, components are deployed in different files/folders or servers, giving the illusion of independent services, but they re-

main tightly interdependent, sharing databases, requiring coordinated deployments, and relying heavily on synchronous communication. These systems suffer from the worst of both worlds: the operational complexity of a distributed system without the benefits of service independence. Despite their architectural differences, what ties these three forms together is their reliance on **tight coupling and shared deployment concerns**, which hinder independent scaling, fault isolation, and true service autonomy. They reflect different stages of evolution within the monolithic paradigm but remain bound by the same underlying constraints.

In the monolithic architecture, Figure 1.1 based on the OVS example, all core functionalities—such as user authentication, ballot management, vote submission, tallying, and auditing—are tightly integrated into a single application. This means that all components share the same codebase and database, and are deployed as one indivisible unit.

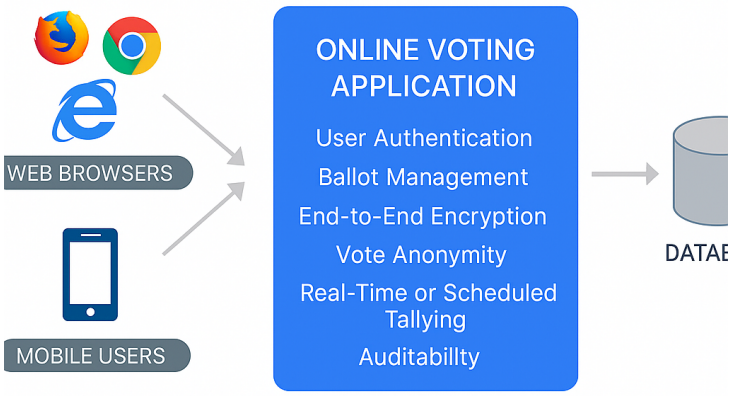


Figure 1.1. Monolithic Architecture for an Online Voting System

1.1.1 Implementation

Let Figure 1.2 be a base folder (bash) structure for the OVS example. To implement an online voting system using a monolithic architecture, a programmer begins by setting up a single unified project where all components—authentication, ballot management, vote casting, and result tallying—reside in one codebase. The project structure is carefully organized with folders for each functional area, such as ‘auth‘, ‘voting‘, ‘tallying‘, and shared components like ‘utils‘, ‘templates‘, and ‘static‘ assets. At the heart of the system is a single entry point (such as ‘main.py‘ or ‘app.js‘), which initializes the application, loads configurations, and sets up all routes.

In this monolithic setup, the programmer designs centralized routing where all user requests —whether for login, voting, or result viewing— are directed through a unified controller. All modules share the same database models, which are defined in one place and used across the system. For example, user authentication, ballot generation, and vote tallying all interact directly with the same database, creating **tight coupling** between modules.

Shared business logic, like verifying user identity or generating voting tokens, is placed in utility functions and used across different parts of the application. All features and functions are integrated directly within the application rather than as separate services, which simplifies communication and deployment. The frontend is also embedded in the same application —HTML templates or static files are served directly by the backend, with forms and AJAX calls interacting seamlessly with server-side routes.

Deployment is straightforward. The entire application is packaged and deployed together, often via a virtual machine using a simple web server (a single Docker image can be used, but let's downplay this fact for a moment). Logging and configuration are centralized, with all modules using the same environment file and logging system. The programmer also sets up a basic CI/CD pipeline to handle code checks, testing, and deployment as one unit.

In a monolithic design, all parts of the system are tightly integrated, which makes development and testing relatively straightforward at a small scale. However, this tight coupling also means that a change in one module —such as authentication— can directly impact other modules like vote submission, as seen in our online voting system example. This illustrates both the simplicity and the inherent limitations of the monolithic approach.

```
/voting-system
├── /auth              # Authentication logic
├── /ballots           # Ballot creation and management
├── /voting            # Vote casting logic
├── /tallying          # Vote counting and results
├── /static            # CSS, JS, and images
├── /templates         # HTML templates or views
├── /db                # Database models and migrations
├── /utils             # Shared utility functions
├── config.env         # Environment variables
├── main.py / app.js   # Entry point
```

Figure 1.2. Base folder structure

1.2 Microservices Architecture

A fundamental principle of software design is to promote **loose coupling** between components to enhance flexibility, scalability, and maintainability. Traditional monolithic architectures, however, are often characterized by *tight coupling*, where components are closely interdependent and changes in one part can affect the entire system. To address these limitations, **microservices architecture** has emerged as an alternative approach that structures applications as a collection of small, independently deployable services (called microservices), each responsible for a specific business function, thereby enabling greater modularity and system resilience.

In microservices architecture, large applications are developed as a suite of loosely coupled modules/components in which each component supports a specific business goal/service. Each component is equipped with a well-defined interface to communicate with other sets of services and manage its database. A service managing its database implies that the service can use a type of database that is best suited to its needs. This boosts security based on loose coupling.

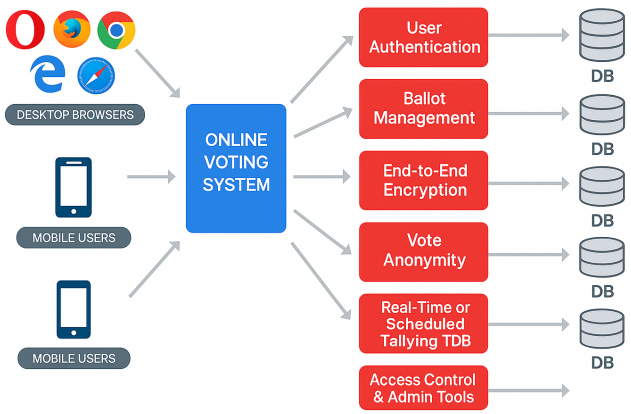


Figure 1.3. Microservices Architecture for an Online Voting System

The microservices architecture (Figure 1.3) breaks down the system into independent, loosely coupled services. Each core functionality (e.g., authentication, encryption, tallying) is implemented as a standalone service, which can be developed, deployed, and scaled independently. This approach offers greater flexibility, fault isolation, and maintainability, especially in cloud-native and large-scale environments. Observe how the database is handled in this architecture and compare it to the tightly coupled database dependency typically found in monolithic architectures.

These visual representations help clarify the differences in structure, communication, and responsibility/resource allocation between the two architectural styles, using the online voting system as a practical example.

1.3 Monolithic vs Microservices Architecture

Monolithic and microservices architectures represent two ends of the spectrum in software design. Monoliths favor simplicity and centralized control, while microservices offer scalability, modularity, and resilience at the cost of increased complexity. Understanding their trade-offs is essential for designing systems that align with technical goals and organizational capabilities. Table 1.1

Table 1.1. Comparative Summary of Monolithic and Microservices

Dimension	Monolith	Microservices
Deployment unit	Single artifact	Many small services
Scaling granularity	Whole application	Per service
Data model	Single shared schema	Decentralized; per-service storage
Coupling	Tight, in-process	Loose, over network APIs
Release cadence	Synchronized	Independent
Operational complexity	Low for small apps	High (orchestrators, service mesh)
Suitable team size	Small, cohesive teams	Multiple autonomous teams
Typical use cases	Proof-of-concepts, simple line-of-business apps	Large-scale, cloud-native platforms

1.3.1 Benefits of Microservices Architecture

Microservices architecture offers a powerful approach for building and scaling large, complex applications by breaking them down into smaller, independently deployable services. One of its greatest strengths lies in enabling continuous delivery and deployment, allowing teams to release updates more frequently and with reduced risk. Each microservice is small and self-contained, making it easier to understand, test, and maintain. This leads to better testability, as individual services can be tested quickly and in isolation, and better deployability, since updates to one service do not require redeploying the entire system.

Microservices also support organizational scalability by aligning development efforts with team boundaries. Teams can take ownership of specific services, developing, testing, deploying, and scaling them independently of others. This independence not only fosters faster development cycles but also removes applications' downtime, reduces inter-team dependencies, and bottlenecks.

From a developer's perspective, working on a smaller codebase improves focus and comprehension. Thereby, improving the product quality with reduced errors and defects. The IDE remains responsive, the application starts faster, and development workflows become more productive. These seemingly small advantages significantly boost efficiency,

especially during iterative development.

Another major benefit is fault isolation. In a monolithic system, a single faulty component—such as a memory leak—can crash the entire application. With microservices, such faults are contained within the failing service, allowing the rest of the system to remain operational.

Finally, microservices promote technological flexibility. Teams can choose the most appropriate technology stack for each service, and even rewrite services using new stacks when necessary, eliminating long-term lock-in and enabling technological evolution over time.

In essence, microservices not only make large systems easier to manage and scale, but they also empower teams to work more independently and deliver value faster, all while improving system resilience and long-term maintainability.

1.3.2 Challenges of Microservices Architecture

While microservices architecture offers significant benefits in terms of scalability, agility, and fault isolation, it also introduces a number of challenges that developers and operations teams must carefully manage. One of the foremost drawbacks is the added complexity of building a distributed system. Unlike monolithic applications, where components communicate via direct function calls, microservices require developers to implement and manage inter-service communication mechanisms—often involving APIs, message queues, or service meshes. This not only adds development overhead but also complicates debugging and failure handling.

Moreover, developer tools and IDEs are generally optimized for monolithic development and provide limited native support for managing multiple services, making tasks like refactoring, tracing, and dependency management more cumbersome. Testing microservices is also more difficult, as it requires simulating or integrating with multiple services across networks, increasing the need for sophisticated test environments and mocks.

Implementing business use cases that span multiple services can be particularly challenging. Without distributed transactions, ensuring consistency across services demands complex patterns like event sourcing or sagas. Additionally, coordinating changes across services often requires tight collaboration between multiple teams, increasing communication overhead and delaying delivery timelines.

From an operational standpoint, deployment and runtime management become significantly more complex. A production microservices system involves orchestrating, monitoring, and scaling dozens or even hundreds of individual services, each with its own lifecycle and deployment pipeline. This can result in increased memory consumption, as each service typically runs in its isolated container, multiplying resource usage compared to a monolithic setup.

In summary, while microservices bring flexibility and scalability, they come at the cost of higher development, testing, coordination, and operational complexity, which must be weighed carefully when deciding on an architectural approach.

Let's walk through how to add a second microservice to your Flask-based app—specifically, a database microservice (e.g., PostgreSQL or MySQL)—and connect it using Docker Compose. 7. To reinforce the above concepts, students or developers might engage in a basic cloud-native exercise such as:

Building a simple RESTful microservice.

Creating a Docker container for the service.
Writing a Kubernetes deployment file to orchestrate the service.
Setting up a CI/CD pipeline to automate build and deployment.
Enabling service monitoring using Prometheus and Grafana.
Each step of the exercise reflects a real-world practice aligned with cloud-native principles.

1.4 Exercises

1. Explain the core characteristics of monolithic architecture. Describe how components such as authentication, business logic, and data access are structured and interact within a monolithic application. Highlight the implications for scalability, maintainability, and fault tolerance.
2. Compare and contrast monolithic and microservices architectures. Identify at least four differences in design principles, deployment strategies, and operational concerns. Use real-world examples to support your comparison.
3. Discuss the advantages and disadvantages of monolithic architecture. In your answer, include scenarios where a monolithic approach is appropriate and when it becomes a limitation.
4. Define microservices architecture and explain its core principles. Describe how microservices differ from traditional monolithic systems in terms of structure, communication, and deployment.
5. Explain the role of service discovery and API gateways in microservices architecture. Why are these components important, and how do they contribute to the reliability and scalability of the system?
6. Discuss the benefits and drawbacks of microservices architecture. Include real-world use cases where microservices are particularly beneficial, and describe challenges that teams may face.
7. Design a basic monolithic architecture for an online voting system. Create a component diagram showing the major modules (e.g., User Interface, Ballot Management, Authentication, Tallying, Database). Explain how these modules interact and are deployed.
8. Given a monolithic application with a single codebase, suggest a strategy to refactor it for better modularity. Identify potential boundaries (e.g., separating UI logic from business logic) and outline steps to improve code maintainability within the monolithic structure.
9. Simulate a failure in a tightly coupled monolithic system. Describe a scenario (e.g., failure in the authentication module) and analyze how it would affect the rest of the system. Suggest ways to mitigate the impact while still remaining within a monolithic design.
10. Design a microservices-based architecture for an online voting system. Identify key services (e.g., User Service, Ballot Service, Voting Service, Tally Service), describe their responsibilities, and outline how they communicate.

CHAPTER 2

Cloud-Native Development

Cloud computing has transformed the way software applications are developed, deployed, and scaled. By providing on-demand access to computing resources such as servers, storage, intelligence, software, agility, and databases over the internet, cloud computing eliminates the need for organizations to invest in and manage their own physical infrastructure. This shift has paved the way for **cloud-native development** —an approach that embraces the full potential of the cloud to build applications that are inherently scalable, resilient, and agile.

Cloud-native development is a modern software engineering paradigm that leverages the full capabilities of cloud computing platforms to build scalable, resilient, and maintainable applications. Unlike traditional development, where applications are often built for static, on-premise environments, cloud-native development is about designing systems that thrive in dynamic, distributed cloud ecosystems, where services can be deployed, scaled, and managed independently. It involves leveraging key cloud technologies and practices, such as **containers**, **microservices**, **continuous integration and deployment** (CI/CD), and **orchestration** platforms like Kubernetes. These tools enable developers to create applications that are not only optimized for the cloud but also capable of rapid evolution and seamless deployment at scale.

A cloud-native application is designed from the ground up to run within a cloud computing environment. It leverages the inherent characteristics of the cloud. Cloud-native applications are not simply applications that happen to run in the cloud; they are built specifically to take advantage of cloud infrastructure and its benefits.

This section provides a hands-on conceptual overview of the essential components and practices involved in cloud-native development. While detailed implementations may vary depending on platform and technology stack, the principles and aim remain consistent: to create scalable $\Leftrightarrow$ portable $\Leftrightarrow$ maintainable software.

- **Containerization** At the core of cloud-native development is the use of containers, which encapsulate an application and all its dependencies into a lightweight, portable unit. Developers typically use tools like **Docker** to define and build container **images** using a **Dockerfile**. Containers ensure consistency across development, testing, and production environments and are the building blocks for scaling services across cloud infrastructure.

  Docker: Think of Docker as a tool for packaging software so that it can run consistently on any hardware or environment. To better understand this, let's use a cooking analogy. Imagine you are a chef preparing a meal:

The Dockerfile is like a recipe card. It lists all the ingredients (dependencies), the steps to cook the dish (installation commands), and the final presentation instructions (how to run the app).

The Docker image is the prepared meal, fully cooked and packaged based on the recipe. It contains everything needed to serve the dish, ready to eat, and consistent in flavor, no matter who prepares it.

The container is the serving plate or lunchbox. It holds the meal (image) and isolates it from the kitchen (host system), so it doesn't mix with other dishes or recipes. You can place this lunchbox in any kitchen/dining area, and the meal inside will always taste the same.

In the same way, Docker ensures that software behaves consistently across development, testing, and production environments. The Dockerfile defines the instructions, the image is the executable package, and the container is the running, isolated instance of the application.

- **Microservices Architecture** Cloud-native applications are often structured using the microservices architecture, where the system is decomposed into a set of small, independent services (as already discussed in Chapter 1.2). Each microservice is responsible for a specific business capability and communicates with other services through lightweight protocols such as *HTTP/REST or gRPC*. This design promotes loose coupling, modularity, and independent deployment, all of which enhance maintainability and scalability.
- **Orchestration with Kubernetes** To manage the deployment, scaling, and operation of containers, cloud-native systems rely on orchestration platforms such as Kubernetes. Kubernetes automates tasks such as load balancing, service discovery, health checks, self-healing, and resource scheduling. Developers define the desired application state using *YAML* configuration files, enabling declarative infrastructure management and reproducible environments.
- **Continuous Integration and Continuous Deployment (CI/CD)** A critical driver of agility in cloud-native development is the use of Continuous Integration and Continuous Deployment (CI/CD) pipelines. These pipelines foster efficient collaboration among development teams by automating the processes of building, testing, and deploying applications to cloud environments. Popular tools such as GitHub Actions, GitLab CI, Jenkins, and Jira (for tracking and coordination) are widely employed to implement and manage CI/CD workflows. By streamlining these workflows, automation not only accelerates development cycles but also minimizes human error and enables advanced deployment strategies like canary releases and blue-green deployments, ensuring safer and more reliable software delivery. CI/CD is a core practice that enables and embodies the goals of the DevOps philosophy.
- **Other Native Standards** Given the distributed nature of cloud-native systems, **observability** is essential for operational success. This includes logging, metrics, and tracing. Tools like Prometheus, Grafana, ELK Stack, and Jaeger provide insights into system behavior, performance bottlenecks, and failures. Observability

enables proactive system management and supports incident response and optimization.

Cloud-native applications are designed to **scale horizontally**, allowing multiple instances of services to handle increased load. Platforms like Kubernetes support features such as the Horizontal Pod Autoscaler (HPA) to automatically adjust the number of running instances based on CPU usage or custom metrics. **Resilience** is further enhanced through patterns like circuit breakers, retries, and distributed failover.

Lab 2.0.1 Hands-On Practice (Practical Outline)

1. **Environment Setup**:

   (A) Install Python and Visual Studio Code (if not already installed):

   - Download and install Python (version 3.7 or later) from: https://www.python.org/downloads/
   - Download and install Visual Studio Code (VS Code) from: https://code.visualstudio.com/

   (B) Install Docker:

   - Visit https://docs.docker.com/desktop/ and select the appropriate Docker Desktop version for your operating system. Check the Kubernetes box during installation.
   - After selecting your version, follow the installation instructions on the next page and complete all steps provided.

   (C) Create a basic microservice using your preferred programming language (e.g., Python Flask, Node.js, Go):

   - To use Flask, visit: https://flask.palletsprojects.com/en/stable/installation/
   - Carefully read and follow the installation instructions and run the required command prompt or in the VS Code terminal to install Flask.
   - Once installed, click on “Quickstart” at the bottom of the web page to run the “Hello World!” example, read, and explore additional options.

   (D) Ensure that "myProject" directory created earlier is where you have the Flask virtual environment (venv) and where you will put all the other microservices (like the Flask-app and Dockerfiles). An example directory is "C:/Users/john/

Documents/flask-app"

2. **Containerize the Application**: To containerize an application, such as flask-app, for example, the "hello.py" (found on the next page after clicking "Quickstart"), you need to create a Dockerfile in your "myProject" directory/folder. That is, you should now have hello.py and Dockerfile in the project directory. Here is a Dockerfile sample:

```
FROM python:3.10
WORKDIR /app
COPY . .
RUN pip install flask
CMD ["python", "hello.py"]
```

Listing 2.1. Dockerfile example

Next, open any command-line interface (CLI) and navigate to the directory where your "Dockerfile" and "hello.py" are located. In that directory, run the command "docker build -t flask-app ." — the dot at the end tells Docker to look in the current folder for the Dockerfile, while the ‘"t flask-app" flag tags the resulting image with the name "flask-app". If the build is successful, you should see a message like “Successfully tagged flask-app latest.” Once the image is built, you can run the container and expose it to your host system using the command "docker run -p 5000:5000 flask-app". This maps port 5000 of the container to port 5000 on your local machine, allowing you to access the application in your web browser. Finally, open your browser and go to "http://localhost:5000/", or press Ctrl + Click on the link in the terminal if supported, to launch and view your Flask app.

So far, you’ve just successfully:

- Built a Flask microservice
- Containerized it with Docker
- Exposed it to the web via localhost
- Most likely solved some real-world deployment issues

2.1 Exercises

1.
2. Containerize multiple microservices using Docker and orchestrate them with Docker Compose. Build and deploy at least three interdependent services (e.g., frontend, backend, and database) and demonstrate how they run in isolation and together.
3. Describe how DevOps practices align with microservices architecture. In your answer, discuss CI/CD, containerization, and infrastructure automation in the context of microservices.

CHAPTER 3

API Design & Service Communication

Modern distributed systems rely extensively on Application Programming Interfaces (APIs) to facilitate communication between services, clients, and data sources. As applications grow in complexity and experience increased traffic volumes, selecting an appropriate API design paradigm and gateway becomes essential for achieving high performance, reliability, maintainability, and scalability. This chapter examines key principles of RESTful API design, provides an introduction to GraphQL and gRPC, and explores the role of API gateways such as Kong, Traefik, and NGINX—with particular emphasis on gRPC's effectiveness for high-performance machine learning (ML) inference scenarios.

API Application programming interface is a set of defined rules and protocols that allow different software components to communicate with one another over a network.

Clients are Consumers of APIs, users sometimes, and typically frontend applications or services that send requests and process responses.

Resources refer to representations of data or services—such as users, transactions, or models—that are typically identified by unique URIs and accessed using standard HTTP methods (e.g., GET, POST, PUT, DELETE). Resources constitute the information made available by applications to their clients and can include images, videos, text, numerical data, or any other data type. The system that hosts and delivers these resources to clients is commonly referred to as the **server**.

3.1 RESTful API Design Principles

REST (Representational State Transfer) is a widely adopted architectural style inspired by the web used for designing networked applications. REST is not a protocol or a standard, but its simplicity, statelessness, and reliance on standard HTTP methods make it a go-to approach for web APIs.

Here is a description using the electronic voting system example to introduce the need for REST architecture, see Figure 3.1. As you would expect, in the electronic voting system, multiple components —such as the voter interface, vote tallying service, authentication module, and administrative dashboard— must all communicate seamlessly and reliably over a network. Voters interact with a web or mobile interface to cast their votes, which must then be securely transmitted to the backend, recorded, validated, and

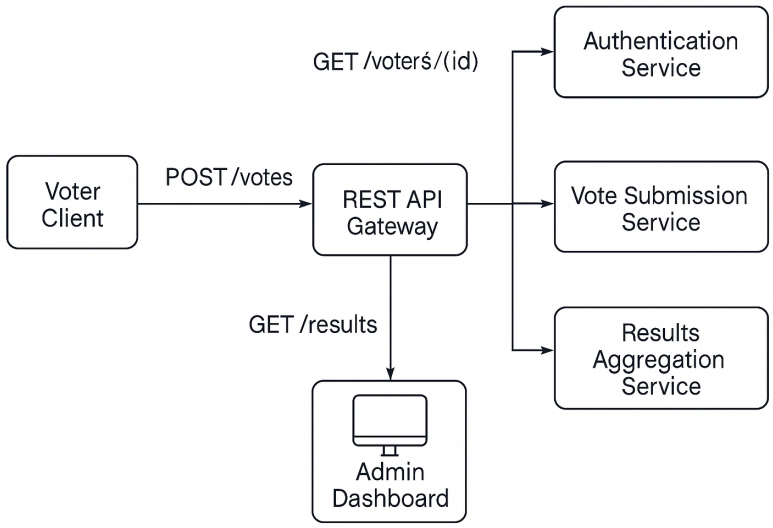


Figure 3.1. Electronic Voting System with REST interactions

aggregated in real time.

To enable this distributed architecture, a standardized, scalable, and stateless communication mechanism is essential. This is where the REST (Representational State Transfer) architecture becomes crucial. RESTful APIs provide a uniform way for each component in the voting system to communicate over HTTP using well-defined endpoints and methods. For example:

- POST /votes – to submit a new vote
- GET /results – to retrieve aggregated voting results
- GET /voters/id – to validate a voter's eligibility
- PUT /voters/id – to update voter information if needed

REST's stateless nature ensures that each request contains all the information needed for processing, through guided principles and constraints, which simplifies horizontal scaling and fault tolerance —both critical in a high-load system like national or statewide elections. Additionally, REST aligns well with existing web infrastructure, supports a wide variety of clients, and enables loose coupling between services (resources/servers), making the system easier to maintain and evolve.

3.1.1 Key Principles of RESTful API Design

REST is based on some constraints and principles that promote simplicity, scalability, and statelessness in the design. Some guiding principles or constraints of the RESTful architecture are:

Uniform Interface

A uniform interface is a foundational principle of RESTful web service design. It simplifies system architecture by enforcing a standardized way for clients and servers to interact, regardless of the underlying implementation. This abstraction enhances the visibility, scalability, and evolvability of distributed systems. The server transfers information in a consistent format known as a *representation*, which may differ from the internal format used by the server application. For example, data stored as plain text may be returned in an HTML or JSON format.

To achieve a uniform REST interface, four architectural constraints must be satisfied:

- Identification of Resources – Each resource must be uniquely identified using a Uniform Resource Identifier (URI). This allows clients to target specific data or services within the system.
- Manipulation of Resources Through Representations – Resources are represented in standardized formats (e.g., JSON, XML) that allow clients to retrieve, modify, or delete them. The server includes metadata in the response to guide these interactions.
- Self-Descriptive Messages – Each message must include enough contextual information (such as media type, status code, or available methods) for the client to understand how to process it, independent of prior knowledge.
- Hypermedia as the Engine of Application State (HATEOAS) – Clients should start with a single known URI and then discover other resources dynamically through hyperlinks embedded in the representation. This allows the application to evolve without breaking existing clients.

In practice, REST APIs typically use standard HTTP methods—such as GET, POST, PUT, and DELETE—to operate on resources, and rely on URIs to locate them. By adhering to these constraints, RESTful systems maintain a consistent interface that promotes loose coupling, strong cohesion, improves interoperability, and supports the dynamic evolution of services.

The Client-Server design pattern

This is a foundational principle of REST architecture that enforces the separation of concerns between the client (user interface) and the server (data storage and business logic). This separation allows both components to evolve independently, enhancing the portability of the client across multiple platforms and improving the scalability and maintainability of the server-side logic.

By clearly delineating responsibilities, the server-side code (e.g., APIs) and client-side code can be developed, maintained, and deployed independently—as long as the interface or contract between them remains stable. This interface defines how clients interact with the server via consistent endpoints, request methods, and data formats. In a RESTful system, multiple clients (e.g., web apps, mobile apps, IoT devices) can send requests to the same API endpoints, perform the same actions, and receive consistent responses, ensuring interoperability and reducing system coupling.

For example, the client-server separation principle is demonstrated in the context of

our electronic voting system. The client —a mobile or web-based user interface— allows voters to authenticate their credentials, view the ballot, make selections for candidates or propositions, and submit their vote. These user-facing functions are clearly separated from the responsibilities of the server (or API backend), which handles services such as vote authentication, ballot retrieval, vote submission, vote tallying, and result aggregation.

Statelessness

REST architecture's statelessness is a fundamental constraint that dictates that each request from a client to the server must be self-contained—carrying all the information necessary for the server to understand, process, and fulfill the request. This means that the server does not retain any context or session information between requests. Every interaction is independent, and clients can send requests in any order without relying on prior communications.

Caching

Ability to **cache** allows clients or intermediaries to store and reuse responses for future requests. This reduces the need for repeated communication with the server, thereby improving response times and lowering server load.

The cacheable constraint in REST architecture requires that each response explicitly or implicitly indicate whether it is cacheable or non-cacheable. If a response is designated as cacheable, the client is permitted to reuse the data for equivalent requests over a defined period of time.

For example, consider a website where every page contains the same header and footer images. Without caching, the server would need to resend these images on every page load. With caching enabled, the client stores the images after the first request and loads them locally for subsequent pages, eliminating redundant data transfer. RESTful APIs manage this behavior using HTTP caching headers (e.g., Cache-Control, ETag, Expires) to control the validity and reusability of cached responses.

Layered System Architecture

In a layered system architecture, a RESTful service is designed with multiple hierarchical layers —such as security, application logic, and business logic— that work together to process client requests. Each component within this architecture interacts only with its immediate layer, without knowledge of or direct access to components beyond it. This separation of concerns enhances modularity, security, maintainability, and scalability of the system.

Clients may communicate with intermediaries such as load balancers, proxies, or gateways, rather than directly with the final server. These intermediaries can perform functions like authentication, caching, or routing, and may forward requests to other backend servers. Importantly, these layers are transparent to the client, which continues to perceive the interaction as a direct communication with the server.

A practical analogy is the Model-View-Controller (MVC) pattern in software design. Like a layered REST architecture, MVC promotes clear separation between the user interface (View), the data processing logic (Model), and the control flow (Controller),

enabling clean abstraction and easier system evolution over time.

Code-on-Demand Constraint

The code-on-demand constraint in REST architecture allows servers to temporarily extend or customize client functionality by transmitting executable code —such as JavaScript scripts or Java applets— to be executed on the client side. This capability is optional but offers powerful flexibility by reducing the amount of functionality that must be pre-implemented in the client.

By delivering dynamic behavior through downloadable code, the server can simplify the client architecture and delegate complex interactions or validations. For example, when a user fills out a registration form on a website and the browser instantly validates fields like email or phone number, this is often accomplished using JavaScript code that was sent from the server to the client.

Although not required for all RESTful systems, this mechanism supports dynamic client enhancement and can lead to more efficient and responsive user experiences without overloading the client with unnecessary features upfront.

Additional Highlights

Resource-Oriented URLs Use nouns to represent resources or objects (e.g., '/users', '/products/123'). HTTP Methods Align actions with standard verbs: 'GET' (read), 'POST' (create), 'PUT' (update), 'DELETE' (remove). HTTP Status Codes Use standardized status codes to indicate response status (e.g., '200 OK', '404 Not Found', '500 Internal Server Error'). Versioning Version your APIs (e.g., '/api/v1/') to ensure backward compatibility.

3.2 Service Communication

Read Chapter 5 of the course textbook. As input, I will highlight a few salient points.

REST APIs are well-suited for CRUD (Create, Read, Update, Delete) operations and integrate seamlessly with standard HTTP infrastructure such as caching, proxies, and firewalls. This allows clients to construct standardized requests that servers can efficiently process and respond to.

However, REST can be limiting in scenarios that demand high performance, low-latency communication, fine-grained data retrieval, or real-time interactions. In such cases, modern alternatives like **GraphQL** and **gRPC** are often preferred. GraphQL enables clients to query exactly the data they need in a single request, minimizing over-fetching and under-fetching. Meanwhile, gRPC —a high-performance, contract-based RPC framework—supports efficient binary serialization, bi-directional streaming, and strong typing, making it ideal for internal microservices and real-time systems.

3.2.1 gRPC

Software applications are essentially webs of function calls—both **local** and **remote**. A *Remote Procedure Call (RPC)* allows a program to invoke a function that resides on a different machine or process, while abstracting the complexity of the underlying network.

To illustrate this, imagine asking a friend in another city to do your grocery shopping. You give them a list (parameters), they complete the task (execute), and send the groceries back to you (response). Let us call this a *remote shopping call*—perhaps uncommon in practice, but useful for understanding the concept.

In real-world systems, especially those based on **microservices architecture**, RPC plays a critical role. For instance, a *payment service* may call a *user service* to retrieve account information, or serve TensorFlow or PyTorch models via a gRPC-based inference service to downstream applications or microservices. Several technologies enable this type of remote communication between distributed systems:

- **gRPC** is a modern, high-performance RPC framework developed by Google. It uses HTTP/2 and **Protocol Buffers (Protobuf)** for efficient, compact, and type-safe communication.
- **SOAP** is an older, XML-based protocol that offers strict structure and extensive standards, often seen in enterprise environments.
- **JSON-RPC** is a lightweight, language-agnostic protocol that uses JSON for data transmission, making it ideal for simpler web-based systems.
- **Thrift**, developed by Facebook, is a scalable framework that supports multiple languages and transport protocols, designed for cross-platform service communication.

Key Features of gRPC

gRPC follows a **language-agnostic, contract-first design**, using `.proto` files to define service interfaces and data structures. This ensures interoperability and consistency across platforms. It relies on **Protobuf** for efficient binary serialization, enabling fast and compact data exchange.

gRPC also includes built-in support for **authentication, load balancing, and request deadlines**, making it suitable for production systems. Furthermore, it supports multiple communication patterns, HTTP/2 multiplexing and connection reuse, including *unary*, *server-streaming*, *client-streaming*, and *bidirectional streaming*, allowing for flexible and high-performance service interactions.

3.2.2 GraphQL

GraphQL (https://graphql.org) is a query language for APIs developed by Facebook. It allows clients to request only the data they need, reducing over-fetching and under-fetching common in REST APIs.

GraphQL Key Features:

- Single endpoint (‘/graphql‘) for all queries and mutations
- Schema definition for type-safe queries
- Hierarchical and nested data fetching
- Strong tooling ecosystem (Apollo, GraphiQL)

Use Case: GraphQL is ideal for frontend-heavy applications requiring flexible and optimized data retrieval. Here is a realistic example scenario of how GraphQL could be

used in our electronic voting system, illustrating the benefit of fine-grained, type-safe queries through schema definition. Assume a client application (voter's mobile or web app) wants to fetch the current user's voting eligibility, upcoming elections, and their voting status —all in one request. Proceed by first defining the GraphQL schema:

Listing 3.1, the GraphQL schema, defines the core structure for an electronic voting system. The Voter type captures essential details about a user, including their eligibility and voting status, while the Election type outlines key attributes of elections such as title, date, and constituency. The Query type enables clients to retrieve voter information by ID and fetch a list of upcoming elections specific to a constituency. This schema supports type-safe queries, allowing front-end applications to request exactly the data they need with confidence in the structure and data types.

```
    type Voter {
  id: ID!
  name: String!
  isEligible: Boolean!
  votingStatus: String!
  registeredConstituency: String!
}

type Election {
  id: ID!
  title: String!
  date: String!
  isActive: Boolean!
  constituency: String!
}

type Query {
  getVoter(id: ID!): Voter
  upcomingElections(constituency: String!): [Election!]!
}
```

Listing 3.1. GraphQL Schema Definition

Next is the GraphQL query (Listing 3.2), which demonstrates how a client can request specific data from an electronic voting system in a single, efficient call. It retrieves key details about a voter —such as their name, eligibility, voting status, and constituency —using the getVoter query. Simultaneously, it fetches a list of upcoming elections in "District 12," including each election's title, date, and status. This illustrates GraphQL's strength in delivering tailored, type-safe data responses while minimizing network overhead.

```
    query {
  getVoter(id: "voter123") {
    name
    isEligible
    votingStatus
    registeredConstituency
  }
```

```
  upcomingElections(constituency: "District 12") {
    title
    date
    isActive
  }
}
```

Listing 3.2. GraphQL Query (Client Request)

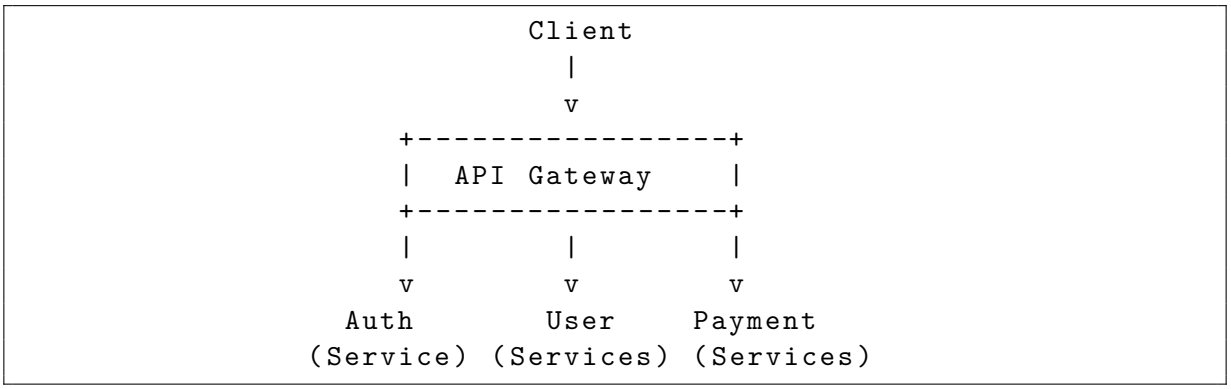
Finally, the GraphQL response Listing 3.3 returns precise, structured data based on the client's query. It confirms that Amina Yusuf is eligible to vote, has not yet voted, and is registered in District 12. Additionally, it provides a list of upcoming elections in that district, including a currently active Local Council Election and an inactive Mayoral Election scheduled for later. This response highlights GraphQL's ability to deliver relevant and targeted information in a single payload. This, in turn, reduces data redundancy while improving performance.

```
    {
  "data": {
    "getVoter": {
      "name": "Amina Yusuf",
      "isEligible": true,
      "votingStatus": "Not Voted",
      "registeredConstituency": "District 12"
    },
    "upcomingElections": [
      {
        "title": "Local Council Election",
        "date": "2025-08-10",
        "isActive": true
      },
      {
        "title": "Mayoral Election",
        "date": "2025-09-15",
        "isActive": false
      }
    ]
  }
}
```

Listing 3.3. Server Response

While GraphQL has become increasingly popular and widely supported across programming languages, it initially faced challenges due to limited tooling and language integration. Early adopters often struggled with the lack of native support, making implementation risky. Another key concern is the **high query processing overhead**, especially when handling deeply nested or complex queries. This can lead to increased server load and performance bottlenecks if not carefully managed through query depth limiting, cost analysis, or caching strategies.

3.3 API Gateways (Kong, Traefik, NGINX)



Listing 3.4. A Gateway Visual Example

See Listing 3.4. API gateways act as intermediaries between clients and services, handling request routing, authentication, rate limiting, and monitoring. Instead of calling Auth, User, and Payment services individually, the client sends requests to the API Gateway, which then manages all communication internally. Think of it as a smart receptionist at a large company: instead of visiting each department individually, you talk to the receptionist who knows where to route your request, like multiple backend services in a microservices architecture. Table 3.3 presents what API Gateway does.

Function	Description
Request Routing	Routes incoming client requests to the appropriate backend service.
Aggregation	Combines responses from multiple services into one.
Authentication & Security	Verifies user credentials and enforces access policies.
Rate limiting	Prevents abuse by limiting how often a client can make requests.
Caching	Stores common responses to reduce backend load and speed up response time.
Logging & Monitoring	Tracks requests and responses for analytics and debugging.
Protocol Translation	Converts between protocols (e.g., HTTP to WebSocket, REST to gRPC).

Several popular API gateways are widely used to manage and streamline communication in distributed systems. **Kong** is an open-source, cloud-native gateway known for its scalability and flexibility. **AWS API Gateway** is a fully managed service tightly integrated with the AWS ecosystem, making it ideal for serverless applications. **NGINX** can be configured as a lightweight API gateway and is often used for its performance and simplicity. **Istio**, using **Envoy** as a sidecar proxy, provides service mesh capabilities with gateway-like behavior for managing microservices traffic. **Traefik** is another powerful and modern API gateway and reverse proxy that's especially popular in cloud-native and containerized environments, etc.

3.3.0.1 More details on

1. Kong Kong
    - Open-source and enterprise-ready gateway built on NGINX
    - Plugin-based architecture (rate limiting, logging, JWT auth)
    - gRPC support with custom plugins
    - Kubernetes-native (via Kong Ingress Controller)
2. Traefik
    - Dynamic reverse proxy and load balancer designed for microservices
    - Native support for Docker, Kubernetes, and Consul
    - Supports HTTP, HTTPS, TCP, and gRPC routing
    - Built-in dashboard and automatic certificate management via Let's Encrypt
3. NGINX
    - High-performance HTTP and reverse proxy server
    - Widely used for load balancing and caching
    - Configurable for REST and gRPC routing
    - Less dynamic than Traefik, but extremely reliable and customizable

3.4 Conclusion

Choosing the right API design and communication mechanism is essential for modern applications. REST remains a reliable standard, while GraphQL offers flexibility in data querying. For performance-critical services such as ML inference, gRPC is the preferred protocol. Complementing these API technologies with robust gateways like Kong, Traefik, or NGINX ensures secure, observable, and scalable system architectures.

3.5 Exercises

1. Implement inter-service communication using RESTful APIs or gRPC. Create two simple services (e.g., User Profile and Notification) that interact using your chosen protocol. Demonstrate how one service calls the other.
2. Simulate a partial failure in a microservices system and analyze its effect. Disable or delay one service (e.g., the authentication service) and observe how it impacts user flow. Suggest mitigation strategies such as retries, circuit breakers, or service

replication.

3. Design and implement a RESTful API for a job listing and application platform similar to Indeed. The API should enable interaction between job seekers, employers, and job postings while following RESTful best practices.
4. Choose any public REST API (e.g., GitHub, OpenWeatherMap) and critically evaluate it against RESTful principles:

   (a) Are resources properly named?
   (b) Are HTTP methods used correctly?
   (c) Is statelessness maintained?
   (d) Is HATEOAS applied (if relevant)?

5. Design and implement a GraphQL API to support operations within a car manufacturing company. The API should enable flexible and efficient querying of production data, vehicle models, and parts inventory.
6. Create a high-performance gRPC service that wraps around a pre-trained machine learning model (e.g., Scikit-learn or TensorFlow model).

   (a) Define '.proto' file for service and message types.
   (b) Implement a gRPC server in Python or Go that loads and uses the model.
   (c) Build a client that sends requests and receives predictions.
   (d) Measure latency compared to a REST endpoint for the same model.

7. Design a use case (e.g., a voting dashboard) and implement the same backend service using both GraphQL and gRPC.

   (a) Evaluate performance (response size and time).
   (b) Discuss developer experience and ease of integration.
   (c) Present the trade-offs.

8. Use Docker or Kubernetes to deploy NGINX/Kong Gateway and expose multiple microservices: Voter service, Election service, and ect services. (or any other applications/product). Perform

   (a) Configure NGINX or Kong to route requests to the right services.
   (b) Apply rate limiting and authentication plugins.
   (c) Use Admin API or Declarative Config for setup.
   (d) Test routing and plugin effects using cURL/Postman.

9. Manually configure NGINX to act as an API gateway for three services (e.g., REST, GraphQL, and gRPC).

   (a) Reverse proxy requests to appropriate services.

   (b) Set up load balancing for one service with multiple instances.

   (c) Enable logging, custom headers, and gzip compression.

CHAPTER 4

Orchestration with Kubernetes

What is the dictionary meaning of orchestration?

Kubernetes (often abbreviated as K8s) has become the industry standard for orchestrating containers in cloud-native environments. It provides a robust, extensible, and declarative platform for automating the deployment, scaling, and operation of containerized applications. This chapter explores Kubernetes architecture, its core components, and how it enables efficient service management, discovery, and load balancing.

Kubernetes is more than just a container orchestrator. It can be thought of as the operating system for cloud-native applications, in the sense that it serves as the platform on which applications run, just as desktop applications run on macOS, Windows, or Linux.

Kubernetes is designed to alleviate the complexity of managing core infrastructure resources like computing, networking, and storage. By handling these responsibilities, it enables developers and operations teams to concentrate on container-focused workflows and empowers self-service application management. It supports the development of tailored automation pipelines and streamlined processes for deploying and maintaining multi-container applications.

Although Kubernetes can orchestrate a broad spectrum of workloads—including monolithic systems, microservices, stateless and stateful applications, batch processing tasks, and beyond—it is particularly well-suited for microservices architectures, which is where it sees the most use in modern systems.

Initially, the platform was primarily adopted for stateless services. However, as its ecosystem matured, the integration of persistent storage solutions expanded, allowing Kubernetes to handle stateful workloads more efficiently and reliably.

One of Kubernetes' defining characteristics is its modularity and extensibility. Users can adopt only the components they need, replace default behaviors with custom implementations, or expand their capabilities through integration with existing enterprise systems and third-party tools.

4.1 Kubernetes Architecture & Key Components

Deploying microservices with Kubernetes involves defining a *pod* to encapsulate the microservice instance, and a *service* to expose the microservice for communication. A *deployment* is then used to manage and apply changes to the running pods over time. Kubernetes is built on a master-worker architecture. The control plane manages the cluster, while worker nodes run the containerized applications, see Figure 4.1.

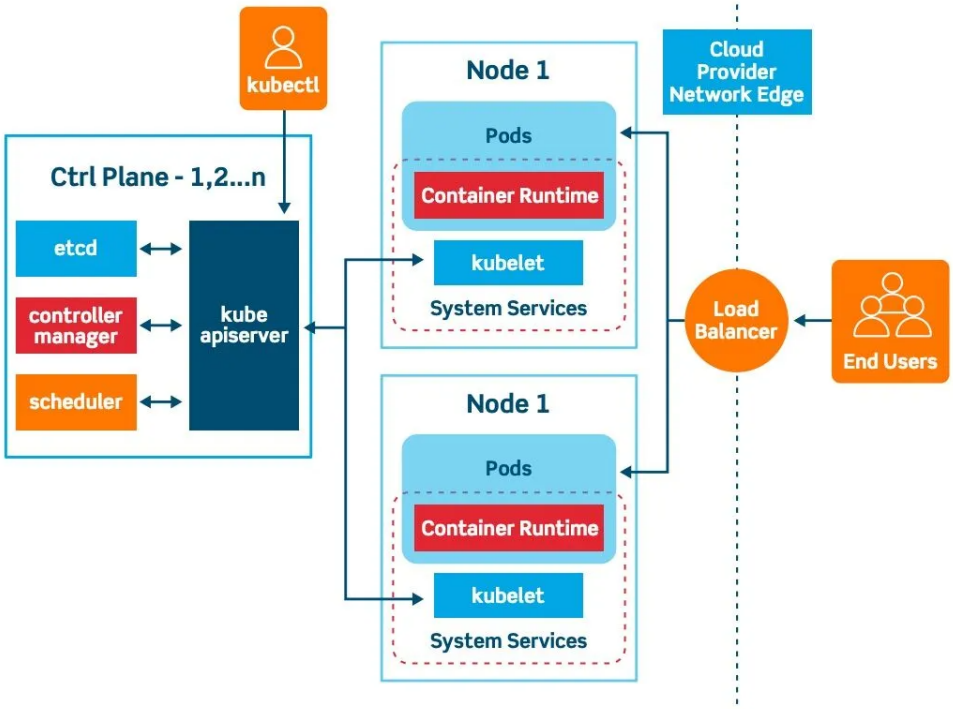


Figure 4.1. Kubernetes Architecture

4.1.1 Core Components

(A) Understanding the Kubernetes Control Plane

The left block of Figure 4.1 is the Kubernetes Control Plane, which is zoomed in on in Figure 4.2. It is the brain of a Kubernetes cluster responsible for global decisions, such as scheduling, maintaining cluster state, and managing workloads.

At the core of the control plane is the **'kube-apiserver'**, which acts as the front door to the cluster. All commands—from users (via **'kubectl'**) or automated processes—go through the kube API server. It validates and configures resources like pods, deployments, and services.

Surrounding the API server are several crucial controllers:

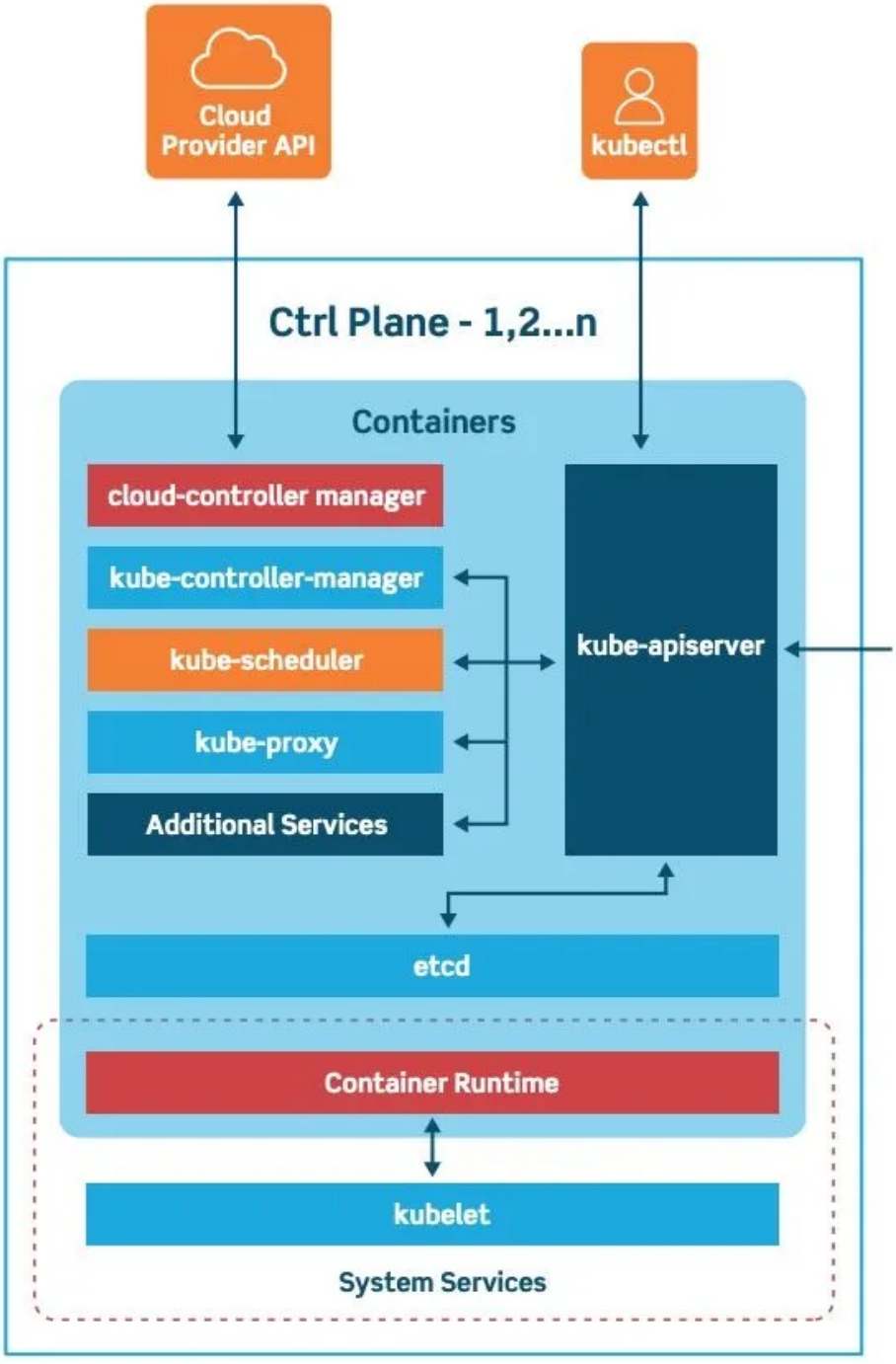


Figure 4.2. Controal Plane

- **kube-scheduler**: Determines which worker node a new pod should run on, based on resource availability and other rules.
- **kube-controller-manager**: Runs controllers that regulate and reconcile the desired cluster state, such as maintaining the right number of pod replicas.
- **cloud-controller-manager**: Interfaces with cloud service providers (e.g., AWS, Azure) to manage infrastructure resources like load balancers or persistent volumes.
- **kube-proxy**: Handles networking and ensures each pod can communicate with other pods and services.

Beneath these services is **'etcd'**, the distributed key-value store that holds the entire state of the cluster—it's the source of truth for Kubernetes.

On the worker nodes, you'll find the **'kubelet'**, an agent that ensures containers are running as specified, and the **container runtime** (like Docker or containerd), which runs the containers.

(B) **Understanding Kubernetes Nodes**

Figure 4.3 is a diagram representing a **Kubernetes Node**, i.e, a worker node in a Kubernetes cluster. Worker nodes are the machines (physical or virtual) where applications run, in the form of **pods**.

At the top, you see pods, which are the smallest deployable units in Kubernetes. Each pod encapsulates one or more containers that share storage, networking, and specifications. Think of a pod as a wrapper around your application container(s).

To support pod communication, Kubernetes runs **kube-proxy** on each node. This component manages networking rules and ensures that traffic is routed correctly between services and pods, both within and across nodes.

Beneath that, you have the **container runtime**, such as Docker, containerd, or CRI-O. This is the engine that pulls container images and runs them inside the pod environment.

At the base is the **kubelet**, the Kubernetes agent responsible for interacting with the control plane. It receives instructions (usually in the form of pod specifications) and ensures containers are running as expected on the node. If a container crashes, the kubelet helps restart it according to the declared desired state.

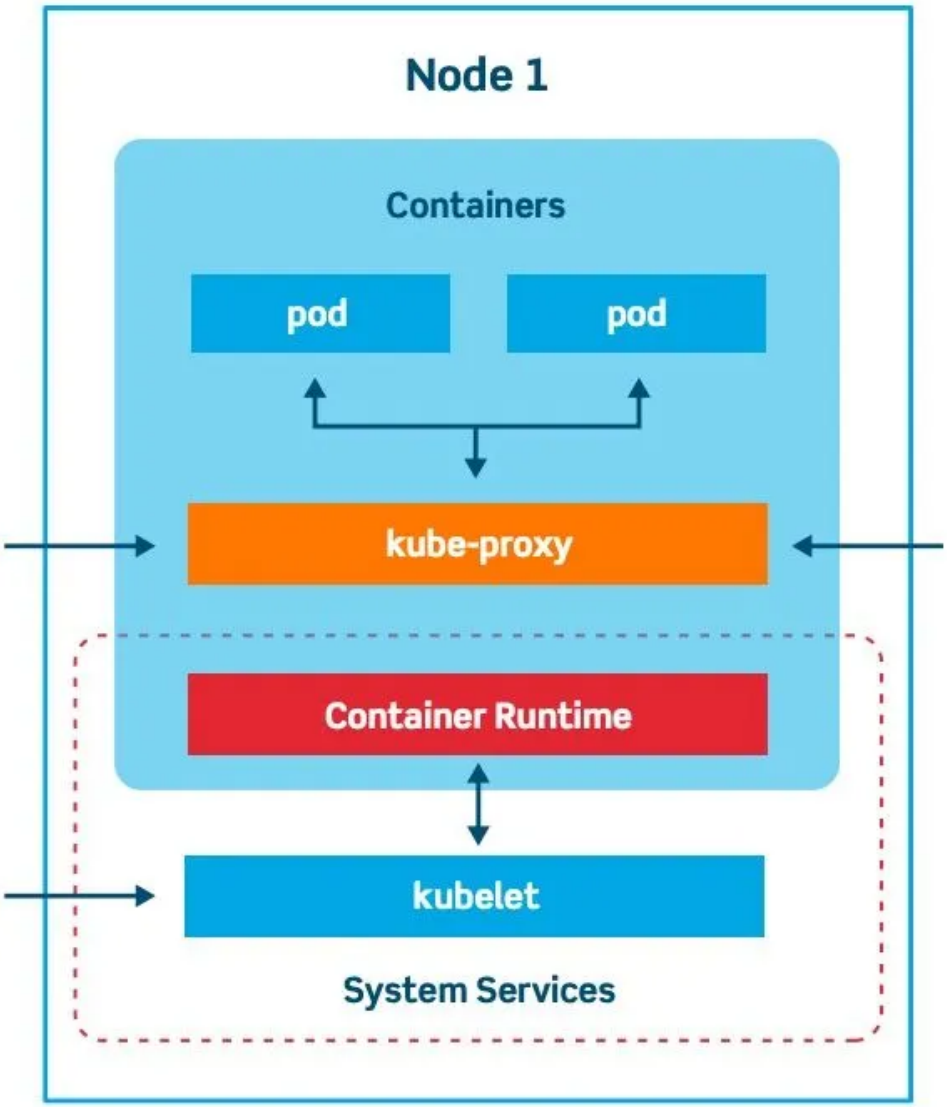


Figure 4.3. Kubernetes Node

4.2 Managing Deployments, Services & ConfigMaps

Constantly refer Figures 4.1, 4.2, or 4.3, to aid understanding of this section.

Kubernetes provides declarative objects to define how applications are deployed, exposed, and configured.

4.2.1 Deployments

A Deployment defines the desired *state of an application*, including the number of replicas, the container image to use, and the update strategy (such as rolling updates and rollbacks). Kubernetes continuously monitors this state and automatically replaces failed pods to maintain the desired configuration.

To achieve this, Kubernetes follows a declarative model, where you *describe what you want, not how to do it*, using **YAML files** as configuration manifests. YAML (YAML Ain't Markup Language) files are used to define **Pods**, **Deployments**, **Services**, **ConfigMaps**, **Volumes**, **Namespaces**, etc. For example, a Deployment YAML has four sections that describe: The number of replicas, the container image to run, labels/selectors, resource requests/limits, update strategies, etc., as follows

Table 4.1. Main Sections of a Kubernetes YAML Object

Section	Purpose
`apiVersion: apps/v1`	Specifies the Kubernetes API version to ensure compatibility and correct parsing.
`kind: Deployment`	Indicates the type of Kubernetes object being defined, such as `Pod`, `Service`, `Deployment`, or `ConfigMap`.

`metadata:`
`name: my-app`
`labels:`
`app: web`
	Provides identifying information about the object. Common fields include:
	`name` – unique name of the object,
	`namespace` – (optional) namespace in which the object resides,
	`labels` and `annotations` for categorization and metadata.

`spec:`
`replicas: 3`
`selector:`
`matchLabels:`
`app: my-app`
	Describes the desired state of the object. This section is specific to the type of object.
	For a `Deployment`: includes replicas, selector, pod template, etc.
	For a `Service`: includes type, ports, selector, etc.

Example Kubernetes Deployment YAML

```yaml
apiVersion: apps/v1                    # API version
kind: Deployment                       # Type of object
metadata:                              # Metadata like name, labels
  name: my-app
spec:                                  # Specification of the desired
   state
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:                            # Pod template
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: nginx:latest
```

Listing 4.1. Kubernetes Deployment YAML Example

4.2.2 SERVICES

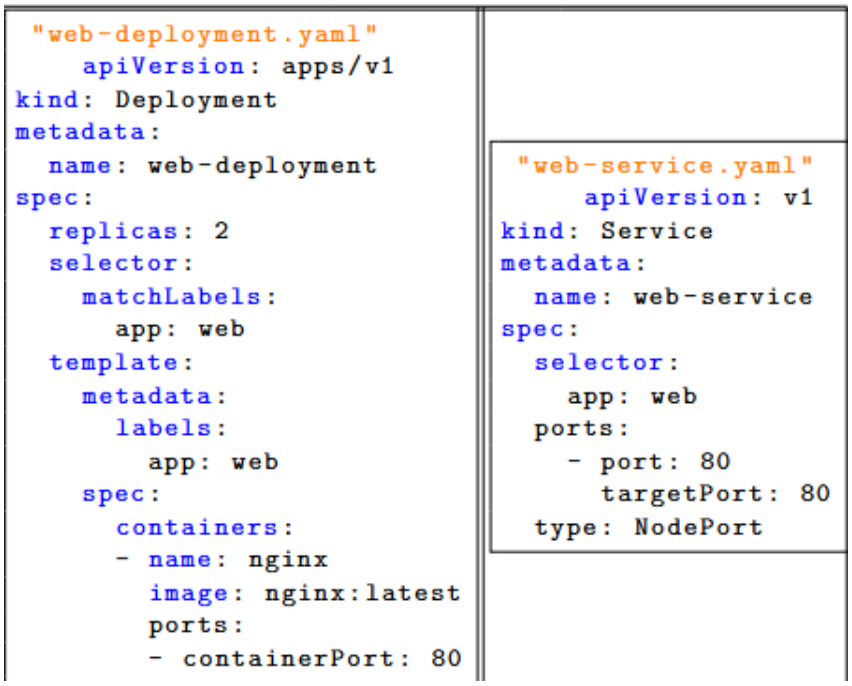
In Kubernetes, a Service provides a consistent and discoverable endpoint to route traffic to a group of dynamically created pods. Since pods are ephemeral—they can be terminated, rescheduled, or replaced at any time due to scaling events, failures, or updates—the Service acts as a stable abstraction layer, maintaining a fixed network identity within the cluster. This decouples the communication layer from the transient nature of individual pod instances.

Kubernetes offers several types of Services—such as **ClusterIP**, **NodePort**, and **LoadBalancer** —to manage access based on different networking needs. Rather than assigning static IPs, Services use label selectors to automatically match and route traffic to the appropriate pods. This approach simplifies deployment and scaling: any pod with matching labels is automatically included in the Service, making version updates or scaling seamless.

By default, a ClusterIP Service is only accessible within the cluster. For external exposure, types like LoadBalancer are used, particularly in cloud environments. However, assigning a dedicated load balancer to each Service can be both costly and complex.

To address this, Kubernetes includes Ingress, a higher-level abstraction that manages external HTTP/HTTPS access to Services using routing rules based on hostnames or paths. With Ingress, multiple Services can be exposed through a single external IP and load balancer, reducing both cost and configuration overhead.

Ingress controllers—such as NGINX, Ambassador, and those provided by cloud vendors like AWS, Google Cloud, and Azure —support advanced capabilities including request routing, rate limiting, timeouts, TLS termination, and authentication. This makes



```
"web-deployment.yaml"
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

```
"web-service.yaml"
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 80
  type: NodePort
```

Figure 4.4. Sample YAML files. We're using NodePort so you can access the service on localhost.

Ingress a powerful solution for managing external access and traffic control in Kubernetes environments.

Example:

Let's follow this example using a Deployment to run a simple web app and a Service to expose it.

When you install Docker Desktop, it includes Kubernetes, but it may not be enabled by default. To verify that **kubectl** is available, open your command-line interface (CLI) and run the following commands:

```
kubectl version --client
kubectl config get-contexts
```

If you encounter an error—likely because Kubernetes was not enabled during installation—do the following: (other errors/problems will be handled in class)

1. Open Docker Desktop.
2. Go to Settings > Kubernetes.
3. Check "Enable Kubernetes".
4. Click Apply & Restart.

Wait a few minutes for Kubernetes to initialize. Once it's ready, you can begin deploying your application using your (sample) YAML files (below, Figure 4.4).

web-deployment.yaml: This YAML file defines a Kubernetes Deployment named ‘web-deployment‘ that ensures two replicas of a pod are always running. Each pod runs a single container based on the ‘nginx:latest‘ image, which serves a simple web server. The ‘replicas: 2‘ field tells Kubernetes to maintain two identical pods at all times. The ‘selector‘ and ‘template.metadata.labels‘ ensure that the Deployment correctly identifies and manages the pods it creates using the label ‘app: web‘. Inside the pod, the container is named ‘nginx‘ and is configured to expose port 80, which is the default HTTP port. Kubernetes assigns dynamic IPs to each pod, and this Deployment ensures high availability and automatic replacement if any pod crashes or is terminated. Identify and relate this file to the file in Listing 4.1.

web-service.yaml This YAML file defines a Kubernetes Service named web-service that provides a stable network endpoint for accessing pods labeled with app: web. The selector field connects the Service to the corresponding pods created by the web-deployment, ensuring that traffic is routed only to those pods. The Service listens on port 80 and forwards incoming traffic to the same port (targetPort: 80) on the matching pods. The type: NodePort declaration exposes the Service outside the cluster by opening a randomly assigned high port (typically between 30000 and 32767) on each node’s IP address. This allows users to access the application through http://localhost:<nodePort>, enabling external traffic to reach the NGINX containers running inside the pods.

Let’s plow ahead by applying the YAML files

```
kubectl apply -f web-deployment.yaml
kubectl apply -f web-service.yaml
```

And verify they are running using these commands

```
kubectl get pods
kubectl get services
```

Access the NGINX web app in your browser by navigating to http://localhost:31234. To determine the correct port, run the command "kubectl get svc web-service" and look for a port mapping in the form of 80:31234/TCP. The second number (e.g., 31234) is the NodePort assigned to your localhost. Once you enter the address in your browser, you should see the NGINX welcome page. When you’re done testing, you can optionally clean up the created resources by running "kubectl delete -f web-deployment.yaml" and "kubectl delete -f web-service.yaml" to remove the deployment and service from your cluster.

4.2.3 ConfigMaps

A **ConfigMap** allows you to **decouple configuration data from application code**, enabling greater flexibility and reusability. ConfigMaps can be mounted into pods either as files or as environment variables. This approach promotes clean separation of concerns by keeping configuration external to the container image. A comprehensive overview of ConfigMaps is available in the official Kubernetes documentation at https://kubernetes.io/docs/concepts/configuration/configmap/. The discussion is further extended in Listing 4.2, where we examine a sample ConfigMap declaration.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

Listing 4.2. Sample YAML ConfigMap File

This YAML file (Listing 4.2) defines a Kubernetes ConfigMap named game-demo, which is used to store non-sensitive configuration data separately from application code. In this example, the data section contains both simple key-value pairs (like `player_initial_lives: "3"`) and multiline, file-like values (using the "|" syntax). The ConfigMap includes settings such as "enemy.types" and UI-related properties, simulating configuration files like game.properties and user-interface.properties. This allows developers to externalize environment-specific configuration from container images, making apps more portable and easier to manage. Rather than hard-coding these values inside the application or container, Kubernetes ConfigMaps let you inject them as environment variables, command-line arguments, or mounted configuration files. This promotes flexibility and enables you to update configurations without rebuilding or redeploying the container. For emphasis, ConfigMaps help decouple app configuration from the app itself, supporting better modularity and environment-specific customization.

4.3 Service Discovery & Load Balancing in Kubernetes

Kubernetes provides powerful primitives for building distributed applications by managing dynamic service endpoints and balancing traffic efficiently. This chapter explains the concepts of **service discovery** and **load balancing** in Kubernetes using an online voting system as a practical example.

Figure 4.5 illustrates how Kubernetes handles service discovery and load balancing in an online voting system. On the left, a user sends a request to a Kubernetes Service named vote, which acts as a stable entry point. The service is responsible for locating healthy pods based on label selectors and routing the request to one of several identical instances of the Voting App running within the cluster. Kubernetes automatically distributes incoming traffic across these instances using internal load balancing mechanisms, ensuring high availability and scalability. This setup decouples the user-facing endpoint from the

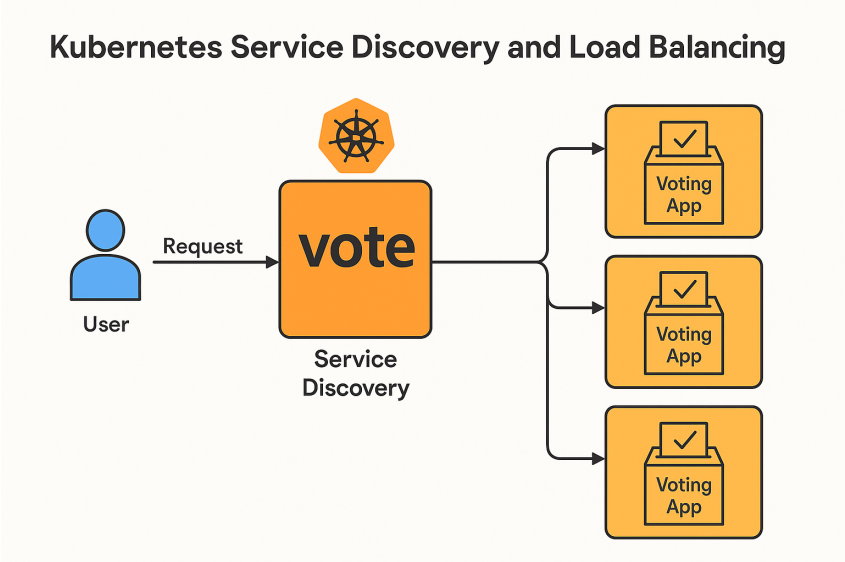


Figure 4.5

individual application pods, allowing for dynamic scaling and fault tolerance without impacting user access.

4.3.1 Service Discovery

Service discovery is the mechanism by which Kubernetes facilitates communication between application components without the need to hardcode IP addresses. Each Service in Kubernetes is assigned both a DNS name and a cluster-internal IP address, enabling applications to discover and communicate with each other in a dynamic environment without manual IP management.

In Kubernetes, pods are ephemeral and are assigned random internal IPs that can change over time due to scaling, updates, or failures. To abstract this volatility, Kubernetes employs Services in conjunction with label selectors to create a stable and consistent communication interface between components. This decoupling of service endpoints from pod lifecycles simplifies application design and deployment. The configuration examples in Listings 4.3 and 4.4 demonstrate how this service discovery mechanism is implemented in practice.

How It Works

- Each pod in a Deployment gets a dynamically assigned internal IP.
- A Service is defined to group a set of pods using a common label.
- Kubernetes uses DNS to make the Service name discoverable within the cluster.

```
   apiVersion: apps/v1
kind: Deployment
metadata:
  name: vote-app-deployment
spec:
```

```
  replicas: 3
  selector:
    matchLabels:
      app: vote-app
  template:
    metadata:
      labels:
        app: vote-app
    spec:
      containers:
      - name: vote-app
        image: myregistry/vote-app:latest  # Replace with your
           actual image
        ports:
        - containerPort: 80
```

Listing 4.3. Service Discovery YAML file

```
 apiVersion: v1
kind: Service
metadata:
  name: vote-service
spec:
  selector:
    app: vote-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 31234  # Optional: set or allow K8s to auto-
         assign
  type: NodePort
```

Listing 4.4. Load Balancing YAML file

In the current YAML example, the Service vote-service identifies and routes traffic to all pods matching the selector label "app: vote-app", irrespective of their individual IP addresses. This abstraction enables other components within the cluster to interact with the voting application through a stable DNS name, such as `vote-service.default.svc.cluster.local`, thereby decoupling service access from the underlying pod infrastructure.

4.3.2 Load Balancing

Load balancing in Kubernetes ensures that incoming traffic is evenly distributed across multiple healthy pods, thereby enhancing application availability, fault tolerance, and performance even as pods are dynamically added, removed, or rescheduled. Kubernetes Services include built-in load balancing mechanisms. When a client sends a request to a Service —such as ‘vote-service‘ in our example— Kubernetes automatically routes the request to one of the available backend pods. This process is typically managed by ‘kube-proxy‘ using a round-robin algorithm.

Kubernetes supports both internal and external load balancing strategies: **Internal Load Balancing** Traffic is distributed among pod endpoints within the cluster using ‘kube-proxy‘. While **External Load Balancing** Services of type ‘LoadBalancer‘ integrate with cloud provider-specific load balancers (e.g., AWS ELB, Azure Load Balancer) to route external traffic to the cluster.

In the current configuration, the Service is exposed externally via the NodePort type. To apply the configuration and access the application from outside the cluster, run the following commands in sequence:

```
    kubectl apply -f vote-app-deployment.yaml
kubectl apply -f vote-app-service.yaml
kubectl get svc vote-service
```

You should expect output similar to the following:

```
NAME           TYPE       CLUSTER-IP   EXTERNAL-IP PORT(S)        AGE
vote-service   NodePort   10.96.0.1    <none>      80:31234/TCP   10m
```

This indicates that the service is accessible via ‘http://localhost:31234‘, where Kubernetes will distribute incoming requests across the defined pod replicas.

Then visit the URL in your browser to interact with the app. A sample request to interact with the app could be

```
curl -X POST http://localhost:31234/vote \
     -H "Content-Type: application/json" \
     -d '{"candidate": "OptionA"}'
```

Summary

Kubernetes provides a powerful framework for container orchestration, enabling dynamic scaling, self-healing, and infrastructure abstraction. Understanding its architecture and core objects such as Deployments, Services, and ConfigMaps is essential for deploying reliable, cloud-native applications. Through integrated service discovery and load balancing, Kubernetes ensures that services are discoverable, accessible, and resilient in production environments.

4.4 Exercises

1. Compare synchronous and asynchronous communication between microservices. Describe scenarios where each type is preferable and analyze their impact on performance and fault tolerance.
2. Discuss the security considerations associated with the use of ConfigMaps in Kubernetes. What elements of a ConfigMap have implications for application or cluster security, and how can these be effectively managed?
3. In Figure 4.5 which of the element(s) are the pod(s) and the service?
4. How many pods are running, based on Listing 4.3?

CHAPTER 5

CI/CD for Microservices

5.1 Introduction to CI/CD Pipelines

Continuous Integration (CI) and **Continuous Deployment/Delivery (CD)** are foundational practices in modern software engineering that enable teams to deliver code changes more frequently and reliably.

With CI, the goal is to keep developers in sync with each other through frequent verification that newly checked-in code properly integrates with existing code. That is, the new code compiles, passes tests, integrates with the old codebase, and validates without breaking existing functionality.

In the same vein, continuous delivery addresses the principle that each check-in should be a release candidate (i.e., deployable). On the other hand, continuous deployment refers to the approach where every check-in must be validated using automated mechanisms (e.g., tests) and automatically deployed to production if validation is successful.

In microservices architectures, where systems are composed of many independently deployable components, CI/CD practices become critical to maintaining consistency, automation, and speed.

Key Concepts

- **CI (Continuous Integration):** Automatically integrates code changes into a shared repository several times a day, ensuring new code does not break the existing application.
- **CD (Continuous Delivery/Deployment):** Automates the release process so that new changes can be deployed to production (or staging) environments automatically or with minimal manual intervention.

Benefits for Microservices

- Faster iteration and release cycles
- Isolated service deployment
- Improved rollback strategies
- Reduced manual errors and bottlenecks

Application Examples of CI/CD Tools

GitHub Actions

GitHub Actions is a cloud-native CI/CD tool integrated with GitHub repositories. It uses YAML files to define workflows triggered by events such as commits or pull requests.

Use Case Example:

- Automatically lint, test, build Docker images, and push them to a container registry when code is pushed to the main branch. {See Listing 5.1.}
- Trigger Helm chart deployment on successful tests. {borrow a leaf from Listing 5.4.}

Solution

```
name: CI/CD Pipeline
on:
push:
branches:
\- main
jobs:
build-and-deploy:
runs-on: ubuntu-latest
steps:
\- uses: actions/checkout\@v2
\- name: Set up Docker
uses: docker/setup-buildx-action\@v1
\- name: Build and Push Docker Image
run: |
docker build -t myapp\:latest .
docker push myregistry/myapp\:latest
```

Listing 5.1. GitHub Actions Workflow

Jenkins

Similar to GitHub Actions in support of CI/CD automation is Jenkins. Jenkins is a powerful, open-source automation server ideal for teams that require full control over their CI/CD environment. It is especially well-suited for organizations managing complex enterprise pipelines, monorepos, or on-premise infrastructure. With its extensive (1000+) plugin ecosystem, Jenkins enables custom integrations and tailored workflows that go beyond the capabilities of most cloud-native CI/CD tools. Whether you're orchestrating multi-stage builds or deploying across hybrid environments, Jenkins offers the flexibility and control needed for sophisticated DevOps pipelines.

Use Case Example:

- Set up multibranch pipelines for different microservices.
- Use Kubernetes and Helm to deploy updates upon successful builds.

Solution

```
pipeline {
agent any
stages {
stage('Build') {
steps {
sh 'docker build -t myapp:\${BUILD\_NUMBER} .'
    }
 }
stage('Push') {
steps {
sh 'docker push myregistry/myapp:\${BUILD\_NUMBER}'
   }
 }
stage('Deploy') {
steps {
sh 'helm upgrade --install myapp ./charts/myapp'
       }
     }
   }
}
```

Listing 5.2. Jenkins Pipeline

GitLab CI/CD

GitLab provides built-in CI/CD functionality tightly integrated with its Git repository and DevOps features.

Use Case Example:

- Trigger builds and deployments automatically on merge requests.
- Use GitLab Runners to deploy to Kubernetes clusters.

Solution

```
stages:

* build
* deploy

build:
script:
\- docker build -t registry.gitlab.com/mygroup/myapp:\$CI\_COMMIT
   \_SHORT\_SHA .
\- docker push registry.gitlab.com/mygroup/myapp:\$CI\_COMMIT\
   _SHORT\_SHA

deploy:
script:
```

```
\- helm upgrade --install myapp ./charts/myapp
only:
\- main
```

Listing 5.3. GitLab CI/CD Script

Automating Deployments with Helm Charts

Helm is the package manager for Kubernetes. It simplifies the deployment and management of applications through reusable YAML templates called charts.

Key Features

- Templating engine to manage complex Kubernetes manifests
- Version-controlled application releases
- Easy rollbacks and upgrades

Example

Use Helm to deploy a microservice with configurable parameters:

```
helm install myservice ./charts/myservice&#x20;
\--set image.tag=1.0.0&#x20;
\--set replicaCount=3
```

Listing 5.4. Helm Deployment

Helm charts can be integrated directly into CI/CD pipelines to automate production-grade Kubernetes deployments.

Lab: Implementing a CI/CD Pipeline for Microservices

Scenario: Build and deploy a simple Node.js microservice using GitHub Actions and Helm on a Kubernetes cluster.

Lab Steps

1. **Initialize the Project:**
   - Create a simple Node.js app with a Dockerfile.
   - Push the code to a GitHub repository.
2. **Create GitHub Actions Workflow:**
   - Build and push Docker image to Docker Hub or GitHub Container Registry.

- Deploy the image using Helm.

3. **Create a Helm Chart:**

   - Scaffold a basic Helm chart using `helm create myservice`.
   - Replace default templates with custom configurations.

4. **Connect to Kubernetes Cluster:**

   - Use GitHub secrets to store kubeconfig or access token.

5. **Test the Pipeline:**

   - Make a code change, push, and verify automated deployment.

Chapter Exercises

1. Explain the difference between Continuous Integration and Continuous Deployment.
2. Describe how GitHub Actions and GitLab CI/CD differ in setup and usage.
3. Create a simple `.gitlab-ci.yml` file that builds and deploys a Python Flask app using Docker.
4. List three benefits of using Helm in a microservices deployment strategy.
5. Implement a Jenkins pipeline that tests, builds, and deploys a Go-based microservice to Kubernetes.
6. Set up a basic CI/CD pipeline for a microservice. Choose any CI/CD tool (e.g., GitHub Actions, Jenkins, GitLab CI) and show how changes to a microservice trigger automated testing and deployment.

CHAPTER 6

Logging, Monitoring & Tracing

Appendices

APPENDIX A

Title

There once was a very smart but sadly blind duck. When it was still a small duckling it was renowned for its good vision. But sadly as the duck grew older it caught a sickness which caused its eyesight to worsen. It became so bad, that the duck couldn't read the notes it once took containing much of inline math just like its favoured equation: $d = u_c \cdot k$. Only displayed equations remained legible so it could still read

$$d = ra^k e.$$

That annoyed the smart duck, as it wasn't able to do its research any longer. It called for its underduckling and said: "Go, find me the best eye ducktor there is. He shall heal me from my disease!"

Bibliography

[Aut] Kubernetes Authors, *Configmaps*, https://kubernetes.io/docs/concepts/configuration/configmap/.

[AWS] 2025 Amazon Web Services, Inc, *What is a restful api?*, https://aws.amazon.com/what-is/restful-api/.

[Kin] Yasar Kinza, *What is a cloud-native application?*, https://www.techtarget.com/searchcloudcomputing/definition/cloud-native-application#:~:text=A%20cloud%2Dnative%20application%20is,private%2C%20public%20and%20hybrid%20clouds.

[Pla] Platform9, *Kubernetes concepts and architecture*, https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts/.

[Pud25a] Mattia Puddu, *Title*, Publisher, 2025.

[Pud25b] ———, *Title*, Journal (2025), 1–100.

[Sam21] Newman Sam, *Building microservices*, O'reilly, 2021.