CAB432 Cloud Computing

Lecture Notes

Queensland University of Technology

Unit Coordinator: Dr. Felipe Gonzalez

Semester 2, 2025

Queensland University of Technology
Faculty of Engineering
GPO Box 2434
Gardens Point Campus
Brisbane, Australia, 4001.

Contents

1 Introduction to Cloud Computing

1.1 What Is Cloud Computing?

Cloud computing refers to the on-demand delivery of IT resources and services—such as compute power, storage, and databases—over the internet with pay-as-you-go pricing. It is often described as **"Elastic Utility Computing at Scale"**, which reflects its three essential characteristics:

Utility Computing

Cloud resources are provisioned like utilities, such as electricity or water. Users consume resources as needed and are billed only for their usage.

- On-demand access to computing services (compute, storage, networking).
- Cost follows a metered billing model—there are no upfront hardware investments.
- Services are provisioned automatically via self-service portals or APIs.

Elastic Provisioning

Elasticity allows systems to automatically adjust their resource usage based on demand.

- Resources can scale up (add more instances) or down (remove excess capacity) dynamically.
- This is especially useful for applications with variable traffic, such as e-commerce or streaming platforms.

Scale

Cloud computing enables global access and mass scalability.

- Services are delivered from data centres distributed worldwide.
- Applications can serve thousands to millions of users without architectural changes.

Cloud Service Models and Deployment Models

Service Model	Description
Infrastructure as a Service (**IaaS**)	Provides virtualised hardware, giving users control over operating systems, storage, and deployed applications (e.g. AWS EC2).
Platform as a Service (**PaaS**)	Delivers a managed platform including OS, runtime, and libraries for application development (e.g. Heroku, AWS Elastic Beanstalk).
Software as a Service (**SaaS**)	Delivers software applications over the internet with no infrastructure management required (e.g. Google Docs, Dropbox).

Deployment Model	Description
Public Cloud	Infrastructure is shared across multiple tenants and hosted by third-party providers (e.g. AWS, Microsoft Azure).
Private Cloud	Infrastructure is provisioned for exclusive use by a single organisation, either on-premises or via a private host.
Hybrid Cloud	Combines public and private clouds, enabling data and applications to be shared across them for flexibility and control.
Community Cloud	Shared infrastructure used by a specific community of users with common objectives or compliance requirements.

1.2 Brief History of Cloud Computing

Although the term "cloud computing" is relatively modern, the concept of shared computing resources dates back to the era of mainframes in the 1960s. Users accessed centralised computing systems via dumb terminals—an early form of utility computing.

Modern cloud computing emerged in the early 2000s:

- In 2002, Amazon launched AWS with two services:
  - **EC2** (Elastic Compute Cloud): Virtual machines on demand.
  - **S3** (Simple Storage Service): Scalable object storage.
- By 2024, AWS had expanded to over 325 services, ranging from AI/ML and blockchain to robotics and IoT.

Major competitors such as Microsoft Azure, Google Cloud Platform (GCP), and Oracle Cloud soon followed, offering a rich ecosystem of interoperable and competing services.

1.3 Advantages of Cloud Computing

Cloud computing offers numerous benefits that support agility, scalability, and cost optimisation in modern IT operations.

Economic Efficiency

- Cloud providers can amortise infrastructure costs across clients.
- Pay-per-use reduces idle capacity waste.
- No capital expenditure on physical hardware or maintenance.

Outsourcing Risk

Businesses often lack the resources or expertise to build resilient, scalable systems from scratch. Cloud providers reduce risk by:

- Providing fault-tolerant and redundant systems.
- Offering automated scaling to absorb traffic spikes.
- Delivering security hardening, logging, and compliance tools as managed services.

Pre-Built Solutions

Most cloud platforms offer plug-and-play services to solve standard needs:

- Load balancers, CI/CD pipelines, databases, API gateways.
- Pre-configured machine images, runtime environments, and container orchestration.
- Cloud-native design patterns support faster innovation and reduced downtime.

Access to Specialised Resources

Cloud platforms provide access to otherwise expensive or rare tools:

- High-performance GPUs, FPGAs, and TPUs for AI/ML workloads.
- Global CDN networks and edge caching for fast content delivery.
- Access to professional support, managed updates, and SLA-backed uptime.

1.4 Disadvantages and Risks of Cloud Computing

Despite its advantages, cloud computing introduces trade-offs and limitations.

Vendor Lock-In

- Proprietary APIs and configurations make migration difficult.
- Some services are only available through specific providers.
- Data egress fees and integration effort discourage portability.

Not Universally Applicable

- Poor internet connectivity or remote environments limit usability.
- On-premise workstations, industrial control systems, and IoT devices may be unsuitable for full cloud integration.

Cost Concerns

While initially cost-effective, cloud can become expensive:

- **Serverless** models (e.g. Lambda, Firestore) may scale in cost unpredictably.
- For steady workloads, a self-hosted or reserved instance model may be more economical.
- Cost monitoring and budgeting tools are essential to prevent bill shock.

Security and Compliance

- Data sovereignty laws (e.g. GDPR, Australian Privacy Act) may restrict cloud usage.
- Industries like defence, finance, and healthcare often require higher security postures than standard public cloud offerings.
- The shared responsibility model means customers must still manage application-level security.

1.5 Alternatives to Public Cloud

Not all organisations adopt full public cloud. Alternatives include:

On-Premises / Self-Hosting

- Offers full control and physical data ownership.
- Preferred when privacy or data residency is a concern.

Hybrid Cloud

- Combines flexibility of public cloud with the control of private infrastructure.
- Often used in regulated industries or during cloud migration phases.

Private Cloud

- Leverages cloud-native technologies (e.g. Kubernetes, OpenStack) on owned infrastructure.
- Useful for internal services or sensitive workloads.

Vendor-Independent Tools

- Promotes portability and standardisation.
- Common tools include:
  - **Appwrite**, **Supabase** – Backend-as-a-Service (BaaS)
  - **MySQL**, **PostgreSQL**, **Redis** – Open-source databases
  - **Docker**, **Kubernetes** – Containerisation and orchestration

1.6 Case Study: Appwrite

Appwrite is an open-source backend server designed to abstract and simplify common backend tasks. It acts as a lightweight alternative to proprietary solutions like Firebase and AWS Amplify.

Features

- Static hosting for web applications.
- User authentication (OAuth2, federated login, MFA).
- Managed document databases and object storage.
- Serverless functions with support for multiple runtimes.
- Event-driven pub/sub messaging.

Deployment

- Containerised and deployed using Docker.
- Can run on local, private, or public cloud infrastructure.
- Supports scaling via Docker Swarm.

Use Case

- Ideal for student projects, startups, or MVPs needing rapid deployment.
- Reduces backend complexity and developer overhead.
- Offers flexibility and data ownership not possible with managed SaaS platforms.

2 Containers

2.1 What Are Software Containers?

Containerisation is a form of operating system virtualisation that allows you to run applications in isolated environments called containers. Unlike virtual machines, which virtualise hardware, containers isolate the software stack at the process level.

Aspect	Virtual Machines vs Containers
OS Architecture	• VMs include a full OS, kernel, and virtualised hardware per instance. • Containers share the host OS kernel but isolate the user space.
Isolation	• VMs provide full hardware-level isolation. • Containers isolate processes, filesystems, and networks at the OS level.
Performance	• VMs are typically heavier and slower to boot. • Containers are lightweight and start almost instantly.
Portability	• VMs are less portable due to OS dependencies. • Containers can be deployed easily across environments.
Examples	• VMs: AWS EC2, VirtualBox • Containers: Docker, Podman

EC2 and the AWS Nitro System

Amazon EC2 uses the **AWS Nitro System**, a proprietary lightweight hypervisor. While not based on KVM, it offers similar isolation by allocating memory and CPU directly via hardware acceleration. Nitro delivers near bare-metal performance and is used for most modern EC2 instance types.

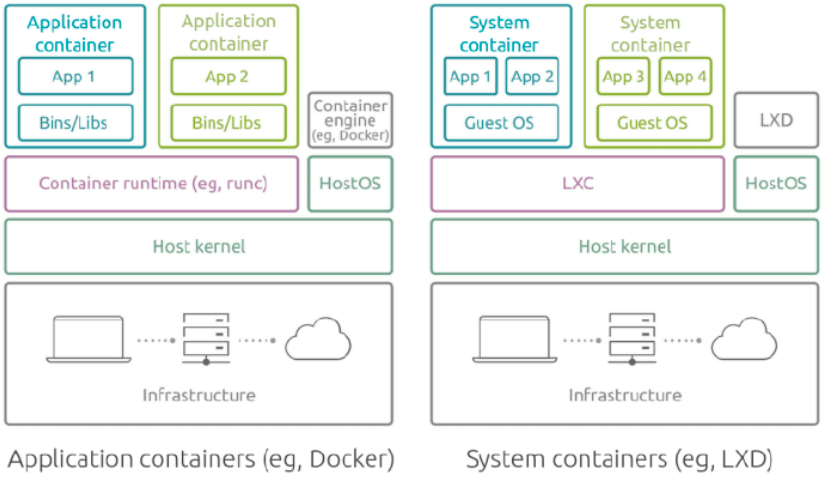

Figure 1: Application vs System Containers

2.2 Types of Containers

Container Type	Description
System Containers	• Contain a full OS environment (e.g. LXC). • Support multiple processes. • Suited for system-level isolation.
Application Containers	• Designed to run a single process per container. • Ideal for scalable apps and microservices. • Docker is the most common tool.

2.3 Docker

Docker allows developers to package applications and their dependencies into a standardised unit called a container.

- Docker containers run in isolated environments on any system that supports the Docker Engine.
- They are lightweight and consistent across development, testing, and production.
- Docker includes tools to build, manage, and share containers.

Key Terminology

- **Image**: A snapshot of an application environment. Immutable and versioned.
- **Container**: A running instance of an image.
- **Volume**: A persistent storage mechanism mounted to a container.
- **Dockerfile**: A script that defines how to build a Docker image.

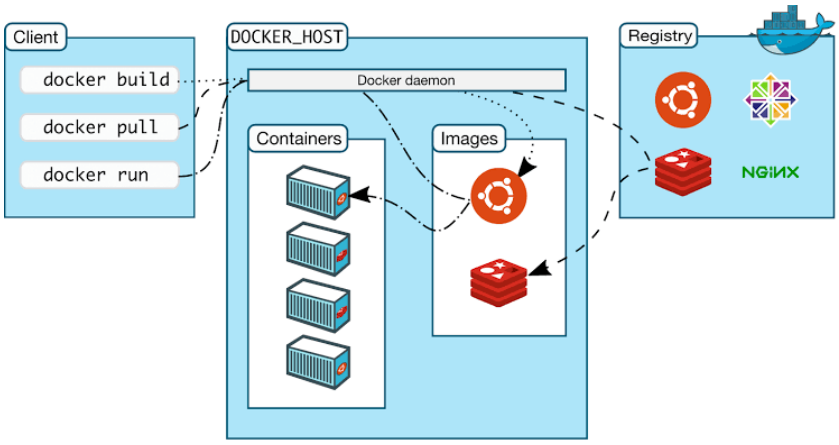

Figure 2: Docker Engine Architecture

Docker Engine Architecture

- **Docker CLI (`docker`)**: Command-line interface used by developers.
- **Docker Daemon (`dockerd`)**: Background process that manages containers and images.
- Uses a client-server model. CLI commands are sent to the daemon, which performs actions.

2.4 Working With Docker

Running Containers

- Use `docker run` to create and start a container from an image.
- Add `-it` for interactive terminals, and `--rm` to auto-remove on exit.

```
docker run -it --rm node
```

Volumes and Persistence

By default, data written inside a container does not persist. Use volumes to retain data.

- Syntax: `-v <host_path>:<container_path>`

```
docker run my-node-app -v /home/jake:/jake
```

Networking

- Use `-p <host_port>:<container_port>` to expose container ports.
- For complex networking, use Docker Compose.

```
docker run -p 8080:80 my-node-app
```

2.5 Docker Compose and Orchestration

Multi-Container Applications

Docker Compose allows orchestration of applications consisting of multiple containers using a YAML file (`compose.yaml`).

- Define services, volumes, networks, and configuration.
- Launch a full stack with:

```
docker compose up
```

2.6 Building Images

Using a Dockerfile

A Dockerfile automates the creation of Docker images through scripted instructions.

```
# Use the official Node.js image as the base image
FROM node:14

# Set the working directory inside the container
WORKDIR /app

# Copy the package.jsonand package-lock.jsonfiles to the working directory
COPY package*.json ./

# Install the dependencies (including express)
RUN npm install

# Copy the rest of the application code to the working directory
COPY . .

# Expose the port on which the app will run
EXPOSE 3000

# Command to run the application
CMD ["node", "app.js"]
```

Another example of using Dockerfile:

```
FROM python:3.12
WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir-r requirements.txt

# Copy in the source code
COPY src./src
EXPOSE 5000

# Setup an app user so the container doesn't run as the root user
RUN useraddapp
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Build and Run

```
docker build -t my-node-app .
docker run -p 3000:3000 my-node-app
```

.dockerignore

To exclude files from the image build process:

```
/node_modules
/tests
*.log
```

Best Practices

- Use multi-stage builds to reduce image size.
- Avoid running as root; create and use an app-specific user.
- Use '.dockerignore' to exclude unnecessary files.
- Pin base images to specific versions for reproducibility.

2.7 Docker in AWS

Amazon ECR (Elastic Container Registry)

- Fully managed Docker-compatible registry.
- Use `docker push` and `docker pull` with ECR.
- Free tier includes 500MB of private and 50GB of public storage.

Amazon ECS (Elastic Container Service)

- Fully managed orchestration for container-based workloads.
- Deep AWS integration.
- You pay only for the EC2/Fargate compute resources.

ECS Best Practices

- Keep containers stateless.
- Store logs to `stdout`/`stderr`.
- Use tags for versioning.
- Gracefully handle termination signals.

2.8 Docker CLI Essentials

Docker CLI command references can be viewed at: docs.docker.com/reference/cli/docker

Common Commands

```
docker --help
docker --version
docker info
docker images
docker ps -a
docker container rm <container_id>
```

Search and Run a Prebuilt Image

```
docker search node
docker run -it --rm node
```

Mount a Volume

```
docker run -it -v C:\container-output:/host bash
```

3 Application Server Fundamentals

3.1 Web Applications

Web applications are software systems accessed through web browsers over the internet. These applications interact with users via graphical interfaces rendered in HTML, CSS, and JavaScript. Client-server communication is facilitated using the Hypertext Transfer Protocol (HTTP), which follows a request-response model. Web applications may be simple (e.g., a static webpage) or complex (e.g., multi-tiered systems integrating APIs, authentication, and databases).

World Wide Web

- The World Wide Web (WWW) originated in 1985 and enables access to interlinked resources using URLs and HTTP.
- It operates on a client-server model where browsers (clients) send requests to servers, which respond with the requested content.
- **Static resources:** Delivered exactly as stored on the server (e.g., `.html`, `.png`, `.css`).
- **Dynamic resources:** Generated on-the-fly, often using server-side languages (e.g., PHP, Python, Node.js) or frameworks (e.g., Django, Express).
- Dynamic responses may also include data served from databases or API endpoints.

Web Architecture

Modern web applications follow distributed architecture, often integrating multiple services:

- A single page may request resources from several servers (e.g., web server, video server, advertising server).
- The application frontend issues HTTP requests for HTML content, CSS styles, JavaScript logic, media files, and API data.
- Examples of requests:
  - `GET /index.html` – retrieves main page markup.
  - `GET /styles.css` – retrieves styling.
  - `GET /video.mp4` – requests video content from a media server.
- These requests are often executed concurrently and are resolved by separate backend services or content delivery networks (CDNs).

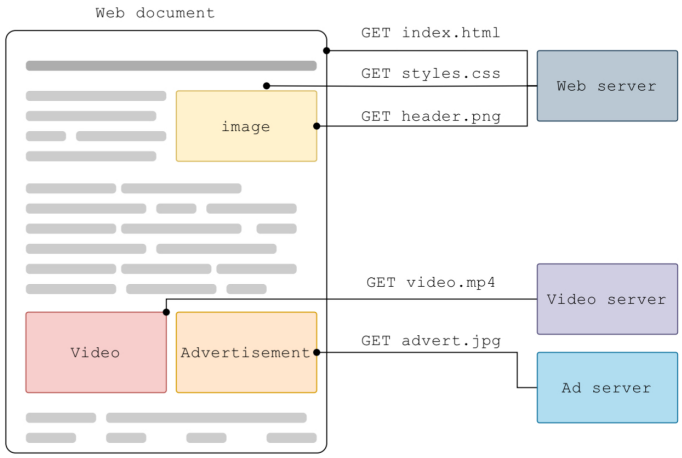

Figure 3: Example Web Architecture Diagram

Browser Characteristics

- Browsers (e.g., Chrome, Firefox, Safari, Edge) are full-featured user agents that interpret and render web content.
- They support:
  - HTML parsing and rendering.
  - JavaScript execution in a sandboxed environment.
  - Security models including Same-Origin Policy and sandboxing.
  - Extensions for HTTP caching, compression, cookies, and developer tools.
- Browsers implement web standards maintained by the W3C and WHATWG.
- They support modern APIs like Web Storage, WebSockets, and IndexedDB.

Web Applications are Asynchronous

- HTTP requests made from a web page may complete out-of-order or fail due to network issues.
- JavaScript employs the asynchronous programming model using:
  - `Promise` objects
  - `async`/`await` syntax
  - Event-driven callbacks (e.g., XHR, Fetch API)
- Asynchronous architecture allows multiple requests to be handled concurrently without freezing the user interface.
- Non-blocking behaviour is crucial for responsive UX.
- It is common to separate frontend and backend codebases:
  - Frontend (client-side) built using frameworks like React, Vue, Angular.
  - Backend (server-side) implemented in Express, Flask, Django, etc.
- Initially, monolithic applications are acceptable for simplicity, but microservice or decoupled architectures are preferable for scalability.

Web Tools

In addition to browsers, developers use command-line tools for HTTP communication and testing:

- `wget` – a non-interactive downloader for HTTP, HTTPS, and FTP.
- `curl` – supports various HTTP methods and headers, often used for API testing.
- These tools enable automation, scripting, and diagnostics for server responses.

```
wget https://canvas.qut.edu.au -O -
```

The above command fetches the HTML of the target webpage and outputs it to the terminal (stdout).

3.2 HTTP

Definition

Hypertext Transfer Protocol (**HTTP**) is a stateless, request-response protocol operating at the application layer of the OSI model. It is the foundational protocol for data communication on the World Wide Web. Clients, such as browsers or command-line tools, send requests to servers, which respond with data such as HTML pages, images, or JSON.

Figure 4: Uniform Resource Locator (URL)

HTTP Request/Response Cycle

- The client initiates a TCP connection to the server, typically on port 80 (HTTP) or 443 (HTTPS).
- It sends an HTTP request line and headers, e.g.:

```
GET / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
```

- The server responds with a status line, headers, and optionally a body:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024
```

- The TCP connection may be kept alive or closed, depending on the `Connection` header.

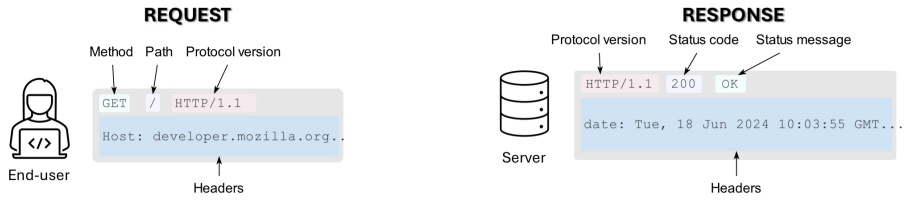

Figure 5: Response Cycle

HTTP Core Concepts and Media Types

HTTP is the foundational protocol used by web clients and servers to communicate. It follows a stateless model, meaning that each request-response cycle is treated independently with no retained memory of previous interactions.

While statelessness promotes scalability and simplicity, some applications require continuity across multiple requests. This is where HTTP cookies come into play. Cookies enable stateful behaviour by

storing session information on the client side, which the server can retrieve and use across requests to maintain user sessions or preferences.

Core Concepts

- HTTP is inherently stateless.
- Each request-response cycle is isolated and has no knowledge of previous interactions.
- HTTP cookies are used to store session state across multiple requests, introducing stateful behaviour.

Media types, also known as MIME types (Multipurpose Internet Mail Extensions), are used to indicate the format of content transmitted via HTTP. These types help browsers and other clients understand how to process the received data. For example, distinguishing whether the data is plain text, an image, or executable JavaScript.

Although files typically have extensions such as '.html' or '.jpg', these extensions should not be used as the sole indicator of content type. Instead, browsers rely on the MIME type declared in the HTTP response header. Servers may also utilise MIME types for optimisation purposes, such as enabling compression, concatenation, or intelligent caching.

Media Types / MIME Types

- MIME types describe the format of data sent in HTTP responses.
- File extensions are unreliable indicators of actual content.
- Browsers determine how to handle files based on the declared MIME type.
- Some servers use MIME types to perform optimisations.

Common HTTP Methods

HTTP defines a set of request methods that indicate the desired action to be performed on a given resource. These methods have standardised semantics and are crucial in the design of RESTful APIs.

- **GET:** Used to request and retrieve data from a server without causing any side effects. It is safe and idempotent.
  - Commonly used to fetch resources like HTML pages, images, CSS files, or JSON data.
  - GET requests do not include a request body; parameters are usually passed via the URL query string.
  - Repeating a GET request should not alter server state.

Figure 6: GET

- **POST:** Sends data to the server to create or process a resource. Unlike GET, POST is not idempotent and may change the server state.

  - Common use cases include submitting form data, uploading files, or creating database entries.
  - Each POST request can result in a different outcome (e.g., placing an order multiple times).
  - The request body contains the data being sent to the server.
  - *Example:* Posting a new comment, registering a user, uploading an image.

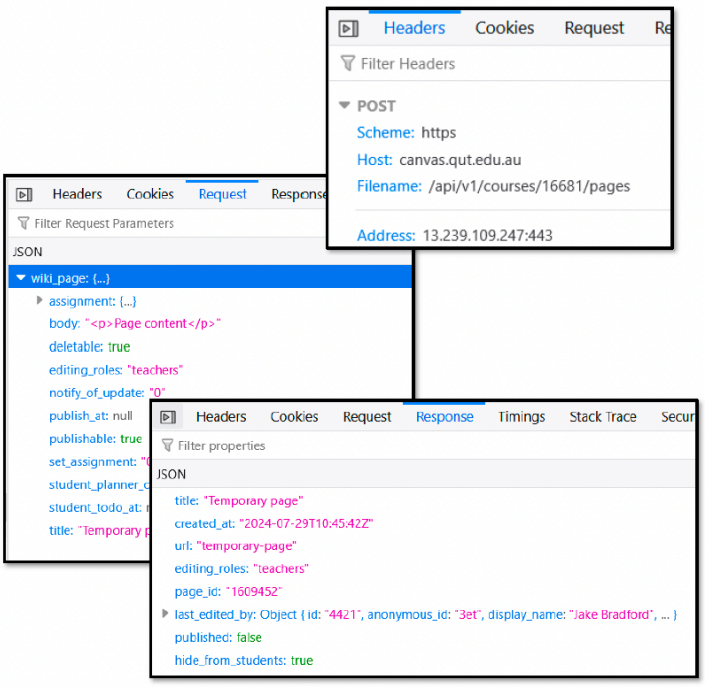

Figure 7: POST

- **PUT:** Replaces an existing resource or creates a new one at a specific URI. PUT is idempotent.
  - The entire state of the resource is provided in the request body.
  - Sending the same PUT request multiple times should result in the same resource state.
  - Common in APIs for updating or creating objects at a known URI.

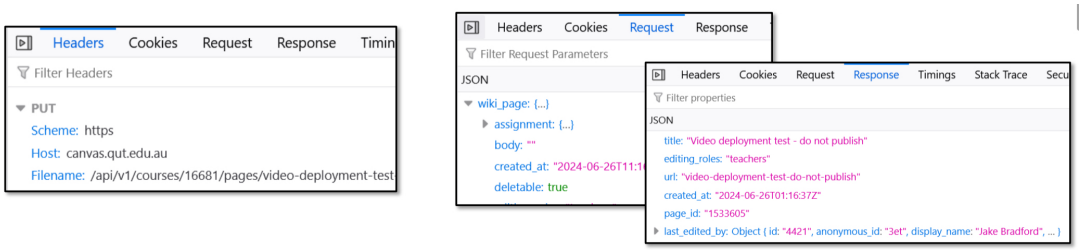

Figure 8: PUT

- **DELETE:** Requests that the server remove the specified resource. Also idempotent.
  - Repeated DELETE requests for the same resource have no additional effect once it is removed.
  - Typically used to delete database entries or API resources.

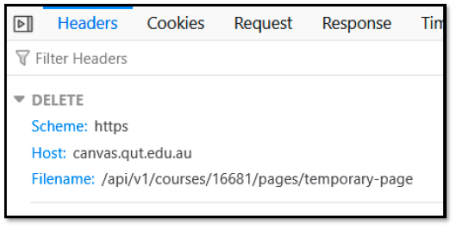

Figure 9: DELETE

- **PATCH:** Applies partial updates to a resource.
  - Unlike PUT, which sends a full replacement, PATCH includes only the fields to be updated.
  - Often used for efficiency and to avoid overwriting unchanged data.
  - May or may not be idempotent depending on implementation.
- **OPTIONS:** Describes the communication options available for a target resource.
  - Allows the client to determine which methods are supported (e.g., for CORS preflight checks).
  - No request body is typically required.
  - Response includes allowed methods in the ‘Allow‘ header.

Status Codes

HTTP status codes are grouped by category:

- **1xx – Informational:** Request received, continuing process.
- **2xx – Success:** Request processed successfully (e.g., 200 OK, 201 Created).
- **3xx – Redirection:** Further action needed (e.g., 301 Moved Permanently, 302 Found).
- **4xx – Client Error:** Request has errors (e.g., 400 Bad Request, 404 Not Found).
- **5xx – Server Error:** Server failed to fulfil request (e.g., 500 Internal Server Error, 503 Service Unavailable).

Cookies

Cookies are small key-value data stored on the client and sent with each request to the server. They are often used to:

- Maintain user sessions (e.g., session IDs).
- Track user preferences or activity.
- Support authentication and personalisation.

They can have attributes such as `HttpOnly`, `Secure`, and `SameSite` for added security.

Media (MIME) Types

The `Content-Type` header specifies the media type (also known as MIME type) of the data sent:

- `text/html` – standard HTML content.
- `application/json` – structured data (e.g., API responses).
- `image/jpeg` – image data.
- `text/css`, `text/javascript`, `audio/mpeg`, etc.

Clients use these types to decide how to process the response.

3.3 Advanced HTTP

Content Security Policy (CSP)

Content Security Policy (CSP) is a security feature that helps detect and prevent certain types of attacks, including Cross-Site Scripting (XSS) and data injection. It works by restricting the sources from which content can be loaded, thereby reducing the attack surface of a web page.

CSP policies are defined using the `Content-Security-Policy` HTTP response header. For example, to ensure that all content is loaded over HTTPS from a trusted domain:

```
Content-Security-Policy: default-src https://onlinebanking.example.com
```

A more complex policy might allow:

- All default resources (e.g., HTML, CSS, scripts) only from the same origin.
- Images from any source.
- Media (audio/video) only from `example.org` and `example.net`.
- Scripts exclusively from `userscripts.example.com`.

This can be written as:

```
Content-Security-Policy: default-src 'self'; img-src *;
media-src example.org example.net; script-src userscripts.example.com
```

Using CSP: To implement CSP, configure your web server to include the appropriate `Content-Security-Policy` header in HTTP responses. This tells the browser what content is allowed to load and execute.

- Writing effective CSP policies is complex but valuable.
- While CSP is primarily enforced by browsers, other tools making HTTP requests (e.g., curl, Postman) do not enforce these policies.
- CSP is a browser-side control mechanism intended to protect users in a controlled environment.

CSP Reporting: You can monitor violations using the `Content-Security-Policy-Report-Only` header. This allows you to test policies without enforcing them. Example:

```
Content-Security-Policy-Report-Only: default-src https:; report-to /csp-violation-report-endpoint/
```

A sample CSP violation report:

```
{
  "csp-report": {
    "blocked-uri": "http://example.com/css/style.css",
    "disposition": "report",
    "document-uri": "http://example.com/signup.html",
    "effective-directive": "style-src-elem",
    "original-policy": "default-src 'none'; style-src cdn.example.com; report-to
     ↪  /_/csp-reports",
    "referrer": "",
    "status-code": 200,
    "violated-directive": "style-src-elem"
  }
}
```

Cross-Origin Resource Sharing (CORS)

CORS controls how a browser permits resources requested from a domain different than the one serving the webpage. This is essential for API access from frontend applications. A server can allow cross-origin requests with:

```
Access-Control-Allow-Origin: *
```

More secure usage would specify a trusted domain:

```
Access-Control-Allow-Origin: https://frontend.example.com
```

Additional headers like `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers` define what is permitted.

Authentication

Authentication mechanisms in HTTP enable clients to prove their identity when accessing protected resources. Several schemes are in common use:

- **Basic Authentication:**
  - Sends a Base64-encoded `username:password` in the `Authorisation` header.
  - Example: `Authorisation: Basic dXNlcjpwYXNz`
  - Not encrypted — must only be used over HTTPS.
- **Bearer Authentication:**
  - Uses an encrypted token string, typically generated by the server.
  - Frequently used with OAuth 2.0 for delegated access.
  - Supplied in the request header: `Authorisation: Bearer mF_9.B5f-4.1JqM`
- **API Keys:**
  - A simple token-like string uniquely associated with a client or application.
  - Each API may have its own usage rules — always consult documentation.
  - May be sent in the header or as a query parameter, e.g., `api_key=ABC123DEF456`
- **JWT (JSON Web Token):**
  - Defined in RFC 7519.
  - A compact, URL-safe token format used for transmitting claims between parties.
  - Digitally signed to ensure authenticity and integrity.
  - Can be sent as a bearer token or within a cookie.
  - Often used in stateless API authentication and session management.

Caching

Caching improves performance by reducing redundant server requests. HTTP headers control how and when resources are cached:

```
Cache-Control: max-age=604800
```

This example allows caching the response for 7 days. Other headers include:

- `ETag` – a hash for conditional requests.
- `Last-Modified` – timestamp for changes.
- `Expires` – absolute expiry time.

Compression

Compression reduces payload size, improving transfer speed and saving bandwidth. Clients specify supported encodings:

```
Accept-Encoding: gzip, deflate, br
```

Servers respond using the best supported method:

```
Content-Encoding: gzip
```

Modern browsers and servers support Brotli (`br`) for optimal compression ratios.

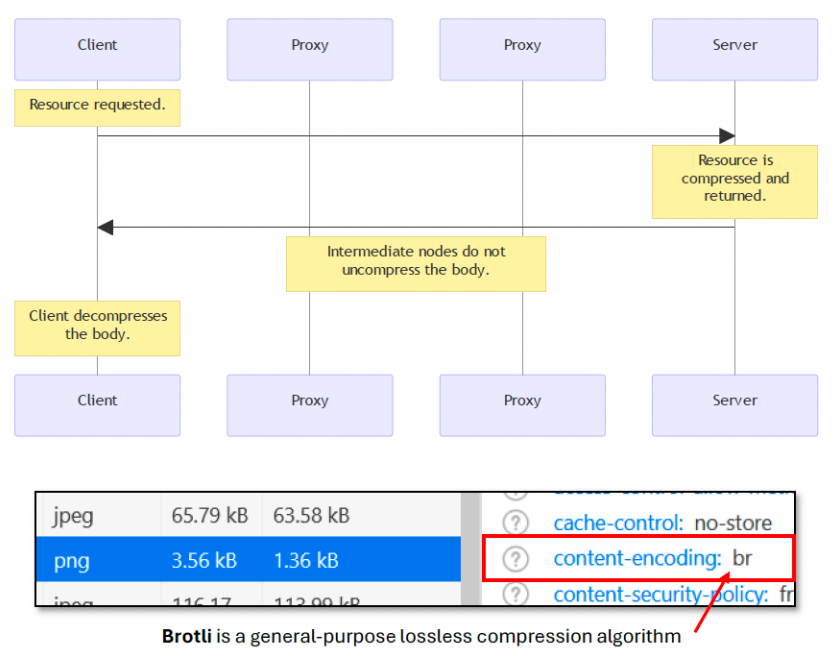

Figure 10: Diagram of Compression

3.4 Representational State Transfer (REST)

REST is an architectural style for building scalable, stateless networked applications. It is not a formal standard or protocol, but rather a set of design principles aligned with the HTTP protocol. RESTful APIs are widely adopted for their simplicity, scalability, and compatibility with web technologies.

REST Basics

A REST API exposes an interface to a software system using standard HTTP methods to operate on resources. Each resource is represented by a URI and is typically manipulated using:

- **Nouns in Endpoints:** Resources are represented as nouns (e.g., `/user`, `/post`, `/article`).
- **Verbs as HTTP Methods:** Operations such as retrieving, creating, or deleting are handled using HTTP methods (e.g., `GET`, `POST`, `PUT`, `DELETE`).
- **JSON Payloads:** Both request and response bodies often carry structured JSON data.

Endpoint and Verb Example

REST APIs should reflect real-world operations through simple and descriptive endpoint-method pairings:

- `GET /api/v1/user` → Retrieve user list
- `POST /api/v1/user` → Create a new user
- `GET /api/v1/user/42` → Retrieve user with ID 42
- `DELETE /api/v1/user/42` → Delete user with ID 42

Best Practices

Following best practices ensures REST APIs remain intuitive, predictable, and scalable.

- Use nouns for endpoint paths — avoid embedding verbs (e.g., use `/user/42`, not `/getuser/42`).
- Maintain statelessness; each request must be self-contained and not dependent on prior interactions.
- Structure your API so each endpoint returns a single type of resource.
- Always use appropriate HTTP methods for actions and keep responses predictable.

HTTP Status Codes

Returning correct HTTP status codes provides immediate context about the outcome of a request.

- `200 OK` — Request succeeded.
- `201 Created` — Resource successfully created.
- `204 No Content` — Action successful, but no content to return.
- `400 Bad Request` — The request was malformed or invalid.
- `401 Unauthorized` — Authentication required.
- `403 Forbidden` — Authenticated but lacking permission.
- `404 Not Found` — Resource does not exist.
- `500 Internal Server Error` — Unhandled server-side issue.

Filtering, Sorting, and Pagination

APIs should efficiently support large datasets without overloading the client or server.

- **Filtering:** Narrow down the results using query parameters (e.g., `/users?role=admin`).
- **Sorting:** Return results in a specified order (e.g., `/posts?sort=date_desc`).
- **Pagination:** Deliver results in manageable chunks (e.g., `/articles?page=2&limit=10`).

Versioning APIs

As APIs evolve, breaking changes may occur. Versioning helps prevent disruption to existing users.

- **Path-based:** Embed the version in the URI (e.g., `/api/v1/users`).
- **Header-based:** Indicate version with request headers (e.g., `Accepts-Version: 1.0`).
- **Query parameter:** Specify via query string (e.g., `/users?version=1.0`).

Security Practices

RESTful APIs must be secure by design to prevent common vulnerabilities.

- Always use HTTPS to encrypt communications.
- Authenticate using API keys, OAuth2, or JWT-based Bearer tokens.
- Avoid exposing credentials or sensitive data in URIs or query parameters.
- Validate all incoming data to prevent injection and misuse.

Respecting HTTP Concepts

RESTful APIs are designed to align with the core principles of HTTP. Violating these principles can result in APIs that are harder to scale, maintain, or consume reliably.

Statelessness Each HTTP request is treated as an independent transaction that is unrelated to any previous request. The server should not retain any client context between requests. For example, if a user uploads a file in multiple parts, each part must include all the information required for the server to process it correctly. Although some APIs (e.g., the Canvas LMS API) use multi-step processes that span multiple requests, this is generally discouraged and should be handled cautiously.

Metadata in Headers HTTP headers should be used to transmit metadata that is not specific to the resource path, such as authentication tokens, pagination details, or content type. While URI query parameters are useful for filters and sort orders, general metadata like rate limits or user-agent identification belongs in headers. This separation of concerns keeps URIs clean and promotes modularity.

One Resource per Endpoint Each endpoint should correspond to exactly one resource. Avoid mixing unrelated data in a single response. For instance, the following is good practice:

- `GET /api/v1/user/42` returns a single user
- `GET /api/v1/user/42/posts` returns posts for that user

Combining these into one request (e.g., returning both user data and all their posts from `/user/42`) makes it harder to maintain and optimise the API, and violates REST's principle of resource separation—akin to serving both `/index.html` and `/index.css` as one response.

Adhering to HTTP concepts ensures your API remains intuitive, scalable, and compatible with industry-standard tools and expectations.

API Consumers

REST APIs are designed to be platform- and language-agnostic, making them accessible to a wide range of client applications. Their simplicity and reliance on HTTP and JSON—both universally supported—have made RESTful APIs the de facto communication standard in modern software systems.

- **Web Browsers and Front-end Applications:** JavaScript-based front-end frameworks (e.g., React, Angular, Vue) commonly interact with REST APIs using `fetch` or `XMLHttpRequest` to dynamically retrieve or update content.
- **Command-Line Interfaces (CLIs):** Tools like `curl`, `httpie`, or Postman are frequently used to test and debug REST APIs, or to automate tasks in scripts.
- **Mobile and Desktop Applications:** Apps running on Android, iOS, Windows, or macOS often consume REST APIs to synchronise user data, pull live content, or interact with cloud services.
- **Backend Services and Microservices:** In distributed architectures, REST APIs serve as a common interface for communication between microservices or backend components. Since all major programming languages support HTTP clients and JSON parsers, interoperability is straightforward.

- **Third-party Integrations:** REST APIs also enable third-party developers to extend functionality, integrate external services, or embed application data into other platforms.

The broad compatibility and low barrier to entry make REST APIs an essential part of modern application ecosystems, supporting both internal and external communication patterns.

3.5 Node.js and Express

Node.js

Node.js is a lightweight, efficient JavaScript runtime built on Chrome's V8 engine. It is designed for building fast and scalable network applications, particularly suitable for I/O-heavy operations such as web servers, APIs, and real-time services. Node uses an event-driven, non-blocking I/O model, which allows it to handle large volumes of concurrent connections with minimal overhead. It is commonly used in backend development and is popular in modern full-stack JavaScript applications.

`npm` (Node Package Manager) is the default package manager for Node.js. It allows developers to install, share, and manage third-party libraries and dependencies efficiently.

Express

Express is a minimal and flexible web framework for Node.js that provides a robust set of features for building web and mobile applications. It simplifies HTTP request routing, middleware management, and response handling, enabling developers to build RESTful APIs and web services quickly. Express supports middleware chaining and custom middleware integration, making it highly extensible.

Hello World

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

This basic Express server listens on port 3000 and responds to HTTP GET requests to the root path by sending a plain text message.

REST Routes

In a RESTful API, routing defines how the server responds to various HTTP requests to specific endpoints. Each route consists of an HTTP method and a path that maps to a handler function. These handlers typically retrieve, create, update, or delete data.

The example below illustrates basic GET and POST routes using Express.js:

```
// In-memory "database"
let users = [];

// GET: Retrieve all users
app.get('/users', (req, res) => {
  res.json(users);
});

// POST: Create a new user
app.post('/users', (req, res) => {
```

```
  const newUser = req.body;
  users.push(newUser);
  res.status(201).json(newUser);
});
```

The `GET /users` route responds with the list of all users, while the `POST /users` route allows the client to send user data in the request body to add a new entry.

Route Parameters Route parameters are dynamic segments of a route's path, allowing the API to capture values directly from the URL. These values are accessible via `req.params`.

```
// Route with parameters
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params);
});
```

A request to `http://localhost:3000/users/34/books/8989` will produce the following response:

```
{ "userId": "34", "bookId": "8989" }
```

Full CRUD Example The following example demonstrates a more complete RESTful API using all major HTTP methods:

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

let users = [];

// GET: Retrieve all users
app.get('/users', (req, res) => {
  res.json(users);
});

// POST: Add a new user
app.post('/users', (req, res) => {
  const newUser = req.body;
  users.push(newUser);
  res.status(201).json(newUser);
});

// PUT: Update a user by ID
app.put('/users/:id', (req, res) => {
  const userId = req.params.id;
  const updatedUser = req.body;
  let user = users.find(u => u.id === userId);
  if (user) {
    Object.assign(user, updatedUser);
    res.json(user);
  } else {
    res.status(404).send('User not found');
  }
});

// DELETE: Remove a user by ID
```

```
app.delete('/users/:id', (req, res) => {
  const userId = req.params.id;
  users = users.filter(u => u.id !== userId);
  res.status(204).send();
});

// Start the server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

This server supports creating, reading, updating, and deleting user resources in accordance with REST principles. Each route is clearly mapped to a specific HTTP method and path pattern, ensuring clarity and maintainability.

Middleware

In Express, middleware functions are executed during the request-response lifecycle. They have access to the request object `(req)`, the response object `(res)`, and the `next()` function. Middleware can:

- Modify the request or response objects.
- End the request-response cycle (e.g., by sending a response).
- Pass control to the next middleware in the stack using `next()`.

Middleware is used for tasks such as logging, parsing request bodies, authentication, and more. Built-in middleware like `express.json()` allows the server to handle incoming JSON payloads.

Custom Middleware Example

```
const express = require('express');
const app = express();

// Custom middleware to attach request timestamp
const requestTime = function (req, res, next) {
  req.requestTime = Date.now();
  next();
};

app.use(requestTime);

app.get('/', (req, res) => {
  let responseText = 'Hello World!<br>';
  responseText += `<small>Requested at: ${req.requestTime}</small>`;
  res.send(responseText);
});

app.listen(3000);
```

This example defines a custom middleware function called `requestTime` that adds a timestamp to every incoming request. When a user visits the root endpoint `/`, the response includes the time the request was received.

Custom middleware is a flexible and powerful way to inject additional behaviour across multiple routes in a consistent and reusable manner.

3.6 HTTP and APIs in AWS

Node in EC2

Running Node.js applications in AWS EC2 (Elastic Compute Cloud) provides fine-grained control over server environments. However, for portability and ease of deployment, it's best practice to containerise Node applications using Docker. This ensures consistent behaviour across development and production environments.

Example Dockerfile

```
FROM node:latest
COPY . /app
WORKDIR /app
RUN npm install
CMD ["node", "myapp.js"]
```

This Dockerfile sets up a container with Node.js, copies the application code, installs dependencies, and runs the app.

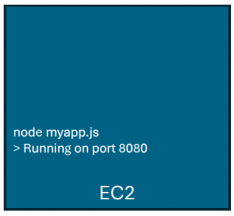

Figure 11: NodeJS in EC2

EC2 Hosting

To make the containerised app accessible from the internet, configure the EC2 security group to allow inbound traffic on the exposed port (e.g., 3000). Map the internal container port (e.g., 8080) to the host port:

```
docker run -p 3000:8080 my-node-app
```

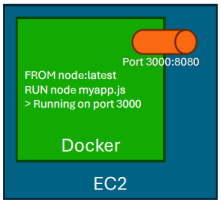

Figure 12: Docker in EC2

API Gateway

API Gateway is a fully managed AWS service that enables developers to create, publish, maintain, monitor, and secure APIs at scale. It supports REST, HTTP, and WebSocket APIs, and integrates with AWS services like Lambda and DynamoDB.

- Provides caching, throttling, and monitoring out of the box.
- Acts as a proxy to microservices or backend systems.
- Frequently paired with serverless backends, but less used in container-based CAB432 projects.

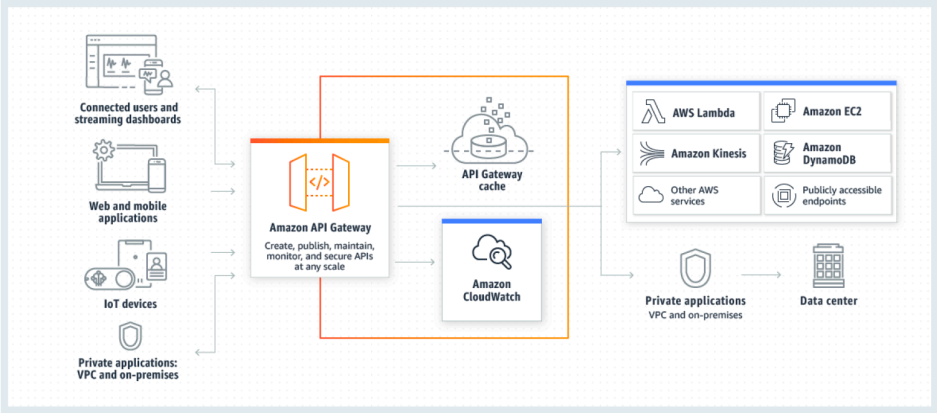

Figure 13: AWS API Gateway

3.7 Third-party APIs

Modern applications frequently integrate external services through Application Programming Interfaces (APIs). These APIs allow programs to exchange data or functionality, enabling developers to extend capabilities without building every feature in-house.

Types of APIs

- **Internal APIs:** Used within an organisation or system, particularly in microservices architectures. For example, an authentication service may expose an internal API used by a front-end or other backend services for verifying user credentials.
- **AWS APIs:** Amazon Web Services exposes APIs for programmatically managing cloud resources. For example, an application running on EC2 might use the S3 API to upload files, or use CloudWatch APIs to retrieve performance metrics.
- **Public APIs:** Offered by third parties to provide external services. These include music metadata from Spotify, weather forecasts from OpenWeatherMap, language models from OpenAI, or photo services from Flickr. Public APIs vary in reliability, cost, and licensing terms.

Use Cases

- **Data enrichment:** Applications can use external APIs to fetch real-time weather, currency exchange rates, news articles, or product reviews.
- **Functionality outsourcing:** Common functionality such as maps (Google Maps), payment processing (Stripe), or natural language processing (OpenAI) can be integrated using APIs.
- **User experience:** Social login via Google, Facebook, or Amazon, media integrations such as "currently playing" Spotify tracks, or calendar syncs with Google Calendar all rely on third-party APIs.
- **Backend orchestration:** Services in a cloud-native architecture may interact through APIs to trigger workflows or coordinate deployments (e.g., using AWS SDKs).

Guidance and Best Practices

- **Choose reputable providers:** Prefer well-documented APIs with active support and reliable uptime. Be wary of obscure or deprecated APIs.
- **Respect terms and conditions:** Many public APIs enforce strict terms of service, and violating them—especially around credential usage or data redistribution—can result in revoked access.

- **Manage usage constraints:** APIs may have free tiers with rate limits (e.g., 60 requests/min). Monitor quota usage and avoid overuse, particularly during assessment tasks.
- **Secure your credentials:** API keys or access tokens must not be embedded in client-side code or public repositories. Instead, store them securely on the backend or in environment variables.
- **Backend proxies:** Route API requests through your own server to add caching, authentication, error handling, or rate-limiting controls.
- **Avoid reliance on unstable services:** In coursework such as CAB432, if a third-party API becomes unavailable before a milestone, support cannot be guaranteed. Choose APIs with stable endpoints and predictable availability.

Public API Integration in CAB432

This unit encourages the exploration of real-world APIs as part of assessment. The aim is to promote hands-on experience working with live data sources and real service constraints. Some example APIs include:

- **Flickr** – image metadata and photo galleries
- **Spotify** – music search, playback, and user activity
- **OpenWeatherMap** – weather forecasts and historical data
- **OpenAI** – natural language and image generation
- **Google APIs** – Maps, Gmail, Calendar, Drive and more
- **Canvas LMS (QUT)** – access to learning data and student submissions

Working with third-party APIs allows students to develop practical integration skills, make real-time decisions about architecture and security, and experience the trade-offs of modern web service development.

4 Persistence

4.1 Introduction to Cloud Architecture

Cloud architecture refers to the composition and integration of cloud services to form a computing environment that supports an application's operational requirements. A well-designed architecture influences cost efficiency, performance, scalability, flexibility, and security.

Key Design Factors

- **Modularity:** Applications are decomposed into independent components such as:
  - *Compute:* Responsible for executing logic (e.g., web servers, transcoding workers).
  - *Storage:* Responsible for persistence (e.g., databases, blob storage).
  - *Support Services:* Networking, user authentication, DNS, TLS, etc.
- **Service Matching:** Each application component should use cloud services best suited to its requirements:
  - Use blob storage for large files (e.g., video, images).
  - Use SQL databases for structured metadata.
  - Use NoSQL for flexible schema or massive scale.
- **Service Categories:**
  - **Generic:** Low-level, reusable services like virtual machines.
  - **Specialised:** Task-optimised services like serverless compute.
  - **Managed:** Pre-configured, scalable services like AWS RDS (managed database).
- **Benefits of Specialisation:**
  - Cost-effective due to performance optimisation.
  - Reduced development and operational overhead.
  - Transparent scalability and high availability.

Cloud Service Examples

- **Compute:** EC2, Lambda, ECS
- **Storage:** S3 (blob), EBS (block), RDS (SQL), DynamoDB (NoSQL)
- **Networking:** Load balancers, VPCs, Route 53
- **Management:** IAM (Identity and Access Management), CloudFormation, CloudWatch

Example Dockerfile for packaging a Node.js app:

```
FROM node:latest
WORKDIR /app
COPY . .
RUN npm install
CMD ["node", "index.js"]
```

4.2 Scaling

Scaling refers to adjusting system resources to accommodate varying loads or performance requirements. It can be performed vertically or horizontally, with cloud architecture supporting both strategies.

Why Scale?

- To handle increased or fluctuating user demand.
- To distribute workload across geographic locations.
- To reduce latency and improve user experience.
- To increase availability and fault tolerance.

Vertical Scaling (Scaling Up)

- Involves upgrading an individual resource (e.g., bigger VM or DB instance).
- **Pros:** Simple to implement.
- **Cons:** Limited by hardware capacity; less fault tolerant; downtime may be required.

Horizontal Scaling (Scaling Out)

- Involves adding more instances of a resource.
- **Pros:** Enables high scalability, better fault tolerance, supports geo-distribution.
- **Cons:** Requires load balancing, shared state management, and orchestration.

Cloud-Aided Scaling Strategies

- **Decouple Compute and Persistence:** Allows independent scaling.
- **Use Load Distribution:** Load balancers, message queues, DNS-based routing.
- **Auto-Scaling:** Automatically increase or decrease instances based on metrics.

4.3 Persistence

Persistence refers to the **ability to store data beyond the lifetime of the process that created it**. Unlike transient data, which exists only in memory (RAM) during computation, persistent data is saved to a medium such as disk or SSD, allowing it to be retrieved and used later.

Examples: Files, databases, object storage.

Persistence Roles in Cloud

In a cloud architecture, persistence services play several crucial roles:

- **Storage Efficiency:** More cost-effective than keeping data on persistent VM disks.
- **Separation of Concerns:** Decouples data from compute resources, increasing flexibility and modularity.
- **Support for Serverless Architectures:** Serverless functions are stateless and require external storage.
- **Advanced Features:** Include backup, replication, versioning, and automated scaling.

Types of Persistence

- **Unstructured:** Used for storing raw data, such as images, videos, and log files. Typically stored in blob/object storage (e.g., AWS S3).
- **Structured:** Organised data stored in databases, such as SQL (relational) or NoSQL (non-relational) databases.

4.4 ACID

ACID is an acronym that describes four key properties of database transactions that ensure reliable processing even in the presence of concurrency, failures, or system crashes. Transactions group multiple operations into a single unit of work to maintain data integrity and consistency.

- **Atomicity:** Ensures that all operations within a transaction are completed successfully; if any part fails, the entire transaction is rolled back.
- **Consistency:** Guarantees that the database transitions from one valid state to another, adhering to all defined rules such as constraints and foreign keys.
- **Isolation:** Prevents interference between concurrent transactions by ensuring they appear to execute sequentially.
- **Durability:** Ensures that once a transaction is committed, the changes are permanently saved—even in the event of a system failure.

Cloud ACID Databases

Most relational databases such as MySQL, PostgreSQL, and MariaDB implement full ACID compliance. Some NoSQL databases like AWS DynamoDB offer configurable ACID transactional capabilities.

4.5 Distributed Data

Distributed data systems are essential in cloud computing to handle increasing data volumes, serve global users efficiently, and ensure system resilience. By spreading data across multiple machines and regions, they improve both performance and fault tolerance.

Goals of Distributed Data Systems:

- **Scalability:** Increase storage capacity by distributing data across multiple nodes.
- **Performance:** Improve read and write throughput by parallelising access across regions and servers.
- **Low Latency:** Serve users from geographically close locations to reduce response time.

Challenges of Distribution:

- **Consistency:** Maintaining a consistent state across multiple nodes is difficult, especially under network partitions or asynchronous communication.
- **Availability:** Ensuring that systems remain operational during failures or maintenance requires redundancy and robust failover strategies.
- **Conflict Resolution:** Concurrent updates to the same data on different nodes can lead to conflicts that need to be reconciled automatically or manually.

Consistency Models:

- **Eventual Consistency:** Prioritises availability and partition tolerance. Updates are allowed to propagate asynchronously; all nodes will eventually reach a consistent state if no further updates occur. This model is commonly used in NoSQL systems such as DynamoDB and Cassandra.
- **Strong Consistency with Replication:** Involves a designated leader (primary) node that handles all writes, which are then synchronously or asynchronously propagated to replica (follower) nodes. This model offers consistency guarantees but can introduce write latency and bottlenecks. It is often used in SQL databases like PostgreSQL and MySQL.

4.6 Relational Databases

Relational Database Management Systems (**RDBMS**) are a foundational technology for structured data storage and retrieval. They organise data into rows and columns, enforcing a predefined schema and supporting declarative access via SQL (Structured Query Language). These systems are widely used in enterprise applications due to their robustness, maturity, and strong consistency guarantees.

RDBMS Overview

Relational databases store data in *tables*, where:

- Each **table** defines a specific data entity (e.g., `Users`, `Orders`).
- Each **row** is a single record or tuple.
- Each **column** represents a typed attribute (e.g., `username`, `email`, `created_at`).
- Relationships between tables are established via primary and foreign keys.

Key Features

- **ACID Compliance:** Provides strong guarantees around data integrity and transaction safety.
- **Complex Query Support:** Enables powerful data retrieval using joins, subqueries, aggregations, and filters.
- **Schema Enforcement:** Enforces constraints (e.g., data types, uniqueness, nullability) to maintain data correctness.
- **Transaction Isolation Levels:** Supports multiple levels (e.g., Read Committed, Repeatable Read, Serializable) to manage concurrent access safely.

Common Use Cases

- Applications requiring strong consistency and strict data validation (e.g., finance, e-commerce, health).
- Migrating legacy systems that already use relational models.
- Systems with stable, well-defined schemas and predictable workloads.

Scale-Out Strategies

- **Replication:** Maintains one or more read-only copies of the database for load balancing and fault tolerance. Can be synchronous or asynchronous.
- **Partitioning (Sharding):** Distributes subsets of data across different physical nodes to improve scalability. Requires careful handling of cross-shard joins and consistency.
  - *Horizontal Partitioning:* Distributes rows across nodes.
  - *Vertical Partitioning:* Distributes columns or attributes across nodes.

Limitations

- **Schema Rigidity:** Making frequent changes to schemas (e.g., adding new columns or altering types) can be disruptive in production environments.
- **Scaling Challenges:** Joins and transactions across shards can lead to complexity and performance degradation.
- **Resource Constraints:** Vertical scaling is often used but has physical limits (CPU, RAM, disk I/O).

Examples of RDBMS

- **MySQL:** Widely used open-source RDBMS.
- **PostgreSQL:** Feature-rich, standards-compliant open-source RDBMS with strong community support.
- **MariaDB:** A fork of MySQL with performance and security improvements.
- **AWS Aurora:** Cloud-optimised RDBMS compatible with MySQL and PostgreSQL, offering auto-scaling and managed backups.

4.7 NoSQL

NoSQL (Not Only SQL) databases are non-relational data stores designed to address the limitations of traditional RDBMS in terms of flexibility, scalability, and performance. They are ideal for modern web-scale applications where data structures are often unpredictable or rapidly evolving.

Overview

Unlike relational databases, NoSQL systems do not require fixed schemas, and they scale out horizontally across multiple nodes. They prioritise availability, partition tolerance, and performance—often at the cost of immediate consistency (as per the CAP theorem).

Common characteristics:

- Designed for distributed architectures.
- Optimised for specific access patterns (e.g., key lookups, document queries).
- Schema-less or schema-optional designs.

Data Models

NoSQL systems are often categorised by their internal data model:

- **Key-Value Stores:** Store data as a collection of key-value pairs. Highly performant for caching and session management.
  - **Examples:** Redis, Memcached
  - **Example usage in Redis:**

```
SET user:1001 "Alice"
GET user:1001
```

- **Document Stores:** Store structured or semi-structured data as documents (e.g., JSON or BSON). Ideal for hierarchical and nested data.
  - **Examples:** MongoDB, CouchDB
  - **Example usage in MongoDB:**

```
db.users.insertOne({
  name: "Alice",
  age: 29,
  email: "alice@example.com"
});
```

- **Wide Column Stores:** Use a tabular format with flexible columns per row. Efficient for large-scale analytical workloads.
  - **Examples:** Apache Cassandra, HBase
- **Graph Databases:** Represent entities as nodes and relationships as edges. Useful for social networks, recommendation systems, and fraud detection.

  - **Examples:** Neo4j, RedisGraph
  - **Example Cypher Query (Neo4j):**

```
MATCH (a:Person)-[:FRIEND]->(b:Person)
WHERE a.name = "Alice"
RETURN b.name
```

Key Features

- **Schema Flexibility:** Fields can vary between records, enabling rapid prototyping and schema evolution.
- **High Throughput:** Suited to real-time workloads with frequent reads/writes.
- **Horizontal Scaling:** Data can be sharded and distributed across many nodes, improving availability and fault tolerance.
- **ACID Support:** While traditionally weaker in consistency, some systems like DynamoDB support ACID-compliant transactions.

Use Cases

NoSQL databases are a strong choice in the following scenarios:

- **Real-Time Analytics:** Ingest and query massive volumes of data quickly (e.g., time-series data in IoT).
- **Flexible Schema Requirements:** Ideal when data structure evolves frequently, such as in agile development or early-stage startups.
- **Social Networks and Recommendations:** Graph or document models help model complex relationships.
- **In-Memory Caching:** Reduce latency by storing frequently accessed data (e.g., Redis sessions or API responses).
- **Event-Driven Systems:** Event logs or user activity tracking where immutability and append-only structures are preferred.

4.8 Other Types of Persistence

Blob Storage (Object Storage)

- Suitable for large, unstructured data (e.g., images, video).
- Data accessed via HTTP(S) interface.
- Highly scalable and cost-effective.
- **Example:** AWS S3

Block Storage

- Presents data as raw disk blocks.
- Attached to VMs/containers as persistent volumes.
- Commonly used for databases or VM root disks.
- **Example:** AWS EBS

Network File Storage

- Provides shared file system access to multiple VMs/containers.
- Suitable for collaborative workloads or legacy apps.
- Offers POSIX-like filesystem interface over a network.
- **Example:** AWS EFS