Contents

2: App Structure

Description on front end and back end

Front-end Development: Front-end development, also known as client-side development, refers to the development of the user interface of a software application or a website. This includes everything that users see, hear, and interact with, such as layouts, colors, fonts, buttons, forms, and animations. Front-end developers use HTML, CSS, and JavaScript to create the visual and interactive components of the application or website that users directly interact with.

Back-end Development: Back-end development, also known as server-side development, refers to the development of the server-side logic of a software application or a website. This includes everything that happens behind the scenes, such as data storage, data retrieval, user authentication, and business logic. Back-end developers use programming languages like Python, PHP, Java, Ruby, and frameworks like Node.js, Django, Laravel, and Ruby on Rails to build the server-side components of the application or website that users don't directly see, but that enable the front-end to function properly.

In summary, front-end development deals with the visual and interactive components of an application or website, while back-end development deals with the server-side components that support the front-end and enable the application or website to function.

What is front end frameworks? and why

A front-end framework is a pre-designed and pre-built collection of HTML, CSS, and JavaScript code that provides developers with a set of tools and features for building the user interface of a software application or a website.

Front-end frameworks can include pre-built UI components such as buttons, forms, dropdown menus, and modals, as well as pre-designed layouts and styles for typography, colors, and responsive design. These pre-built components can help developers save time and effort in creating consistent and visually appealing user interfaces.

Some popular front-end frameworks include Bootstrap, Foundation, Materialize, Bulma, and Tailwind CSS. These frameworks are designed to be flexible and customizable, allowing developers to choose and modify the components and styles that fit their specific needs and preferences.

Using a front-end framework can have several benefits, such as:

1. Faster development: Front-end frameworks provide pre-built components and styles, which can help developers save time in building the user interface of an application or website.
2. Consistency: Front-end frameworks provide a consistent design language and style guide, making it easier to maintain consistency across multiple pages and sections of an application or website.
3. Responsive design: Many front-end frameworks are built with responsive design in mind, which means that they can automatically adjust the layout and style of the user interface based on the screen size and orientation of the device.

4. Cross-browser compatibility: Front-end frameworks are often tested across multiple browsers and devices, which can help ensure that the user interface looks and works consistently across different platforms.

Overall, front-end frameworks can be a useful tool for developers looking to build consistent, responsive, and visually appealing user interfaces for their applications or websites.

Compare between React, Angular, Ember, Vue and Backbone

React, Angular, Ember, Vue, and Backbone are all popular front-end JavaScript frameworks that provide developers with a set of tools and features for building interactive and dynamic user interfaces. Here's a comparison of some key aspects of each framework:

1. Learning curve: React and Vue are generally considered to have a lower learning curve compared to Angular and Ember, which can be more complex and require more setup and configuration. Backbone is the most lightweight and flexible of the five frameworks, but it also has a more limited feature set.
2. Performance: React and Vue are known for their fast rendering performance, while Angular and Ember can be slower due to their more complex architecture and features.
3. Flexibility: Backbone is the most flexible of the five frameworks, allowing developers to choose their own architecture and libraries. Vue and React also offer a high degree of flexibility, while Angular and Ember have more opinionated architectures and can be less customizable.
4. Community support: React and Angular have the largest and most active communities, with a wealth of resources, plugins, and documentation available. Vue has a growing and enthusiastic community, while Ember and Backbone have smaller communities but are still widely used in certain industries and projects.
5. Features: Angular and Ember are both full-featured frameworks that provide a wide range of tools and features out of the box, such as routing, data binding, and testing. React and Vue are more lightweight and focused on the view layer, but offer a wide range of third-party libraries and plugins that can be added for additional functionality.

Overall, the choice of framework depends on the specific needs and preferences of the project and development team. React and Vue are often recommended for smaller projects and startups due to their ease of use and flexibility, while Angular and Ember may be better suited for larger, more complex projects with more stringent requirements. Backbone is a good option for simple projects that require a lightweight and customizable framework

Name	Type	Shadow DOM EcmasScript 6+	Relative Popularity	Difficulty ofLearning
React	Library	Supported	*****	*****
Angular	Framework	Supported	***	*****
Ember	Framework	Supported	*	*****
Vue	Library	Supported	**	***
Backbone	Framework	Supported	*	***

Server-side architecture

Server-side architecture refers to the design and structure of the back-end components that power a software application or a website. The architecture of the back-end determines how data is stored, retrieved, processed, and transmitted between the client-side (front-end) and the server-side.

Here are some common server-side architectures:

1. Monolithic architecture: In a monolithic architecture, all the components of the back-end are tightly coupled and run in a single process or thread. This architecture is simple and easy to develop, but can be difficult to scale and maintain as the application grows.
2. Microservices architecture: In a microservices architecture, the back-end is broken down into smaller, independent services that can be developed, deployed, and scaled independently. Each service is responsible for a specific task or functionality, and communicates with other services via APIs. This architecture allows for greater scalability and flexibility, but can be more complex to develop and manage.
3. Serverless architecture: In a serverless architecture, the back-end is built using cloud-based services that provide compute, storage, and other resources on demand. This architecture allows for highly scalable and cost-effective back-ends, as resources are only used when needed. However, serverless architectures can be more complex to develop and may require a different approach to development and deployment.
4. Service-oriented architecture (SOA): In a service-oriented architecture, the back-end is broken down into modular services that can be reused across multiple applications. This architecture is similar to microservices, but emphasizes the use of standardized interfaces and protocols for service communication.

Overall, the choice of server-side architecture depends on the specific needs and requirements of the application or website. Factors such as scalability, flexibility, maintenance, and cost will all play a role in determining the most appropriate architecture.

Web Server

A web server is a software program or application that delivers web content, such as web pages, files, and multimedia, to client devices over the internet. The web server is responsible for receiving and processing requests from clients, and returning the appropriate response.

Web servers use a client-server model to communicate with clients. When a client device, such as a web browser, requests a web page or file from a server, the request is sent over the internet to the web server. The web server receives the request, processes it, and returns the requested content to the client device. This process is known as the request-response cycle.

There are many different web server software programs available, including Apache, Nginx, Microsoft IIS, and Google Web Server. These servers can run on a variety of operating systems, including Windows, Linux, and macOS.

Web servers are an essential component of the World Wide Web, allowing users to access and view web content from anywhere in the world. Without web servers, it would not be possible to share and access information over the internet in the way that we do today.

Web Server – Types

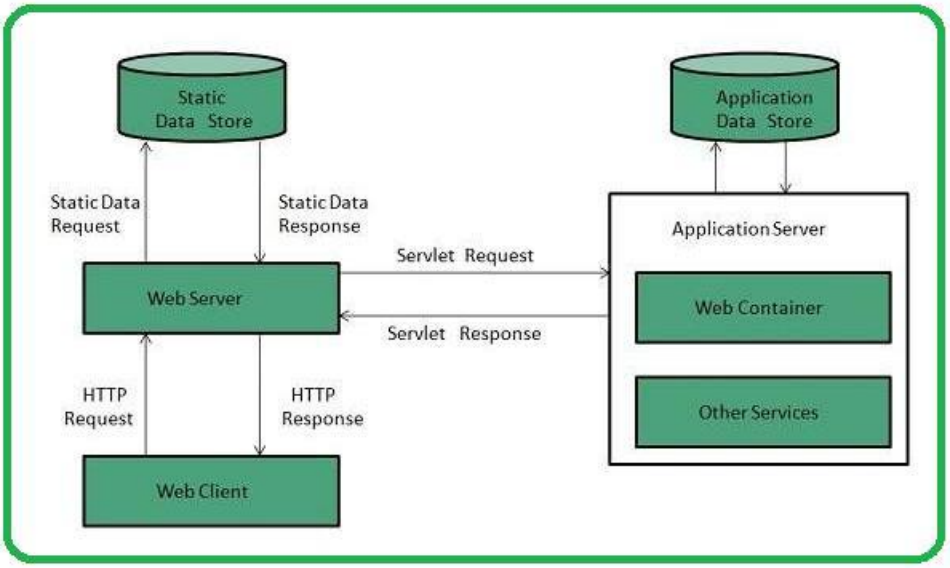
Dynamic vs. static web servers

Dynamic and static web servers are two different types of web servers that serve different types of web content.

Static web servers deliver static web content, such as HTML, CSS, and images, that do not change frequently. These files are stored on the web server and are delivered to the client as-is when requested. Static web servers are simple and efficient, as they do not require any processing or database access. Examples of static web servers include Apache and Nginx.

Dynamic web servers, on the other hand, deliver dynamic web content, such as web pages that are generated on-the-fly based on user requests or database queries. Dynamic web servers use server-side scripting languages, such as PHP, Python, or Ruby, to generate HTML content in response to user requests. Dynamic web servers require more resources and processing power than static web servers, but are necessary for websites that require user interaction or access to databases. Examples of dynamic web servers include Apache with PHP and Microsoft IIS with ASP.NET.

Overall, the choice of web server type will depend on the specific needs and requirements of the website or application. Websites that deliver primarily static content, such as blogs or portfolio sites, can benefit from using a static web server for its simplicity and efficiency. Websites that require dynamic content, such as e-commerce sites or social media platforms, will require a dynamic web server to generate content on-the-fly based on user requests.



Server-side Components - Servers: The machinery

Servers are another essential component of server-side architecture, as they provide the physical hardware and infrastructure necessary to run web applications. They are often referred to as the "machinery" of web applications, as they provide the computing power and resources needed to handle large amounts of traffic and data.

Servers can be physical or virtual, depending on the needs and requirements of the application. Physical servers are dedicated hardware that is installed on-premise or in a data center. They offer high performance and reliability, but can be expensive to purchase and maintain. Virtual servers, on the other hand, are created by partitioning a physical server into multiple virtual machines. They offer flexibility and cost-effectiveness, but may not offer the same level of performance and reliability as physical servers.

There are several types of servers available, including web servers, application servers, and database servers. Web servers, such as Apache and Nginx, are designed to handle HTTP requests and deliver static web content, such as HTML, CSS, and images. Application servers, such as Tomcat and JBoss, are designed to run server-side applications, such as Java and Ruby applications. Database servers, such as MySQL and PostgreSQL, are designed to store and manage large amounts of data for web applications.

Servers can also be managed in various ways, including on-premise management, managed hosting, and cloud hosting. On-premise management involves purchasing and maintaining physical servers on-site, while managed hosting involves outsourcing server management to a third-party provider. Cloud hosting, such as Amazon Web Services and Microsoft Azure, involves renting virtual servers and infrastructure from a cloud provider.

Overall, servers are a critical component of server-side architecture, and are essential for the performance, reliability, and scalability of web applications. The choice of server will depend on the specific needs and requirements of the application, as well as factors such as performance, scalability, and cost

Server-side Components - Databases: The brains

Databases are an essential component of server-side architecture, as they store and manage large amounts of data for web applications. They are often referred to as the "brains" of web applications, as they provide the foundation for data storage and retrieval, and enable web applications to be more powerful and useful.

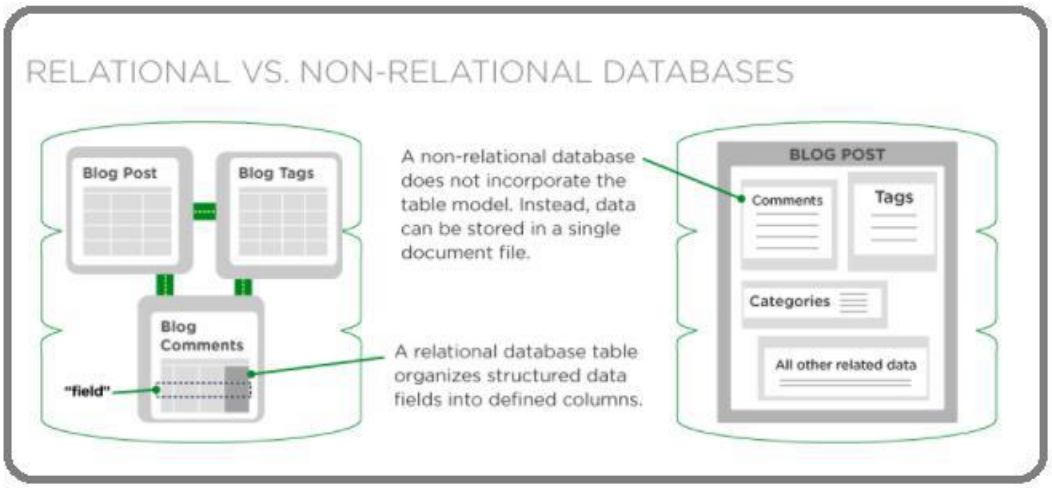
There are several types of databases available, including relational databases, NoSQL databases, and graph databases. Relational databases, such as MySQL and PostgreSQL, are structured and organized in tables, with data stored in rows and columns. They are ideal for applications that require complex queries and data relationships, such as e-commerce websites and content management systems.

NoSQL databases, such as MongoDB and Cassandra, are designed for large-scale, distributed, and unstructured data storage. They are often used in applications that require high availability and scalability, such as real-time analytics and social media platforms.

Graph databases, such as Neo4j and OrientDB, are used to store and manage complex data relationships, such as social networks and recommendation systems. They are ideal for applications that require efficient querying and visualization of complex data.

Databases can also be hosted locally or in the cloud. Local databases are installed on the server itself, while cloud databases are hosted remotely and accessed over the internet. Cloud databases offer many benefits, including scalability, high availability, and disaster recovery.

Overall, databases are a critical component of server-side architecture, and are essential for the storage and management of data for web applications. The choice of database will depend on the specific needs and requirements of the application, as well as factors such as scalability, performance, and security.



Server-side Components - Middleware: The plumbing

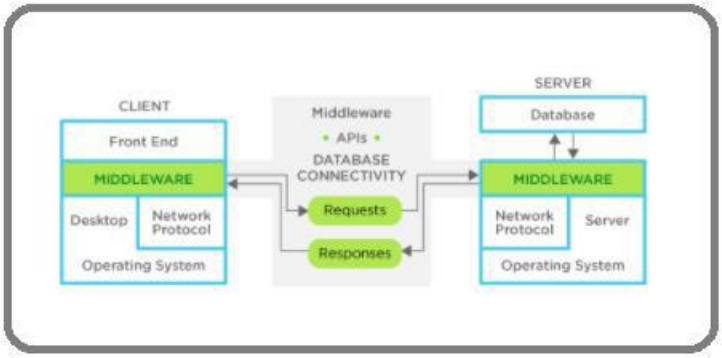
Middleware is another important component of server-side architecture, acting as the "plumbing" that connects different components and services within a web application. Middleware serves as a bridge between the application and the underlying hardware and operating system, allowing different components to communicate with each other seamlessly.

Middleware includes a wide range of software components, including web servers, application servers, message queues, and authentication systems. These components work together to provide the necessary functionality and services for a web application to function.

Web servers, such as Apache and Nginx, act as middleware by handling HTTP requests and delivering static web content, such as HTML, CSS, and images. Application servers, such as Tomcat and JBoss, provide middleware services for running server-side applications, such as Java and Ruby applications.

Message queues, such as RabbitMQ and ActiveMQ, act as middleware by providing a messaging system that enables communication between different components of a web application. Authentication systems, such as OAuth and SAML, provide middleware services for managing user authentication and authorization.

Overall, middleware is an essential component of server-side architecture, and is critical for the proper functioning and communication between different components of a web application. The choice of middleware will depend on the specific needs and requirements of the application, as well as factors such as performance, scalability, and security.



Server-side Components - Programming languages & frameworks: The nuts & bolts

Programming languages and frameworks are the "nuts and bolts" of server-side architecture, providing the tools and building blocks for creating web applications. They are used to write the code that runs on the server, and are essential for building complex, dynamic web applications.

There are many programming languages and frameworks available for server-side development, including popular languages such as Java, Python, Ruby, and PHP. Each language has its own strengths and weaknesses, and the choice of language will depend on the specific needs and requirements of the application.

Java, for example, is a popular choice for building enterprise-level applications, as it offers strong support for multithreading, scalability, and security. Python, on the other hand, is known for its ease of use and versatility, and is often used for building web applications and data analysis tools.

Frameworks, such as Spring for Java and Django for Python, provide a set of pre-built libraries and tools that simplify the development process and speed up time-to-market. Frameworks also provide a consistent structure and architecture for web applications, making it easier to maintain and update code over time.

Overall, programming languages and frameworks are essential components of server-side architecture, and are critical for the development of complex, dynamic web applications. The choice of language and framework will depend on the specific needs and requirements of the application, as well as factors such as performance, scalability, and maintainability

Server-side Components - APIs: Crucial tech in Back-End programming

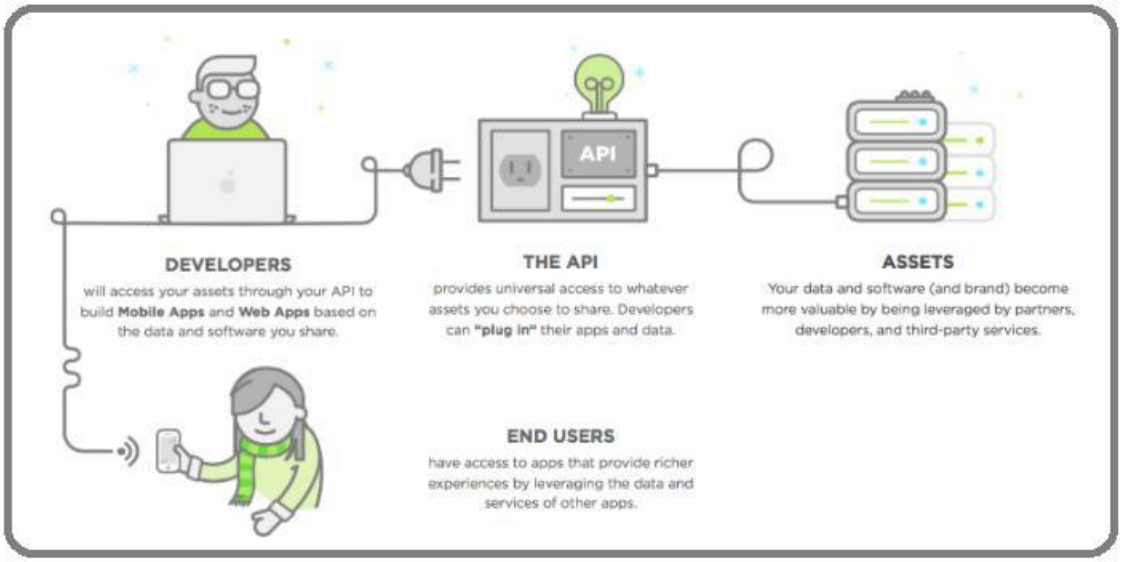
APIs, or application programming interfaces, are a crucial component of server-side architecture. APIs allow different software components and services to communicate with each other, making it possible to create complex, integrated web applications.

APIs provide a set of predefined rules and protocols for interacting with a web application, making it possible for developers to build new functionality on top of existing services. APIs can be used to connect to databases, perform calculations, or retrieve data from external sources.

There are many different types of APIs, including RESTful APIs, SOAP APIs, and GraphQL APIs. RESTful APIs are the most common type of API used in web development, and provide a simple, lightweight way to interact with web applications using HTTP requests.

APIs can be used to build custom integrations between different software components, or they can be used to expose functionality to third-party developers. This allows developers to build new applications on top of existing services, creating a vibrant ecosystem of third-party applications and services.

APIs are a critical component of modern server-side architecture, and are essential for building complex, integrated web applications. The choice of API will depend on the specific needs and requirements of the application, as well as factors such as performance, scalability, and security.



Server Side Scripting

Server-side scripting is a type of programming that runs on the server, rather than on the client-side (i.e. in the user's web browser). Server-side scripting is used to generate dynamic content, such as web pages that change depending on user input or data retrieved from a database.

Server-side scripting languages include PHP, Python, Ruby, and ASP.NET, among others. These languages are designed to interact with servers and databases, making it possible to build complex, dynamic web applications.

Server-side scripting works by running scripts on the server in response to user requests. The server processes the request, executes the script, and generates a response that is sent back to the user's web browser. This allows for the generation of dynamic content that can change in response to user input or changes in the underlying data.

Server-side scripting is often used in combination with other server-side technologies, such as databases, web servers, and middleware, to create complete web applications. Server-side scripting is also essential for creating web applications that require authentication, such as e-commerce websites and web-based applications.

Overall, server-side scripting is an essential component of modern web development, and is critical for building complex, dynamic web applications that can interact with servers, databases, and other components of a web application

Tech stack

Tech stack refers to the set of technologies and tools that are used to build a web application. A typical tech stack for a web application includes the following components:

1. Front-end Framework: This includes the libraries, frameworks, and tools used to build the user interface of the application. Popular front-end frameworks include React, Angular, Vue.js, and Ember.
2. Back-end Framework: This includes the tools and frameworks used to build the server-side of the application, including the APIs, databases, and server-side scripting. Popular back-end frameworks include Node.js, Django, Ruby on Rails, and Laravel.
3. Database: This includes the technology used to store and manage the data used by the application. Popular databases include MySQL, PostgreSQL, MongoDB, and Redis.
4. Cloud Services: This includes the cloud-based services and platforms used to host and deploy the application, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform.
5. DevOps Tools: This includes the tools and processes used for continuous integration and deployment (CI/CD), testing, monitoring, and debugging. Popular DevOps tools include Jenkins, Docker, Kubernetes, and GitLab.

The specific tech stack used for a web application will depend on a variety of factors, including the requirements of the application, the development team's experience and expertise, and the budget and timeline for the project. Choosing the right tech stack is essential for building a scalable, maintainable, and secure web application

REST vs gRPC vs GraphQL

REST, gRPC, and GraphQL are three popular approaches to building APIs for modern web applications. Each approach has its own strengths and weaknesses, and the choice of approach will depend on the specific requirements of the application.

1. REST (Representational State Transfer): REST is a widely adopted architectural style for building web services. REST APIs use a set of standard HTTP methods (GET, POST, PUT, DELETE) to interact with resources. REST APIs are simple to build, easy to understand, and work well with a wide range of client applications. REST APIs are also stateless, which makes them scalable and easy to cache.
2. gRPC: gRPC is a high-performance, low-latency framework for building remote procedure call (RPC) APIs. gRPC uses Protocol Buffers, a language-agnostic binary format, to serialize data and supports multiple programming languages. gRPC is particularly well-suited to building microservices and is designed to be fast, efficient, and scalable.
3. GraphQL: GraphQL is a query language for APIs that provides a more flexible, efficient, and powerful alternative to REST. GraphQL allows clients to request exactly the data they need, reducing over-fetching and under-fetching of data. GraphQL also allows clients to define their own data structures and relationships, making it easier to iterate on the API and build new features. However, implementing GraphQL can be more complex than REST, and it may not be suitable for all use cases.

In summary, REST is a simple, flexible, and widely adopted approach to building APIs. gRPC is a fast and efficient framework for building microservices. GraphQL provides a powerful and flexible query

language for APIs, but may be more complex to implement. The choice of approach will depend on the specific requirements of the application, including factors such as performance, scalability, flexibility, and developer experience.

Backend as a Service (BaaS)

Backend as a Service (BaaS) is a cloud computing model where a third-party provider hosts and manages the back-end infrastructure for web and mobile applications. BaaS providers offer pre-built back-end services such as databases, storage, user authentication, push notifications, and other functionality that developers can access through an API.

BaaS providers can help simplify the development process by removing the need for developers to build and manage their own back-end infrastructure. This can save time and resources, as developers can focus on building the front-end user interface and integrating with the BaaS API.

Some advantages of using BaaS include:

1. Speed of development: BaaS can help accelerate development by providing pre-built back-end services that can be easily integrated into an application.
2. Scalability: BaaS providers can offer scalable infrastructure that can handle large volumes of data and traffic.
3. Reduced cost: BaaS can help reduce the cost of building and maintaining a back-end infrastructure, as developers do not need to invest in servers, databases, and other infrastructure components.
4. Security: BaaS providers can offer built-in security features such as user authentication and access control, helping to protect applications from security threats.

Some popular BaaS providers include Firebase, Parse, AWS Amplify, and Backendless. However, it's important to note that BaaS may not be suitable for all applications, particularly those with complex back-end requirements or high levels of customization

Why Backend as a service?

Backend as a Service (BaaS) can provide several benefits for developers and organizations. Here are some reasons why someone might choose to use a BaaS solution:

1. Faster development: BaaS can help developers accelerate the development process by providing pre-built back-end components and services that can be easily integrated into an application. This can save developers significant time and effort in building, testing, and maintaining their own back-end infrastructure.
2. Reduced costs: Building and managing a back-end infrastructure can be expensive, particularly for small businesses and startups with limited resources. BaaS providers can offer cost-effective solutions that eliminate the need for investing in hardware, software licenses, and infrastructure maintenance.

3. Scalability: BaaS providers can offer scalable infrastructure that can handle large volumes of data and traffic. This can be particularly useful for applications that experience rapid growth or have unpredictable usage patterns.
4. Security: BaaS providers can offer built-in security features such as user authentication and access control, helping to protect applications from security threats.
5. Integration: BaaS providers often offer pre-built integrations with other services and platforms, such as third-party APIs, social media platforms, and messaging services. This can help developers to easily add new functionality to their application without having to build it from scratch.
6. Focus on core business: By outsourcing the back-end infrastructure to a BaaS provider, developers and organizations can focus on building and improving their core business offerings rather than spending time on back-end infrastructure.

Overall, Backend as a Service can provide significant benefits in terms of speed, cost, scalability, security, and integration, making it an attractive option for developers and organizations looking to build and deploy web and mobile applications

DBMS

A DBMS, or database management system, is software that manages the storage, organization, and retrieval of data in a database. It provides an interface for users and applications to interact with the database, allowing them to create, read, update, and delete data.

A DBMS is designed to handle the following tasks:

1. Data storage and retrieval: The DBMS manages the physical storage of data on disk or in memory, and provides mechanisms for retrieving and updating the data.
2. Data organization: The DBMS provides tools for organizing data into tables, columns, and rows, and enforcing relationships between them.
3. Data manipulation: The DBMS provides tools for querying and manipulating data, such as SQL.
4. Data security: The DBMS provides mechanisms for controlling access to data, enforcing data privacy, and protecting against unauthorized access or data breaches.
5. Data backup and recovery: The DBMS provides tools for backing up data, restoring it in case of a failure, and recovering data from corrupted or lost data files.
6. Performance optimization: The DBMS provides tools for optimizing database performance, such as indexing, caching, and query optimization.

Some popular DBMS systems include Oracle, MySQL, PostgreSQL, Microsoft SQL Server, MongoDB, and Cassandra. Each system has its own strengths and weaknesses, and the choice of a DBMS depends on the specific requirements of the application or organization

Advantages Disadvantages of DBMS

Advantages of DBMS:

1. Data integration: DBMS allows the integration of data from multiple sources, which can be stored and managed centrally. This enables efficient management of large volumes of data.

2. Data sharing: With a DBMS, multiple users can access the same data simultaneously. This enables real-time collaboration and decision making.
3. Data consistency: DBMS ensures data consistency by enforcing data integrity rules, which prevent the insertion of incorrect or inconsistent data.
4. Data security: DBMS provides security features such as access control, authentication, and encryption to ensure that data is protected from unauthorized access.
5. Backup and recovery: DBMS allows for automated backups and data recovery, ensuring data is available in the event of a disaster.

Disadvantages of DBMS:

1. Cost: DBMS can be expensive to set up and maintain, as it requires specialized hardware, software, and skilled personnel to operate.
2. Complexity: DBMS can be complex to manage, requiring significant training and expertise to design and maintain the database.
3. Performance: DBMS can sometimes suffer from performance issues, particularly when managing large volumes of data. This can be mitigated by optimizing the database design and query execution.
4. Single point of failure: DBMS can be a single point of failure, as all data is stored in a central location. This can lead to data loss or downtime if the system fails.
5. Vendor lock-in: DBMS can sometimes create a vendor lock-in, as it can be difficult to migrate data to a different system if the organization decides to switch vendors.

When not to use DBMS

There are certain situations where using a DBMS may not be the best choice. Some examples include:

1. Small-scale applications: If an application only requires a small amount of data to be stored and does not require complex data management, a DBMS may be unnecessary. In such cases, a simpler storage solution such as a file system may be more appropriate.
2. Performance-critical applications: If an application requires extremely high performance, such as in real-time systems or high-frequency trading, a DBMS may not be the best choice. In such cases, a specialized solution such as an in-memory database or a key-value store may be more appropriate.
3. Limited budget: If an organization has a limited budget, implementing a DBMS may be prohibitively expensive. In such cases, open-source DBMS options or cloud-based DBMS services may be more cost-effective.
4. Security concerns: If an application requires a high level of security, such as in the case of sensitive data, a DBMS may not be the best choice. In such cases, a specialized security solution such as an encrypted file system or a data encryption tool may be more appropriate.

5. Unstructured data: If an application deals with unstructured data such as images, videos, or audio, a DBMS may not be the best choice. In such cases, a specialized storage solution such as a content management system or a multimedia database may be more appropriate

Classification of DBMS

There are several ways to classify DBMS, including:

1. Based on data model: DBMS can be classified based on the data model they use to organize and store data. Examples include:

- Relational DBMS (RDBMS): These are based on the relational model, where data is organized into tables consisting of rows and columns.
- Object-oriented DBMS (OODBMS): These are based on the object-oriented model, where data is organized into objects that encapsulate data and behavior.
- NoSQL DBMS: These are non-relational DBMS that do not use the traditional tabular format. Instead, they use a variety of data models such as document-oriented, key-value, and graph.

2. Based on number of users: DBMS can be classified based on the number of users that can access the system concurrently. Examples include:

- Single-user DBMS: These are designed for use by a single user at a time.
- Multi-user DBMS: These can be used by multiple users simultaneously.

3. Based on architecture: DBMS can be classified based on their architecture. Examples include:

- Centralized DBMS: These have a single point of control and are managed centrally.
- Distributed DBMS: These are spread across multiple locations and can be managed centrally or locally.

4. Based on functionality: DBMS can be classified based on their functionality. Examples include:

- Transactional DBMS: These are designed to manage transactions and ensure data consistency.
- Analytical DBMS: These are designed for data analysis and decision-making.
- Data warehousing DBMS: These are designed for storing and managing large volumes of data.

These are just a few examples of the ways in which DBMS can be classified

Relational DBMS

A Relational DBMS (RDBMS) is a type of DBMS that is based on the relational model. In the relational model, data is organized into tables consisting of rows and columns. Each table represents a specific type of entity, such as customers or orders, and each row represents a specific instance of that entity.

RDBMS use SQL (Structured Query Language) to interact with the database. SQL is a standard language for managing relational databases and allows users to create, read, update, and delete data in the database.

Some of the key features of RDBMS include:

1. ACID Compliance: RDBMS ensure that transactions are Atomic, Consistent, Isolated, and Durable, which ensures data consistency and reliability.
2. Data Integrity: RDBMS enforce data integrity rules, such as primary key and foreign key constraints, to ensure that data is accurate and consistent.
3. Scalability: RDBMS can scale up or down depending on the size of the data, making them suitable for small as well as large enterprises.
4. Security: RDBMS provide a range of security features such as authentication, authorization, and encryption to ensure that data is secure.
5. Flexibility: RDBMS allow for ad-hoc queries and reports, which makes it easy to retrieve data from the database.

Some popular RDBMS include MySQL, Oracle, Microsoft SQL Server, PostgreSQL, and SQLite.

SQL

SQL stands for Structured Query Language and it is a standard programming language used for managing and manipulating relational databases. SQL allows users to create, modify, and query databases, as well as perform other tasks such as adding and deleting data, creating tables and indexes, and managing transactions.

SQL is a declarative language, meaning that users describe what they want the database to do, rather than how to do it. SQL uses a set of commands or statements to interact with the database. Some of the commonly used SQL commands include:

- SELECT: Used to retrieve data from one or more tables.
- INSERT: Used to insert new data into a table.
- UPDATE: Used to modify existing data in a table.
- DELETE: Used to delete data from a table.
- CREATE: Used to create a new table, index or view.
- ALTER: Used to modify an existing table, index or view.
- DROP: Used to delete a table, index or view.

SQL is used by many relational database management systems, including Oracle, Microsoft SQL Server, MySQL, PostgreSQL, and SQLite. Although these RDBMS may have some differences in syntax and features, the basic SQL language is largely the same across all RDBMS.

SQL is a powerful language that allows users to extract and analyze data from a database, and it is widely used by developers, data analysts, and data scientists.

Characteristic of NoSQL

NoSQL (Not Only SQL) is a non-relational database management system that is designed to handle large volumes of unstructured and semi-structured data. Here are some of the characteristics of NoSQL databases:

1. Schema-less: NoSQL databases are schema-less, which means that data can be stored without a pre-defined schema or structure. This allows for greater flexibility in handling different types of data.
2. Distributed: NoSQL databases are designed to work with distributed systems, which means that they can handle large volumes of data across multiple servers.
3. High scalability and availability: NoSQL databases are highly scalable and can handle large amounts of data with ease. They can also provide high availability by replicating data across multiple servers.
4. Designed for big data: NoSQL databases are designed to handle large volumes of data, which makes them ideal for big data applications.
5. Support for unstructured data: NoSQL databases can handle unstructured and semi-structured data, such as text, images, videos, and other multimedia formats.
6. High performance: NoSQL databases are optimized for high performance and can handle large volumes of data with low latency.
7. Horizontal scaling: NoSQL databases can scale horizontally, which means that new servers can be added to the cluster to handle additional load.

NoSQL databases are widely used in applications that require high scalability, availability, and performance, such as social media platforms, e-commerce websites, and big data analytics applications. However, they may not be suitable for all applications, especially those that require complex querying and transactional support

Categories of NoSQL

There are several categories of NoSQL databases, each with its own set of characteristics and use cases. The main categories of NoSQL databases are:

1. Document-oriented databases: These databases store data in a document format, such as JSON or XML. Document-oriented databases are flexible and can handle unstructured data, making them well-suited for content management systems, e-commerce websites, and other applications that require storing complex data structures.
2. Key-value stores: Key-value stores are simple databases that store data in a key-value format. They are highly scalable and can handle large volumes of data with low latency. Key-value stores are commonly used in caching, session management, and real-time applications.
3. Column-family stores: Column-family stores store data in columns rather than rows, making them well-suited for data warehousing, data analytics, and other applications that require fast queries on large data sets.
4. Graph databases: Graph databases store data in a graph format, with nodes representing entities and edges representing relationships between them. Graph databases are ideal for

applications that require complex relationship queries, such as social networks and recommendation engines.

5. Object-oriented databases: Object-oriented databases store data in objects, making them well-suited for object-oriented programming languages such as Java and C#. Object-oriented databases are commonly used in complex applications such as CAD/CAM systems and scientific simulations.

Each category of NoSQL database has its own strengths and weaknesses, and the choice of database depends on the specific requirements of the application.

Key-value stores	Column-Oriented	Document based	Graph based
Riak	Cassandra	MongoDB	Neo4J
Redis	Hbase	CouchDB	InfiniteGraph
Membase	HyperTable	RavenDB	AllegroGraph

Document-oriented databases

Document-oriented databases are a type of NoSQL database that store data in a document format, such as JSON or XML. Document-oriented databases are designed to handle unstructured data and provide a flexible data model that can adapt to changing requirements. Here are some of the characteristics of document-oriented databases:

1. Flexible schema: Document-oriented databases do not require a fixed schema, allowing data to be stored in a flexible and dynamic format. This makes it easier to handle unstructured and semi-structured data, such as social media posts, product reviews, and user-generated content.

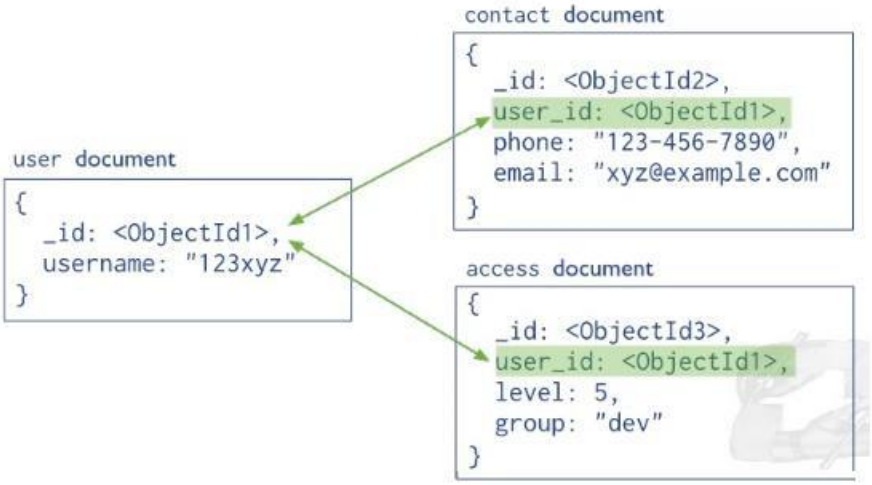
2. Scalable: Document-oriented databases are highly scalable and can handle large volumes of data with ease. They are designed to work with distributed systems, which means that they can be scaled horizontally by adding new nodes to the cluster.

3. Rich query language: Document-oriented databases provide a rich query language that allows for complex queries on nested data structures. This makes it easier to search for specific data within a document.

4. Easy integration with programming languages: Document-oriented databases are designed to work with programming languages such as Java, Python, and Ruby, making it easier to integrate them into modern application architectures.

5. High availability and fault tolerance: Document-oriented databases provide high availability and fault tolerance by replicating data across multiple nodes. This ensures that data is always available, even in the event of a node failure.

Document-oriented databases are commonly used in content management systems, e-commerce websites, and other applications that require storing complex data structures. Some popular document-oriented databases include MongoDB, Couchbase, and Amazon DocumentDB.



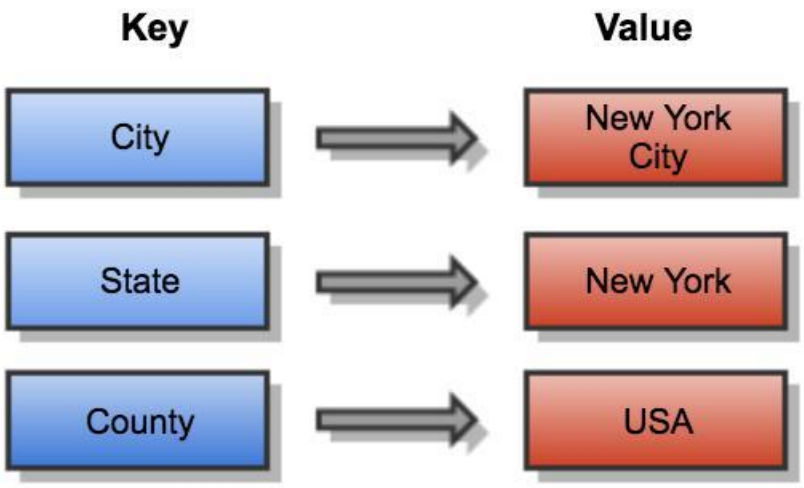
Key-value stores

Key-value stores are a type of NoSQL database that stores data in a simple key-value format. In a key-value store, each data item is stored as a key-value pair, where the key is a unique identifier for the data and the value is the actual data itself. Here are some of the characteristics of key-value stores:

1. Simple data model: Key-value stores have a simple data model that is easy to understand and use. Each key-value pair is stored as a separate item, making it easy to add, update, and delete data.
2. High performance: Key-value stores are designed for high performance and can handle large volumes of data with low latency. They are often used for caching and real-time applications where fast access to data is critical.
3. Scalable: Key-value stores are highly scalable and can be easily scaled horizontally by adding new nodes to the cluster. This makes them ideal for applications that require high scalability and availability.
4. Distributed architecture: Key-value stores are designed to work with distributed systems and can be replicated across multiple nodes for increased availability and fault tolerance.
5. Limited query capabilities: Key-value stores have limited query capabilities compared to other types of databases. They are designed for simple lookup and retrieval operations based on the key, rather than complex queries.

Key-value stores are commonly used for caching, session management, and real-time applications, where fast access to data is critical. Some popular key-value stores include Redis, Riak, and Amazon DynamoDB

Key	Value
2014HW112220	{ Santosh,Sharma,Pilani}
2018HW123123	{Eshwar,Pillai,Hyd}



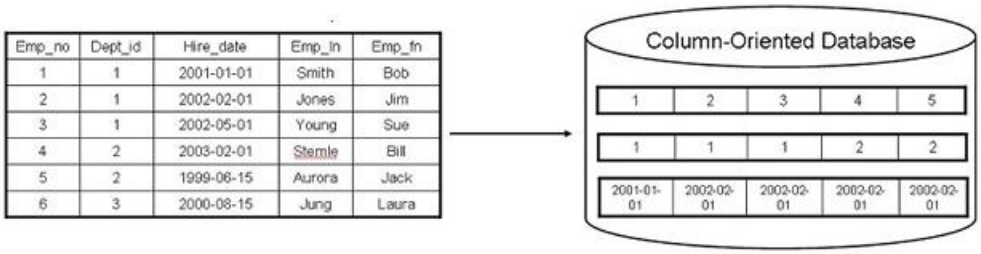
Column based

Column-based databases, also known as column-family databases, are a type of NoSQL database that stores data in columns instead of rows. In a column-based database, data is organized into columns and column families, with each column containing a specific type of data. Here are some of the characteristics of column-based databases:

1. Column-oriented storage: Column-based databases store data in columns instead of rows, making them ideal for applications that require complex analytics and reporting. By storing data in columns, column-based databases can perform fast aggregation, filtering, and sorting operations.
2. Schema flexibility: Column-based databases have a flexible schema that allows for changes to be made to the data model without disrupting the database. This makes them ideal for applications with rapidly changing data requirements.
3. Scalability: Column-based databases are highly scalable and can be easily scaled horizontally by adding new nodes to the cluster. This makes them ideal for applications that require high scalability and availability.
4. High performance: Column-based databases are designed for high performance and can handle large volumes of data with low latency. They are often used for applications that require real-time analytics and reporting.

5. Limited transaction support: Column-based databases have limited transaction support compared to relational databases. They are designed for applications that require fast read and write operations, rather than complex transactions.

Column-based databases are commonly used for analytics, reporting, and real-time applications that require fast access to data. Some popular column-based databases include Apache Cassandra, HBase, and Amazon SimpleDB.

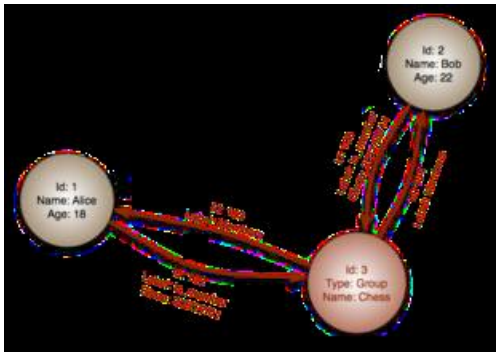


Graph Based

Graph-based databases are a type of NoSQL database that stores data in a graph format, where nodes represent entities and edges represent relationships between entities. In a graph-based database, data is stored in nodes and edges, and each node and edge can have its own properties and attributes. Here are some of the characteristics of graph-based databases:

1. Graph data model: Graph-based databases use a graph data model to represent data, which is ideal for applications that require complex relationships between entities. The graph model is highly flexible and can easily handle complex relationships and data structures.
2. High performance: Graph-based databases are designed for high performance and can handle complex queries and traversals with low latency. They are often used for applications that require real-time analytics and recommendation engines.
3. Scalability: Graph-based databases are highly scalable and can be easily scaled horizontally by adding new nodes to the cluster. This makes them ideal for applications that require high scalability and availability.
4. Schema flexibility: Graph-based databases have a flexible schema that allows for changes to be made to the data model without disrupting the database. This makes them ideal for applications with rapidly changing data requirements.
5. Limited query capabilities: Graph-based databases have limited query capabilities compared to relational databases. They are designed for complex graph traversals and pattern matching, rather than traditional SQL queries.

Graph-based databases are commonly used for recommendation engines, social networks, and fraud detection systems. Some popular graph-based databases include Neo4j, ArangoDB, and OrientDB.



Today’s application requirements

Today's application requirements are diverse and often complex, requiring developers to use a variety of technologies and tools to meet the needs of modern applications. Here are some of the key requirements for modern applications:

1. Scalability: Applications need to be able to scale horizontally and vertically to handle increased traffic and data volumes.
2. Performance: Applications must be designed for high performance and low latency, to ensure a smooth and responsive user experience.
3. Security: Applications need to be secure and protect sensitive data from unauthorized access and attacks.
4. Availability: Applications must be highly available and reliable, with minimal downtime or service interruptions.
5. Interoperability: Applications must be able to integrate with other systems and technologies, to enable seamless data exchange and communication.
6. Agility: Applications need to be agile and adaptable, with the ability to quickly respond to changing business requirements and market conditions.
7. Cloud compatibility: Applications must be able to run on cloud platforms and take advantage of cloud services and technologies.
8. Mobile compatibility: Applications must be able to run on mobile devices and provide a seamless mobile experience.
9. Analytics: Applications need to collect and analyze data to provide insights and drive business decisions.
10. User experience: Applications must provide an engaging and intuitive user experience, with responsive and easy-to-use interfaces

Zero Downtime: This refers to the ability of an application to remain available and responsive to users even during updates or maintenance. Achieving zero downtime requires careful planning and implementation of strategies such as rolling updates, blue-green deployment, and canary release.

Shortened Feedback Cycles: Modern applications require rapid feedback cycles to ensure that they are meeting the needs of users and the business. Continuous integration and delivery (CI/CD) pipelines, automated testing, and real-time monitoring can help developers get fast feedback on the performance and quality of their applications.

Mobile and Multi-Device Support: With the widespread use of mobile devices, applications must be able to run seamlessly on multiple devices and platforms. This requires the use of responsive design, adaptive layouts, and other techniques that ensure that applications are optimized for each device.

Connected Devices: The Internet of Things (IoT) has brought about a proliferation of connected devices that require applications to interact with them in real-time. This requires the use of technologies such as edge computing, message queues, and IoT platforms that enable seamless communication and data exchange between devices and applications.

Data-Driven: Applications must be able to collect and analyze data to provide insights and drive business decisions. This requires the use of technologies such as data warehouses, data lakes, and business intelligence tools that enable organizations to store, process, and analyze large amounts of data.

Architectural Styles Overview

Architectural styles refer to the patterns or frameworks used to design and build software applications. They define the overall structure of an application and provide guidelines for organizing code and data, as well as for communicating between different components.

Here is an overview of some of the most common architectural styles:

1. Monolithic architecture: In a monolithic architecture, all components of an application are integrated into a single codebase and deployed as a single unit. This can make the application easier to develop and deploy, but can also make it harder to scale and maintain.
2. Service-Oriented Architecture (SOA): In an SOA, an application is divided into loosely coupled, independent services that communicate with each other through standardized protocols such as REST or SOAP. This allows for greater flexibility and scalability, but can be more complex to design and maintain.
3. Microservices architecture: Similar to SOA, microservices architecture involves breaking an application into smaller, independent services. However, microservices are typically smaller and more focused on specific tasks, making them easier to develop and deploy.
4. Event-Driven architecture: In an event-driven architecture, components of an application communicate with each other by sending and receiving events, or messages. This can make it easier to create scalable and resilient applications, but can also be more complex to design and maintain.
5. Layered architecture: In a layered architecture, components of an application are organized into layers that communicate with each other in a hierarchical fashion. This can make it easier to design and maintain complex applications, but can also result in tightly coupled components that are difficult to modify or replace.

6. Domain-Driven Design: In a domain-driven design architecture, the focus is on modeling an application around the core business domain, with clear boundaries between different domains. This can help to create more maintainable and scalable applications, but requires a deep understanding of the business domain and may be more difficult to implement.

3 Modern App Architectures

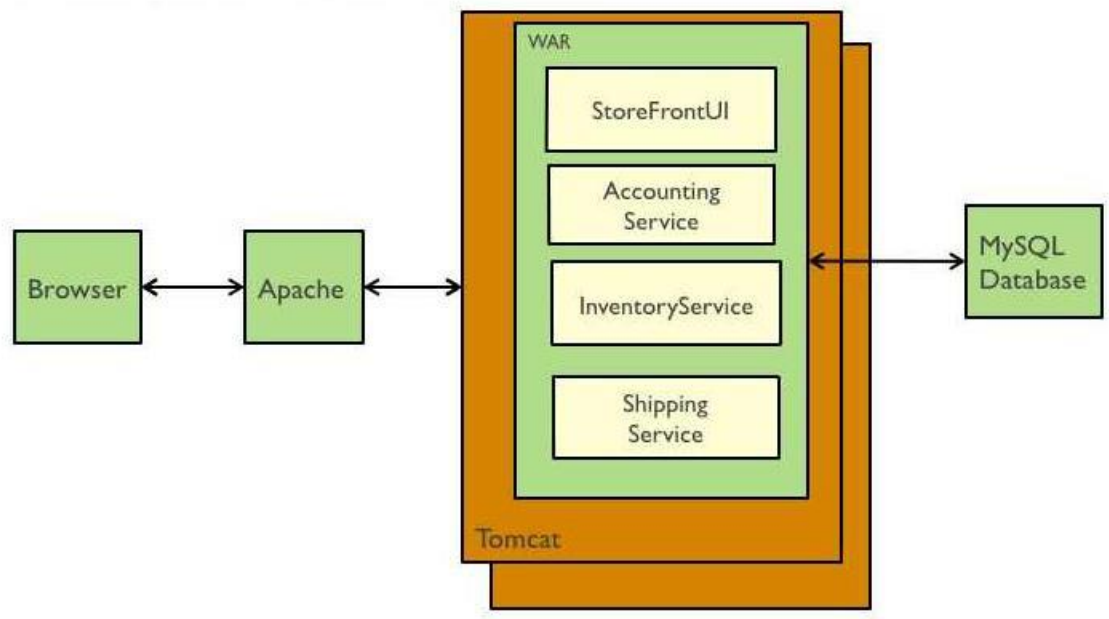
Monolithic Architecture

In a monolithic architecture, an application is developed as a single, self-contained unit with all its components tightly integrated and deployed as a single artifact. This architecture pattern has been in use since the inception of software development and is still commonly used today.

In a monolithic architecture, the entire application is built, tested, and deployed as a single unit. The application logic and user interface are tightly coupled, making it easier to develop and deploy the application as a whole. This also means that there is a single codebase for the entire application, making it easier to manage the code and maintain consistency across the application.

However, monolithic architecture can also have its drawbacks. As the application grows in complexity and size, it can become more difficult to maintain and scale. Additionally, any changes made to one part of the application may affect other parts, making it harder to make changes and deploy updates.

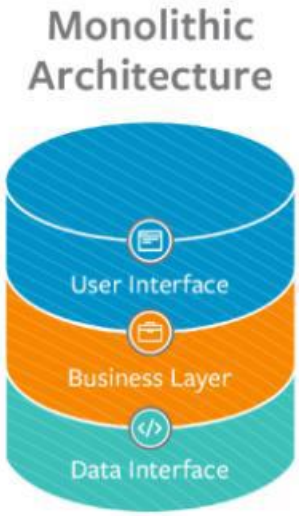
Monolithic architecture can work well for smaller applications or for teams with limited resources, but it may not be the best option for larger, more complex applications. In these cases, other architecture patterns such as microservices or service-oriented architecture may be more appropriate.



Types of Monoliths

there are a few other types that are worth mentioning:

1. Single process monolith: In a single process monolith, the entire application runs in a single process or container. This makes it easier to manage and deploy the application, but also makes it harder to scale or distribute the workload.



2. Distributed monolith: In a distributed monolith, the application is divided into multiple processes or containers that communicate with each other over a network. This allows the application to scale and distribute the workload across multiple machines, but also makes it more complex to manage and deploy.
3. Third party black-box systems: Some monolithic applications may rely heavily on third-party black-box systems, such as commercial software or cloud services. These systems may provide important functionality for the application, but can also introduce dependencies and limitations that can make it harder to modify or scale the application.

Each of these types has its own unique characteristics and trade-offs, and the choice of architecture will depend on the specific requirements and constraints of the application

Modular monolith

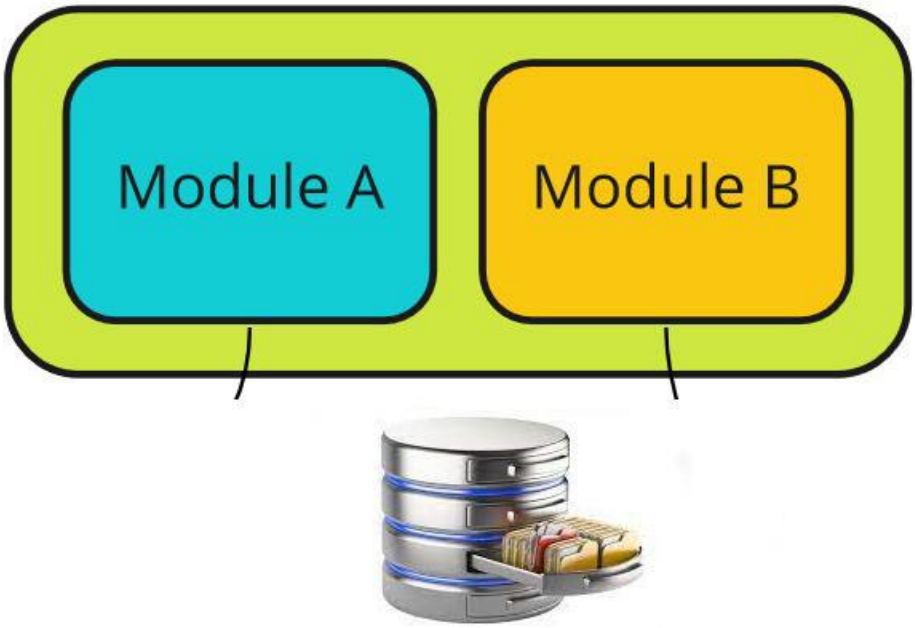
A modular monolith is a type of monolithic architecture that attempts to address some of the limitations of traditional monoliths by organizing the codebase into modular components that can be developed, tested, and deployed independently.

In a modular monolith, the application is still deployed as a single unit, but the codebase is divided into smaller, more manageable modules. Each module is responsible for a specific set of features or functions, and can be developed and tested independently. The modules communicate with each other through well-defined interfaces or APIs, which helps to decouple the modules and reduce dependencies.

One advantage of a modular monolith is that it can make it easier to scale and maintain the application over time. For example, if a specific module needs to be scaled up to handle more traffic, it can be deployed independently of the rest of the application. This can also make it easier to

develop new features or make changes to existing ones, since the impact of any changes can be more easily contained.

However, a modular monolith can still suffer from some of the limitations of traditional monoliths, such as increased complexity and slower development cycles as the codebase grows. Additionally, since the modules are still deployed as a single unit, there can still be limitations to scalability and fault tolerance.

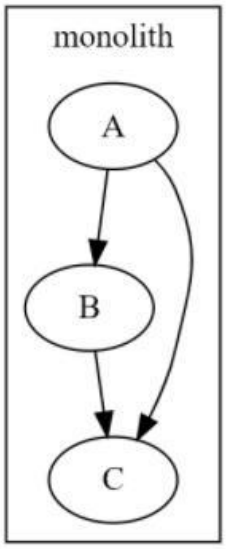


Distributed monolith

A distributed monolith is a type of monolithic architecture that attempts to address scalability and availability issues by distributing the monolith across multiple servers or nodes. In a distributed monolith, the application is still deployed as a single unit, but different parts of the monolith are deployed on different nodes.

Each node typically runs its own instance of the monolith, but the nodes are interconnected and communicate with each other through well-defined interfaces or APIs. This allows the application to scale horizontally by adding more nodes to handle increased traffic, and can also provide better fault tolerance since failures in one node can be isolated and do not necessarily affect the entire application.

However, a distributed monolith can also introduce new complexities, such as managing the inter-node communication and ensuring consistency across different nodes. Additionally, since the monolith is still deployed as a single unit, changes to the codebase can still have a wide-ranging impact and can require coordination across all nodes.



Third party black-box systems

In monolithic architecture, third-party black-box systems refer to external services or systems that the application relies on for certain functionalities, but the code for those functionalities is not part of the monolith codebase. These systems are usually accessed through APIs or other interfaces and are treated as a "black box" by the monolith, since the inner workings of the system are not visible or modifiable by the monolith.

Third-party black-box systems can provide certain advantages, such as allowing the monolith to delegate certain functionalities to specialized services that can handle them more efficiently or securely. However, they can also introduce certain complexities, such as requiring the monolith to handle different communication protocols or manage multiple authentication and authorization mechanisms. Additionally, since the code for these systems is not under the control of the monolith development team, changes or updates to these systems can have unforeseen impacts on the monolith and may require additional testing or integration efforts.

Micorservices

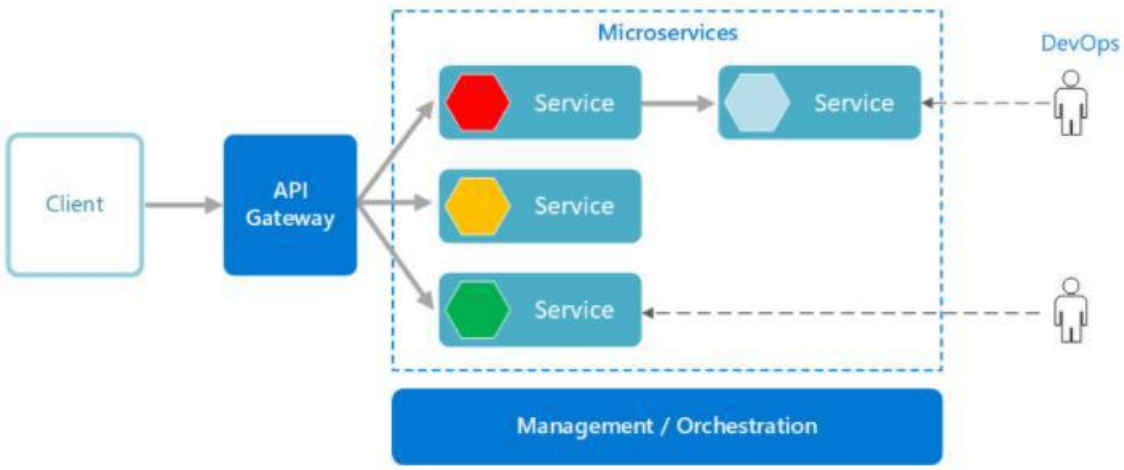
Microservices is an architectural style that structures an application as a collection of loosely coupled services, each running in its own process and communicating with lightweight mechanisms such as HTTP APIs. Each microservice is designed to handle a specific business capability and can be developed, deployed, and scaled independently of the rest of the application.

Microservices architecture aims to solve the problems of traditional monolithic architecture by breaking down the application into smaller, independently deployable services that can be developed and managed separately. This approach offers several advantages, including:

1. Scalability: Each microservice can be scaled independently, allowing the application to handle a larger number of requests as needed.
2. Resilience: If a single microservice fails, the rest of the application can continue to function normally, reducing the risk of a catastrophic failure.

3. Flexibility: Microservices can be developed and deployed independently, making it easier to iterate on individual services and add new features to the application.
4. Technology diversity: Different microservices can be built with different technologies, allowing teams to use the best tool for the job.

However, microservices also introduce new challenges, including increased complexity in deployment, testing, and monitoring, as well as the need for more sophisticated communication between services.



Microservices architecture style

Microservices architecture is an architectural style that structures an application as a collection of small, independently deployable services. Each service is designed to handle a specific business capability and can be developed, deployed, and scaled independently of the rest of the application.

The key characteristics of microservices architecture include:

1. Service-oriented: Each microservice is designed to provide a specific service or business capability. Services communicate with each other through APIs, typically using lightweight protocols such as HTTP.
2. Decentralized: Microservices are deployed independently of each other, and each service is responsible for its own data management and business logic. There is no central point of control or coordination.
3. Independent scalability: Each microservice can be scaled independently of the rest of the application, allowing the system to handle high loads efficiently.
4. Resilience: In a microservices architecture, a failure in one service does not necessarily cause the entire application to fail. Services are designed to be fault-tolerant, and the system can automatically recover from failures.
5. Technology diversity: Different microservices can be developed using different technologies and programming languages, allowing teams to use the best tool for the job.
6. Continuous delivery: Microservices architecture supports continuous delivery, allowing teams to release new features and updates quickly and frequently.

Microservices architecture has become increasingly popular in recent years, as organizations have sought to break down monolithic applications into smaller, more manageable components. However, microservices architecture is not a silver bullet, and there are trade-offs and challenges associated with this approach. For example, the increased complexity of distributed systems can make it more difficult to test, monitor, and troubleshoot applications.

Microservices – Benefits & Challenges

Microservices architecture has several benefits and challenges:

Benefits of Microservices:

1. Scalability: Microservices architecture enables easy scalability of individual services that need to handle varying loads.
2. Flexibility: Each microservice can be built and deployed independently, allowing developers to use different technologies and programming languages as needed.
3. Resilience: With a distributed architecture, if one service fails, the others can still function independently.
4. Maintainability: Microservices make it easier to maintain large applications by breaking them down into smaller, more manageable pieces.
5. Faster deployment: Microservices allow for independent deployment of each service, reducing the time it takes to deploy the entire application.

Challenges of Microservices:

1. Increased complexity: Microservices architecture introduces additional complexity to the application due to the need to manage multiple services.
2. Distributed system management: With microservices, developers must manage a distributed system, which can be challenging.
3. Data management: Microservices architecture can lead to data management challenges because each service manages its own data, which can lead to data inconsistency.
4. Testing complexity: Testing becomes more complex with microservices architecture, as each service needs to be tested individually and as part of the overall system.
5. Network latency: With microservices architecture, communication between services can lead to network latency issues, which can impact performance.

Microservice Application Platforms

Microservice application platforms are software frameworks that provide developers with the tools to build, deploy, and manage microservices-based applications. Some of the popular microservice application platforms include:

1. Kubernetes: Kubernetes is an open-source container orchestration platform that provides a scalable and highly available environment for deploying, managing, and scaling microservices.

2. Docker Swarm: Docker Swarm is a container orchestration platform that provides a simple and easy-to-use way to deploy and manage microservices.
3. Apache Mesos: Apache Mesos is a distributed systems kernel that provides resource management and scheduling for large-scale distributed systems.
4. Amazon Web Services (AWS) Elastic Beanstalk: AWS Elastic Beanstalk is a fully managed platform for deploying and running web applications and services.
5. Google App Engine: Google App Engine is a platform for developing and hosting web applications in Google-managed data centers.
6. Microsoft Azure: Microsoft Azure is a cloud computing platform that provides a wide range of services for building, deploying, and managing microservices-based applications.
7. Red Hat OpenShift: Red Hat OpenShift is an open-source container application platform that provides a complete development and deployment environment for containerized applications.

These platforms provide developers with the necessary tools to build, deploy, and manage microservices-based applications efficiently and effectively. They provide features such as auto-scaling, load balancing, fault tolerance, and centralized management of microservices

Distributed Systems and Fallacies

Distributed systems are computer systems that are composed of multiple interconnected components that work together to achieve a common goal. These components can be geographically distributed across different locations and can communicate with each other over a network. The development of distributed systems has revolutionized the way we build and deploy large-scale applications, allowing us to achieve high levels of scalability, availability, and performance.

However, building and operating distributed systems is not without its challenges. One of the most significant challenges is the fallacies of distributed computing. The fallacies of distributed computing are a set of assumptions that developers make when building distributed systems that are not necessarily true. These fallacies include:

1. The network is reliable: Developers often assume that the network connecting the different components of a distributed system is reliable and will always be available. However, the reality is that networks can be unreliable and can fail at any time.
2. Latency is zero: Developers often assume that there is no delay or latency when communicating between different components of a distributed system. However, the reality is that there is always some latency, and this can have a significant impact on the performance of a distributed system.
3. Bandwidth is infinite: Developers often assume that there is unlimited bandwidth available to communicate between different components of a distributed system. However, the reality is that bandwidth is limited, and it can become a bottleneck if not managed properly.
4. The topology doesn't change: Developers often assume that the topology of a distributed system remains static and doesn't change over time. However, the reality is that components can be added or removed from the system, and the topology can change dynamically.

5. There is one administrator: Developers often assume that there is a single administrator who is responsible for managing and monitoring the distributed system. However, the reality is that distributed systems can have multiple administrators, and it can be challenging to coordinate their efforts.
6. Transport cost is zero: Developers often assume that there is no cost associated with transporting data between different components of a distributed system. However, the reality is that there is always some cost associated with data transfer, and this can become a significant factor in the overall performance and cost of a distributed system.
7. The system is homogeneous: Developers often assume that all components of a distributed system are identical and behave in the same way. However, the reality is that different components can have different characteristics, and it can be challenging to manage and coordinate their interactions.

Understanding and addressing these fallacies is crucial when building and operating distributed systems. Developers need to be aware of these assumptions and design their systems to be resilient, flexible, and scalable in the face of these challenges

Distributed Systems and Fallacies

Distributed systems are computer systems that are composed of multiple interconnected components that work together to achieve a common goal. These components can be geographically distributed across different locations and can communicate with each other over a network. The development of distributed systems has revolutionized the way we build and deploy large-scale applications, allowing us to achieve high levels of scalability, availability, and performance.

However, building and operating distributed systems is not without its challenges. One of the most significant challenges is the fallacies of distributed computing. The fallacies of distributed computing are a set of assumptions that developers make when building distributed systems that are not necessarily true. These fallacies include:

1. The network is reliable: Developers often assume that the network connecting the different components of a distributed system is reliable and will always be available. However, the reality is that networks can be unreliable and can fail at any time.
2. Latency is zero: Developers often assume that there is no delay or latency when communicating between different components of a distributed system. However, the reality is that there is always some latency, and this can have a significant impact on the performance of a distributed system.
3. Bandwidth is infinite: Developers often assume that there is unlimited bandwidth available to communicate between different components of a distributed system. However, the reality is that bandwidth is limited, and it can become a bottleneck if not managed properly.
4. The topology doesn't change: Developers often assume that the topology of a distributed system remains static and doesn't change over time. However, the reality is that

components can be added or removed from the system, and the topology can change dynamically.

5. There is one administrator: Developers often assume that there is a single administrator who is responsible for managing and monitoring the distributed system. However, the reality is that distributed systems can have multiple administrators, and it can be challenging to coordinate their efforts.
6. Transport cost is zero: Developers often assume that there is no cost associated with transporting data between different components of a distributed system. However, the reality is that there is always some cost associated with data transfer, and this can become a significant factor in the overall performance and cost of a distributed system.
7. The system is homogeneous: Developers often assume that all components of a distributed system are identical and behave in the same way. However, the reality is that different components can have different characteristics, and it can be challenging to manage and coordinate their interactions.

Understanding and addressing these fallacies is crucial when building and operating distributed systems. Developers need to be aware of these assumptions and design their systems to be resilient, flexible, and scalable in the face of these challenges.

Fallacies of Distributed System

The fallacies of distributed computing are a set of misconceptions or incorrect assumptions that developers may make when designing and implementing distributed systems. They were first described by Peter Deutsch in 1994 and later expanded upon by James Gosling. The fallacies include:

1. The network is reliable: Developers often assume that the network will always be available and perform well, but in reality, network failures and latency can occur.
2. Latency is zero: Developers often assume that communication between distributed components happens instantaneously, but in reality, there is always some latency, which can vary based on factors like network distance and congestion.
3. Bandwidth is infinite: Developers often assume that there are no limits to the amount of data that can be transmitted across the network, but in reality, there are bandwidth constraints that can affect system performance.
4. The network is secure: Developers often assume that the network is secure and that their applications are immune to attacks, but in reality, there are various threats to network security, such as eavesdropping and denial-of-service attacks.
5. Topology doesn't change: Developers often assume that the topology of the network will remain constant, but in reality, nodes can be added or removed, and the network topology can change dynamically.
6. There is one administrator: Developers often assume that there is a single administrator who is responsible for managing the network, but in reality, there can be multiple administrators with different levels of control and access.

7. Transport cost is zero: Developers often assume that there are no costs associated with transporting data across the network, but in reality, there can be costs such as network usage fees or data transfer fees.
8. The network is homogeneous: Developers often assume that all components in the network are similar and can communicate with each other easily, but in reality, there can be differences in hardware, software, and protocols that can make communication more complex.

These fallacies highlight the importance of designing and testing distributed systems carefully and accounting for the inherent complexities and limitations of distributed computing.

Fallacy	Solutions
The network is reliable	Automatic Retries, Message Queues
Latency is zero	Caching Strategy, Bulk Requests, Deploy in AZs near client
Bandwidth is infinite	Throttling Policy, Small payloads with Microservices
The network is secure	Network Firewalls, Encryption, Certificates, Authentication
Topology does not change	No hardcoding IP, Service Discovery Tools
There is one administrator	DevOps Culture eliminates Bus Factor
Transport cost is zero	Standardized protocols like JSON, Cost Calculation
The network is homogenous	Circuit Breaker, Retry and Timeout Design Pattern

12 Factor App

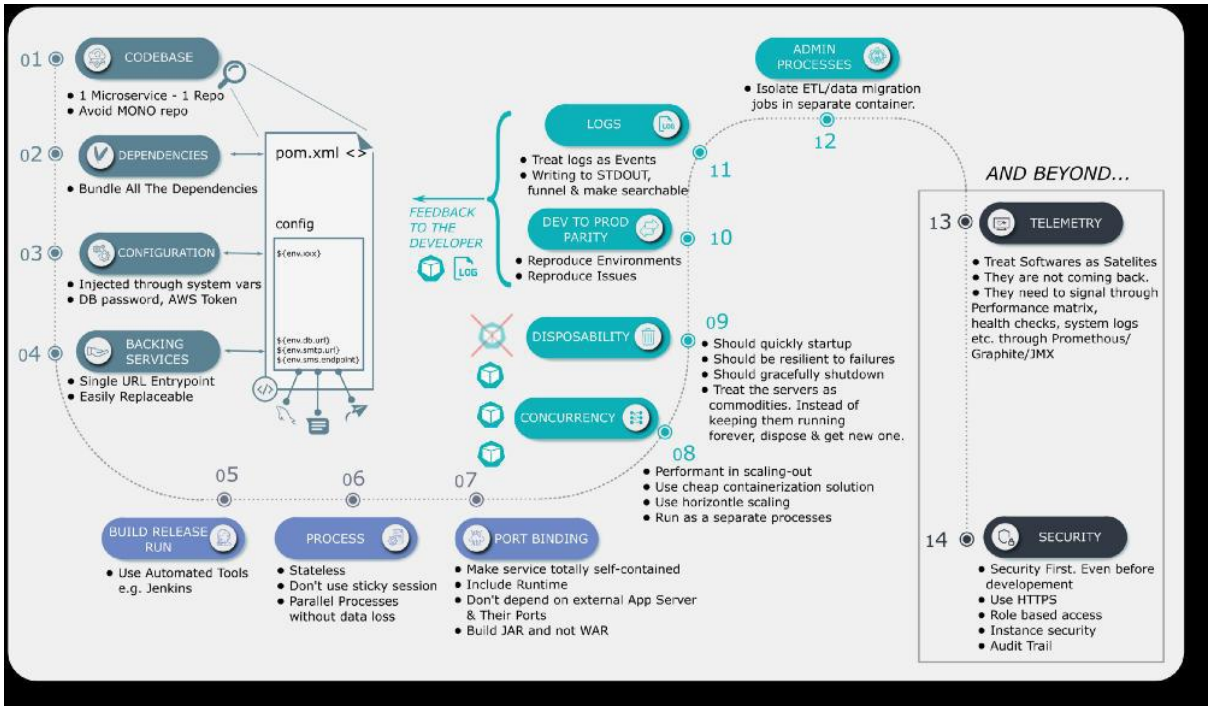
The 12 Factor App is a set of principles for building software applications that are easy to maintain and scale in a modern cloud environment. The principles were developed by Heroku, a cloud-based platform-as-a-service (PaaS) provider, and are widely used as a guideline for building cloud-native applications.

Here are the 12 factors that define a 12 Factor App:

1. Codebase: A single codebase is used for each application and is version-controlled using a modern version control system such as Git.
2. Dependencies: Dependencies are explicitly declared and isolated, making it easy to manage application dependencies.
3. Config: Configuration is stored in the environment and not in the code, allowing for easy deployment and scaling.
4. Backing services: All services, including databases, are treated as attached resources and accessed over the network.

5. Build, release, run: A strict separation between building the application, releasing it, and running it in production is maintained.
6. Processes: Applications are executed as stateless processes, which can be scaled horizontally.
7. Port binding: Services export themselves via a port binding, which makes them available to the network.
8. Concurrency: Processes are designed to be highly concurrent and can be scaled horizontally.
9. Disposability: Processes are designed to start up and shut down quickly, allowing for easy scaling and high availability.
10. Dev/prod parity: Development, staging, and production environments should be as similar as possible.
11. Logs: Applications should generate logs as event streams, which can be easily aggregated and analyzed.
12. Admin processes: Administrative tasks are treated as one-off processes, separate from the main application processes.

By following these principles, developers can build cloud-native applications that are easy to deploy, manage, and scale.



Cloud Native Architecture

Cloud Native Architecture is a way of building applications that utilize cloud computing capabilities, such as scalability, availability, and resilience. It refers to a set of practices, technologies, and tools that are used to develop and deploy applications that can run in the cloud environment.

The main principles of Cloud Native Architecture are:

1. Microservices: Applications are designed as a collection of loosely-coupled microservices that can be developed, deployed, and scaled independently.
2. Containers: Applications are packaged into containers, which allow them to be easily deployed and run consistently across different environments.
3. Orchestration: Containers are managed and orchestrated using platforms like Kubernetes, which automates deployment, scaling, and management of containerized applications.
4. DevOps: The development and operations teams work collaboratively using DevOps practices and tools to ensure fast and reliable delivery of applications.
5. Infrastructure as Code: The infrastructure needed to run the application is managed using code, making it easier to deploy and manage across different environments.

The benefits of Cloud Native Architecture are:

1. Scalability: Applications can easily scale up or down to meet changing demands, without requiring manual intervention.
2. Availability: Applications are designed to be highly available, with built-in failover and self-healing capabilities.
3. Agility: Applications can be developed, deployed, and updated quickly and easily, allowing organizations to respond to changing business needs faster.
4. Cost-effectiveness: Cloud Native Architecture allows organizations to optimize their resource usage, leading to cost savings.

However, there are also some challenges associated with Cloud Native Architecture, including:

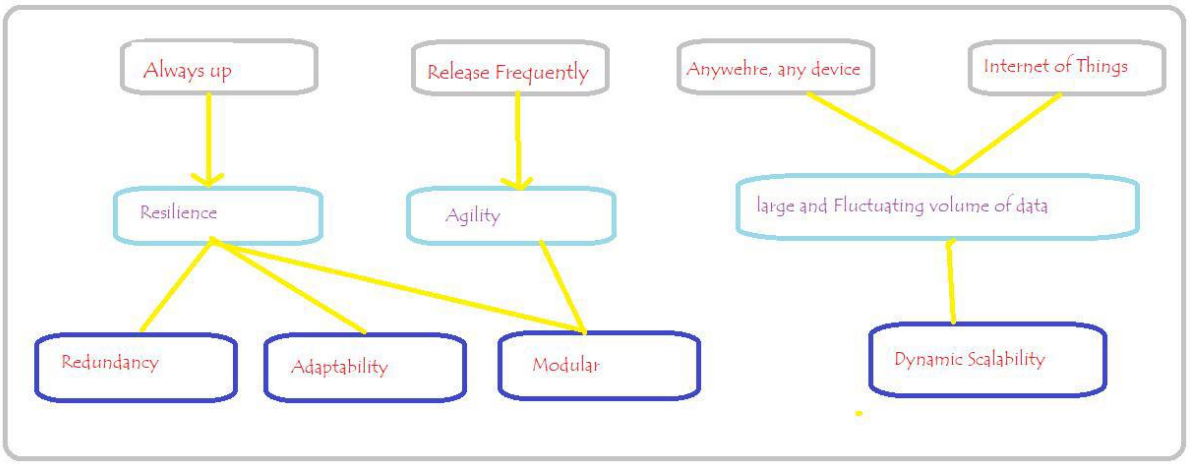
1. Complexity: The use of microservices and containers can increase the complexity of the application, requiring more management and monitoring.
2. Skill set: Developing and managing Cloud Native applications requires specialized skills, which may not be readily available in an organization.
3. Security: As applications are distributed across different environments, ensuring the security of the entire system becomes more challenging.
4. Vendor lock-in: Using Cloud Native technologies and platforms can result in vendor lock-in, making it difficult to switch to a different provider

What characterizes cloud-native software?

Cloud-native software is designed and built specifically to run on cloud infrastructure. It is characterized by the following:

1. Microservices architecture: Cloud-native applications are typically designed as a collection of loosely coupled microservices, which can be independently deployed and scaled.
2. Containers: Containers are used to package and deploy individual microservices, enabling them to be deployed and run consistently across different environments.

3. DevOps culture: Cloud-native development is typically characterized by a DevOps culture, with a focus on automation, continuous integration and delivery, and collaboration between development and operations teams.
4. Scalability and elasticity: Cloud-native applications are designed to scale and handle high levels of traffic, by dynamically allocating and deallocating resources as needed.
5. Resilience: Cloud-native applications are designed to be resilient to failures, by using techniques such as redundancy, load balancing, and fault tolerance.
6. Cloud infrastructure: Cloud-native applications are designed to run on cloud infrastructure, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure, taking advantage of the platform's built-in services and features.



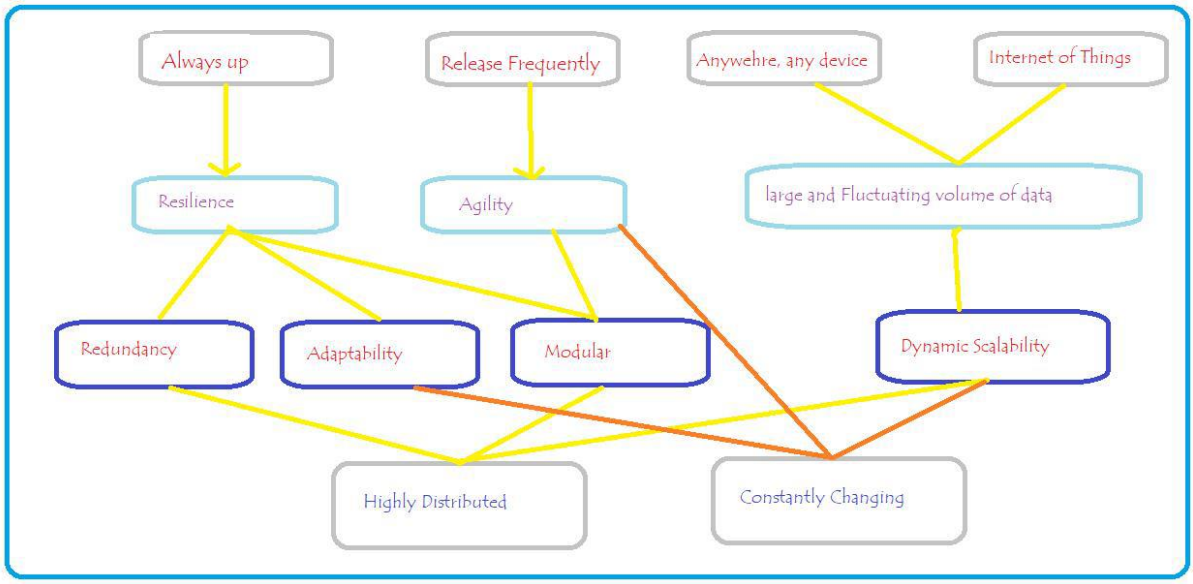
What defines “cloud native” software?

Cloud-native software refers to applications that are designed and optimized to run in cloud environments such as public, private, and hybrid clouds. The key defining characteristics of cloud-native software are:

1. Containerization: Cloud-native applications are typically packaged and deployed as containers using containerization platforms like Docker.
2. Microservices: Cloud-native applications are designed as a collection of loosely-coupled microservices that can be developed, deployed, and scaled independently.
3. DevOps: Cloud-native software is developed and managed using a DevOps approach that emphasizes automation, continuous delivery, and agile development practices.
4. Scalability: Cloud-native applications are designed to scale easily and efficiently, both horizontally and vertically, in response to changes in demand.
5. Resiliency: Cloud-native applications are built with resiliency in mind, with features like automatic failover, self-healing, and high availability.
6. Infrastructure as code: Cloud-native applications are deployed and managed using infrastructure as code tools that allow for version-controlled, repeatable deployments.

7. Cloud-native data services: Cloud-native software often makes use of cloud-native data services like object storage, managed databases, and data streaming services.

Overall, the goal of cloud-native software is to enable organizations to build and deploy applications more quickly, efficiently, and reliably in the cloud.



Two important Characteristics

Two important characteristics of cloud-native software are:

1. Containerization: Cloud-native software is designed to run inside containers, which are lightweight and portable units of software that can be run consistently across different environments. Containers make it easy to package and deploy software, as well as to scale applications up or down as needed. They also enable the efficient use of resources, since multiple containers can run on the same host.
2. Microservices: Cloud-native software is often built using a microservices architecture, which involves breaking an application down into smaller, independent services that can be developed, deployed, and scaled independently. This makes it easier to maintain and update applications, as well as to scale them horizontally to meet demand. Microservices also promote flexibility and agility, since individual services can be replaced or updated without affecting the entire application.

Virtual Machines

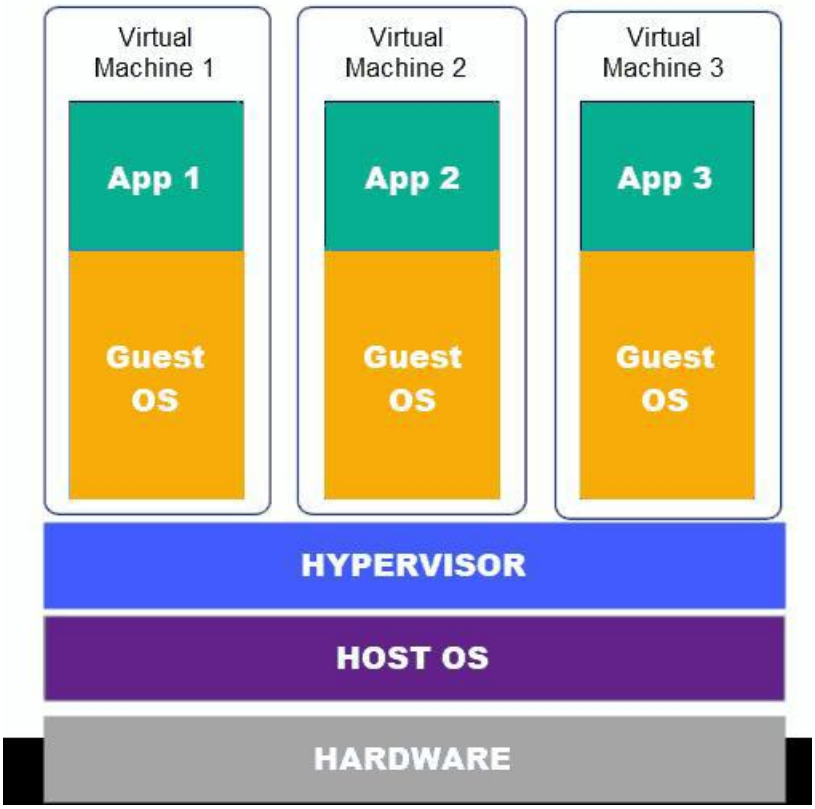
A virtual machine (VM) is a software program that simulates a computer system, enabling the execution of an operating system and applications. It is an abstraction of a physical computer, allowing multiple virtual machines to be run on a single physical machine, each with its own operating system and applications.

Virtual machines offer several benefits, including:

1. Isolation: Virtual machines are isolated from one another, so if one virtual machine crashes or is compromised, it won't affect the others.
2. Resource allocation: Virtual machines allow for flexible allocation of resources, including CPU, memory, and storage.
3. Portability: Virtual machines can be moved easily between physical machines, making it easy to migrate applications or distribute workloads.
4. Testing: Virtual machines provide a safe environment for testing software and applications, without the risk of affecting production systems.

However, virtual machines also have some drawbacks, including:

1. Overhead: Running multiple virtual machines on a single physical machine can result in overhead, which can affect performance.
2. Resource limitations: The resources available to a virtual machine are limited by the resources available on the physical machine.
3. Complexity: Managing and configuring multiple virtual machines can be complex, especially as the number of virtual machines grows.



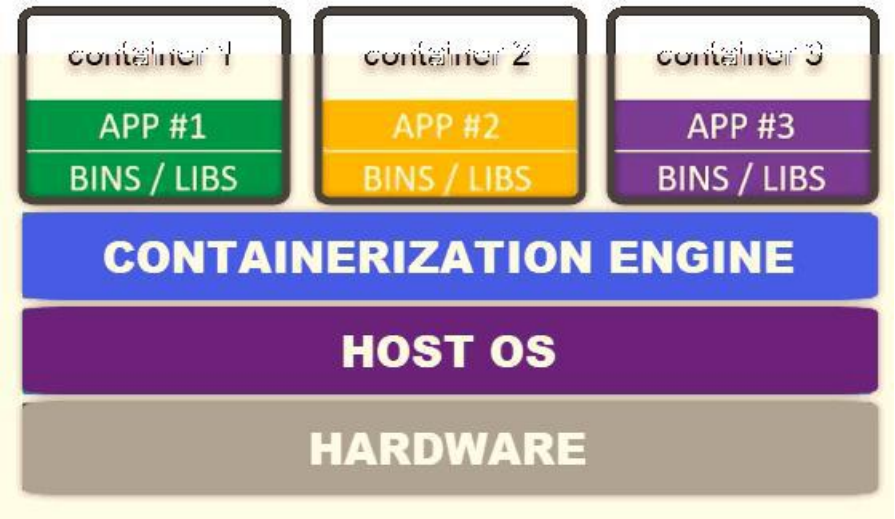
Containers

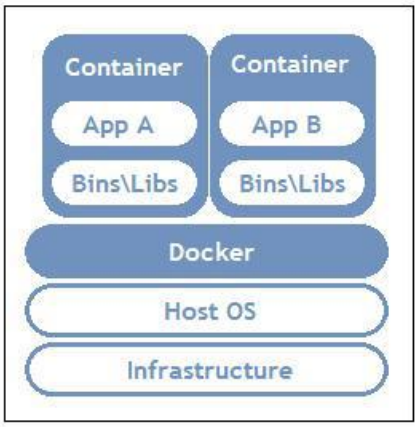
Containers are a lightweight, portable and self-contained way to package and run software applications, which allow for easier deployment, scaling and management of applications. Containers provide a way to package all the necessary components of an application, including code, libraries, and system tools, into a single executable package that can run consistently across

different environments, whether it's a developer's laptop, a testing server, or a production environment.

Containers are isolated from each other and from the host operating system, using a technology called containerization. This means that each container has its own file system, network stack, and process space, and can run its own isolated instances of applications and services. Containers are also highly scalable, as they can be easily replicated and distributed across a cluster of servers.

Containers are often used in cloud-native architectures and DevOps workflows, as they can help organizations achieve faster time-to-market, greater agility, and better scalability and reliability of their applications. Popular container platforms include Docker, Kubernetes, and OpenShift.





Benefits of containers

There are several benefits of using containers in software development and deployment:

1. Portability: Containers provide a consistent environment for applications to run, making it easier to move applications between different environments, such as development, testing, and production.
2. Scalability: Containers can be easily scaled horizontally, meaning that more instances of the same container can be added to handle increased traffic or demand.

3. Efficiency: Containers have a smaller footprint than virtual machines, which makes them faster to start up and easier to manage. They also use fewer resources and can be run on the same machine as other containers.
4. Isolation: Containers provide a high level of isolation between applications and their dependencies, making it less likely that one application will interfere with another.
5. Consistency: Containers can help ensure that applications are running in a consistent environment, with the same versions of libraries and dependencies.
6. Security: Containers can be designed with security in mind, making it easier to isolate and secure applications and their dependencies.

Overall, containers can help make software development and deployment more efficient, consistent, and secure.

Container use cases

Containers have many use cases in the modern IT landscape. Some of the most common use cases include:

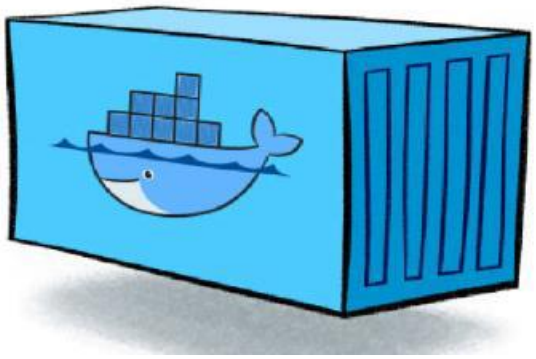
1. Application deployment: Containers are widely used to deploy applications in a consistent and repeatable manner, regardless of the underlying infrastructure.
2. Microservices: Containers are particularly well-suited for deploying microservices, which are small, independently deployable services that work together to form a larger application.
3. DevOps: Containers are a key tool in many DevOps workflows, allowing for faster and more reliable software delivery.
4. Cloud computing: Containers are often used to deploy and manage cloud-native applications in public, private, or hybrid cloud environments.
5. Testing and QA: Containers make it easy to test applications in isolated environments that closely mimic production environments, helping to ensure that software works as expected before it is released.
6. Disaster recovery: Containers can be used to quickly and easily spin up backup environments in the event of a disaster, reducing downtime and ensuring business continuity.

Containers in Cloud native

Containers play a critical role in cloud-native architecture as they provide a lightweight and portable solution for deploying and running microservices. In a cloud-native environment, containers are used to encapsulate individual microservices and their dependencies, allowing them to be easily deployed, scaled, and managed.

Some of the key use cases for containers in a cloud-native environment include:

1. Microservices: Containers are the ideal deployment model for microservices because they allow each service to be packaged and deployed independently. This provides greater flexibility, scalability, and resilience to the application.
2. DevOps: Containers streamline the development and deployment process by providing a consistent environment for development, testing, and production. This allows developers to focus on writing code rather than managing infrastructure.
3. Scalability: Containers can be easily scaled up or down based on demand. This makes them ideal for applications that experience fluctuations in traffic and usage.
4. Hybrid Cloud: Containers are highly portable and can be deployed across different cloud platforms and infrastructure. This makes it easier for organizations to adopt a hybrid cloud strategy and take advantage of the benefits of multiple cloud providers.
5. Infrastructure as Code: Containers can be managed and configured using code, allowing for greater automation and standardization of infrastructure management. This makes it easier to deploy and manage applications in a consistent and repeatable manner.



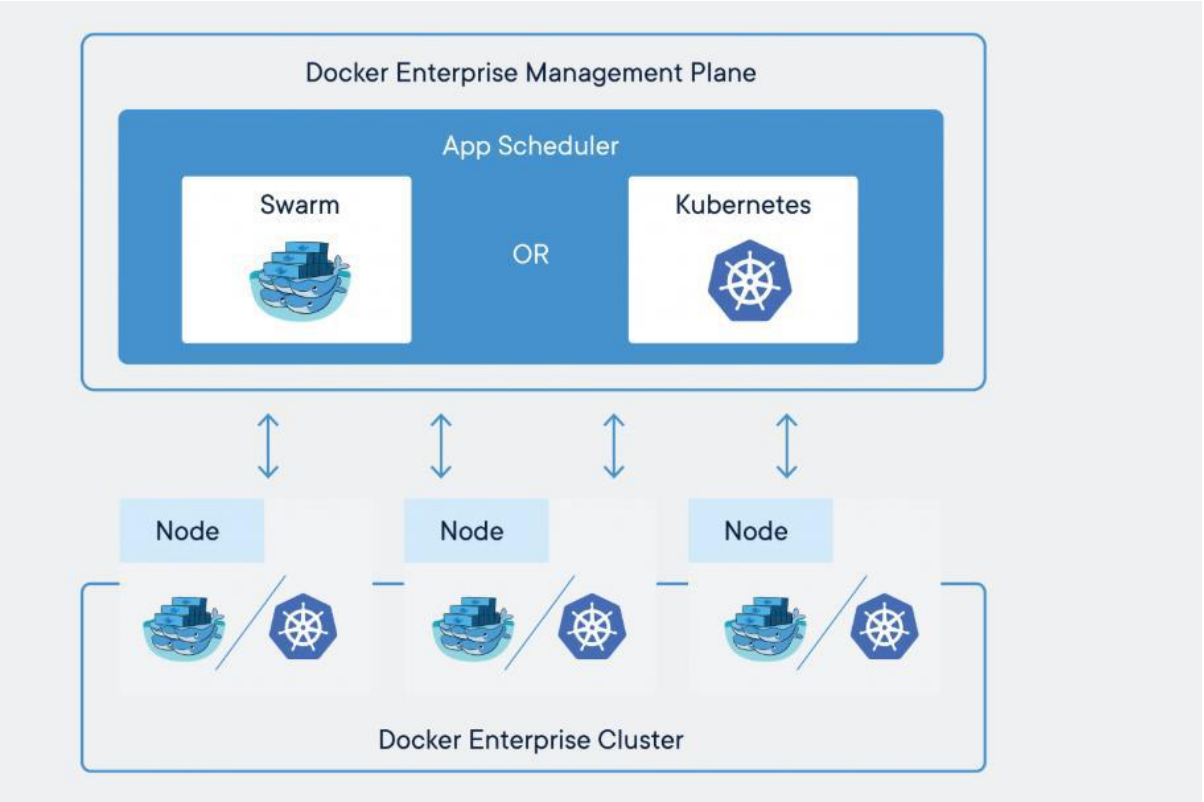
Containers in Production

Containers have become an essential tool for deploying and managing applications in production. Here are some of the ways containers are used in production environments:

1. Application Deployment: Containers are used to package applications and their dependencies into a portable format, which makes it easy to deploy and run the application in different environments without the need for complex configuration.
2. Scaling: Containers can be easily scaled up or down based on the demand for the application. This is done by creating multiple instances of the container, and distributing the load across these instances using a load balancer.
3. High Availability: Containers can be used to achieve high availability of applications by running multiple instances of the container across different nodes in a cluster. If one node fails, the container can be automatically moved to another node to maintain service availability.
4. DevOps: Containers are often used in DevOps workflows to simplify the deployment and testing of applications. By packaging the application and its dependencies into a container, developers can easily test the application in a production-like environment, and quickly deploy new versions of the application.

5. Cloud Migration: Containers can be used to migrate applications from on-premise infrastructure to the cloud. By packaging the application and its dependencies into a container, the application can be easily moved to the cloud, and run on different cloud platforms without the need for complex configuration.

Overall, containers have revolutionized the way applications are deployed and managed in production environments, by providing a flexible and portable platform for running applications.



Container Orchestration

Container orchestration is the process of automating the deployment, scaling, and management of containerized applications. It involves managing the lifecycles of containers, monitoring their health, and ensuring that they are always available and running as expected. Container orchestration also involves managing the interactions between containers, as well as with other components in the infrastructure, such as load balancers, databases, and other services.

Container orchestration is necessary when managing large, complex, and dynamic containerized applications. It provides the ability to automate many tasks, such as scaling, load balancing, failover, and self-healing, which helps to reduce the workload on developers and operations teams.

Some popular container orchestration platforms include Kubernetes, Docker Swarm, and Apache Mesos. These platforms provide powerful tools for automating the deployment and management of containerized applications, and they are widely used in cloud-native environments.

Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) is a cloud computing service model where computing resources, such as virtual machines (VMs), storage, and networking, are provided over the internet. With IaaS, customers can rent computing infrastructure from a cloud provider, and use it to run their applications and services.

IaaS is a popular choice for businesses because it provides the flexibility and scalability needed to handle varying workloads and changing business requirements. Instead of building and maintaining their own on-premises infrastructure, businesses can leverage IaaS to quickly provision and deploy resources on demand, paying only for what they use.

Some examples of IaaS providers include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP).

Platform as a Service (PaaS)

Platform as a Service (PaaS) is a cloud computing model in which a third-party provider delivers a computing platform that allows customers to develop, run, and manage their applications without having to worry about the underlying infrastructure. In PaaS, the provider typically manages the operating system, servers, storage, and networking, while the customer focuses on building and deploying their applications.

PaaS can be a useful option for businesses that want to develop and deploy their applications quickly and efficiently, without the hassle of managing the underlying infrastructure. It can also provide cost savings by reducing the need for in-house infrastructure and personnel. PaaS is commonly used for web application development, mobile application development, and API development. Some popular PaaS providers include Google App Engine, Microsoft Azure, and Heroku.

Software as a Service (SaaS)

Software as a Service (SaaS) is a cloud computing model that delivers software applications over the internet to end-users. In this model, the provider hosts the software, manages the infrastructure, security, and upgrades, and users can access the application through a web browser or mobile app.

SaaS applications can be customized according to user needs, and they are available on a subscription-based model, where users pay for the software on a monthly or yearly basis. This eliminates the need for companies to invest in expensive hardware, software, and IT infrastructure to support the application, reducing the total cost of ownership.

SaaS is a popular model for delivering business applications such as customer relationship management (CRM), human resource management (HRM), accounting, and project management software, among others. SaaS is also commonly used for email, collaboration, and communication tools, such as Google Workspace, Microsoft 365, and Zoom, among others.

Container as a Service

Container as a Service (CaaS) is a type of cloud computing service that enables users to deploy, manage, and scale containers without the need for complex infrastructure management. In a CaaS model, the cloud provider manages the underlying infrastructure and provides the necessary resources for running containers, such as computing, networking, and storage.

CaaS enables developers and IT teams to easily deploy and manage containerized applications, allowing them to focus on the application logic rather than the infrastructure. It also allows for easy scaling and load balancing of containerized applications, and provides a consistent environment for running applications across different deployment environments.

Some popular CaaS platforms include Amazon Elastic Container Service (ECS), Google Kubernetes Engine (GKE), and Microsoft Azure Container Instances (ACI).

Functions as a Service (FaaS)

Functions as a Service (FaaS) is a type of cloud computing service where developers can execute code in response to events without the need for server management. FaaS is also known as serverless computing.

In FaaS, the cloud provider manages the underlying infrastructure, including the server, operating system, and runtime environment. Developers write small pieces of code, known as functions, that perform specific tasks or respond to specific events, such as an HTTP request. When an event occurs, the cloud provider runs the corresponding function, executes the code, and returns the result.

FaaS has several advantages over traditional server-based computing, including reduced operational costs, automatic scaling, and faster development cycles. FaaS also enables developers to focus on writing code without worrying about infrastructure management, and to pay only for the computing resources used during function execution.

Evaluation of Cloud Services

When evaluating cloud services, there are several factors to consider, including:

Cost: This includes both the upfront costs and ongoing costs of using the cloud service. Some cloud services may have hidden costs, such as charges for data transfers or storage beyond a certain amount.

Performance: Cloud services should be evaluated for their ability to handle the workload of the application. This includes factors such as response time, scalability, and reliability.

Security: Cloud services should be evaluated for their security features and the measures they take to protect data and applications from breaches and attacks.

Availability: Cloud services should be evaluated for their uptime and availability. Service level agreements (SLAs) should be reviewed to ensure that they meet the requirements of the application.

Integration: Cloud services should be evaluated for their ability to integrate with other systems and applications.

Support: Cloud services should be evaluated for the level of support they provide, including technical support, customer service, and documentation.

Flexibility: Cloud services should be evaluated for their ability to meet changing needs and requirements. This includes the ability to easily scale up or down, add new features or functionality, and switch between different cloud providers if necessary.

Vendor lock-in: Cloud services should be evaluated for the potential for vendor lock-in, which can make it difficult or costly to switch to a different cloud provider in the future.

Serverless

Serverless is a cloud computing model where the cloud provider manages the infrastructure and the server-side processes required for executing a piece of code or function. In serverless architecture, the developers only need to provide the code or function, and the cloud provider takes care of the rest, including scaling, resource allocation, and availability.

The term "serverless" does not mean that there are no servers involved. It refers to the abstraction of servers and infrastructure management away from developers. Instead of managing the servers, developers can focus on writing code that responds to specific events or triggers.

Serverless has gained popularity due to its cost-effectiveness, scalability, and ease of use. It allows developers to create applications that can scale automatically and handle large workloads without worrying about the underlying infrastructure. It also reduces operational costs by only charging for the time that the function runs.

Some examples of serverless platforms include AWS Lambda, Google Cloud Functions, and Azure Functions.

Serverless components

Serverless computing is an execution model where the cloud provider dynamically manages the allocation and provisioning of servers, and the customer only pays for the actual usage of resources. Serverless components refer to the various services that make up a serverless architecture, including:

Functions as a Service (FaaS): Serverless computing platforms provide FaaS capabilities, allowing developers to deploy functions and execute them in a serverless environment.

Backend as a Service (BaaS): BaaS provides serverless access to backend services, such as authentication, database, storage, and APIs.

Event-driven computing: Serverless computing platforms are designed to execute code in response to events, such as an HTTP request, database change, or message from a message queue.

Stateless computing: Serverless computing platforms are typically stateless, meaning that there is no need to maintain a persistent connection to a server.

Microservices: Serverless computing promotes a microservices architecture, where small, independent, and decoupled services are deployed and executed independently.

Containerization: Containerization is used to package the code and its dependencies in a container, which can then be deployed on a serverless computing platform.

Advantages of Serverless

Serverless has several advantages, including:

Cost savings: With serverless, you only pay for what you use, and you don't have to worry about the cost of managing and maintaining servers, which can be a significant cost savings.

Scalability: Serverless architectures allow for automatic scaling of resources, which means that applications can handle a sudden increase in traffic without any manual intervention.

Reduced time-to-market: Developers can focus on writing code instead of managing infrastructure, which can help speed up development and deployment times.

Increased reliability: Serverless architectures are designed to be highly available, and service providers often provide redundancy and failover mechanisms to ensure that services are always available.

Simplified management: Serverless architectures can be easier to manage than traditional architectures, as there are fewer moving parts to worry about.

Flexibility: Serverless architectures allow developers to use a wide range of programming languages, frameworks, and tools, giving them greater flexibility in designing and implementing applications.

IAAS, PAAS, Container, Serverless

	IaaS	PaaS	Container	Serverless
Scale	VM	Instance	App	Function
Abstracts	Hardware	Platform	OS Host	Runtime
Unit	VM	Project	Image	Code
Lifetime	Months	Days to Months	Minutes to Days	Milliseconds to Minutes
Responsibility	Applications, dependencies, runtime, and operating system	Applications and dependencies	Applications, dependencies, and runtime	Function

- **Scale** refers to the unit that is used to scale the application
- **Abstracts** refers to the layer that is abstracted by the implementation
- **Unit** refers to the scope of what is deployed
- **Lifetime** refers to the typical runtime of a specific instance
- **Responsibility** refers to the overhead to build, deploy, and maintain the application

Serverless Patterns

Serverless patterns are specific architectures that are optimized for serverless computing. They provide a structure for building serverless applications and address specific use cases. Some common serverless patterns include:

1. Event-driven computing: In this pattern, the application is triggered by an event, such as a file being uploaded to a storage service. The application performs a specific action in response to the event and then exits.
2. Batch processing: This pattern is used for processing large volumes of data in parallel. The application is triggered by a schedule or a specific event, such as a file being uploaded. The application then processes the data in batches and writes the results to a storage service.
3. Real-time stream processing: In this pattern, the application processes data as it is generated in real-time. The application is triggered by a stream of events, such as sensor data from IoT devices. The application processes the data and writes the results to a storage service or triggers another action.
4. RESTful microservices: This pattern is used for building microservices that expose RESTful APIs. Each microservice performs a specific function, such as processing payments or retrieving data from a database. The microservices can be independently deployed and scaled.
5. Mobile and IoT backend: This pattern is used for building backends for mobile and IoT applications. The backend provides authentication, storage, and other services for the application. The backend is triggered by events from the application, such as a user logging in or a device sending data.

These patterns provide a starting point for building serverless applications and can be customized to meet specific requirements.

Command and Query Responsibility Segregation (CQRS)

Command and Query Responsibility Segregation (CQRS) is a design pattern that separates the responsibility for handling commands (i.e., modifying data) from the responsibility for handling queries (i.e., retrieving data) in a software application.

The core idea behind CQRS is to use different models for reading and writing data. The write model is optimized for fast write operations, while the read model is optimized for fast read operations. This approach allows for better scalability, performance, and flexibility in complex applications.

In a CQRS architecture, write operations are typically handled by a command model, which receives commands from clients, processes them, and updates the write database. Read operations are handled by a separate query model, which is optimized for reading data and returns data to clients.

CQRS can be used in conjunction with other architectural patterns, such as event sourcing and microservices, to build highly scalable and flexible systems. It is especially useful in applications where read and write operations have different performance requirements, or where data needs to be processed in real-time.

Event-based processing

Event-based processing is a software design pattern that allows systems to react to events and messages in real-time as they occur, rather than relying on batch processing or polling mechanisms. In this pattern, a system is designed to emit and consume events, which represent state changes in the system or external inputs. The system can then use these events to trigger actions or updates, either within the same system or in other connected systems.

Event-based processing is often used in distributed systems, microservices, and serverless architectures. It allows for decoupling of components and services, as well as scalability and fault-tolerance. By emitting and consuming events, services can remain loosely coupled, and failures can be isolated to individual components rather than causing a cascading failure across the entire system.

There are several tools and technologies available for implementing event-based processing, including message brokers, event sourcing, and stream processing frameworks. These technologies allow for the efficient and reliable processing of large volumes of events in real-time.

File triggers and transformations

File triggers and transformations refer to a pattern of processing files based on certain events or triggers and transforming them as needed. The pattern involves setting up a file system watcher to monitor a directory or file for any changes or updates. When a new file is detected or an existing file is updated, the watcher triggers a set of actions, such as reading the file, performing some processing or transformation, and storing the output in a different location.

This pattern is commonly used in data processing applications, such as ETL (Extract, Transform, Load) pipelines and batch processing systems. For example, a data pipeline may monitor a directory for new data files generated by a source system, transform the data into a specific format or schema, and load it into a database or data warehouse for further analysis.

File triggers and transformations can be implemented using a variety of technologies and tools, including shell scripts, programming languages such as Python or Java, and specialized frameworks such as Apache NiFi or AWS Glue. The pattern can also be extended to include more advanced features, such as data validation, error handling, and workflow management.

Web apps and APIs

Web applications and APIs (Application Programming Interfaces) are two common types of software applications used to deliver services over the internet.

A web application is a software application that runs on a web server and is accessed via a web browser. It typically consists of HTML, CSS, and JavaScript that is rendered by the browser, and it communicates with the server using HTTP requests and responses.

An API, on the other hand, is a set of rules and protocols that defines how software components should interact with each other. It enables software applications to communicate with each other and share data without the need for direct integration. APIs can be used to access data, services, or functionality from third-party applications or web services.

Web applications and APIs are often used together to provide a comprehensive solution for delivering software services over the internet. Web applications can consume APIs to provide additional functionality, while APIs can be used to integrate disparate software components into a cohesive solution.

Data pipeline

A data pipeline is a series of automated processes that move and transform data from source systems to destination systems. It is used to manage the flow of data from one place to another and to ensure that the data is accurate, consistent, and timely. The pipeline can be used to collect data from various sources, process the data, and then store it in a destination system. Data pipelines are essential for organizations that rely on data-driven decisions to make business decisions. A well-designed data pipeline can help to streamline the data collection process, reduce errors, and ensure that data is consistent and accurate.

Stream processing

Stream processing is a computing approach that involves analyzing and processing real-time data streams, such as data generated by sensors, clickstreams, or financial transactions. The main idea behind stream processing is to analyze and act on data as it flows through a system, rather than waiting to store it in a database or data warehouse before analyzing it. This approach enables businesses to gain insights into their operations, detect anomalies or fraud in real-time, and respond quickly to changing conditions.

Stream processing typically involves a distributed system that ingests and processes high-volume, high-velocity data streams in real-time. The system may use complex event processing (CEP) techniques to detect patterns and anomalies in the data, and machine learning algorithms to make predictions and recommendations. Stream processing can be implemented using a variety of tools and technologies, including Apache Kafka, Apache Flink, and Apache Spark Streaming.

Stream processing has a wide range of use cases across different industries, including financial services, healthcare, retail, and transportation. For example, stream processing can be used in fraud detection systems to detect and block fraudulent transactions in real-time, or in predictive maintenance systems to identify potential equipment failures before they occur. Stream processing can also be used in real-time recommendation engines for e-commerce or media streaming services, or in monitoring and analyzing social media feeds for sentiment analysis and brand reputation management.

API gateway

An API gateway is a server that acts as an entry point for a group of microservices in a service-oriented architecture (SOA). It is responsible for routing client requests to the appropriate microservice, as well as providing additional features such as authentication, rate limiting, and monitoring. An API gateway can act as a security layer by protecting backend services from direct client access and reducing the attack surface. It can also simplify the client code by exposing a unified API for all the backend services, instead of requiring the client to interact with each service separately.

What is low code?

Low code is a software development approach that emphasizes visual, drag-and-drop interfaces and prebuilt application components that allow users to create and customize applications with little to no coding required. Low code platforms often include tools for model-driven development, business process automation, and integration with other systems, enabling users to quickly create applications that meet their specific needs. The goal of low code is to streamline the application development process and allow non-technical users to build applications without relying on a dedicated development team.

But why did low-code come about?

Low-code development came about as a response to the growing demand for software solutions to meet rapidly evolving business requirements, while also dealing with a shortage of skilled developers. Traditional software development approaches required extensive coding, testing, and maintenance, which could be time-consuming and expensive. With low-code development, non-technical business users can build applications with little or no coding knowledge, which can significantly reduce the time and cost involved in developing new software solutions. Additionally, low-code platforms provide pre-built components and templates, making it easier for developers to build applications quickly and efficiently.

LOW-CODE: TRANSFORM IDEAS TO INNOVATION

Low-code is a software development approach that enables people with little or no coding experience to create applications quickly and easily using a visual interface with drag-and-drop components, pre-built templates, and automated workflows. The low-code approach is designed to address the challenges faced by organizations in developing software applications, such as the lack of IT skills and the high cost and time required for traditional software development. With low-code, users can create applications that automate business processes, analyze data, and improve customer experience without the need for extensive coding.

Low-code platforms provide an environment for creating and deploying applications with minimal coding. They offer a range of pre-built components, such as data connectors, integrations, workflows, and analytics, that can be easily combined to create complex applications. Low-code platforms also include features for testing, debugging, and deployment, which help organizations to reduce the time and cost involved in software development.

The low-code approach has become increasingly popular in recent years due to its ability to help organizations quickly develop and deploy applications that address specific business needs. With low-code, businesses can respond quickly to changing market conditions, customer demands, and other factors that impact their operations. Low-code platforms also provide a way for companies to improve their agility, flexibility, and innovation, as well as their ability to scale up and down as needed.

Overall, low-code platforms are transforming the way organizations approach software development, enabling them to quickly turn their ideas into innovative solutions that drive growth and competitive advantage.

Key features of low-code

Some key features of low-code platforms are:

1. Visual development: Low-code platforms allow developers to drag-and-drop pre-built components to create applications visually, rather than writing code from scratch. This makes it easier and faster to build applications, even for non-technical users.
2. Rapid application development: Low-code platforms enable faster application development by providing pre-built components, templates, and integrations. Developers can create applications quickly, reducing time-to-market.
3. Process automation: Low-code platforms often include workflow engines and business process management tools, allowing users to automate processes and workflows without writing code.
4. Integration capabilities: Low-code platforms provide out-of-the-box integrations with other systems, such as databases, APIs, and third-party services. This simplifies integration and reduces the need for custom code.
5. Collaboration and governance: Low-code platforms provide tools for team collaboration and version control, ensuring that all stakeholders can work together effectively and securely.
6. Mobile and responsive design: Low-code platforms enable developers to create mobile applications and responsive web designs without additional coding.
7. Scalability: Low-code platforms provide a scalable architecture that can support large-scale, enterprise-level applications.
8. Security: Low-code platforms provide security features such as authentication, authorization, and data encryption to ensure data protection and compliance with industry regulations.

Developing Low Code Apps

Developing low-code apps typically involves the following steps:

1. Defining requirements: The first step is to clearly define the requirements of the app. This includes understanding the user needs, defining the features and functionality required, and determining any integrations with other systems or services.
2. Selecting a low-code platform: Once the requirements are defined, the next step is to select a low-code platform that best meets the needs of the project. There are a variety of low-code platforms available, each with their own strengths and weaknesses, so it's important to do research and select the platform that best fits the project requirements.
3. Designing the user interface: The next step is to design the user interface (UI) of the app. Most low-code platforms provide a visual UI designer that allows developers to drag and drop UI components and customize their properties.

4. Configuring the data model: After the UI is designed, the next step is to configure the data model. This involves defining the data schema and creating the database tables and fields required to store the app's data.
5. Configuring business logic: Once the data model is configured, the next step is to configure the app's business logic. This involves defining the rules and processes that govern how the app functions. Most low-code platforms provide a visual process designer that allows developers to define business logic using a drag-and-drop interface.
6. Testing and deployment: The final step is to test the app and deploy it to production. Most low-code platforms provide testing tools that allow developers to test the app's functionality and performance. Once the app is tested and approved, it can be deployed to production either on-premise or in the cloud.

Get started with low-code apps in 5 simple steps

Here are 5 simple steps to get started with low-code apps:

1. Identify the problem you want to solve: Before you start building an app, you need to identify the problem you want to solve. Determine the scope of the app and the features that you need to include to address the problem.
2. Choose a low-code platform: Once you have identified the problem and the scope of the app, you need to choose a low-code platform. There are many options available in the market such as Microsoft Power Apps, Appian, Salesforce Lightning, Mendix, etc.
3. Design the app: Once you have chosen the platform, you can start designing the app. Most low-code platforms offer drag-and-drop interfaces, which makes it easy to design the app without writing any code.
4. Configure the app: After designing the app, you need to configure it. This involves setting up the database, creating workflows, and defining the business logic.
5. Test and deploy the app: Once you have configured the app, you can test it to ensure that it works as expected. After testing, you can deploy the app to the cloud or on-premise servers.

By following these simple steps, you can create low-code apps quickly and easily, without requiring any programming expertise.

4: Serverless Apps

Backend as a Service (BaaS)

Backend as a Service (BaaS) is a model in which a third-party service provider offers cloud-based backend services for mobile or web applications. These services typically include features such as user authentication, data storage, push notifications, and serverless functions.

BaaS providers enable developers to focus on the frontend of their applications, as opposed to the backend development that would normally be required. This means that developers do not have to

worry about infrastructure, database management, or server configuration, allowing them to focus on the features and user experience of the application.

BaaS providers typically offer APIs that enable developers to integrate backend services into their applications seamlessly. These services are typically available on a subscription basis, and pricing is often based on usage metrics such as API calls or data storage.

BaaS has become increasingly popular in recent years as more developers turn to cloud-based solutions to simplify and speed up application development. Some of the most popular BaaS providers include Firebase, AWS Amplify, and Kinvey.

Why Backend as a service?

Backend as a Service (BaaS) provides an easy and cost-effective way to develop and manage the backend of a mobile or web application. It allows developers to focus on building the frontend of an application and not worry about managing the backend infrastructure, server maintenance, database, API, and security. With BaaS, developers can easily access pre-built backend components through APIs and integrate them into their application. BaaS providers offer features such as user authentication, data storage, push notifications, social media integration, and analytics. This makes the development process faster and more efficient, as developers can focus on building the features that matter most to their users rather than building the infrastructure from scratch. Additionally, BaaS providers typically offer scalable and reliable infrastructure that can handle large amounts of traffic, ensuring that the application is always available and performing well.

Mobile Backend as a Service MBaaS

Mobile Backend as a Service (MBaaS) is a cloud-based platform that provides mobile app developers with a set of tools and services to help them quickly and easily build the backend infrastructure of their mobile applications.

Traditionally, building a backend infrastructure for a mobile app required significant time and effort, as developers had to build and maintain their own server-side APIs, databases, and authentication systems. MBaaS eliminates the need for developers to write backend code by providing pre-built backend services such as data storage, user management, file storage, and serverless functions.

MBaaS allows developers to focus on building the frontend of their mobile applications, as they can simply integrate the pre-built backend services through APIs. This reduces development time and enables rapid app development, as developers do not need to worry about backend infrastructure complexities.

MBaaS also provides scalability and flexibility to mobile apps, as the backend infrastructure is managed by the MBaaS provider, which ensures high availability, reliability, and security of the mobile app. Additionally, MBaaS providers offer various pricing models, including pay-as-you-go and subscription-based models, which allows developers to choose a plan that best fits their needs and budget.

MBaaS Providers

There are several MBaaS providers available in the market, including:

1. Firebase: Firebase is a Google-backed MBaaS provider that offers a range of features for mobile app development, including real-time database, user authentication, and cloud storage.
2. AWS Amplify: AWS Amplify is a cloud-based platform that offers a range of tools and services for mobile app development, including backend services, data storage, and push notifications.
3. Kinvey: Kinvey is an MBaaS provider that offers a range of features for mobile app development, including backend services, data integration, and push notifications.
4. Back4App: Back4App is a cloud-based platform that offers a range of tools and services for mobile app development, including backend services, data storage, and push notifications.
5. Microsoft Azure Mobile Services: Microsoft Azure Mobile Services is a cloud-based platform that offers a range of features for mobile app development, including backend services, data storage, and push notifications.
6. Parse: Parse is an open-source MBaaS platform that offers a range of features for mobile app development, including backend services, data storage, and push notifications.
7. Kuzzle: Kuzzle is an open-source, self-hosted MBaaS platform that offers a range of features for mobile app development, including backend services, data storage, and push notifications.
8. Backendless: Backendless is an MBaaS platform that offers a range of features for mobile app development, including backend services, data storage, and push notifications.
9. DreamFactory: DreamFactory is an open-source MBaaS platform that offers a range of features for mobile app development, including backend services, data storage, and push notifications.

These are just a few examples of MBaaS providers available in the market. The choice of MBaaS provider may depend on factors such as the specific features required for the app, the cost of the service, and the level of support offered by the provider.

Function as a Service

Function as a Service (FaaS) is a cloud computing model where the cloud provider is responsible for executing a piece of code in response to an event. In this model, developers write small, self-contained functions that perform a specific task or operation. When an event occurs, such as a user accessing a web page or uploading a file, the function is triggered to run and perform its task.

FaaS provides a way to execute code without the need to manage and maintain servers or infrastructure. It is a form of serverless computing, where the focus is on writing and deploying code rather than managing and maintaining servers.

FaaS has become popular for building event-driven, serverless architectures and for developing microservices-based applications. It is often used for tasks such as data processing, file processing, and real-time data analysis. FaaS providers include Amazon Web Services (AWS) Lambda, Google Cloud Functions, and Microsoft Azure Functions

Characteristics of FaaS

Function as a Service (FaaS) has several key characteristics, including:

1. Event-driven: FaaS is designed to execute code in response to specific events, such as incoming data or user actions.
2. Scalable: FaaS platforms can automatically scale to handle large volumes of requests, ensuring that functions can run as needed without performance issues.
3. Stateless: FaaS functions are stateless, meaning that they do not maintain a persistent connection to any specific user or request. This makes them more flexible and easier to scale.
4. Pay-per-use: FaaS platforms typically charge based on the number of function invocations and the resources used during execution, allowing developers to pay only for what they actually use.
5. Platform-agnostic: FaaS platforms can be used with any programming language or development environment, making them highly flexible and compatible with a wide range of use cases.
6. Short-lived: FaaS functions are typically designed to execute quickly and complete their tasks within a few seconds or less. This helps to minimize costs and improve performance.

Benefits of FaaS

Here are some benefits of FaaS:

1. Cost-effectiveness: FaaS providers charge only for the time your functions are running, making it a cost-effective solution for businesses.
2. Scalability: FaaS allows for automatic scaling, meaning that you don't have to worry about managing server capacity, load balancing, and infrastructure.
3. Reduced complexity: With FaaS, you only need to write and deploy code for specific functions, instead of having to manage an entire application stack.
4. Faster time-to-market: FaaS allows developers to focus on writing code instead of dealing with infrastructure and deployment, allowing for faster time-to-market.
5. Easy integration: FaaS integrates with other cloud services, APIs, and databases, making it easier to develop and deploy serverless applications.

Serverless Architecture? How it works?

Serverless architecture is a cloud computing model where the cloud provider manages the infrastructure and automatically allocates resources as needed for running applications. It is also known as Function-as-a-Service (FaaS) architecture. In serverless architecture, developers only need to focus on writing the code for the specific task or function, and the cloud provider takes care of the rest.

The serverless architecture works by breaking down applications into smaller, independent functions or microservices. These functions are executed only when triggered by an event or request, such as a user request to a web application or an event from an IoT device. The cloud provider allocates the necessary computing resources for executing the function and automatically scales up or down based on demand.

Serverless architecture is typically used for event-driven and compute-intensive workloads. It offers several benefits such as cost efficiency, scalability, and reduced time-to-market for applications. However, it also has some limitations such as longer cold-start times and limited control over the infrastructure.

FaaS vs Serverless

FaaS and Serverless are related but not the same.

FaaS stands for Function-as-a-Service, and it is a model in which developers create small, single-purpose functions that can be executed on demand, in response to specific events or triggers. FaaS providers take care of the underlying infrastructure, including the servers, networking, and other resources required to run the functions. Popular FaaS providers include AWS Lambda, Azure Functions, and Google Cloud Functions.

Serverless architecture, on the other hand, is a broader concept that encompasses FaaS, as well as other services such as API gateways, databases, and messaging services. With serverless architecture, developers focus on writing code that performs specific tasks or services, rather than managing the infrastructure required to run that code. The serverless approach allows for more efficient and cost-effective development, as developers only pay for the resources they use, and don't need to worry about the underlying infrastructure.

In summary, FaaS is a specific type of serverless architecture that allows developers to create small functions that can be executed on demand. Serverless architecture is a broader concept that includes FaaS, but also encompasses other services and technologies that enable more efficient and cost-effective development.

PaaS vs FaaS

Platform as a Service (PaaS) and Function as a Service (FaaS) are two different cloud computing models that offer unique features and benefits.

PaaS provides a complete platform for developers to build and deploy applications without having to worry about the underlying infrastructure. It offers a range of services such as databases, middleware, application servers, and development tools that developers can use to build, test, and deploy their applications. PaaS is a good choice for teams that want to build and deploy complex applications that require a high degree of customization and control.

FaaS, on the other hand, is a cloud computing model that allows developers to write and run code without having to manage the underlying infrastructure. FaaS providers such as AWS Lambda, Azure Functions, and Google Cloud Functions offer a platform where developers can write small pieces of code, called functions, and execute them in response to events such as API calls, file uploads, or

database updates. FaaS is an ideal choice for teams that want to build and deploy event-driven, serverless applications that scale dynamically and are cost-effective.

In summary, PaaS provides a complete platform for building complex applications, while FaaS offers a lightweight, event-driven model for building serverless applications. Both models have their strengths and weaknesses, and the choice between them depends on the specific needs and requirements of your project.

Serverless on AWS

AWS offers a wide range of serverless services that developers can use to build and deploy their applications without having to manage the underlying infrastructure. Some of the popular AWS serverless services are:

1. AWS Lambda: AWS Lambda is a serverless compute service that lets you run your code without provisioning or managing servers. You can write Lambda functions in Node.js, Python, Java, Go, C#, and Ruby.
2. Amazon API Gateway: Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. It supports REST and WebSocket APIs and integrates with AWS Lambda, AWS Step Functions, and other AWS services.
3. Amazon DynamoDB: Amazon DynamoDB is a fast and flexible NoSQL database service that provides consistent single-digit millisecond latency at any scale. It is fully managed and supports document and key-value data models.
4. Amazon S3: Amazon S3 is a highly scalable, durable, and secure object storage service. It can be used to store and retrieve any amount of data from anywhere on the web.
5. Amazon SQS: Amazon SQS is a fully managed message queuing service that enables decoupling and scaling of microservices, distributed systems, and serverless applications.
6. AWS Step Functions: AWS Step Functions is a serverless workflow service that lets you coordinate distributed applications and microservices using visual workflows.

These services are designed to work together seamlessly, making it easy to build and deploy serverless applications on AWS.

AWS Lambda and its benefits

AWS Lambda is a serverless computing platform provided by Amazon Web Services (AWS) that allows developers to run code without having to provision or manage servers. Some of the benefits of using AWS Lambda include:

1. Cost savings: With AWS Lambda, you only pay for the computing time that you consume. There are no upfront costs, and you don't have to pay for idle resources.

2. Scalability: AWS Lambda automatically scales your application to handle any amount of traffic without the need for manual intervention. This makes it easy to handle sudden spikes in traffic.
3. Fast development: AWS Lambda allows developers to focus on writing code without worrying about managing infrastructure. This makes it possible to develop and deploy applications faster.
4. High availability: AWS Lambda automatically replicates your code across multiple availability zones to ensure high availability and reliability.
5. Integration with other AWS services: AWS Lambda can easily integrate with other AWS services, such as Amazon S3, Amazon DynamoDB, and Amazon Kinesis, making it easy to build powerful serverless applications.

AWS Lambda? how it works

AWS Lambda is a compute service provided by Amazon Web Services (AWS) that allows developers to run code without managing or provisioning servers. It is a serverless computing platform that enables developers to build and run applications and services without worrying about the underlying infrastructure.

When a developer creates a function in AWS Lambda, they specify the amount of memory required to run the function, as well as other configuration settings. Once the function is created, the developer can then configure events to trigger the function, such as changes to data in an Amazon S3 bucket or an update to an Amazon DynamoDB table.

When an event triggers the function, AWS Lambda automatically provisions the required resources to execute the code and manages the compute capacity for the developer. Once the function has completed, the resources are automatically released, so the developer only pays for the time the function was actually running.

AWS Lambda supports a variety of programming languages, including Node.js, Java, C#, Python, and Go, among others. It also integrates with other AWS services, such as Amazon S3, Amazon DynamoDB, and Amazon API Gateway, making it easy to build serverless applications and services on AWS.

use case of AWS Lambda

AWS Lambda can be used in various use cases, some of them are:

1. Event-driven computing: AWS Lambda allows developers to build event-driven applications without worrying about the underlying infrastructure. For example, developers can use AWS Lambda to process data streams, generate notifications, and perform other tasks triggered by events from various AWS services like S3, DynamoDB, and Kinesis.
2. Serverless web applications: AWS Lambda can be used to build serverless web applications by integrating it with other AWS services like API Gateway, S3, and DynamoDB. Developers can use AWS Lambda to create the application logic, and API Gateway to handle the HTTP requests.

3. Real-time data processing: AWS Lambda can be used to process real-time data by integrating it with other AWS services like Kinesis and AWS IoT. Developers can use AWS Lambda to analyze the data streams and generate insights in real-time.
4. Back-end processing: AWS Lambda can be used as a back-end processing service for mobile and web applications. Developers can use AWS Lambda to execute back-end tasks like user authentication, data validation, and database operations.
5. Image and video processing: AWS Lambda can be used to process images and videos in real-time by integrating it with other AWS services like Amazon Rekognition and Amazon Transcribe. Developers can use AWS Lambda to analyze the images and videos and generate insights in real-time.

Google Cloud Platform

Google Cloud Platform (GCP) is a cloud computing platform offered by Google that provides a range of infrastructure and platform services for building, deploying, and managing applications and services. Some of the key services offered by GCP include:

1. Compute: GCP provides a range of compute options, including virtual machines (VMs) with customizable hardware configurations, container engine, and serverless computing with Google Cloud Functions.
2. Storage: GCP offers a variety of storage services, including object storage (Google Cloud Storage), block storage (Google Cloud Persistent Disks), and file storage (Google Cloud Filestore).
3. Networking: GCP provides a global, low-latency network with advanced features like load balancing, DNS, and VPN.
4. Big Data and Machine Learning: GCP provides a range of big data and machine learning services, including BigQuery, Cloud Dataflow, Cloud Dataproc, Cloud ML Engine, and more.
5. Developer Tools: GCP offers a range of tools for developing, debugging, and deploying applications, including Google Cloud SDK, Cloud Build, and Cloud Source Repositories.
6. Identity and Security: GCP provides robust security features and tools for identity and access management, encryption, and compliance.
7. IoT: GCP provides a range of IoT services, including IoT Core for managing IoT devices and Cloud IoT Edge for processing IoT data at the edge.

Overall, GCP is a comprehensive cloud computing platform that offers a wide range of services and tools for building and running modern applications and services.

Google Cloud Platform - serverless

Google Cloud Platform offers several serverless services that allow users to build and run applications without the need to manage servers or infrastructure. Some of the serverless services offered by Google Cloud Platform are:

1. Cloud Functions: Google Cloud Functions is a serverless computing service that allows users to run code in response to events, such as changes to data in a storage bucket or the creation of a new instance of a virtual machine.
2. Cloud Run: Google Cloud Run is a fully managed serverless platform that allows users to deploy and run containerized applications. It allows users to run any stateless HTTP request/response-based service, including web applications, microservices, and HTTP APIs.
3. App Engine: Google App Engine is a fully managed serverless platform that allows users to build and deploy web applications and APIs using several programming languages, including Java, Python, PHP, Node.js, Ruby, and .NET. App Engine automatically scales the resources required to run applications based on the traffic they receive.
4. Cloud Firestore: Google Cloud Firestore is a fully managed NoSQL document database that allows users to store, synchronize, and query data for their mobile, web, and IoT applications. Cloud Firestore is designed to automatically scale based on user demand and requires no server management.
5. Cloud Pub/Sub: Google Cloud Pub/Sub is a fully managed message queuing service that allows users to send and receive messages between independent applications. It supports many-to-many asynchronous messaging, provides reliable message delivery, and scales automatically based on demand.

These serverless services offered by Google Cloud Platform provide benefits such as reduced operational overhead, automatic scaling, and pay-per-use pricing.

Google Cloud Platform - benefits

Google Cloud Platform offers a range of benefits, including:

1. Scalability: GCP offers high scalability, allowing users to scale up or down their resources as needed.
2. Reliability: GCP's global network of data centers and advanced infrastructure ensures high availability and reliability for users.
3. Security: GCP provides a variety of security features, including encryption, identity and access management, and DDoS protection.
4. Cost-effectiveness: GCP offers a pay-as-you-go model, which allows users to only pay for the resources they use.
5. Innovation: GCP provides access to a wide range of innovative tools and technologies, such as machine learning, big data processing, and serverless computing.
6. Flexibility: GCP offers a range of services and deployment options, including virtual machines, containers, and serverless computing.
7. Integration: GCP integrates with a wide range of other Google services, as well as third-party tools and services.
8. Support: GCP provides a range of support options, including documentation, community support, and paid support plans.

GCP Serverless products

Google Cloud Platform provides several serverless products, including:

1. Cloud Functions: A fully managed service for running event-driven, serverless applications written in Node.js, Python, and Go.
2. Cloud Run: A fully managed serverless platform for running containerized applications.
3. App Engine: A fully managed platform for building and deploying web applications and APIs.
4. Cloud Tasks: A fully managed service for managing and executing asynchronous tasks in the cloud.
5. Cloud Pub/Sub: A fully managed messaging service for sending and receiving messages between independent applications.
6. Cloud Firestore: A fully managed NoSQL document database for storing, syncing, and querying data.
7. Cloud Storage: A fully managed object storage service for storing and serving files and data.

These serverless products provide developers with the flexibility to build and deploy applications without worrying about managing servers or infrastructure.

Google Cloud Functions

Google Cloud Functions is a serverless compute platform that allows you to build and run event-driven applications and microservices. It provides a fully managed environment where you can write code in your preferred language without worrying about infrastructure management. Cloud Functions lets you trigger your code in response to a variety of events, such as changes in data stored in Google Cloud Storage, incoming HTTP requests, or messages in Cloud Pub/Sub.

Cloud Functions supports a variety of programming languages, including Node.js, Python, Go, and Java, and integrates with other Google Cloud services such as Cloud Firestore, Cloud Pub/Sub, Cloud Storage, and Cloud Vision API.

One of the key benefits of Cloud Functions is its pay-per-use model, where you only pay for the number of function invocations and the duration of the function's execution, with no upfront costs or reservations required. This allows you to build cost-efficient and scalable applications, as well as experiment with new ideas without having to worry about infrastructure costs.

Question paper Mid term solution

Q.1 Answer in brief: **[2 * 3 = 6]**

(a) Explain with an example, relationship between microservices, Docker and kubernetes.

(b) Consider a case of composite pattern - where an application fetches data from different APIs. For example, a dashboard that retrieves the required data from different sources such as logging services APIs, cloud based backends for consumption numbers, third-party analytics services to capture end-user interactions etc. Which type of API design style will be suitable for this scenario? Justify?

(c) Serverless computing can be thought of as no servers instead of less servers. Justify/Invalidate.

1Ans:

(a)

Let's take an example scenario of a web-based e-commerce platform to explain the relationship between microservices, Docker, and Kubernetes.

The e-commerce platform is built using a microservices architecture, where each microservice is responsible for a specific functionality of the platform such as product catalog, shopping cart, payment processing, order management, etc. Each microservice is developed, deployed, and managed independently of the others.

To run each microservice, Docker containers are used. Each microservice is packaged as a Docker container along with all its dependencies, libraries, and configurations. This ensures that each microservice can run in any environment with the same behavior, regardless of the underlying infrastructure.

Kubernetes is used as the container orchestration platform to manage and deploy these microservices at scale. Kubernetes helps to automate the deployment, scaling, and management of containerized applications. It provides features such as load balancing, automatic scaling, rolling updates, self-healing, and more.

With Kubernetes, each microservice is deployed as a set of containers, called a Kubernetes pod. These pods are deployed across a cluster of nodes, which can be physical or virtual machines. Kubernetes monitors the health of each pod and automatically restarts failed pods to ensure high availability and reliability of the application.

Overall, the combination of microservices, Docker, and Kubernetes allows for the development and deployment of scalable, resilient, and agile applications that can be easily maintained and updated without disrupting the entire system.

(b) The suitable API design style for a composite pattern scenario where an application fetches data from different APIs is the API Gateway pattern. An API Gateway acts as a single entry point for multiple microservices and APIs, providing a unified interface for clients to access the various services. The API Gateway can handle requests from clients, route them to the appropriate service, and aggregate responses from multiple services to provide a single consolidated response. This pattern is suitable for scenarios where there are multiple independent services, and clients require a unified view of these services.

(c) The statement "Serverless computing can be thought of as no servers instead of less servers" can be justified. Serverless computing, also known as Function-as-a-Service (FaaS), allows developers to write and deploy code without having to worry about managing servers or infrastructure. With serverless computing, the cloud provider manages the underlying infrastructure, and developers only need to write the code for their specific functions. This can reduce the operational overhead and costs associated with managing servers, and allow developers to focus more on writing code that solves business problems. However, serverless computing still relies on servers and infrastructure, but the difference is that the management of these resources is abstracted away from the developer.

Q.2 Rishikesh is an enthusiastic traveler. When travelling he uses his DSLR camera a lot to capture the pictures of the surroundings. Also he uses these pictures in blogposts which narrates his journeys and experiences of the places which he has visited in the past. As his blogs are very informative, many readers find them quite useful when they plan their journeys to those places, hence many feedbacks are also shared by the readers.

Answer the following sub-questions based on the above narrative: **[2 + 2 + 2 = 6]**

(a) What type of data is captured in this narrative?
(b) Whether a file based system will be appropriate choice for such type of data? Why?
(c) What other type of data storage will be suitable for such type of data? Justify with example.

2Ans:

(a) The narrative involves capturing and storing multimedia data (images), textual data (blogposts), and feedback data.

(b) A file-based system may not be an appropriate choice for this type of data as it can become difficult to manage and organize large amounts of multimedia and textual data in a hierarchical file system. Additionally, it may be challenging to search and retrieve specific data from the file system.

(c) A cloud-based object storage system such as Amazon S3 or Google Cloud Storage would be a suitable choice for this type of data. These storage systems are designed to handle large amounts of unstructured data, provide easy and flexible access to the data, and offer advanced search and retrieval capabilities. For example, Rishikesh could store his images in Amazon S3 buckets and his blogposts in Google Cloud Storage buckets, and access them through respective APIs.

Q.3 Let's assume that you need to design a mobile app for English language dictionary. It should serve as English language learning tool and provide word games built for every level of learner. With trusted definitions and synonyms plus word puzzles, language quizzes, and spelling quizzes, this English dictionary and thesaurus app for Android should be optimized with your mobile device in mind to help you learn English or improve your English vocabulary. In addition to the trusted reference content from Dictionary.com and Thesaurus.com, this education app should include:

- Word Puzzle ► To Improve your vocabulary with fun spelling quizzes and vocabulary challenges.
- Word of the Day ► Learn a new word each day and expand your vocabulary education.
- Synonyms ► Get thesaurus content alongside your dictionary definitions.
- Audio pronunciations ► Never mispronounce another word.
- Voice search ► Find the definitions you're looking for anywhere, anytime. The app even offers up English spelling help. Not sure how a word is spelled? Say it out loud, and this app will find it for you.
- Grammar help ► Get grammar tips, word usage, and more to improve your writing.
- Favorite words and search history ► Customize your recently searched word list, and never forget the newest words you've learned
- Learner's dictionary ► Includes extra information about word usage for English learners

Answer the following sub questions based on this scenario: **[2 + 2 + 1 + 2 + 2 + 1 = 10]**

a) What will be the type of mobile dictionary application? Justify briefly.
b) What features of the mobile phone will be leveraged by this dictionary application?
c) What framework and programming language will be suitable for development of this type of application?
d) If this application needs to work in offline manner, then using a file for storing dictionary data will be an appropriate choice? Justify.
e) Every day when user first time opens up the application, a new "word of day" needs to be shown to him. Describe the factors that you will consider while designing this feature?
f) Whether using any Backend-as-a-Service (BaaS) for dictionary app will make any difference?

3Ans:

a) The type of mobile dictionary application will be an educational app that includes a dictionary and thesaurus, word puzzles, grammar help, word of the day, audio pronunciations, voice search, and favorite words and search history. This will help users learn English or improve their English vocabulary.

b) The mobile phone features that will be leveraged by this dictionary application include voice search, audio pronunciations, and offline functionality.

c) A suitable framework for the development of this type of application is React Native, which allows developers to build cross-platform mobile apps using a single codebase. The programming language will be JavaScript.

d) If this application needs to work in an offline manner, using a file for storing dictionary data will not be an appropriate choice as it will increase the app size and will not be easily updatable. Instead, a local database such as SQLite can be used for offline storage.

e) Factors to consider while designing the "word of the day" feature include selecting an appropriate algorithm to choose the word, ensuring the word is relevant to the user's level, and providing additional information about the word such as synonyms, antonyms, and usage examples.

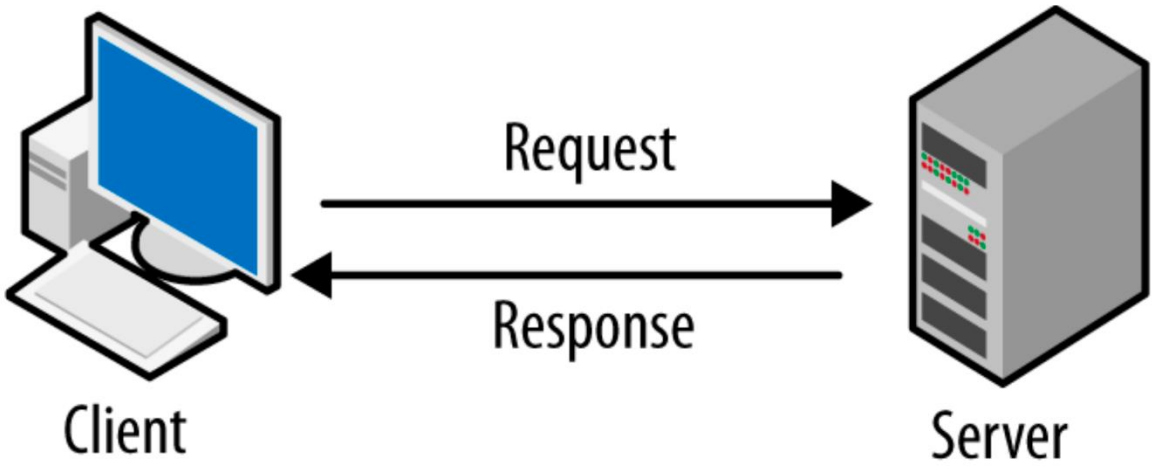
f) Using a Backend-as-a-Service (BaaS) for the dictionary app can make a difference in terms of scalability, data storage, and user management. It can help developers focus on building the core features of the app while relying on the BaaS provider for backend infrastructure.

Q.4 For the below mentioned cases, identify a suitable architectural style (discussed in the class) and provide an architectural block diagram and short description narrating the request-response flow between various components involved. **[8]**

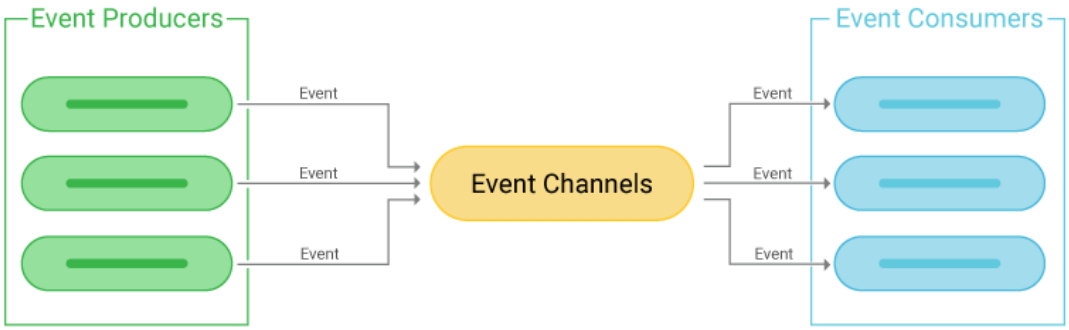
a) Yours is a unified payment interface that enables transfer of money from one back account to another account and also has plans in mind to extend it for transfer of money between bank account and credit cards.
b) Yours is credit score management system that tracks the loans taken by the customer and updates the credit score on regular basis when an EMI is paid by the customer
c) Yours is a business that wants to adapt mobile-only application for supporting the business transactions and does not want to take headache associated with management and maintenance of infrastructure required for the application
d) You quickly need to build a prototype of the product before embarking on a more ambitious project, its less complex in nature and the team has expertise into conventional development and deployment approaches

4Ans:

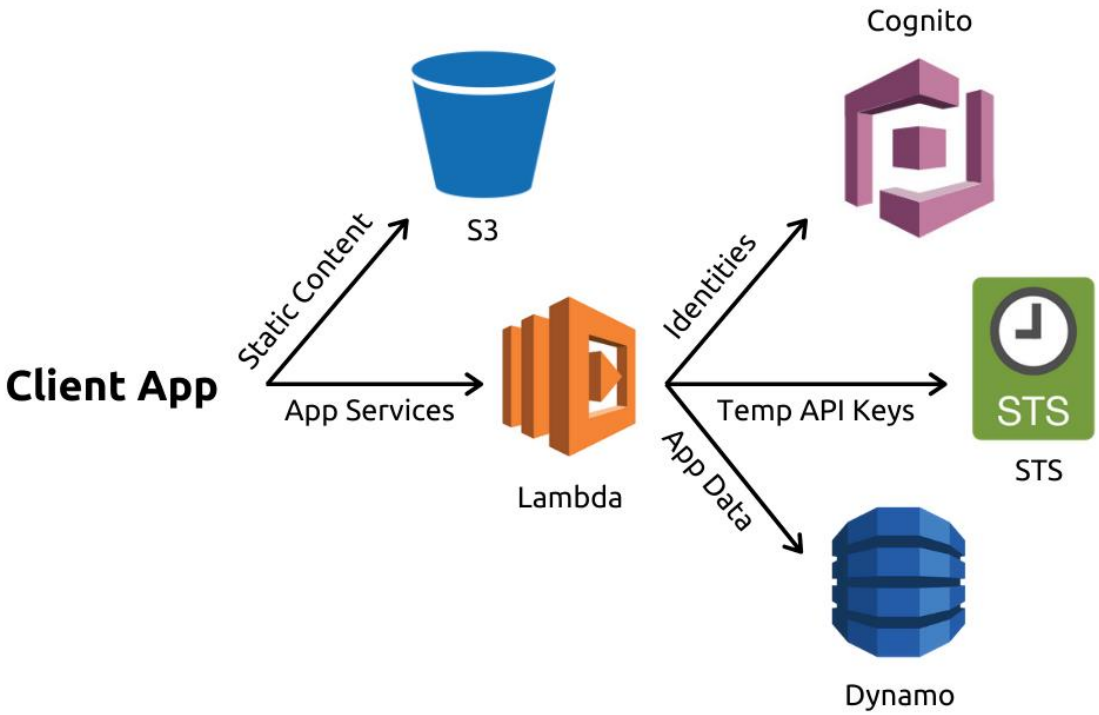
a) For the first case of a unified payment interface, a suitable architectural style is the client-server architecture. In this architecture, the client, which could be a mobile app or a web application, sends a request to the server to initiate a transfer of money. The server validates the request and processes the transfer of funds between the bank accounts or credit cards.



b) For the credit score management system, a suitable architectural style is the event-driven architecture. In this architecture, the system listens for events such as EMI payments and updates the credit score accordingly.



c) For the mobile-only application, a suitable architectural style is serverless architecture. In this architecture, the application is built using cloud-based services, and the infrastructure management and maintenance is taken care of by the cloud provider.



d) For the prototype product, a suitable architectural style is the monolithic architecture. In this architecture, the application is built as a single unit, with all the components integrated together.

Monolithic Architecture