Task 1:

Layered pattern:

We often use ‘N-tier architecture’, or ‘Multi-tiered architecture’ to denote “layered architecture pattern”. It’s one of the most commonly used patterns where the code is arranged in layers. The key characteristics of this pattern are as follows:

- The outermost layer is where the data enters the system. The data passes through the subsequent layers to reach the innermost layer, which is the database layer.
- Simple implementations of this pattern have at least 3 layers, namely, a presentation layer, an application layer, and a data layer. Users access the presentation layer using a GUI, whereas the application layer runs the business logic. The data layer has a database for the storage and retrieval of data.

This pattern has the following advantages:

- Maintaining the software is easy since the tiers are segregated.
- Development teams find it easy to manage the software infrastructure, therefore, it’s easy to develop large-scale web and cloud-hosted apps.

Popular frameworks like Java EE use this pattern.

There are a few disadvantages too, as follows:

- The code can become too large.
- A considerable part of the code only passes data between layers instead of executing any business logic, which can adversely impact performance.

Client-Server pattern:

“Client-server software architecture pattern” is another commonly used one, where there are 2 entities. It has a set of clients and a server. The following are key characteristics of this pattern:

- Client components send requests to the server, which processes them and responds back.
- When a server accepts a request from a client, it opens a connection with the client over a specific protocol.
- Servers can be stateful or stateless. A stateful server can receive multiple requests from clients. It maintains a record of requests from the client, and this record is called a ‘session’.

Email applications are good examples of this pattern. The pattern has several advantages, as follows:

- Clients access data from a server using authorized access, which improves the sharing of data.
- Accessing a service is via a ‘user interface’ (UI), therefore, there’s no need to run terminal sessions or command prompts.
- Client-server applications can be built irrespective of the platform or technology stack.
- This is a distributed model with specific responsibilities for each component, which makes maintenance easier.

Some disadvantages of the client-server architecture are as follows:

- The server can be overloaded when there are too many requests.
- A central server to support multiple clients represents a ‘single point of failure’.

Master-Slave pattern:

“Master-slave architecture pattern” is useful when clients make multiple instances of the same request. The requests need simultaneous handling. Following are its’ key characteristics:

- The master launches slaves when it receives simultaneous requests.
- The slaves work in parallel, and the operation is complete only when all slaves complete processing their respective requests.

Advantages of this pattern are the following:

- Applications read from slaves without any impact on the master.
- Taking a slave offline and the later synchronization with the master requires no downtime.

Any application involving multi-threading can make use of this pattern, e.g., monitoring applications used in electrical energy systems.

There are a few disadvantages to this pattern, e.g.:

- This pattern doesn't support automated fail-over systems since a slave needs to be manually promoted to a master if the original master fails.
- Writing data is possible in the master only.
- Failure of a master typically requires downtime and restart, moreover, data loss can happen in such cases.

Pipe-Filter pattern:

Suppose you have complex processing in hand. You will likely break it down into separate tasks and process them separately. This is where the "Pipe-filter" architecture pattern comes into use. The following characteristics distinguish it:

- The code for each task is relatively small. You treat it as one independent 'filter'.
- You can deploy, maintain, scale, and reuse code in each filter.
- The stream of data that each filter processes pass through 'pipes'.

Compilers often use this pattern, due to the following advantages:

- There are repetitive steps such as reading the source code, parsing, generating code, etc. These can be easily organized as separate filters.
- Each filter can perform its' processing in parallel if the data input is arranged as streams using pipes.

- It’s a resilient model since the pipeline can reschedule the work and assign to another instance of that filter.

Watch out for a few disadvantages:

- This pattern is complex.
- Data loss between filters is possible in case of failures unless you use a reliable infrastructure.

Broker pattern:

Consider distributed systems with components that provide different services independent of each other. Independent components could be heterogeneous systems on different servers, however, clients still need their requests serviced. “Broker architecture pattern” is a solution to this.

It has the following broad characteristics:

- A broker component coordinates requests and responses between clients and servers.
- The broker has the details of the servers and the individual services they provide.
- The main components of the broker architectural pattern are clients, servers, and brokers. It also has bridges and proxies for clients and servers.
- Clients send requests, and the broker finds the right server to route the request to.
- It also sends the responses back to the clients.

Message broker software like IBM MQ uses this pattern. The pattern has a few distinct advantages, e.g.:

- Developers face no constraints due to the distributed environment, they simply use a broker.
- This pattern helps using object-oriented technology in a distributed environment.

Peer-to-Peer pattern:

“Peer-to-peer (P2P) pattern” is markedly different from the client-server pattern since each computer on the network has the same authority. Key characteristics of the P2P pattern are as follows:

- There isn’t one central server, with each node having equal capabilities.
- Each computer can function as a client or a server.
- When more computers join the network, the overall capacity of the network increases.

File-sharing networks are good examples of the P2P pattern. Bitcoin and other cryptocurrency networks are other examples. The advantages of a P2P network are as follows:

- P2P networks are decentralized, therefore, they are more secure. You must have already heard a lot about the security of the Bitcoin network.
- Hackers can’t destroy the network by compromising just one server.

Under heavy load, the P2P pattern has performance limitations, as the questions surrounding the Bitcoin transaction throughout show.

Event-Bus patter:

There are applications when components act only when there is data to be processed. At other times, these components are inactive. “Event-bus pattern” works well for these, and it has the following characteristics:

- A central agent, which is an event-bus, accepts the input.
- Different components handle different functions, therefore, the event-bus routes the data to the appropriate module.
- Modules that don’t receive any data pertaining to their function will remain inactive.

Think of a website using JavaScript. Users’ mouse clicks and keystrokes are the data inputs. The event-bus will collate these inputs and it will send

the data to appropriate modules. The advantages of this pattern are as follows:

- This pattern helps developers handle complexity.
- It's a scalable architecture pattern.
- This is an extensible architecture, new functionalities will only require a new type of events.

This software architecture pattern is also used in Android development.

Some disadvantages of this pattern are as follows:

- Testing of interdependent components is an elaborate process.
- If different components handle the same event require complex treatment to error-handling.
- Some amount of messaging overhead is typical of this pattern.

The development team should make provision for sufficient fall-back options in the event the event-bus has a failure.

Model-View-Controller patter:

"Model-View-Controller (MVC) architecture pattern" involves separating an applications' data model, presentation layer, and control aspects. Following are its' characteristics:

- There are three building blocks here, namely, model, view, and controller.
- The application data resides in the model.
- Users see the application data through the view, however, the view can't influence what the user will do with the data.
- The controller is the building block between the model and the view. View triggers events, subsequently, the controller acts on it. The action is typically a method call to the model. The response is shown in the view.

This pattern is popular. Many web frameworks like Spring and Rails use it, therefore, many web applications utilize this pattern. Its' advantages are as follows:

- Using this model expedites the development.
- Development teams can present multiple views to users.
- Changes to the UI is common in web applications, however, the MVC pattern doesn't need changes for it.
- The model doesn't format data before presenting to users, therefore, you can use this pattern with any interface.

There are also a few disadvantages, for e.g.:

- With this pattern, the code has new layers, making it harder to navigate the code.
- There is typically a learning curve for this pattern, and developers need to know multiple technologies.

Blackboard pattern:

Emerging from the world of 'Artificial Intelligence' (AI) development, the "Blackboard architecture pattern" is more of a stop-gap arrangement. Its' noticeable characteristics are as follows:

- When you deal with an emerging domain like AI or 'Machine Learning' (ML), you don't necessarily have a settled architecture pattern to use. You start with the blackboard pattern, subsequently, when the domain matures, you adopt a different architecture pattern.
- There are three components, namely, the blackboard, a collection of knowledge resources, and a controller.
- The application system stores the relevant information in the blackboard.
- The knowledge resources could be algorithms in the AI or ML context that collect information and updates the blackboard.

- The controller reads from the blackboard and updates the application ‘assets’, for e.g., robots.

Image recognition, speech recognition, etc. use this architecture pattern. It has a few advantages, as follows:

- The pattern facilitates experiments.
- You can reuse the knowledge resources like algorithms.

There are also limitations, for e.g.:

- It’s an intermediate arrangement. Ultimately, you will need to arrive at a suitable architecture pattern, however, you don’t have certainty that you will find the right answer.
- All communication within the system happens via the blackboard, therefore, the application can’t handle parallel processing.
- Testing can be hard.

Interpreter pattern:

A pattern specific to certain use cases, the “Interpreter pattern” deals with the grammar of programming languages. It offers an interpreter for the language. It works as follows:

- You implement an interface that aids in interpreting given contexts in a programming language.
- The pattern uses a hierarchy of expressions.
- It also uses a tree structure, which contains the expressions.
- A parser, external to the pattern, generates the tree structure for evaluating the expressions.

The use of this pattern is in creating “Classes” from symbols in programming languages. You create a grammar for the language, so that interpretation of sentences becomes possible. Network protocol languages and SQL uses this pattern.

Task 2:

An ER diagram gives you the visual outlook of your database. It details the relationships and attributes of its entities, paving the way for a smooth database development in the steps ahead.

EER diagrams, on the other hand, are perfect for taking a more detailed look at your information. When your database contains a larger amount of data it is best to turn to an enhanced model to more deeply understand your model.

So when should you use which? Honestly, both are useful, and it depends mostly on the size and detail of your data. The more complicated the data, the more likely you'll need to use an EER diagram to make sure you're properly organizing every relationship.

Task 3:

List of NoSQL Databases:

- MongoDB
- Cassandra
- Redis
- Couchbase
- Amazon DynamoDB
- Apache HBase
- Neo4j
- CouchDB
- Riak

- Apache Cassandra
- MarkLogic
- OrientDB
- ArangoDB
- RavenDB
- Couchbase Server

Task 4:

Power Query is a data transformation and integration tool that is part of Microsoft Power BI, Excel, and other Microsoft products. Here are some preferences and advantages associated with Power Query:

1. Data Transformation: Power Query allows users to transform and shape data from multiple sources. It provides a user-friendly interface for performing operations such as filtering, sorting, merging, appending, and pivoting data. This makes it easy to clean and prepare data for analysis or reporting.
2. Data Connectivity: Power Query supports a wide range of data sources, including databases, Excel files, CSV files, SharePoint lists, web pages, and more. It provides built-in connectors for popular data sources, enabling seamless data extraction and integration.
3. Query Folding: Power Query has the ability to push certain data transformation operations back to the data source, known as query folding. This can significantly improve performance by leveraging the processing power of the underlying data platform and reducing the amount of data transferred over the network.
4. Data Mashup: Power Query allows users to combine data from multiple sources into a single query. It supports data mashup scenarios where you can merge, join, or append data from different sources, making it easy to integrate data from various systems and create unified datasets.
5. Data Profiling and Quality: Power Query includes features for data profiling and data quality assessment. It can analyze the structure and content of the

data, identify data quality issues such as missing values or duplicates, and provide suggestions for data cleansing and transformation.

6. Reusability and Automation: Power Query allows users to create reusable data transformation steps and functions. These transformations can be saved as queries and applied to new data with a single click. Power Query also supports automation through scheduled data refreshes, ensuring that your data is always up to date.
7. User-Friendly Interface: Power Query provides a user-friendly interface with a visual editor, formula bar, and intuitive navigation. It allows users to interactively build and modify data transformation steps without the need for complex programming or scripting.
8. Integration with Power BI and Excel: Power Query is tightly integrated with Microsoft Power BI and Excel, providing seamless data integration and analysis capabilities. It allows users to load transformed data directly into Power BI models or Excel worksheets, enabling powerful data analysis and visualization.

Task 5:

	AWS	**Azure**	**GCP**
Market Share and Maturity	AWS is currently the largest and most mature cloud provider, with a significant market share and a wide array of services.	Azure is the second-largest cloud provider and has been rapidly gaining market share. It benefits from Microsoft's extensive enterprise presence and integration with other Microsoft products.	GCP is the third-largest cloud provider, with a smaller market share compared to AWS and Azure. However, it has been growing steadily and is known for its innovation and advanced technologies.

Services and Features	AWS has the most comprehensive service portfolio, offering a vast range of services in areas such as computing, storage, databases, networking, analytics, AI/ML, IoT, and more.	Azure provides a broad range of services, including computing, storage, databases, networking, analytics, AI/ML, IoT, and integration with Microsoft products like Office 365 and Dynamics 365.	GCP offers a diverse set of services, including computing, storage, databases, networking, big data, AI/ML, IoT, and specialized offerings like Google Kubernetes Engine and BigQuery for data analytics.
Global Infrastructure	AWS has the most extensive global infrastructure, with data centers in multiple regions worldwide, allowing for broad coverage and low-latency access.	Azure has a widespread global presence, with data centers in many regions, providing good global coverage and enabling compliance with local data regulations.	GCP has a growing global infrastructure, with data centers in multiple regions, providing decent global coverage. However, it has fewer regions compared to AWS and Azure.
Pricing and Cost	AWS offers various pricing models, including pay-as-you-go, reserved instances, and spot instances. It	Azure has competitive pricing options, including pay-as-you-go, reserved instances, and hybrid benefits	GCP also offers competitive pricing models, including on-demand pricing and sustained use discounts. It

	often provides more pricing options and flexibility.	for customers with Microsoft software licenses.	provides transparent pricing and a pricing calculator for cost estimation.
Integration and Ecosystem	AWS has a vast ecosystem of services, partner solutions, and integrations. It offers extensive APIs and SDKs, enabling seamless integration with third-party tools.	Azure benefits from strong integration with the Microsoft ecosystem, including Windows Server, Active Directory, Visual Studio, and other Microsoft products.	GCP has a growing ecosystem of services and integrations. It has strong ties to popular Google services like Google Cloud AI and Google Kubernetes Engine.
Enterprise Focus	AWS has a strong presence in the enterprise market, offering a wide range of services and features that cater to enterprise needs, including security, compliance, and hybrid cloud solutions.	Azure is well-positioned for enterprise customers, leveraging Microsoft's enterprise background and integration with existing Microsoft technologies.	GCP is increasingly focusing on the enterprise market, offering enterprise-grade services, strong security features, and compliance certifications.

Task 6:

RTOS stands for Real-Time Operating System. It is an operating system specifically designed to handle real-time applications that require precise and deterministic timing. Real-time systems are those that must respond to events or stimuli within strict timing constraints.
Here are some key characteristics and features of an RTOS:

1. Deterministic Response: An RTOS provides deterministic response times, meaning it guarantees that tasks or processes will be executed within specific time constraints. This is crucial for applications where timing is critical, such as industrial control systems, embedded systems, robotics, and aerospace systems.
2. Task Scheduling: An RTOS includes a scheduling mechanism that allows tasks to be prioritized and scheduled for execution based on their priority levels. Common scheduling algorithms used in RTOS include priority-based scheduling, round-robin scheduling, and rate-monotonic scheduling.
3. Interrupt Handling: An RTOS has efficient interrupt handling mechanisms to quickly respond to and service hardware interrupts. Interrupts are events generated by hardware devices that require immediate attention.
4. Resource Management: An RTOS manages system resources such as CPU time, memory, and peripherals. It provides mechanisms for task synchronization, inter-task communication, and resource sharing to ensure efficient utilization of system resources.
5. Kernel Services: RTOS kernels provide a set of services to manage tasks, scheduling, synchronization, and communication. These services may include task creation and deletion, context switching, semaphores, mutexes, message queues, and timers.
6. Small Footprint: RTOSs are designed to have a small memory footprint and low overhead to run efficiently on resource-constrained embedded systems.
7. Real-Time Constraints: An RTOS distinguishes between hard real-time and soft real-time requirements. Hard real-time systems have strict timing constraints, where missing a deadline can lead to catastrophic

consequences. Soft real-time systems have timing constraints, but occasional missed deadlines may not be critical.

8. Preemptive and Cooperative Multitasking: An RTOS supports multitasking, allowing multiple tasks to execute concurrently. It can implement either preemptive multitasking, where higher-priority tasks can interrupt lower-priority tasks, or cooperative multitasking, where tasks voluntarily yield the CPU to other tasks.

Popular examples of RTOSs include FreeRTOS, VxWorks, QNX, uC/OS, and ThreadX. These RTOSs provide a range of features, services, and optimizations to cater to different application requirements.