**Interview Questions asked in BarRaiser**

* 1. **DHP deployment**

* 1. **To onboard new API to DHP , environment.yml file is the first place to start with the changes. A new swimlane has to be identified here under the respective environment**
  2. **Swimlane is a logical group of all business applications that could be hosted on single or multiple EC2 instances**
  3. **The names for the swimlane and infrastructure definition should follow the naming convention rule**
  4. **Once swimlane defined then create/update specific yaml file depending on the security group in which Api needs to be hosted**
  5. **Define load balancer related configurations. Target group should be identified and health check endpoints also identified**
  6. **Once config are defined please raise PR. Once PR merged to master teamcity jobs corresponding to config defined will be created**
  7. **Next, provision the infrastructure as per config by running teamcity pipeline**
  8. **Also we need access to vault**
  9. **Finally deploy swimlane job will be triggered under the respective environment in teamcity**

* 1. **BIAN Architecture**

**BIAN Architecture Overview**

**BIAN is based on dividing banking operations into modular Service Domains, each representing a specific business capability. These domains interact with each other via standardized services and APIs, ensuring seamless communication.**

**Example: Loan Processing**

**Scenario:**

**A customer applies for a loan at a bank. The bank processes the loan application, assesses the customer’s creditworthiness, approves the loan, and disburses the amount.**

**Steps in the BIAN Architecture:**

**Visual Representation of Interaction (Conceptual):**

**Customer Offer → Collects loan application.**

**Loan Processing → Processes application.**

**Credit Risk Assessment → Evaluates the applicant.**

**Decisioning → Approves or rejects the application.**

**Payments Execution → Transfers the loan amount.**

**Loan Management → Tracks repayment.**

* 1. **What is ECS and EKS and EC2**

|  |  |  |
| --- | --- | --- |
| **ECS(Elastic Container Service)** | **EKS(Elastic Kubernetes Service)** | **EC2(Elastic Compute Cloud)** |
| **•Amazon ECS is a fully managed container orchestration service that makes it easy to deploy, manage, and scale containerized applications using Docker.** | **•Amazon EKS is a managed Kubernetes service that makes it easy to run Kubernetes on AWS without needing to install and operate your own Kubernetes control plane or nodes.** | **•Amazon EC2 provides resizable compute capacity in the cloud. It allows you to launch virtual servers (instances) and manage them as needed.** |
| **•ECS: Managed container orchestration service for Docker containers.** | **•EKS: Managed Kubernetes service for running Kubernetes clusters.** | **•EC2: Resizable virtual servers for running any workload with full control over the environment.** |
| **Scenario:**  **•        You have a banking application with multiple microservices, such as account management, transaction processing, and customer support, all packaged as Docker containers.**  **•        Deploy and manage these microservices using ECS.** | **Scenario:**  **•        Your banking application uses Kubernetes for container orchestration, and you want to leverage Kubernetes features like custom resource definitions (CRDs) and Helm charts.**  **•        Deploy and manage your Kubernetes cluster using EKS.** | **Scenario:**  **•        You need to run a legacy banking application that requires a specific operating system and custom configurations.**  **Launch and manage EC2 instances to run this legacy application.** |
| **Use ECS to manage containerized microservices for your banking application, such as account management and transaction processing.** | **Use EKS to leverage Kubernetes features for managing your banking application's microservices, ensuring high availability and scalability.** | **Use EC2 to run legacy banking applications that require specific configurations and full control over the environment** |

* 1. **Difference b/ Devops and Devsecops**

|  |  |  |
| --- | --- | --- |
| **Parameters** | **DevOps** | * + 1. **DevSecOps** |
| **Purpose** | **DevOps is primarily focused on increasing the speed and quality of software development and delivery.** | **DevSecOps aims to secure the software development process by integrating security early and throughout the software development life cycle.** |
| **Teams** | **Developers and operations teams work together.** | **Developers, operations teams, and security teams work together.** |
| **Processes** | **The processes in DevOps are typically continuous integration (CI) and continuous delivery (CD).** | **The processes in DevSecOps are typically CI/CD plus additional security-related processes.** |
| **Tools** | **Puppet, Chef, Ansible, Jenkins** | **Puppet, Chef, Ansible, Jenkins, & security-specific tools like Veracode, Burp Suite, OWASP ZAP Proxy, etc.** |
| **Vulnerabilities** | **Vulnerabilities are not always addressed throughout the development life cycle.** | **Vulnerabilities are addressed throughout the software development life cycle.** |
| **Definition** | **A set of practices combining software development (Dev) and IT operations (Ops).** | **Extends DevOps by integrating security (Sec) into the process from the start.** |
| **Focus** | **Shortening the development life cycle, continuous delivery with high software quality.** | **Integrating security practices within the DevOps process without compromising speed or agility.** |
| **Security Integration** | **Often considers security as a final step in development.** | **Embeds security practices throughout the software development life cycle.** |

* 1. **MVC architecture**

**MVC (Model-View-Controller) is a design pattern used for developing web applications. It separates an application into three main logical components: the Model, the View, and the Controller. This separation helps in organizing code, making it more modular, maintainable, and testable.**

**Components of MVC**

**Model:**

**Represents the data and the business logic of the application.**

**Interacts with the database and performs data-related operations.**

**Notifies the View of any data changes.**

**View:**

**Represents the presentation layer of the application.**

**Displays data to the user and sends user commands to the Controller.**

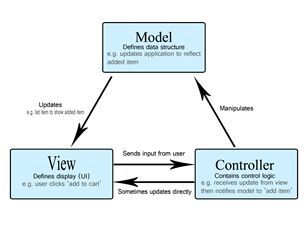
**Updates the UI based on changes in the Model.**

**Controller:**

**Acts as an intermediary between the Model and the View.**

**Handles user input and updates the Model.**

**Selects the View to display based on user actions and Model state.**

****

* 1. **Exception handling in MVC**
     1. **Exception handling is a crucial aspect of building robust and reliable web applications. In ASP.NET MVC, there are several ways to handle exceptions effectively to ensure that your application can gracefully recover from errors and provide meaningful feedback to users.**
     2. **Methods**
        1. **Try-catch block**
        2. **Custom error page**
        3. **Global exception handling**
        4. **Logging exception details in files**

* 1. **What layer does does forgate take care**

**AWS Fargate is a serverless compute engine for containers provided by Amazon Web Services (AWS). It allows users to run containerized applications without managing the underlying servers or infrastructure. With Fargate, you don't need to provision, configure, or scale clusters of virtual machines (VMs) to run containers, as the service handles these tasks automatically.**

**Key Features:**

**Serverless: No need to manage EC2 instances; AWS automatically provisions and scales the resources.**

**Scalability: Fargate automatically adjusts the required compute resources based on your application's needs.**

**Cost-Effective: You pay only for the resources you use (CPU and memory) while your containers are running.**

**Integration: Works with Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).**

**Security: Offers isolation between tasks, enhancing security by running each task in its own kernel.**

**AWS Fargate operates at the \*\*Container Orchestration Layer\*\* within the \*\*Infrastructure as a Service (IaaS)\*\* stack. Here's how it fits into the overall architecture:**

**1. \*\*Cloud Service Model\*\*:**

**- Fargate is part of IaaS because it provides underlying infrastructure resources like compute, storage, and networking but abstracts the management of these resources.**

**2. \*\*Container Ecosystem\*\*:**

**- Within the containerization stack, Fargate resides in the orchestration and execution layer. It runs containerized workloads orchestrated by either:**

**- \*\*Amazon Elastic Container Service (ECS)\*\*.**

**- \*\*Amazon Elastic Kubernetes Service (EKS)\*\*.**

**3. \*\*Abstraction Level\*\*:**

**- Fargate abstracts the \*\*infrastructure provisioning and management layer\*\* typically associated with virtual machines or servers. Developers only define:**

* 1. **Load balancer and auto scaling**

**Load Balancer**

**A load balancer is a system that distributes incoming network traffic across multiple servers or resources to ensure no single server becomes overwhelmed. It improves application reliability, scalability, and performance by evenly spreading the load.**

**In AWS, you can use the Elastic Load Balancer (ELB), which offers three types:**

**Application Load Balancer (ALB):**

* 1. **Best for HTTP/HTTPS traffic.**
  2. **Works at Layer 7 (Application Layer), providing features like host-based and path-based routing.**

**Network Load Balancer (NLB):**

* 1. **Best for high-performance and low-latency TCP/UDP traffic.**
  2. **Operates at Layer 4 (Transport Layer).**

**Gateway Load Balancer (GLB):**

* 1. **Designed for deploying and scaling virtual appliances.**

**Benefits of Load Balancers:**

* 1. **Distributes traffic evenly across instances.**
  2. **Automatically reroutes traffic to healthy instances.**
  3. **Provides scalability and fault tolerance.**
  4. **Offers SSL/TLS termination for security.**

**Auto Scaling**

**Auto Scaling is a feature that automatically adjusts the number of instances or resources in your environment based on demand. This ensures your application remains performant while optimizing costs by scaling resources up or down dynamically.**

**AWS provides Auto Scaling in two key areas:**

**Amazon EC2 Auto Scaling:**

* 1. **Adjusts the number of EC2 instances in an Auto Scaling group based on policies or metrics like CPU utilization.**

**Service Auto Scaling:**

* 1. **Scales AWS services like ECS, DynamoDB, and RDS based on predefined conditions.**

**Components of Auto Scaling:**

**Scaling Policies:**

* 1. **Defines rules for scaling (e.g., add/remove instances if CPU exceeds 80%).**

**Launch Template:**

* 1. **Specifies the configuration for new instances (e.g., instance type, AMI, security group).**

**Scaling Metrics:**

* 1. **Metrics like CPU usage, request count, or custom CloudWatch metrics trigger scaling.**

* 1. **How have you helped in improving performance in your application**

**Performance Testing**

**•        Conducting Load Testing: Simulate multiple users accessing the application simultaneously to identify performance issues under load.**

**•        Example: Using tools like Apache JMeter or Visual Studio Load Test to simulate user load and measure response times.**

**. Stress Testing**

**•        Identifying Breaking Points: Determine the application's breaking point by gradually increasing the load until the system fails.**

**•        Example: Using stress testing tools to push the application beyond its limits and observe its behavior.**

**Profiling and Monitoring**

**•        Using Profiling Tools: Identify performance bottlenecks by profiling the application during execution.**

**•        Example: Using Visual Studio Profiler to analyze CPU and memory usage.**

**Analyzing Logs**

**•        Reviewing Application Logs: Analyze logs to identify slow queries, exceptions, and other performance-related issues.**

**•        Example: Using log analysis tools like ELK Stack (Elasticsearch, Logstash, Kibana) to visualize and analyze logs.**

**Automated Performance Regression Testing**

**•        Ensuring Consistent Performance: Implement automated tests to ensure that performance improvements are maintained over time.**

**•        Example: Using NUnit or other testing frameworks to automate performance tests.**

* 1. **Github actions**

**GitHub Actions is a CI/CD (Continuous Integration/Continuous Deployment) and automation service provided by GitHub. It allows you to automate workflows directly within your GitHub repositories. These workflows are defined in YAML files and can be triggered by various events, such as pushing code, opening pull requests, or even on a schedule.**

**With GitHub Actions, you can automate tasks like:**

* 1. **Running tests**
  2. **Building and deploying code**
  3. **Code linting and formatting**
  4. **Packaging and publishing software**
  5. **Managing issues and pull requests**

**It integrates seamlessly with GitHub, making it a popular choice for developers looking to automate development pipelines and streamline their software lifecycle management.**

* 1. **How your application works, what are the different components in your application**

* 1. **How you feel testing experience will help in development**

**Testing plays a crucial role in development by ensuring the reliability, functionality, and quality of the software. Here’s how the testing experience can benefit development:**

**1. \*\*Early Bug Detection\*\*: Testing helps in identifying issues and bugs early in the development process, which can prevent them from becoming more complicated and harder to fix later.**

**2. \*\*Improved Code Quality\*\*: Through unit testing, integration testing, and code reviews, developers are forced to write cleaner, modular, and more maintainable code, leading to overall better software quality.**

**3. \*\*Faster Development Cycle\*\*: Automated tests allow for continuous feedback. Developers can run tests frequently to ensure that new changes don’t break existing functionality. This accelerates development and reduces the time spent on debugging.**

**4. \*\*Reduced Manual Effort\*\*: Automated tests eliminate the need for manual testing of repetitive tasks, allowing developers to focus more on writing new features.**

**5. \*\*Confidence in Refactoring\*\*: When developers know that they have a solid suite of tests in place, they can refactor code with more confidence, ensuring that new changes do not introduce bugs or regressions.**

**6. \*\*Better Collaboration\*\*: Test cases and results provide documentation for how the system is expected to behave, which aids collaboration among team members. Test failures can also highlight misunderstandings or mismatches in requirements.**

**7. \*\*Code Coverage\*\*: Testing provides metrics such as code coverage, which helps ensure that all parts of the application are being tested and increases the robustness of the application.**

**8. \*\*Continuous Improvement\*\*: Testing encourages a culture of continuous integration and delivery (CI/CD), where code is constantly integrated, tested, and deployed, leading to more efficient development workflows.**

**In summary, testing enhances development by ensuring software reliability, promoting best practices, reducing manual work, and allowing for continuous improvement.**

* 1. **Infrastructure of underlying api Architecture of your application**

* 1. **Synchronous and Asynchronous**

**Synchronous**

**Definition:**

**•        Synchronous programming means that tasks are executed one after another. Each task must complete before the next one starts.**

**Characteristics:**

**•        Blocking: The execution of the program is blocked until the current task is completed.**

**•        Sequential: Tasks are performed in a specific order, and each task waits for the previous one to finish.**

**Example (Banking - Synchronous):**

**Imagine you are at a bank and want to withdraw money. In a synchronous banking system:**

* 1. **You approach the bank teller (the system) and request a withdrawal.**
  2. **The teller first checks your account balance to ensure you have sufficient funds.**
  3. **Then, the teller processes the withdrawal and hands you the cash.**
  4. **You have to wait until the teller finishes all of these steps before you receive the money and leave the bank.**

**Here, the process is blocking because you have to wait for each step (checking the balance, processing the withdrawal) to complete before you can proceed.**

**2. Asynchronous (Non-blocking)**

**Definition:**

**•        Asynchronous programming allows tasks to run independently of the main program flow. Tasks can start and complete at different times without blocking the execution of other tasks.**

**Characteristics:**

**•        Non-blocking: The execution of the program continues while the task is being performed.**

**•        Concurrent: Multiple tasks can be in progress simultaneously, improving responsiveness and performance.**

**Example (Banking - Asynchronous):**

**Now, imagine a scenario where you are performing a banking transaction online through a mobile app with asynchronous behavior:**

* 1. **You request to withdraw money through the app.**
  2. **Instead of waiting for the system to check your balance and process the withdrawal sequentially, the app sends a request to the bank’s server, which starts the balance check in the background.**
  3. **While the system checks your balance, you could be doing other things, like checking your account history or sending another request to pay a bill.**
  4. **Once the balance check is done, the system notifies you (via a callback or a notification) that your withdrawal was successful and that the money is ready.**

**In this asynchronous scenario, the system doesn't block you from performing other actions while it's processing the withdrawal request. It efficiently handles multiple tasks at the same time, improving the overall user experience and system performance.**

**Key Differences**

**1.        Execution Flow:**

**•        Synchronous: Tasks are executed sequentially, one after another.**

**•        Asynchronous: Tasks can be executed concurrently, allowing other tasks to run while waiting for a task to complete.**

**2.        Blocking vs. Non-blocking:**

**•        Synchronous: The program waits (blocks) for a task to complete before moving on to the next task.**

**•        Asynchronous: The program does not wait (non-blocking) and can continue executing other tasks.**

**3.        Performance:**

**•        Synchronous: Can lead to performance bottlenecks, especially for I/O-bound operations (e.g., file I/O, network requests).**

**•        Asynchronous: Improves performance and responsiveness, especially for I/O-bound operations, by allowing other tasks to run concurrently.**

* 1. **How do you integrate automation in CI/CD pipeline?**

**By defining a GitHub Actions workflow, you can automate the CI/CD pipeline for your .NET project. This includes steps for building, testing, and deploying your application, ensuring a streamlined and efficient development process. The use of secrets ensures that sensitive information is securely managed. This approach helps maintain code quality and accelerates the delivery of new features and fixes.**

* 1. **What are microservices and what is your approach to develop a microservice?**

**Microservices are a software architectural style where an application is composed of small, loosely coupled, independently deployable services. Each service focuses on a specific business functionality and communicates with other services over lightweight protocols, such as HTTP/REST, gRPC, or messaging queues.**

**Approach to Developing Microservices**

**Developing microservices involves a systematic approach to ensure scalability, reliability, and maintainability.**

**1. Understand Business Requirements**

* 1. **Identify the core functionalities of the system and split them into smaller, manageable business domains.**
  2. **Define the boundaries of each service using the Single Responsibility Principle (SRP).** 
     1. **Example: In an e-commerce app, services might include Order Service, Inventory Service, and User Service.**

**2. Design the Architecture**

* 1. **API Gateway: Acts as an entry point for external clients, handling authentication, routing, and load balancing.**
  2. **Service Communication: Decide between synchronous (REST/gRPC) or asynchronous (message queues like Kafka, RabbitMQ) communication.**
  3. **Database Design:** 
     1. **Each service should have its own database (Database per service) to avoid tight coupling.**
     2. **Use event-driven architecture for data consistency (e.g., using events for eventual consistency).**

**3. Choose the Technology Stack**

* 1. **Programming Languages: Choose languages based on the team's expertise or service requirements (e.g., Java, Python, Node.js, Go).**
  2. **Frameworks: Use lightweight frameworks like Spring Boot (Java), Flask (Python), or Express.js (Node.js).**
  3. **Databases: Select databases based on use cases (e.g., relational databases like PostgreSQL for transactions, NoSQL databases like MongoDB for unstructured data).**

**4. Implement the Microservices**

* 1. **Define APIs: Create well-documented APIs for communication between services using tools like Swagger/OpenAPI.**
  2. **Service Isolation: Ensure services are developed independently without relying on internal workings of other services.**
  3. **Error Handling: Implement retries, circuit breakers (using libraries like Resilience4j), and fallback mechanisms to improve resilience.**
  4. **Logging and Monitoring: Use centralized tools like ELK Stack, Prometheus, or Grafana for logs and metrics.**

**5. Secure the Services**

* 1. **Authentication: Use standards like OAuth2 or JWT for securing APIs.**
  2. **Authorization: Implement role-based or fine-grained access control.**
  3. **Encrypt sensitive data both in transit (e.g., HTTPS) and at rest.**

**6. Deploy and Scale**

* 1. **Containerization: Use Docker to containerize services for consistency across environments.**
  2. **Orchestration: Use Kubernetes or Docker Swarm for service deployment, scaling, and management.**
  3. **CI/CD Pipelines: Automate testing and deployment using tools like GitHub Actions, Jenkins, or CircleCI.**

**7. Test the Microservices**

* 1. **Unit Testing: Ensure individual functionalities work as expected.**
  2. **Integration Testing: Verify communication between services.**
  3. **End-to-End Testing: Test the entire application flow.**
  4. **Performance Testing: Test the system’s ability to handle load (using tools like JMeter or Gatling).**

* 1. **How do you manage synchronous & asynchronous calls? Also, what would you test these endpoints?**

**Managing synchronous and asynchronous calls in an API involves designing the API's behavior based on the requirements and the nature of the tasks it handles. Here's how you can manage them effectively and test their functionality.**

**Managing Synchronous Calls**

**Synchronous calls wait for a response before proceeding. These are typically used for:**

* 1. **Simple operations (e.g., CRUD operations).**
  2. **Scenarios where the result is needed immediately.**

**How to Manage**

**Process Sequentially:**

* 1. **The server processes the request and sends a response only after completing the task.**
  2. **Example: Fetching user data from a database.**

**Error Handling:**

* 1. **Implement robust error handling for predictable failures.**
  2. **Use HTTP status codes to communicate success (200 OK) or failure (400 Bad Request, 500 Internal Server Error).**

**Timeouts:**

* 1. **Set reasonable timeouts for requests to prevent the client from hanging indefinitely.**

**Asynchronous calls allow other processes to continue while waiting for a task to complete. These are used for:**

* 1. **Long-running tasks (e.g., processing large datasets, sending emails).**
  2. **Non-blocking operations (e.g., using message queues).**

**How to Manage**

**Callbacks or Promises:**

* 1. **Use callbacks, Promises, or async/await for handling asynchronous operations.**
  2. **Example: Querying an external API or processing background tasks.**

**Queues for Long-Running Tasks:**

* 1. **Offload intensive tasks to a queue (e.g., RabbitMQ, Kafka) and process them asynchronously.**
  2. **Example: Sending an email after user registration.**

**Webhooks or Polling for Notifications:**

* 1. **Notify the client when the task completes via webhooks or have the client poll periodically.**

* 1. **How do you restrict the api endpoint for users or public availablity?**

**Restricting API endpoints to specific users or limiting public access is essential for securing your application. This can be achieved through various methods, depending on the level of access control and the nature of the application. Here are some approaches:**

**1. Authentication**

**Authentication ensures that only identified users can access your API.**

**Common Methods:**

**API Keys:**

* 1. **Generate unique keys for each user or application.**
  2. **Require the key to be passed with each request (e.g., as a header or query parameter).**
  3. **Validate the key on the server.**

**OAuth2:**

* 1. **Use OAuth2 for secure user authentication.**
  2. **Provide access tokens after user login.**
  3. **Validate the token with each request.**

**JWT (JSON Web Token):**

* 1. **Authenticate users and generate a JWT on successful login.**
  2. **Validate the token's signature and claims (e.g., expiry, user roles) for each request.**

**2. Authorization**

**Authorization determines what resources a user can access.**

**Role-Based Access Control (RBAC):**

* 1. **Define roles (e.g., admin, user, guest) and assign permissions to roles.**
  2. **Check the user's role before processing the request.**

**3. IP Whitelisting**

**Restrict access based on the client's IP address.**

**4. CORS (Cross-Origin Resource Sharing)**

**Restrict API access to specific domains.**

**5.Rate Limiting**

**Prevent abuse by limiting the number of requests from a user or IP.**

* 1. **How do you implement CORS?**

**CORS (Cross-Origin Resource Sharing) is a security feature implemented in web browsers to control how resources on a web page can be requested from another domain (origin). It ensures that a web page can only request data from the same origin unless the server explicitly allows access to its resources from other origins.**

**Key Terms in CORS**

**Origin: A combination of protocol (HTTP/HTTPS), domain, and port.**

**Example:** [**https://example.com:8080**](https://example.com:8080) **is a different origin from** [**http://example.com**](http://example.com) **or** [**https://anotherdomain.com**](https://anotherdomain.com)**.**

**Same-Origin Policy: A security feature that blocks web applications from making requests to a different origin than the one that served the web page.**

**Preflight Request: For certain types of HTTP requests (e.g., PUT, DELETE, or custom headers), the browser sends an OPTIONS request first to check if the actual request is allowed.**

**How CORS Works**

**When a browser makes a cross-origin request:**

**It sends an Origin header with the request to inform the server about the source of the request.**

**The server responds with specific CORS headers if it allows the request.**

* 1. **When do we see null pointer exception and object reference is not set to instance of an object**

**A NullReferenceException occurs when you try to access a member on an object that is null. Common scenarios include uninitialized objects, dereferencing null objects, collections with null values, and methods returning null. To avoid these exceptions, always check for null, use the null-conditional operator, ensure proper initialization, and provide default values. These practices help ensure your code is robust and less prone to runtime errors.**

**How to Avoid Null Reference Exceptions**

**1.        Check for Null:**

**Always check if an object is null before accessing its members.**

**2.        Use Null-Conditional Operator:**

**Use the null-conditional operator (?.) to safely access members.**

**3.        Initialize Objects:**

**Ensure objects are properly initialized before use.**

**4.        Use Default Values:**

**Provide default values to avoid null references.**

* 1. **What is Routing**

**Routing in AWS (Amazon Web Services) refers to the process of directing network traffic to various resources within the AWS cloud. This is crucial for ensuring that requests reach the correct services and resources, such as EC2 instances, load balancers, or other AWS services. AWS provides several services and features to manage routing effectively.**

**Key AWS Services for Routing**

**1.        Amazon Route 53:**

**A scalable Domain Name System (DNS) web service designed to route end-user requests to AWS resources.**

**Supports various routing policies like simple, weighted, latency-based, failover, and geolocation routing.**

**2.        Elastic Load Balancing (ELB):**

**Distributes incoming application or network traffic across multiple targets, such as EC2 instances, containers, and IP addresses.**

**Supports different types of load balancers: Application Load Balancer (ALB), Network Load Balancer (NLB), and Gateway Load Balancer (GLB).**

**3.        Amazon VPC (Virtual Private Cloud):**

**Allows you to create a logically isolated network within the AWS cloud.**

**Includes features like route tables, internet gateways, and NAT gateways to control routing of traffic within the VPC.**

**4.        AWS Transit Gateway:**

**A network transit hub that you can use to interconnect your VPCs and on-premises networks.**

**Simplifies network architecture by consolidating routing through a single gateway.**

* 1. **Lambda and Athena in AWS Services**

**AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers. You can execute your code in response to events such as changes to data in an Amazon S3 bucket, updates to a DynamoDB table, or HTTP requests via Amazon API Gateway.**

**Key Features of AWS Lambda**

**1.        Event-Driven: Automatically runs your code in response to events.**

**2.        Scalability: Automatically scales your application by running code in response to each trigger.**

**3.        Cost-Effective: You pay only for the compute time you consume.**

**4.        Supports Multiple Languages: Including Node.js, Python, Ruby, Java, Go, .NET Core, and custom runtimes.**

**Amazon Athena**

**Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries you run.**

**Key Features of Amazon Athena**

**1.        Serverless: No need to manage any infrastructure.**

**2.        Standard SQL: Use familiar SQL syntax to query data.**

**3.        Integration with S3: Directly query data stored in Amazon S3.**

**4.        Pay-Per-Query: Only pay for the queries you run.**

**### \*\*Comparison: Lambda vs. Athena\*\***

**| Feature                | \*\*AWS Lambda\*\*                                   | \*\*Amazon Athena\*\*                             |**

**|------------------------|--------------------------------------------------|----------------------------------------------|**

**| \*\*Purpose\*\*            | Run custom code in response to events.           | Query and analyze data stored in S3 using SQL. |**

**| \*\*Type\*\*               | Serverless compute service.                      | Serverless query service.                     |**

**| \*\*Trigger\*\*            | Event-driven (S3, DynamoDB, API Gateway, etc.).  | Query-based (manually or scheduled queries).  |**

**| \*\*Input\*\*              | Data passed as events or parameters.             | Data stored in S3 (structured/semi-structured). |**

**| \*\*Output\*\*             | Processed data, API responses, or events.        | Query results in tabular form (or exported).  |**

**| \*\*Languages\*\*          | Supports multiple programming languages.          | SQL only.                                     |**

**| \*\*Typical Use Case\*\*   | Processing, automation, and microservices.        | Data analytics and ad-hoc querying.          |**

* 1. **Dependency injection in mvc**

**Dependency Injection (DI) is a design pattern used to implement Inversion of Control (IoC) between classes and their dependencies. It allows for better modularity, testability, and maintainability of code by decoupling the creation of dependencies from their usage.**

**Key Concepts**

**Service: A class that provides functionality to other classes.**

**Client: A class that depends on the service.**

**Injector: A class or framework that injects the service into the client.**

**Types of Dependency Injection**

**Constructor Injection - In constructor injection, dependencies are provided through a class constructor.**

**Property Injection - dependencies are provided through public properties of the client class**

**Method Injection - dependencies are provided through method parameters**

* 1. **How do APIs between two domains communicate**

**APIs between two domains communicate over the internet using HTTP/HTTPS protocols. This communication typically involves making HTTP requests from one domain to another and receiving responses. Here are the key concepts and steps involved in enabling such communication:**

**Key Concepts**

**1.        CORS (Cross-Origin Resource Sharing):**

**•        CORS is a security feature implemented by web browsers to control how resources on one domain can be requested from another domain.**

**•        The server hosting the API must include specific headers in its responses to allow cross-origin requests.**

**2.        HTTP Methods:**

**•        Common HTTP methods used in API communication include GET, POST, PUT, DELETE, etc.**

**3.        Authentication and Authorization:**

**•        APIs often require authentication (e.g., API keys, OAuth tokens) to ensure that only authorized clients can access the resources.**

* 1. **What's your understanding on API**

**An API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other. APIs define the methods and data formats that applications can use to request and exchange information.**

**Key Concepts of APIs**

**1.        Endpoints:**

**•        Specific URLs where API services are accessed. Each endpoint corresponds to a specific function or resource.**

**2.        HTTP Methods:**

**•        Common methods include GET (retrieve data), POST (submit data), PUT (update data), DELETE (remove data).**

**3.        Request and Response:**

**•        Request: Sent by the client to the server, includes method, endpoint, headers, and body.**

**•        Response: Sent by the server to the client, includes status code, headers, and body.**

**4.        Status Codes:**

**•        Indicate the result of the request (e.g., 200 OK, 404 Not Found, 500 Internal Server Error).**

**5.        Authentication and Authorization:**

**•        Ensures that only authorized users can access the API. Common methods include API keys, OAuth tokens, and JWT (JSON Web Tokens).**

**6.        Data Formats:**

**•        Common formats for data exchange include JSON (JavaScript Object Notation) and XML (eXtensible Markup Language).**

* 1. **Kafka services**

**Apache Kafka is an open-source distributed event streaming platform used for building real-time data pipelines and streaming applications. It is designed to handle high throughput and low latency data streams, making it suitable for a variety of use cases such as log aggregation, real-time analytics, and event sourcing.**

**Key Concepts of Kafka**

**1.        Producer:**

**•        An application that sends messages to Kafka topics.**

**2.        Consumer:**

**•        An application that reads messages from Kafka topics.**

**3.        Topic:**

**•        A category or feed name to which records are sent by producers. Topics are partitioned and replicated across multiple brokers.**

**4.        Partition:**

**•        A topic is divided into partitions, which are the basic unit of parallelism in Kafka.**

**5.        Broker:**

**•        A Kafka server that stores data and serves clients. Multiple brokers form a Kafka cluster.**

**6.        Zookeeper:**

**•        A centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. Kafka uses Zookeeper to manage and coordinate the Kafka brokers.**

* 1. **Difference Between Authentication & Authorization**

**Authentication**

**Authentication is required to ensure that the API is well secured to protect crucial data**

**There are multiple ways to authenticate API**

* 1. **Basic auth**
  2. **OAuth - Allows third-party applications to access user data without exposing credentials**
  3. **API key**
  4. **IDP token**

**Authorization**

**Ensure that the authenticated users have the necessary permission to access the specific resource or perform action**

* 1. **What is Group IDP and how does it work**

**Group IDP (Identity Provider) is a service that manages user identities and provides authentication and authorization services. It allows users to log in to multiple applications using a single set of credentials. Group IDP typically supports features like Single Sign-On (SSO), Multi-Factor Authentication (MFA), and user provisioning.**

**How Group IDP Works**

**1.        User Authentication:**

**•        Users authenticate themselves to the IDP using their credentials (e.g., username and password, biometric data).**

**•        The IDP verifies the credentials and, if valid, issues an authentication token.**

**2.        Single Sign-On (SSO):**

**•        Once authenticated, users can access multiple applications without needing to log in again.**

**•        The IDP provides an SSO token that applications can use to verify the user's identity.**

**3.        Authorization:**

**•        The IDP manages user roles and permissions.**

**•        Applications can query the IDP to determine what actions a user is authorized to perform.**

**4.        Federation:**

**•        The IDP can federate identities across different domains or organizations.**

**•        This allows users to access resources in different domains using their primary credentials.**

* 1. **Difference Between .Net core and MVC**

**.NET Core**

**.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, and internet-connected applications. It is a general-purpose development platform maintained by Microsoft and the .NET community on GitHub.**

**ASP.NET MVC**

**ASP.NET MVC is a web application framework developed by Microsoft, based on the Model-View-Controller (MVC) architectural pattern. It is part of the ASP.NET framework and is used for building dynamic, data-driven web applications.**

**Cross-platform compatibility: ASP.NET Core can be used on Windows, Mac, or Linux, while ASP.NET MVC is limited to Windows1**[**2**](https://www.bing.com/ck/a?!&&p=a8fd31db5119788acb12d2bb171cea6121ec8d89aa1892ae687df875df9de66dJmltdHM9MTczNjk4NTYwMA&ptn=3&ver=2&hsh=4&fclid=1ed0bd55-1e8a-6a8c-1a5a-a97b1fb96b2f&psq=Difference+Between+.Net+core+and+MVC&u=a1aHR0cHM6Ly93d3cuemVuZXN5cy5jb20vYXNwLW5ldC1jb3JlLXZzLWFzcC1uZXQtbXZjLTU&ntb=1)[**3**](https://www.bing.com/ck/a?!&&p=44a3ebfeb9f02f0a85fc9d48f94f58d4a66fe4cc0ca67d3adab8b4acfb3c42efJmltdHM9MTczNjk4NTYwMA&ptn=3&ver=2&hsh=4&fclid=1ed0bd55-1e8a-6a8c-1a5a-a97b1fb96b2f&psq=Difference+Between+.Net+core+and+MVC&u=a1aHR0cHM6Ly93d3cuZWJpem5lZWRzLmNvbS9ibG9nL2FzcC1uZXQtY29yZS12cy1hc3AtbmV0LW12Yy8&ntb=1)**.**

**Lightweight and portable: ASP.NET Core is lighter and more portable than ASP.NET MVC1.**

**Optimized for use with ASP.NET Core: ASP.NET Core MVC is specifically designed for building web apps and APIs using ASP.NET Core**

* 1. **Monolith architecture vs Microservices**

**. Monolith Architecture**

* 1. **Definition: A monolithic architecture is a single, unified software system where all components (UI, business logic, and database access) are tightly integrated and run as a single process.**
  2. **Features:** 
     1. **Simple to develop and deploy.**
     2. **All functionalities are bundled together in one codebase.**
     3. **Scaling is done as a whole unit (vertical scaling).**
     4. **Any change requires redeploying the entire application.**
  3. **Example: Imagine an e-commerce platform where:** 
     1. **The user interface, order processing, inventory management, and payment systems are all part of a single codebase.**
     2. **If you want to update the payment logic, you have to redeploy the entire application.**
  4. **Use Case: Suitable for small applications or startups with limited resources and simple requirements.**

**2. Microservices Architecture**

* 1. **Definition: A microservices architecture breaks down an application into small, independent services. Each service is responsible for a specific functionality and communicates with others through APIs.**
  2. **Features:** 
     1. **Modular and loosely coupled.**
     2. **Each service can be developed, deployed, and scaled independently.**
     3. **Promotes the use of different technologies for different services.**
     4. **Better fault isolation (failure in one service doesn’t bring down the whole system).**
  3. **Example: In the same e-commerce platform, the system is divided into microservices like:** 
     1. **User Service: Manages user accounts.**
     2. **Order Service: Handles order processing.**
     3. **Inventory Service: Tracks product stock.**
     4. **Payment Service: Manages payment processing.**
     5. **Each service runs independently, possibly on different servers, and communicates via APIs.**
  4. **Use Case: Suitable for complex, large-scale applications with high scalability, flexibility, and teams working on different functionalities.**

* 1. **Difference b/w AWS and DHP**

**Dynamic Hosting Platform (DHP)**

**A Dynamic Hosting Platform typically refers to a hosting environment designed to scale dynamically based on traffic or resource demands. While it's not a specific branded service like AWS, it often describes platforms that offer dynamic scaling, resource allocation, and automated hosting features.**

**Key Features:**

**Dynamic Scaling: Automatically adjusts resources like CPU, memory, and storage based on demand.**

**Resource Optimization: Optimizes usage to ensure cost efficiency.**

**Managed Hosting: May provide pre-configured solutions for applications.**

**Simplified Management: Offers a more straightforward interface for managing hosting requirements.**

**Examples:**

* 1. **Managed hosting providers that offer dynamic scaling, such as Heroku or DigitalOcean's App Platform.**

**Amazon Web Services (AWS)**

**AWS is a comprehensive cloud platform offering a wide range of services, including compute power, storage, databases, machine learning, analytics, and more.**

**Key Features:**

**Scalability: Highly scalable infrastructure to handle workloads of any size.**

**Wide Service Portfolio: Offers 200+ fully-featured services (e.g., EC2, Lambda, S3, RDS).**

**Global Availability: Operates in multiple regions and availability zones worldwide.**

**Pay-as-You-Go Pricing: Charges based on usage, with options for reserved capacity.**

**Flexibility: Supports various operating systems, frameworks, and languages.**

**Examples:**

* 1. **EC2: Virtual servers in the cloud.**
  2. **S3: Scalable object storage.**
  3. **Lambda: Serverless computing to run code without managing servers.**

* 1. **Kill switch vs Feature switch**

**A Kill Switch is a mechanism used to immediately disable a feature or service in a system. It is typically used in emergency situations to prevent potential damage, security breaches, or to stop malfunctioning features.**

**Key Characteristics:**

**•        Purpose: To quickly disable a feature or service in case of emergencies.**

**•        Usage: Used when a feature is causing critical issues or when a security vulnerability is detected.**

**A Feature Switch (also known as a Feature Toggle or Feature Flag) is a technique used to enable or disable features in a system without deploying new code. It allows developers to control the availability of features dynamically, often used for gradual rollouts, A/B testing, or canary releases.**

**Key Characteristics:**

**•        Purpose: To control the availability of features dynamically.**

**•        Usage: Used for gradual rollouts, A/B testing, canary releases, or to enable/disable features based on user segments.**

* 1. **What is stateful and stateless request give example**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Stateful Requests** | **Stateless Requests** |
| **State Retention** | **Server retains client-specific state** | **Server does not retain client-specific state** |
| **Independence** | **Requests depend on previous interactions** | **Each request is independent** |
| **Scalability** | **Harder to scale due to state management** | **Easier to scale due to stateless nature** |
| **Complexity** | **Requires maintaining sessions or state data** | **Simpler architecture** |
| **Example** | **Online banking, shopping carts** | **REST APIs, stock price lookups** |

* 1. **What is session? why do we need to maintain it?**

**A session is a way to store information (state) about a user's interaction with a web application across multiple requests. It allows the server to remember the user's data and preferences as they navigate through the application. Sessions are typically used to maintain user-specific data, such as login status, shopping cart contents, and user preferences.**

* 1. **CRUD Operations using HTTP Verbs**

**CRUD operations (Create, Read, Update, Delete) are fundamental actions performed on data in a web application. These operations are typically mapped to HTTP verbs (methods) to interact with RESTful APIs. Each HTTP verb corresponds to a specific type of CRUD operation.**

* 1. **Green Lane and Blue lane (Deployment Lanes)**

**Green Lane typically refers to the current production environment that is actively serving live traffic. It is the stable version of the application that users are currently interacting with.**

**Blue Lane refers to the new version of the application that is being prepared for deployment. It is the environment where the new version is tested and validated before it is promoted to production.**

* 1. **Middleware in ASP.Net core**

**Middleware in ASP.NET Core is software that is assembled into an application pipeline to handle requests and responses. Each component in the pipeline can either process an incoming request, modify it, pass it to the next component, or handle the response.**

**Key Characteristics:**

**•        Request Processing: Middleware components are executed in the order they are added to the pipeline.**

**•        Response Processing: Middleware can also handle responses, modifying them before they are sent back to the client.**

**•        Chaining: Middleware components are chained together, and each component decides whether to pass the request to the next component or handle it directly.**

**How Middleware Works**

**1.        Incoming Request: When a request comes in, it passes through each middleware component in the order they are registered.**

**2.        Processing: Each middleware component can perform operations on the request, such as authentication, logging, or routing.**

**3.        Next Middleware: After processing, the middleware can pass the request to the next component in the pipeline.**

**4.        Response: Once the request is processed by the final component, the response travels back through the middleware pipeline in reverse order, allowing each component to modify the response if needed.**

* 1. **How will you scale the API**

**Scaling an API involves ensuring that it can handle increasing loads (traffic, data, or requests) efficiently while maintaining performance and reliability. Here's a step-by-step approach to scaling an API:**

**\*\*1. Vertical Scaling (Scale-Up)\*\***

**- \*\*Increase Server Resources:\*\* Upgrade the server hardware (e.g., more CPU, RAM, or faster disks) hosting the API.**

**\*\*2. Horizontal Scaling (Scale-Out)\*\***

**- \*\*Add More Servers:\*\* Distribute API traffic across multiple servers.**

**\*\*3. Caching\*\***

**- \*\*Response Caching:\*\* Use tools like Redis, Memcached, or HTTP caching headers to cache API responses.**

**\*\*4. Database Scalability\*\***

**- \*\*Read/Write Separation:\*\* Use primary-replica architecture for databases, directing reads to replicas and writes to the primary instance.**

**\*\*5. Microservices Architecture\*\***

**- \*\*Decompose Monoliths:\*\* Break your API into smaller, independently deployable services to improve scalability and maintainability.**

**\*\*6. Autoscaling\*\***

**- \*\*Cloud Providers:\*\* Use autoscaling features from cloud platforms like AWS, Azure, or Google Cloud to add/remove instances based on traffic.**

**- \*\*Kubernetes:\*\* Orchestrate containerized API deployments with Kubernetes, which supports horizontal pod autoscaling.**

* 1. **How will you fix the performance issue of the API**

**To fix performance issues in an ASP.NET Core API, you can use a combination of profiling and monitoring, database optimization, caching, optimizing middleware, asynchronous programming, and implementing rate limiting and throttling. By systematically identifying and addressing performance bottlenecks, you can ensure that your API remains performant and reliable under varying load conditions.**

* 1. **What are all the different authentication that CBA use**

**We use oauth for our service api.. each endpoint is configured to have a scope which we create through api catalogue.. any client that needs to connect to the endpoint should first get the client created in api catalogue and then it should be mapped to the scope associated with the endpoint. Using the client id and scope, the client needs to get the token from oauth server, and that token is used as a bearer token to authenticate to the endpoint..**

**For the UI, we use cookie authentication.. we have an authentication middleware which routes to the netbank login page if auth cookie is not found.. the netbank authentication server will generate the cookie and route the request back to our UI..**

**Username and Password**

**API Key**

**Bearer token**

* 1. **oAuth vs IDP**

**OAuth and Identity Providers (IdP) are distinct yet related concepts in authentication and authorization systems. Here's a breakdown of their differences and roles:**

**1. What is OAuth?**

**OAuth is an authorization framework that allows third-party applications to access a user’s resources on another service without exposing the user's credentials.**

* 1. **Purpose: OAuth is designed for delegated authorization, not authentication. However, OAuth 2.0 is often used in combination with OpenID Connect (OIDC) for authentication.**
  2. **How it Works:**
     1. **Users grant permission to a third-party app to access specific resources (e.g., access your Google Drive files).**
     2. **The third-party app receives a token (access token) that it uses to access resources on behalf of the user.**
  3. **Key Components:**
     1. **Resource Owner: The user who owns the data.**
     2. **Client: The app requesting access to the resource.**
     3. **Authorization Server: Issues access tokens after user consent.**
     4. **Resource Server: Hosts the protected resources (e.g., an API).**

**2. What is an Identity Provider (IdP)?**

**An Identity Provider is a service that authenticates users and provides information about their identity to applications. It acts as a source of truth for user credentials and identity.**

* 1. **Purpose: IdPs are responsible for authentication and identity management, often using standards like OAuth, OpenID Connect (OIDC), or SAML.**
  2. **How it Works:**
     1. **Users log in via the IdP.**
     2. **The IdP authenticates the user and provides an identity token or assertion (e.g., JWT, SAML assertion) to the requesting application.**
     3. **The application uses this identity information to verify the user and potentially fetch additional details.**
  3. **Examples of IdPs:**
     1. **Google, Microsoft Azure AD, Okta, Ping Identity, Auth0.**

* 1. **Different load balancers**

**AWS provides several types of load balancers to meet different use cases:**

**•        Application Load Balancer (ALB): Best for HTTP/HTTPS traffic and advanced routing.**

**•        Network Load Balancer (NLB): Best for TCP/UDP traffic requiring high performance and low latency.**

**•        Gateway Load Balancer (GWLB): Best for deploying and managing third-party virtual appliances.**

**•        Classic Load Balancer (CLB): Basic load balancing for legacy applications.**

* 1. **What are all the different monitoring tools available in AWS and which one CBA uses (Observe )**

**Observe and Splunk**

**•        Observe is focused on observability, providing a unified platform for monitoring, logging, and tracing, with an emphasis on ease of use and quick insights for DevOps and SRE teams.**

**•        Splunk is a comprehensive data analytics platform that handles a wide range of data types, offering powerful search, analysis, and visualization capabilities for IT operations, security, business analytics, and more.**

* 1. **How your API automation improved the application performance or consumer usage**

**PI automation can significantly enhance application performance and consumer usage by streamlining processes, reducing errors, and improving efficiency. Here’s an explanation with an example:**

**Key Ways API Automation Improves Performance and Usage:**

**Efficiency and Speed:**

**Automated APIs eliminate the need for manual intervention, reducing delays in data processing and response times. This improves application responsiveness and ensures users have a seamless experience.**

**Error Reduction:**

**Automated API testing ensures that the application’s APIs are working as intended, detecting and fixing bugs early. A reliable API improves the consumer's trust in the application.**

**Scalability:**

**Automated APIs can handle a large number of requests, adapting to increased traffic without degrading performance.**

**Integration and Interoperability:**

**Automated APIs simplify communication between systems, enabling faster data exchanges and improved functionality across integrated platforms.**

**Example: E-commerce Application**

**Imagine an e-commerce app where consumers search for products, add them to their cart, and complete payments.**

**Scenario Without Automation:**

* 1. **API for product availability: Manually tested periodically, increasing the risk of errors.**
  2. **API for payment gateway: If not well-optimized or tested automatically, may cause failures during high traffic, leading to consumer frustration.**

**Scenario With Automation:**

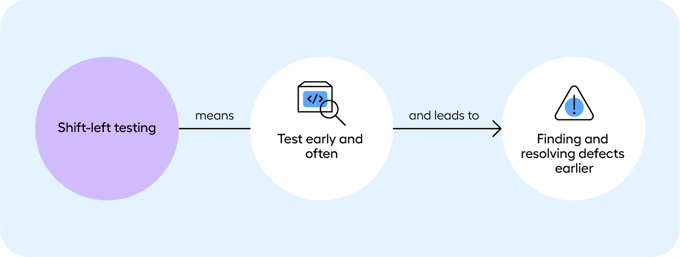
* 1. **Inventory Management Automation: APIs are automated to instantly check product availability in real-time. If a product is out of stock, the system informs users immediately, preventing unnecessary steps in the checkout process.**
  2. **Payment Gateway Testing Automation: Automated API tests ensure the payment system can handle multiple transactions simultaneously during a sale. It also verifies integrations with third-party gateways, ensuring faster, secure payments.**

**Outcome:**

* 1. **Faster checkout times and smoother transactions.**
  2. **Fewer errors during high traffic (e.g., flash sales).**
  3. **Improved user satisfaction, leading to higher retention and sales**

* 1. **Shift left approach**

**Shifting left simply refers to testing to be performed as early as possible in the SDLC process. Below shared testing pyramid illustrates the coverage of testing at each layer to effectively apply this testing strategy for both Manual and Automation Testing.**

****

* 1. **How does deployment happening - Cloud farmation and IAAS**

* 1. **How you are helping your teams growth as a senior**

**As a senior, contributing to team growth involves mentoring, fostering collaboration, and creating an environment where everyone can excel. Here's how I approach this responsibility:**

**1. Mentorship and Knowledge Sharing**

* 1. **Guiding Team Members: I actively mentor junior team members by sharing technical knowledge, best practices, and problem-solving approaches.**
  2. **Conducting Learning Sessions: Regular knowledge-sharing sessions, code reviews, or technical workshops ensure everyone is up-to-date with industry trends and tools.**

**2. Encouraging Collaboration**

* 1. **Promoting Open Communication: I ensure team members feel comfortable sharing ideas, asking questions, and providing feedback.**
  2. **Facilitating Teamwork: Encouraging cross-functional collaboration to break silos and create a cohesive, high-performing team.**

**3. Leading by Example**

* 1. **Demonstrating Best Practices: I consistently follow coding standards, document processes, and approach challenges with a positive attitude, setting a benchmark for others.**
  2. **Accountability: Taking responsibility for decisions and outcomes motivates the team to adopt a similar mindset.**

**4. Supporting Professional Growth**

* 1. **Providing Opportunities: Encouraging team members to take on challenging tasks, learn new technologies, or lead initiatives to grow their skill set.**
  2. **Giving Constructive Feedback: Offering actionable, supportive feedback helps individuals recognize areas of improvement and build confidence.**

**5. Enhancing Productivity**

* 1. **Streamlining Processes: Identifying bottlenecks and proposing solutions or tools to optimize workflows.**
  2. **Fostering Autonomy: Trusting team members with responsibilities while being available for guidance fosters independence and innovation.**

* 1. **How will you perform scaling in EC2 -**

**By enabling the Autoscaling group in EC2 instance**

**Scaling in EC2 can be achieved through vertical scaling (changing instance types) and horizontal scaling (adding/removing instances using Auto Scaling Groups). Vertical scaling is suitable for increasing the capacity of a single instance, while horizontal scaling is ideal for distributing the load across multiple instances to handle varying traffic levels. By setting up Auto Scaling Groups and configuring scaling policies, you can ensure that your application scales automatically based on demand, providing high availability and performance.**

* 1. **DNS service in AWS - Route 53**

* 1. **The complex defect that you identified and fixed in your project**

* 1. **How does deployment happen in DHP**

**Cloud Formation and IaaS**

**AWS CloudFormation is a service provided by Amazon Web Services (AWS) that allows you to define and provision infrastructure as code (IaC). With CloudFormation, you can use a simple text file (in JSON or YAML format) to model and set up all the resources needed for your applications across all regions and accounts in AWS.**

**Key Features:**

**•        Infrastructure as Code: Define your infrastructure in code, making it easy to version control and replicate.**

**•        Automated Provisioning: Automatically create and manage AWS resources like EC2 instances, S3 buckets, and RDS databases.**

**•        Consistency: Ensure consistent environments by using the same template to deploy resources across different stages (development, testing, production).**

**Infrastructure as a Service (IaaS) is a form of cloud computing that provides virtualized computing resources over the internet. IaaS is one of the three main categories of cloud services, alongside Platform as a Service (PaaS) and Software as a Service (SaaS).**

**Key Features:**

**•        Virtual Machines: Provision virtual machines (VMs) with the desired operating system and configuration.**

**•        Storage: Access scalable storage solutions like block storage, object storage, and file storage.**

**•        Networking: Configure virtual networks, load balancers, and other networking components.**

**•        Scalability: Easily scale resources up or down based on demand.**

**Examples of IaaS Providers:**

**•        Amazon Web Services (AWS): EC2, S3, VPC**

**•        Microsoft Azure: Virtual Machines, Blob Storage, Virtual Network**

**•        Google Cloud Platform (GCP): Compute Engine, Cloud Storage, Virtual Private Cloud (VPC)**

**Summary**

**•        AWS CloudFormation: A service that allows you to define and provision AWS infrastructure using code (JSON or YAML templates). It automates the setup and management of AWS resources, ensuring consistency and repeatability.**

**•        Infrastructure as a Service (IaaS): A cloud computing model that provides virtualized computing resources over the internet. IaaS allows you to provision and manage virtual machines, storage, and networking components, offering flexibility and scalability for your infrastructure needs.**

* 1. **Which API have you used?**
  2. **What is the difference b/w REST and SOAP**
  3. **What is SOLID principle.**
  4. **Types of dependency injection**
  5. **How you will do routing? Which components is used for routing**
  6. **What is Amazon Routt 53 ?why it is named as 53**
  7. **How will you increase the scalability of ur application**
  8. **How you do the scaling**
  9. **DHP deployment - Explain in detail**
  10. **What are the systems involved in your application**
  11. **How you improve the monolith architecture? Will you switch the architecture ?**
  12. **Explain about Lambda and Athena**