**AWS Services Notes**

1. **VPC**

**Assignment Tasks (Docker)**

1. Run a Simple Web Server: Use Docker to run a simple web server like Nginx or Apache and access its default web page in your browser. This will introduce you to creating containers, running them, and exposing ports.
2. Dockerize a Static Website: Create a Dockerfile to build a container image for your static website (HTML, CSS, JavaScript). The image should include a web server and copy your website files. Run the container and verify your website is accessible. This teaches building images, working with Dockerfiles, and volume mounting.
3. Run a Database in Docker: Use Docker to run a popular database system like MySQL or PostgreSQL. Explore connecting to the database from your local machine using a client tool. This introduces containerized databases, environment variables, and basic data management with Docker.
4. Link Multiple Containers: Run two Docker containers: one for a database and another for a simple web application. Link the containers so the web application can connect to the database. Verify the functionality of your linked services. This teaches container linking, service discovery, and building simple multi-container applications.
5. Build a Docker Image with Multi-Stage Builds: Create a Dockerfile that utilizes multi-stage builds to create a slim and optimized image for your web application. This could involve building the application code in one stage and then copying only the necessary files to a final image with a web server. This introduces multi-stage builds, optimizing image size, and best practices for container creation.
6. Run a Customized Image: Find a pre-built Docker image from a public registry (e.g., Docker Hub) that aligns with your interests (e.g., a game server, programming language environment). Run the image and explore its functionalities within the container. This introduces using pre-built Docker images.
7. Docker Inspect: Use the docker inspect command to examine the details of a running Docker container. Explore the information provided, such as environment variables, exposed ports, and mounted volumes. This teaches inspecting container configurations.
8. Docker Network: Create a simple Docker network and run multiple containers on that network. Verify container communication within the isolated network. This introduces managing Docker networks and container connectivity.
9. Docker Volume Persistence: Create a Docker volume and use it within a container. Mount the volume to a directory within the container and store data there. Restart the container and verify the data persists across restarts. This teaches data persistence with Docker volumes.
10. Environment Variables in Dockerfile: Define environment variables within your Dockerfile and use them during the build process. Access these variables within your application code running in the container. This introduces managing environment variables in Docker.
11. Dockerfile CMD vs ENTRYPOINT: Create a Dockerfile that utilizes both CMD and ENTRYPOINT instructions. Understand the difference between them and how they influence the container's default command. This teaches advanced Dockerfile instructions.
12. Build Docker Image with Cache: Modify your Dockerfile to leverage Docker build cache. Introduce a change in a later stage of the build process and observe how previous build stages are reused from the cache, optimizing build times. This teaches using Docker build cache for efficiency.
13. Docker Push and Pull: Build a Docker image and push it to a private Docker registry (e.g., Docker Hub with private repository). Then, pull the image from the registry and run a container from it. This introduces sharing and deploying Docker images through registries.
14. Docker Compose - Multi-container App: Create a simple multi-container application using Docker Compose. Define services for multiple containers and their interactions within a docker-compose.yml file. This introduces managing multi-container applications with Docker Compose.
15. Docker Run Detached Mode: Run a Docker container in detached mode (-d flag). Explore how detached containers operate in the background and how to manage them using commands like docker ps and docker stop. This teaches managing background containers.

**Scenario based Question (Docker)**

1. **Container Fails to Start:**

**You try to run a Docker container, but it exits immediately with an error message. How would you diagnose the root cause of the failure?**

* Review the error message in the docker logs (docker logs <container\_name>) for specific clues.
* Check the Dockerfile syntax for errors using docker build -t <image\_name>:<tag> -f Dockerfile --no-cache (replace placeholders with your details).
* Verify that required dependencies or libraries are present within the image.
* Ensure sufficient disk space and resource availability on the Docker host.

1. **Unresponsive Container:**

**Your Docker container is running but appears unresponsive to requests. How would you investigate the issue and identify potential causes?**

* Use docker logs <container\_name> to check for error messages or clues about the unresponsive state.
* Try running docker exec -it <container\_name> sh (or appropriate shell) to access the container's shell and investigate processes.
* Utilize tools like top or htop within the container to identify resource bottlenecks (CPU, memory).

1. **Network Connectivity Issues:**

**Containers within your Docker network cannot communicate with each other or external services. How would you troubleshoot network connectivity problems?**

* Verify the Docker network configuration (docker network inspect <network\_name>) and ensure container IP addresses are assigned correctly.
* Check firewall rules on the Docker host or container that might be blocking communication.
* Use docker ps to confirm containers are running on the same network and have appropriate port mappings.
* Test network connectivity using ping or nc commands within containers.

1. **High CPU Usage:**

**A container is experiencing high CPU utilization, impacting overall performance. What steps would you take to identify the cause and optimize resource usage?**

* Analyze container logs for clues about resource-intensive processes.
* Utilize docker stats to monitor CPU utilization over time.
* Consider profiling tools within the container to pinpoint specific code sections causing high CPU usage.
* Explore container resource limits with docker update --cpu-shares <value> <container\_name> (adjust value based on requirements).

1. **Memory Leaks:**

**You suspect a memory leak within a container, causing it to crash or become unresponsive. How would you diagnose and address the memory leak issue?**

* Monitor memory usage with docker stats. Look for containers with steadily increasing memory consumption.
* Use tools like docker run <image> sh -c 'ps auxf | grep <process\_name>' (replace <process\_name> with the suspected process) within the container to identify memory usage patterns.
* Consider language-specific tools like valgrind or jemalloc for detailed memory leak analysis if applicable.

1. **Log Errors:**

**The container logs are filled with error messages. How would you interpret these logs and identify the underlying problems within the container?**

* Use docker logs <container\_name> to view container logs for error messages and warnings.
* Filter logs based on severity level with options like docker logs <container\_name> --filter level=error.
* Search logs for specific keywords or patterns that might indicate the source of the problem.

1. **Volume Mounting Issues:**

**You encounter errors when mounting volumes to your containers. What could be causing these issues, and how would you troubleshoot them?**

* Verify the volume path exists and has appropriate permissions on the Docker host.
* Ensure the volume driver (if used) is functioning correctly.
* Check for typos or inconsistencies in volume mount configuration within the Docker run command.
* Test volume mounting functionality with a simple test container to isolate the issue.

1. **Image Pull Failures:**

**Docker fails to pull a container image from the registry. What are some potential reasons for this, and how can you debug them?**

* Double-check the image name and repository for typos or missing details.
* Verify your internet connectivity and ensure access to the Docker registry.
* Check for authentication issues with private registries (if applicable).
* Try pulling from a different registry mirror if available.

1. **Incorrect Environment Variables:**

**Your application running within the container malfunctions due to incorrect or missing environment variables. How would you identify and fix this issue?**

* Inspect container logs for errors related to missing or incorrect environment variables.
* Verify your Docker run command (docker run -e KEY=VALUE ...) or Docker Compose configuration for typos or missing environment variables.
* Use env command within the container to view all available environment variables and compare them to your expectations.

1. **Resource Limits Exceeded:**

**A container keeps getting stopped due to exceeding resource limits (CPU, memory). How would you troubleshoot the issue and potentially adjust resource allocation?**

* Analyze container logs and docker stats output to identify resource bottlenecks.
* Adjust resource limits with docker update --cpu-shares <value> --memory <memory\_limit> <container\_name> (customize values based on requirements).
* Consider optimizing the application code or container image to reduce resource consumption.

1. **Conflicting Ports:**

**You try to run multiple containers that require the same port mapping. How can you resolve port conflicts and ensure proper container communication?**

* Use docker ps to identify containers using the same port.
* Remap ports in your Docker run command or Docker Compose configuration to avoid conflicts (e.g., -p 8081:80).
* Utilize a dynamic port allocation strategy with tools like docker run -p 0:80 <image\_name> (maps container port 80
* You can also leverage host networking mode (docker run --network host <image\_name>) for specific containers if they require access to the host's network ports directly (use with caution due to potential security implications).

1. **Restart Policy Issues:**

**Your container keeps restarting unexpectedly despite having a defined restart policy. What could be causing these restarts, and how would you investigate them?**

* Review container logs for any crash messages or clues about the cause of restarts.
* Check the defined restart policy with docker inspect <container\_name>.
* Consider using docker logs <container\_name> --tail=100 to view the most recent log entries for potential restart triggers.
* Troubleshoot the underlying issue within the container causing the restarts (e.g., application errors, resource exhaustion).

1. **Security Vulnerabilities:**

**You discover a security vulnerability within a container image. How would you approach mitigating the risk and potentially updating the image?**

* Use a vulnerability scanner like snyk docker or anchore to identify potential vulnerabilities within your container image.
* Update the base image or relevant packages within the Dockerfile to address the identified vulnerabilities.
* Consider using a Docker Security Scanner (like Aqua or Clair) for continuous vulnerability scanning in your CI/CD pipeline.
* Implement security best practices like following the principle of least privilege and minimizing container access to resources.

1. **Service Discovery Problems:**

**Containers within a multi-container application cannot discover each other using service discovery mechanisms like DNS or links. How can you troubleshoot these issues?**

* Verify that service discovery mechanisms like DNS or links are configured correctly within your Docker network.
* Check if containers can resolve service names to IP addresses using nslookup within a container.
* Ensure containers are joining the intended Docker network where service discovery is happening.
* Test communication between containers using tools like ping or nc with service names or IP addresses.

1. **Debugging Long-Running Processes:**

**You need to debug an issue within a long-running process inside a container. How can you utilize tools like docker exec or attach to the container for troubleshooting?**

* Use docker exec -it <container\_name> sh (or appropriate shell) to access the container's shell and interact with the running process.
* Leverage tools like ps auxf or top within the container to identify the process ID (PID) of the long-running process.
* Utilize debuggers like gdb or strace (if available within the container) to step through the process execution and pinpoint issues.
* Consider logging mechanisms within your application to capture detailed information about the process's behavior.

**Assignment Tasks (Jenkins)**

1. Hello World Job: Create a simple Jenkins job that uses a freestyle project and prints "Hello World" to the console during execution. This will introduce you to basic job creation and running builds in Jenkins.
2. Checkout a Public Repository: Set up a Jenkins job that checks out a public Git repository (e.g., a sample project on GitHub) and displays the contents of a specific file within the project directory. This teaches you how to use Jenkins to interact with version control systems.
3. Build a Docker Image: Create a Jenkins job that utilizes a pipeline to build a simple Docker image from a Dockerfile. This image can be based on a pre-built image and potentially copy additional files. This introduces building Docker images within Jenkins pipelines.
4. Run Shell Script: Design a Jenkins job that executes a basic shell script containing commands like echo or ls within a build stage. This script can display a message or list files in the workspace. This teaches you how to integrate shell scripting into Jenkins pipelines.
5. Schedule a Periodic Build: Configure a Jenkins job to run automatically on a specific schedule (e.g., daily or hourly). This could be useful for repetitive tasks or fetching updates from repositories. This introduces scheduling builds within Jenkins.
6. Pipeline Stages: Create a Jenkins job with a multi-stage pipeline (e.g., checkout, build, test). Each stage should perform a specific task within the build process. This reinforces the concept of breaking down workflows into logical stages.
7. Post-Build Actions: Configure a Jenkins job to perform post-build actions after a successful build. These actions could involve sending notifications (email, chat) or archiving artifacts (build outputs). This teaches managing post-build activities.
8. Build Artifacts: Design a Jenkins job that captures and archives specific files or directories generated during the build process as artifacts. This allows access to these outputs for later reference or deployment. This introduces managing build artifacts.
9. Pipeline Parameters (Multiple): Enhance a Jenkins job to accept multiple parameters during execution. Use these parameters to customize the build process based on user input (e.g., environment variables, build configurations). This extends parameterization concepts.
10. Pipeline Failure Handling: Implement error handling within a Jenkins pipeline. Use conditional logic (e.g., based on build status) to handle failures gracefully (e.g., notify admins, schedule retries). This teaches managing and reacting to build failures.
11. Use Credentials in Jenkins: Create a Jenkins job that utilizes credentials (username/password) to access a private Git repository or another resource requiring authentication. This introduces secure access management within pipelines.
12. Manage Plugins: Install a plugin for Jenkins that extends its functionality (e.g., specific programming language support, notification channels). Explore the Jenkins plugin ecosystem and its impact on job capabilities.
13. Workspace Cleanup: Configure a Jenkins job to clean up its workspace after a build is completed. This can involve deleting temporary files or directories to maintain clean workspaces for subsequent builds. This teaches workspace management practices.
14. Job DSL Script: Write a basic Job DSL script to define a Jenkins job in code. This script can be used to automate job creation or manage configurations through code. This introduces programmatic job configuration.
15. Multibranch Pipeline: Set up a Jenkins job that utilizes a multibranch pipeline. This pipeline can automatically detect changes in multiple Git branches and trigger builds for each branch independently. This teaches handling builds for multiple branches or repositories.
16. Create a Simple Job with Pipeline as Code: Set up a Jenkins job that uses a basic Jenkinsfile to checkout a public Git repository, build a sample project (e.g., compile some code), and print a success message. This will introduce Jenkinsfiles, pipeline stages, and basic build steps.
17. Automate Unit Tests with Jenkins: Modify your previous Jenkins job to include a stage that runs unit tests for your sample project. Utilize a built-in Jenkins plugin or script to execute the tests and capture results (pass/fail). This teaches integrating unit testing with Jenkins pipelines and understanding test reporting.
18. Parameterize Your Jenkins Job: Enhance your existing job to accept a parameter (e.g., branch name) during execution. Use this parameter in the pipeline to checkout a specific branch from the Git repository. This introduces parameterized builds, allowing customization based on user input.
19. Trigger a Build Pipeline on Push to Git: Configure your Jenkins job to trigger automatically whenever there's a push to the Git repository containing your sample project. This could involve webhooks or other notification mechanisms. This teaches build automation and integration with version control systems.
20. Implement a Multi-Stage Pipeline with Conditional Logic: Design a Jenkins pipeline with multiple stages (e.g., checkout, build, test, deploy). Use conditional logic (e.g., based on test results) to skip stages if necessary (e.g., skip deployment if tests fail). This introduces complex pipelines, decision making within pipelines, and best practices for failure handling.

**AWS**

1. Multi-Tier Web Application with Autoscaling:

* Deploy a web application with a separate frontend (S3 or EC2) and backend (EC2) on a VPC.
* Implement an Auto Scaling Group for the backend servers to handle traffic spikes.
* Configure an Elastic Load Balancer to distribute traffic across the backend instances.
* Use CloudWatch to monitor application and server health.

2. Secure File Sharing with Access Control:

* Create an S3 bucket for storing sensitive files.
* Implement IAM policies to restrict access to specific user groups or roles.
* Configure CloudFront as a CDN to deliver files securely with low latency.
* Use IAM roles to grant temporary access to specific users for limited durations.

3. Serverless Application with Notifications:

* Develop a serverless function using Lambda to perform a specific task (e.g., image resizing, data processing).
* Trigger the Lambda function using API Gateway to create a RESTful API endpoint.
* Set up SNS notifications to send alerts when the Lambda function encounters errors.
* Use SQS as a queue to buffer messages if the Lambda function experiences high load.

4. Containerized Microservice with Monitoring:

* Build a Docker image for a microservice using ECR.
* Deploy the containerized microservice to a Fargate cluster within your VPC.
* Use Route 53 to create a hosted zone and route traffic to the Fargate service.
* Configure CloudWatch to monitor container performance and resource utilization.

5. Database Backup and Disaster Recovery:

* Create an RDS database instance for your application.
* Configure automated backups of the RDS database to S3 at regular intervals.
* Set up a CloudTrail trail to track API calls made to RDS for auditing purposes.
* Design a disaster recovery plan to restore your database from backups in case of a failure.

6. Serverless Data Processing with DynamoDB:

* Develop a Lambda function to process data from an SQS queue.
* Use the DynamoDB NoSQL database to store the processed data.
* Implement DynamoDB Streams to capture changes in the database and trigger additional actions.
* Use CloudWatch Logs to monitor the execution of the Lambda function.

7. Static Website with Global Distribution:

* Deploy a static website to an S3 bucket.
* Configure CloudFront as a CDN with edge locations around the world.
* Use Route 53 to create a hosted zone and route traffic to the CloudFront distribution.
* Implement CloudWatch metrics to monitor website traffic and performance.

8. Scalable Batch Processing with ECS:

* Develop a containerized application for batch processing tasks.
* Create an ECS cluster with Fargate service to run your containerized application.
* Use ECS Tasks to define the batch processing jobs and their resource requirements.
* Set up an SQS queue to trigger new processing tasks as needed.

9. Automated Infrastructure Provisioning with CloudFormation:

* Design a CloudFormation template to define your VPC, EC2 instances, and security groups.
* Use CloudFormation to automate the provisioning and configuration of your infrastructure.
* Implement IAM roles for CloudFormation to grant access to necessary AWS resources.
* Use CloudTrail to track infrastructure changes made through CloudFormation deployments.

10. Serverless API with User Authentication:

* Develop a serverless API using Lambda to handle various functionalities.
* Integrate API Gateway with user authentication using Cognito or IAM.
* Implement IAM policies to authorize access to specific API resources based on user roles.
* Use CloudWatch Logs to monitor API requests and identify potential security threats.

11. Securely Accessing Secrets in Your Application:

* Create a Secrets Manager secret to store sensitive application credentials (database passwords, API keys).
* Configure IAM roles for your EC2 instances or Lambda functions to access secrets securely.
* Use the AWS SDK in your application to retrieve secrets from Secrets Manager at runtime.

12. Building a Serverless Data Pipeline with SQS and Lambda:

* Develop a Lambda function to process a stream of data received from an SQS queue.
* Utilize SQS message attributes to provide additional context for each data item.
* Implement error handling within the Lambda function to handle potential processing failures.
* Consider using AWS Step Functions to orchestrate a more complex data processing pipeline with multiple Lambda functions.

13. Building a Serverless Web Application with API Gateway and DynamoDB:

* Develop a serverless API using Lambda functions for various functionalities (user management, data retrieval).
* Integrate API Gateway to expose your Lambda functions as RESTful endpoints.
* Utilize DynamoDB as a NoSQL database to store application data efficiently.
* Implement DynamoDB access control using IAM policies to restrict data access based on user roles.

14. Monitoring Application Logs with CloudWatch Logs Insights:

* Configure CloudWatch Logs to collect and store logs from your EC2 instances, Lambda functions, and other services.
* Utilize CloudWatch Logs Insights for ad-hoc querying and analysis of your application logs.
* Create custom dashboards with visualizations to monitor key application metrics and identify potential issues.

15. Building a Scalable Web Application with ElastiCache:

* Deploy a web application with a separate frontend (S3 or EC2) and backend (EC2) on a VPC.
* Utilize ElastiCache for caching frequently accessed data to improve application performance.
* Configure Auto Scaling for your backend servers to handle traffic spikes efficiently.
* Implement CloudWatch metrics to monitor cache performance and application response times.

16. Developing a Serverless Chat Application with SNS and SQS:

* Develop a serverless Lambda function to handle chat messages received from clients.
* Utilize SNS to broadcast messages to all connected clients in real-time using fan-out architecture.
* Consider using SQS as a buffer for messages if there are spikes in message volume to prevent message loss.
* Implement user authentication using Cognito or IAM to secure chat functionalities.

17. Creating a CI/CD Pipeline with CodeBuild and CodeDeploy:

* Integrate a CodeBuild project to build and test your application code automatically upon code changes in CodeCommit.
* Configure CodeDeploy to automate the deployment of your application code from CodeBuild onto EC2 instances or ECS clusters.
* Consider using CloudWatch Logs to monitor the build and deployment process.
* Implement rollback mechanisms in CodeDeploy to revert to previous deployments in case of issues.

18. Building a Serverless Workflow with Step Functions:

* Design a workflow with multiple steps using AWS Step Functions to automate a complex task sequence.
* Utilize Lambda functions as building blocks within the workflow steps to perform specific actions.
* Implement branching and decision-making within the workflow based on specified conditions.
* Monitor the workflow execution with CloudWatch for insights into success rates, latency, and error handling.

19. Developing a Secure IoT Application with IoT Core and SQS:

* Configure AWS IoT Core to manage your IoT devices and establish secure communication channels.
* Implement Lambda functions to process data received from your IoT devices via SQS queues.
* Utilize IAM policies to restrict access to IoT Core resources and data based on specific devices or user roles.
* Consider using CloudWatch metrics to monitor device health and data flow within your IoT application.

20. Building a Content Delivery Network with WAF and CloudFront:

* Utilize CloudFront as a CDN to deliver content from an S3 bucket with low latency and high availability.
* Configure WAF (Web Application Firewall) on CloudFront to protect your website from common web attacks.
* Implement WAF rules to block malicious traffic and ensure website security.
* Use CloudWatch metrics to monitor the performance of your CDN and identify potential issues.

21. Migrating a Legacy Application to the Cloud with EC2 and RDS:

* + Develop a migration plan to move your on-premises application to a secure VPC on AWS.
  + Deploy your application on EC2 instances within the VPC for scalability and manageability.
  + Utilize RDS to create a managed database service for your application data.
  + Configure security groups to restrict access to your EC2 instances and RDS database.

22. Building a Highly Available Web Application with Autoscaling and Load Balancing:

* + Deploy a web application with a separate frontend (S3 or EC2) and backend (EC2) on a VPC.
  + Implement an Auto Scaling Group for the backend servers to handle traffic spikes automatically.
  + Configure an Elastic Load Balancer to distribute incoming traffic across the backend instances.
  + Integrate CloudWatch monitoring to track application and server performance.

23. Creating a Serverless Static Website with S3, CloudFront, and Route 53:

* + Develop a static website using HTML, CSS, and JavaScript.
  + Deploy the website to an S3 bucket for secure and durable storage.
  + Configure CloudFront as a CDN to deliver website content with low latency globally.
  + Utilize Route 53 to create a hosted zone and route traffic to the CloudFront distribution.
  + Implement CloudWatch metrics to monitor website traffic and performance.

24. Building a Serverless Data Pipeline with SQS, Lambda, and DynamoDB:

* + Develop a Lambda function to process data received from an SQS queue.
  + Implement DynamoDB as a NoSQL database to store the processed data efficiently.
  + Utilize SQS message attributes to provide additional context for each data item.
  + Consider using AWS X-Ray tracing to visualize the execution flow of the Lambda function and DynamoDB interactions.

25. Developing a Secure Microservice Architecture with ECS, Fargate, and IAM:

* + Develop a containerized microservice using Docker and ECR.
  + Deploy the microservice to a Fargate cluster within your VPC for scalability and serverless management.
  + Utilize IAM roles and policies to grant access control to resources within the ECS cluster and Fargate tasks.
  + Configure CloudWatch monitoring to track container resource utilization and service performance.

26. Building a Serverless API with API Gateway, Lambda, and RDS:

* + Develop a serverless API using Lambda functions for various functionalities (user management, data retrieval).
  + Integrate API Gateway to expose your Lambda functions as RESTful endpoints.
  + Utilize RDS as a managed database service for your application data.
  + Implement IAM policies to restrict access to specific API resources based on user roles.

27. Designing a Disaster Recovery Plan with S3 Backups and Route 53 Failover:

* + Create an S3 bucket for storing regular backups of your application data and EC2 instances.
  + Configure automated backups of your RDS database to S3 at regular intervals.
  + Use Route 53 with health checks to automatically failover to a secondary web server in case of a primary server failure.
  + Consider implementing CloudTrail to track API calls made to AWS services for auditing purposes.

28. Building a Scalable Batch Processing System with ECS and SQS:

* + Develop a containerized application for batch processing tasks like data analysis or file conversion.
  + Create an ECS cluster with Fargate service to run your containerized application.
  + Utilize SQS as a queue to buffer tasks and trigger new processing jobs as needed.
  + Implement CloudWatch monitoring to track the queue size and processing time of jobs.

29. Developing a Serverless Workflow with Lambda and Step Functions:

* + Design a workflow with multiple steps using AWS Step Functions to automate a complex task sequence.
  + Utilize Lambda functions as building blocks within the workflow steps to perform specific actions like sending notifications or processing data.
  + Implement branching and decision-making within the workflow based on specified conditions.
  + Utilize CloudWatch Logs to monitor the workflow execution and identify potential errors.

30. Creating a Secure Mobile Backend with API Gateway, Lambda, and Cognito:

* + Develop a serverless API using Lambda functions for various mobile app functionalities (data access, user authentication).
  + Integrate API Gateway to expose your Lambda functions as endpoints for your mobile app.
  + Utilize Cognito for user authentication and authorization within your mobile app.
  + Implement IAM policies to restrict access to specific API resources based on user roles and identity.

1. VPC Setup and Configuration

Create a custom VPC with public and private subnets across multiple availability zones. Set up appropriate route tables, internet gateways, and NAT gateways.

2. EC2 Instance Deployment

Launch an EC2 instance in your custom VPC. Configure security groups to allow HTTP and SSH access. Use a custom AMI for the instance and ensure it is placed in the correct subnet.

3. Auto Scaling and Load Balancing

Configure an Auto Scaling group with an EC2 launch template. Integrate the Auto Scaling group with an Application Load Balancer (ALB) to distribute traffic across multiple EC2 instances.

4. S3 Bucket Configuration

Create an S3 bucket with versioning and server-side encryption enabled. Configure bucket policies to allow public read access to a specific folder while keeping the rest of the bucket private.

5. IAM Roles and Policies

Create IAM roles for an EC2 instance to allow access to S3 and DynamoDB. Attach appropriate policies and verify that the instance can interact with these services.

6. CloudFront Distribution

Set up a CloudFront distribution to serve content from an S3 bucket. Configure caching behaviors and custom error responses.

7. Monitoring with CloudWatch

Set up CloudWatch alarms to monitor CPU utilization of your EC2 instances. Configure SNS notifications to alert you when the alarm state changes.

8. Domain Management with Route53

Register a domain name with Route53. Create a hosted zone and configure DNS settings to route traffic to your ALB.

9. Serverless with Lambda

Develop a Lambda function that processes data from an S3 bucket trigger. Configure the Lambda function with appropriate permissions and environment variables.

10. Container Orchestration with ECS and Fargate

- Deploy a Dockerized application using ECS with Fargate launch type. Use ECR to store your Docker images and configure service discovery within your VPC.

11. API Gateway Integration

- Create a RESTful API using API Gateway. Integrate the API with a Lambda function and configure stages, methods, and usage plans.

12. Relational Database with RDS

- Launch an RDS instance with MySQL or PostgreSQL engine. Configure security groups, parameter groups, and automated backups. Connect an EC2 instance to the RDS instance.

13. NoSQL Database with DynamoDB

- Create a DynamoDB table with appropriate primary keys and indexes. Implement a Lambda function to perform CRUD operations on the table.

14. Notification Service with SNS

- Set up an SNS topic for application alerts. Configure an email subscription and create a Lambda function to publish messages to the SNS topic.

15. Message Queuing with SQS

- Create an SQS queue and configure a Lambda function to process messages from the queue. Implement error handling and message visibility settings.

**Scenario based Question (Linux)**

1. **Your Linux system fails to boot completely, getting stuck on the early boot screen or displaying error messages. How would you diagnose the root cause and attempt to recover?**

* Switch to a virtual terminal (Ctrl+Alt+F1-F6) and examine boot error messages for clues.
* Use emergency boot mode (e.g., init=/bin/bash) to access a basic shell and potentially repair boot issues.
* Check for hardware malfunctions like disconnected disks or faulty RAM using diagnostic tools.

1. **The system becomes unresponsive to keyboard or mouse input, and applications freeze. What steps would you take to identify the issue and regain control?**

* Try using keyboard shortcuts like Ctrl+Alt+Del or SysRq key combinations to regain control (if applicable).
* Use top or htop commands in another virtual terminal to identify resource-intensive processes and potentially terminate them.
* Check system logs (e.g., /var/log/messages) for clues about the cause of the freeze.

1. **You cannot access the internet or network resources from your Linux machine. How would you troubleshoot network connectivity problems?**

* Use ping or traceroute commands to test connectivity to specific destinations and identify potential network bottlenecks.
* Verify network interface configuration with ifconfig or ip addr and ensure your IP address, subnet mask, and gateway settings are correct.
* Check firewall rules with iptables -L (or firewalld for newer systems) to ensure they're not blocking network traffic.

1. **A specific process is consuming excessive CPU resources, impacting overall system performance. What tools and techniques can you use to identify and address the issue?**

* Use top or htop to identify processes consuming excessive CPU resources.
* Analyze process details with ps auxf | grep <process\_name> (replace with the process name) to understand resource usage patterns.
* Consider tools like strace or gdb (for advanced users) to delve deeper into process behavior and pinpoint issues.
* If necessary, utilize kill or killall commands to terminate processes cautiously (be mindful of potential consequences).

1. **The system runs out of disk space, causing applications to malfunction. How would you investigate disk usage and potentially free up space?**

* Use df -h to analyze disk usage and identify space-consuming directories or files.
* Consider tools like du -sh /path/to/directory to drill down into specific directories and locate large files.
* Explore options for cleaning up unnecessary files, logs, or temporary data.
* If necessary, consider resizing partitions or adding additional storage.

1. **You encounter permission errors when trying to access files or directories. What could be causing these errors, and how can you fix them securely?**

* Understand file ownership and permissions using ls -l to identify the user and group associated with the file.
* Use chown and chmod commands to adjust file ownership and permissions cautiously, ensuring proper access control.
* Consider using sudo for administrative tasks that require elevated privileges but avoid granting excessive permissions permanently.

1. **Package installation or updates through your package manager fail with error messages. How would you troubleshoot these issues and ensure successful package management?**

* Review error messages from your package manager (e.g., apt, yum) to understand the cause of the failure.
* Check package repositories for updates or potential outages using your package manager's tools.
* Try refreshing package lists (apt update or yum update) before attempting installation or updates again.
* In rare cases, consider manually downloading and installing packages from trusted sources.

1. **A critical system service fails to start, causing issues with functionalities like networking or printing. What steps would you take to diagnose the problem and get the service running?**

* Use systemctl status <service\_name> to view the service status and potential error messages.
* Check system logs (e.g., /var/log/syslog) for clues about why the service might be failing to start.
* Utilize systemctl enable <service\_name> to ensure the service is enabled for automatic startup.
* If necessary, try restarting the service with systemctl restart <service\_name>.

1. **You cannot log in to your Linux system with your usual credentials. How can you troubleshoot potential login problems and regain access?**

* Verify you're using the correct username and password (case-sensitive).
* Check if the virtual terminal you're using (e.g., Ctrl+Alt+F1) is configured for the login manager you're using (e.g., GDM, LightDM).
* In case of forgotten passwords, consult your system's documentation for password reset procedures (often involving a recovery disk or user with administrative privileges).
* Consider security implications when attempting password resets.

1. **You suspect a hardware component (like RAM or disk) might be malfunctioning. How would you diagnose hardware issues and potentially isolate the faulty component?**

* Utilize system tools like memtest to diagnose potential RAM issues.
* Run SMART tests (specific tools vary by disk manufacturer) to check disk health and identify potential failures.
* Monitor system logs for hardware-related error messages.
* Consider using diagnostic tools provided by your hardware vendor for specific troubleshooting steps.

1. **The system unexpectedly shuts down or restarts without warning. What could be causing these sudden shutdowns, and how can you investigate them?**

* Analyze system logs (e.g., /var/log/kern.log) for kernel panic messages or clues about potential hardware or software issues.
* Check for overheating components using tools like sensors (if available) and ensure proper ventilation.
* Consider recent system changes (hardware or software) that might have introduced instability.
* If the issue persists, consult hardware documentation or seek professional assistance for hardware diagnostics.

1. **You discover a potential security vulnerability in your system. How would you approach mitigating the risk and patching the system securely?**

* Utilize system security scanners like Lynis or OpenVAS to identify potential vulnerabilities.
* Update your system and installed packages regularly using your package manager to address known vulnerabilities.
* Consult security advisories from your distribution provider and apply necessary patches promptly.
* Consider hardening your system configuration by following security best practices (e.g., disabling unused services, implementing strong passwords).

1. **System logs are flooded with messages, making it difficult to identify the root cause of an issue. How can you effectively analyze and filter system logs for troubleshooting purposes?**

* Use tools like grep to filter log messages based on keywords or patterns related to the issue you're troubleshooting.
* Utilize advanced logging utilities like journalctl (on systemd systems) for efficient log filtering and searching.
* Leverage timestamps within logs to identify the timeframe when the issue might have occurred.
* Consult system documentation for specific log file locations and message formats to understand their meaning.

1. **An application you're running throws cryptic error messages or crashes unexpectedly. What debugging techniques can you use to pinpoint the source of the application error?**

* Review application logs (often located in /var/log/<application\_name>.log or similar paths).
* Utilize debugging tools provided by the programming language used for the application (e.g., gdb for C/C++, lldb for Swift).
* Consider enabling debug mode within the application (if available) for more verbose error messages.
* Search online forums or communities for similar errors and potential solutions related to the specific application.

1. **You suspect a misconfiguration in a system or application configuration file might be causing problems. How would you approach troubleshooting and identifying potential configuration errors?**

* Use a text editor like nano or vim to cautiously edit configuration files (always back up before making changes).
* Consult the configuration file's documentation for syntax and parameter details.
* Utilize comments within the configuration file to document changes you make for future reference.
* Consider using configuration management tools like Ansible or Chef to automate configuration management and avoid manual errors.

**Jenkins**

**Install Jenkins on Ubuntu**

Step 1: Install Java on Ubuntu

$ sudo apt update

$ sudo apt sudo apt install openjdk-8-jdk

Alternatively, install version 11:

$ sudo apt install openjdk-11-jdk

Confirm the download by pressing Y and Enter

Step 2: Add the repository key to the system:

$ sudo apt wget -q -O -https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

Step 3: Once the key is added with no errors, append the Debian package repository address

$ sudo sh -c 'echo deb

http://pkg.jenkins.io/debian-stable binary/ >

/etc/apt/sources.list.d/jenkins.list’

Step 4: Run update to use new repository

$ sudo apt update

Step 5: Install Jenkins

$ sudo apt install jenkins

After successful installation let us start Jenkins

$ sudo systemctl start jenkins

The above command will not display any output

To check the running status of Jenkins use the below command

which should show active status on run

$ sudo systemctl status jenkins

**Linux**

**Difference between grep and egrep?**

In egrep, you can search for more than one strings at same time

egrep "key1|key2|key4"

**How can you read a file without using cat command?**

Using less, more, vi command

1) Stopping any service from the command line

On Linux:

sudo systemctl stop jenkins

OR

ps aux | grep jenkins

kill <PID>

On Windows:

net stop jenkins

OR

tasklist | findstr jenkins

taskkill /PID <PID> /F (Windows)

2) To check what is running on a certain port

netstat -ano | find "8080"

3)

File System:

* Understanding the directory structure: Linux uses a hierarchical directory structure, similar to a tree. Knowing how to navigate this structure with commands like cd and pwd is essential.
* File types: Linux differentiates between regular files (containing text or data), directories (folders for organizing files), symbolic links (shortcuts to other files or directories), and special files (devices or resources).
* File permissions: Understanding file permissions (read, write, execute) for users, groups, and others is crucial for file access control. Commands like chmod and chown are used to manage permissions.

Processes and Shells:

* Processes: These are running programs or tasks in the system. Tools like ps and top help you list and monitor processes.
* Shells: The shell is a command-line interface where you interact with the system. Common shells include Bash (Bourne Again SHell) and Zsh.
* Process management: Knowing how to start, stop, and manage processes using commands like kill and jobs is important for controlling system resources.

System Administration:

* Users and groups: Linux systems manage user accounts and groups for access control. Commands like useradd, usermod, and groupadd are used for user and group management.
* Package management: Package managers like APT (Debian/Ubuntu) or Yum (Red Hat/CentOS) help install, update, and remove software packages.
* System logs: System logs record events and messages from the system. Understanding how to read and analyze logs (using tools like tail and grep) is helpful for troubleshooting.

Basic Networking:

* Network interfaces: Linux systems have network interfaces for communication. Tools like ifconfig and ip help you configure and view network settings.
* IP addresses: Each network device has an IP address for identification. Understanding IP addresses and subnet masks is fundamental for network connectivity.
* Basic commands: Commands like ping and traceroute help test network connectivity and identify routing paths.

Text Editors:

* Basic text editors: Vi or nano are common text editors in Linux for editing files. Learning the basic commands for navigating and editing text files is essential.

Additional Concepts:

* Shell scripting: Scripting allows automating repetitive tasks using shell commands.
* Virtualization: Technologies like KVM and Docker allow running multiple operating systems or isolated environments within a single system.
* Security: Understanding basic security principles like user management, firewalls, and secure file permissions is crucial.