**LINUX**

1. \*\*How to Check Linux OS?\*\*

- Run `cat /etc/os-release` command.

2. \*\*How to Check IP Address?\*\*

- Use `ifconfig` or `hostname -I`.

3. \*\*How to Check CPU Information?\*\*

- View CPU info with `cat /proc/cpuinfo`.

4. \*\*How to Check RAM Usage?\*\*

- Check RAM with `free -h`.

5. \*\*How to Check Disk Space?\*\*

- Use `df -h` to check disk space.

6. \*\*How to Check Disk Usage?\*\*

- See disk usage with `du -h`.

7. \*\*Where to Find Logs?\*\*

- Logs are typically found in `/var/appname`.

8. \*\*File Hierarchy Overview:\*\*

- `/etc`: Configuration

- `/bin`: Executable/Binary

- `/opt`: Application

- `/home`: Users

- `/tmp`: Temporary

- `/mnt`: Mount new devices

- `/var`: Logs

- `/proc`: Virtual directory

9. \*\*How to Install Software?\*\*

- Use `apt` for Ubuntu, `yum` for RedHat.

10. \*\*Ways to Connect to Linux:\*\*

- Via credentials (username/password) or SSH keys (recommended).

11. \*\*What is sudo?\*\*

- Command for privileged access.

12. \*\*How to Set Cronjobs?\*\*

- Edit with `crontab -e` or list with `crontab -l`.

13. \*\*Understanding Cronjob Syntax:\*\*

- Minutes, Hours, Day, Month, Weekday.

14. \*\*Checking Port Status:\*\*

- Use `nc -vz` to check if a port is open.

15. \*\*Managing Processes:\*\*

- View with `ps -nlp`, start/stop/restart/status with `systemctl`, kill with `kill -9 appid`.

16. \*\*Monitoring System Performance:\*\*

- Use `top` command.

17. \*\*Using the Find Command:\*\*

- Search for files/directories.

18. \*\*Downloading Files:\*\*

- Use `curl` or `wget

TERRAFORM

1. **Module Types:**
   * **Question:** What are the different types of modules in Terraform?
   * **Answer:** Terraform supports several module types, including root modules, child modules, and remote modules, each serving specific purposes in infrastructure management and organization.
2. **Dynamic Block:**
   * **Question:** How do you use dynamic blocks in Terraform?
   * **Answer:** Dynamic blocks allow you to define repetitive configurations dynamically, enabling flexibility and reducing code duplication. They are useful when you need to generate multiple similar configurations based on a given input.
3. **Unlock Types - Unlock, Release, Delete Lock File:**
   * **Question:** Explain the different types of unlocking mechanisms in Terraform and their purposes.
   * **Answer:** Terraform provides three main unlocking actions: **unlock**, **release**, and **delete lock file**. Each action serves a specific purpose, such as releasing a lock on the state file, unlocking it for further operations, or deleting the lock file altogether in case of emergencies.
4. **Sensitive Information:**
   * **Question:** How do you handle sensitive information in Terraform configurations?
   * **Answer:** Terraform's **sensitive** attribute allows you to mark sensitive data, such as passwords or API keys, preventing them from being displayed in plan output or stored in state files, enhancing security.
5. **Key Vault Integration:**
   * **Question:** How would you integrate Terraform with a key vault for managing secrets?
   * **Answer:** Terraform supports integration with key vault services, enabling secure storage and retrieval of sensitive information such as credentials or encryption keys, enhancing overall security and compliance.
6. **Terraform Workflow - Init, Plan, Apply:**
   * **Question:** Describe the typical Terraform workflow from initialization to applying changes.
   * **Answer:** The Terraform workflow involves initializing a working directory (**terraform init**), generating and reviewing an execution plan (**terraform plan**), and applying changes to infrastructure (**terraform apply**), ensuring controlled and predictable infrastructure management.
7. **Current Version:**
   * **Question:** How do you determine the current version of Terraform installed on your system?
   * **Answer:** The current version of Terraform installed on the system can be checked by running the command **terraform version**, which displays the Terraform version along with other relevant information.
8. **Terragrunt Tool - Multiple Folder:**
   * **Question:** How does Terragrunt facilitate managing Terraform configurations across multiple folders?
   * **Answer:** Terragrunt simplifies the management of Terraform configurations across multiple folders by providing features such as configuration inheritance, remote state management, and environment-specific variables, enhancing code reusability and maintainability.
9. **Import - Why:**
   * **Question:** When and why would you use the **terraform import** command?
   * **Answer:** The **terraform import** command is used to bring existing infrastructure resources under Terraform management. It is particularly useful when adopting Terraform for existing environments or integrating with resources provisioned outside of Terraform.
10. **Using Multiple Users in Terraform:**
    * **Question:** How would you configure Terraform to use multiple users within a team for infrastructure management?
    * **Answer:** Terraform supports multiple authentication methods and access controls, allowing teams to collaborate on infrastructure management securely. By configuring different authentication credentials or access keys for each user, teams can ensure granular access control and accountability.
11. **Multi-Subscription Deployment:**
    * **Question:** Explain how Terraform facilitates multi-subscription deployments in cloud environments.
    * **Answer:** Terraform's flexible configuration management and provider-based architecture enable seamless deployment and management of resources across multiple cloud subscriptions, allowing organizations to maintain consistent infrastructure configurations and policies.
12. **Taint - Destroy/Create:**
    * **Question:** How does the **terraform taint** command affect resources in Terraform?
    * **Answer:** The **terraform taint** command marks a resource for destruction and recreation during the next **terraform apply** operation, ensuring that the resource's state is regenerated based on the current configuration, effectively forcing a recreate action.
13. **Format/Validate:**
    * **Question:** What is the purpose of the **terraform fmt** and **terraform validate** commands?
    * **Answer:** The **terraform fmt** command formats Terraform configuration files to ensure consistent code style and readability, while **terraform validate** checks the syntax and configuration of Terraform files for errors or misconfigurations, ensuring valid and functional configurations.
14. **Branching Strategy or Folder - Dev/QA/Prod:**
    * **Question:** Describe how you would implement a branching strategy or folder structure for managing Terraform configurations across different environments (e.g., development, QA, production).
    * **Answer:** Implementing a folder structure with separate directories for each environment allows for isolation and version control of Terraform configurations, ensuring distinct configurations and workflows for each environment while facilitating code reuse and collaboration.
15. **Backend Statefile:**
    * **Question:** What is the significance of the Terraform backend statefile, and how does it affect infrastructure management?
    * **Answer:** The Terraform backend statefile stores the current state of infrastructure managed by Terraform, providing a centralized repository for tracking resource configurations and dependencies. It enables collaborative workflows, state locking, and remote state management for improved reliability and scalability.
16. **Depends On:**
    * **Question:** How do you use the **depends\_on** attribute in Terraform, and what role does it play in resource management?
    * **Answer:** The **depends\_on** attribute establishes explicit dependencies between resources in Terraform, ensuring that certain resources are created or updated before others, effectively controlling the order of resource provisioning and ensuring correct resource dependencies.
17. **Count[Index] vs For-Each[Map/List]:**
    * **Question:** Compare and contrast the usage of **count[index]** and **for\_each[map/list]** in Terraform configurations.
    * **Answer:** Both **count[index]** and **for\_each[map/list]** are used to create multiple instances of resources in Terraform. While **count[index]** is based on a numerical index and creates a fixed number of resource instances, **for\_each[map/list]** is based on a map or list of values and allows for dynamic creation of resource instances based on variable input.
18. **Another Use Case of Count - Boolean/True-False:**
    * **Question:** Besides numerical values, how else can you use the **count** attribute in Terraform configurations?
    * **Answer:** In addition to numerical values, the **count** attribute can be used with boolean or true-false expressions to conditionally create or disable resource instances based on specific criteria or conditions, providing flexibility in resource provisioning.
19. **Map vs List:**
    * **Question:** Explain the differences between maps and lists in Terraform, and when you would use each data structure.
    * **Answer:** Maps and lists are both data structures used to organize and manage data in Terraform. Maps are key-value pairs

20. \*\*Providers Files:\*\*

- \*\*Question:\*\* What are providers files in Terraform, and how are they used?

- \*\*Answer:\*\* Providers files in Terraform define the configuration for various cloud providers or services that Terraform interacts with to manage infrastructure resources. These files specify provider settings such as authentication credentials, endpoints, and default configurations, allowing Terraform to establish connections and provision resources in the respective environments.

21. \*\*Component - Resource, Providers, Data, Var, Output, Backend, Module, Init/Plan/Apply:\*\*

- \*\*Question:\*\* Explain the different components of Terraform configurations and their roles in infrastructure management.

- \*\*Answer:\*\* Terraform configurations consist of various components:

- Resources define the infrastructure resources to be provisioned or managed.

- Providers specify the cloud providers or services used to provision resources.

- Data sources retrieve information from external sources for use in configurations.

- Variables (`var`) allow for parameterization and customization of configurations.

- Outputs (`output`) define values to be displayed after applying changes.

- Backend configuration specifies settings for storing Terraform state files remotely.

- Modules encapsulate reusable configurations for provisioning infrastructure components.

- Init, plan, and apply are the three main steps in the Terraform workflow: initialization, planning changes, and applying changes to infrastructure.

22. \*\*Lifecycle:\*\*

- \*\*Question:\*\* What is the lifecycle in Terraform, and how does it impact resource management?

- \*\*Answer:\*\* The Terraform lifecycle refers to the sequence of actions performed during resource creation, updating, or deletion. It includes stages such as create, read, update, and delete (CRUD), along with additional lifecycle hooks for customization and automation. Understanding the lifecycle helps ensure proper resource management and behavior in Terraform configurations.

23. \*\*Destroy Specific Resource:\*\*

- \*\*Question:\*\* How do you destroy a specific resource in Terraform without affecting other resources?

- \*\*Answer:\*\* To destroy a specific resource in Terraform, you can use the `terraform destroy -target` option followed by the resource identifier. This command selectively destroys the specified resource while leaving other resources intact, allowing for targeted infrastructure changes.

24. \*\*Root and Child Module:\*\*

- \*\*Question:\*\* What is the difference between a root module and a child module in Terraform?

- \*\*Answer:\*\* A root module is the top-level directory containing the main Terraform configuration files and serves as the entry point for managing infrastructure. Child modules are reusable configurations stored in separate directories that can be called from within the root module or other child modules, enabling modular and scalable infrastructure designs.

25. \*\*Var and Output:\*\*

- \*\*Question:\*\* How do variables (`var`) and outputs (`output`) differ in Terraform, and when would you use each?

- \*\*Answer:\*\* Variables (`var`) are used to parameterize configurations and pass values dynamically, while outputs (`output`) define values to be exposed after applying changes. Variables are inputs to configurations, while outputs are the results or outputs of configurations, useful for sharing information between resources or modules.

26. \*\*Workspace - Default/Dev/QA/Prod:\*\*

- \*\*Question:\*\* What are workspaces in Terraform, and how do they facilitate environment management?

- \*\*Answer:\*\* Workspaces in Terraform provide isolated environments for managing infrastructure configurations, allowing separate configurations and state files for different environments such as development, QA, and production. Workspaces enable teams to work concurrently on different environments while maintaining configuration consistency and isolation.

27. \*\*Resource Graph:\*\*

- \*\*Question:\*\* What is the resource graph in Terraform, and how does it aid in infrastructure management?

- \*\*Answer:\*\* The resource graph in Terraform represents the dependency relationships between resources defined in configurations. It visualizes the order of resource creation, update, and deletion, enabling Terraform to orchestrate changes efficiently and predictably across infrastructure components.

28. \*\*Null Resource = Optional:\*\*

- \*\*Question:\*\* Explain the concept of null resources in Terraform and their role in configuration management.

- \*\*Answer:\*\* Null resources in Terraform represent placeholders for arbitrary actions or configurations that do not correspond to a specific infrastructure resource. They are often used for performing tasks such as local provisioning, script execution, or orchestration of external resources, providing flexibility and extensibility in configuration management.

29. \*\*Provisioner - Exec, Local, Zip, Remote:\*\*

- \*\*Question:\*\* What are provisioners in Terraform, and what types are available?

- \*\*Answer:\*\* Provisioners in Terraform are used to perform actions on local or remote resources during resource creation or modification. Different types of provisioners include `exec` for executing commands, `local-exec` for executing commands locally, `remote-exec` for executing commands on remote machines via SSH, and `file` for transferring files to remote machines.

30. \*\*Implicit and Explicit Depends:\*\*

- \*\*Question:\*\* Differentiate between implicit and explicit dependency management in Terraform.

- \*\*Answer:\*\* Implicit dependencies in Terraform are automatically inferred based on resource relationships defined in configurations, while explicit dependencies are explicitly specified using the `depends\_on` attribute. Implicit dependencies are determined by Terra

**GIT – SOURCE CODE MANAGEMENT**

1. \*\*Git - Azure Repo and GitHub:\*\*

- \*\*Question:\*\* How do you differentiate between Azure Repos and GitHub in terms of Git repository hosting?

- \*\*Answer:\*\* Azure Repos is a Git repository hosting service provided by Microsoft Azure, primarily integrated with Azure DevOps for seamless CI/CD pipelines and collaboration. On the other hand, GitHub is a widely-used Git repository hosting platform, known for its extensive community, open-source projects, and collaboration features beyond version control.

2. \*\*Branching Strategy in Current Project:\*\*

- \*\*Question:\*\* Can you describe the branching strategy used in our current project?

- \*\*Answer:\*\* In our current project, we follow a Git branching strategy based on GitFlow, with main branches such as `master` and `develop`, feature branches for new developments, release branches for stable releases, and hotfix branches for addressing critical issues in production.

3. \*\*Pull Request:\*\*

- \*\*Question:\*\* What is a pull request, and how does it facilitate collaboration in Git-based projects?

- \*\*Answer:\*\* A pull request (PR) is a mechanism in Git for proposing changes and initiating code review before merging them into the main codebase. It allows team members to review, discuss, and provide feedback on proposed changes, ensuring code quality, consistency, and alignment with project goals.

4. \*\*Branch:\*\*

- \*\*Question:\*\* Explain the concept of a branch in Git and its significance in version control.

- \*\*Answer:\*\* In Git, a branch is a parallel line of development that diverges from the main codebase, allowing developers to work on features, fixes, or experiments independently without affecting the main codebase. Branches facilitate collaboration, experimentation, and isolation of changes before merging them back into the main branch.

5. \*\*Pull = Fetch + Merge:\*\*

- \*\*Question:\*\* What is the relationship between `pull`, `fetch`, and `merge` commands in Git?

- \*\*Answer:\*\* In Git, the `pull` command combines the actions of `fetch` and `merge`. It retrieves changes from the remote repository using `fetch` and then merges them into the current branch using `merge`, ensuring synchronization of local and remote repositories.

6. \*\*Merge:\*\*

- \*\*Question:\*\* How does the `merge` operation work in Git, and when would you use it?

- \*\*Answer:\*\* The `merge` operation in Git combines changes from different branches into the current branch, integrating new features, fixes, or updates. It is commonly used to incorporate changes from feature branches into the main branch or to resolve divergent changes between branches.

7. \*\*Merge Conflict:\*\*

- \*\*Question:\*\* What is a merge conflict in Git, and how do you resolve it?

- \*\*Answer:\*\* A merge conflict occurs when Git is unable to automatically merge changes from different branches due to conflicting modifications to the same file or lines of code. To resolve a merge conflict, developers need to manually reconcile the differences, choose the desired changes, and commit the resolution.

8. \*\*Commit History:\*\*

- \*\*Question:\*\* Why is the commit history important in Git, and how do you review it?

- \*\*Answer:\*\* The commit history in Git provides a chronological record of changes made to the repository, including who made the changes, when they were made, and the nature of the changes. Reviewing the commit history helps track project progress, understand code evolution, and identify contributors and their contributions.

9. \*\*Checkout Branch:\*\*

- \*\*Question:\*\* How do you switch to a different branch in Git using the `checkout` command?

- \*\*Answer:\*\* To switch to a different branch in Git, you can use the `checkout` command followed by the name of the target branch. This command updates the working directory and the HEAD pointer to the specified branch, allowing you to continue work on that branch.

10. \*\*Create Branch:\*\*

- \*\*Question:\*\* What is the process for creating a new branch in Git, and why would you create a branch?

- \*\*Answer:\*\* To create a new branch in Git, you can use the `git checkout -b` command followed by the desired branch name. Branches are created to work on new features, bug fixes, or experiments without directly modifying the main codebase, providing isolation and versioning for changes.

11. \*\*Delete Branch:\*\*

- \*\*Question:\*\* How do you delete a branch in Git, and when would you delete a branch?

- \*\*Answer:\*\* To delete a branch in Git, you can use the `git branch -d` command followed by the name of the branch to be deleted. Branches are typically deleted after their changes have been merged into the main codebase, or if they are no longer needed for ongoing development or maintenance.

12. \*\*Reset:\*\*

- \*\*Question:\*\* What does the `git reset` command do, and how is it used in Git?

- \*\*Answer:\*\* The `git reset` command is used to reset the current branch to a specific state, undoing changes or resetting the staging area. It is commonly used to undo local commits, unstage changes, or move the HEAD pointer to a different commit in the commit history.

13. \*\*Git Diff:\*\*

- \*\*Question:\*\* What is the purpose of the `git diff` command in Git, and how is it used?

- \*\*Answer:\*\* The `git diff` command compares changes between different versions of files in the repository, showing the differences line by line. It is used to review changes before committing, identify modifications between branches, or track file changes over time.

14. \*\*Git Stash:\*\*

- \*\*Question:\*\* What is the `git stash` command used for in Git, and when would you use it?

- \*\*Answer:\*\* The `git stash` command is used to temporarily store changes in the working directory without committing them, allowing you to switch branches or work on other tasks. It is useful when you need to switch contexts quickly or save work in progress temporarily.

15. \*\*Git Tag:\*\*

- \*\*Question:\*\* What is a Git tag, and how is it used to mark significant points in a repository's history?

- \*\*Answer:\*\* In Git, a tag is a reference pointing to a specific commit in the repository's history, marking it as a significant point such as a release or milestone. Tags are immutable and provide a stable reference for versioning, release management, and historical tracking of software releases.

1. **Git Release:**

* **Question:** How do you manage releases in Git, and what role do tags play in the release process?
* **Answer:** Git releases are managed by creating tags that mark specific commits as release points. These tags serve as stable references for versioning and release management, enabling teams to track and distribute software releases efficiently.

1. **Git Rebase:**

* **Question:** What is Git rebase, and when would you use it in a collaborative development environment?
* **Answer:** Git rebase is a command used to reapply commits from one branch onto another, typically to integrate changes from one branch into another cleanly. It is commonly used to maintain a linear commit history, resolve merge conflicts, or incorporate changes from a feature branch into the main branch.

1. **Add, Commit, Push:**

* **Question:** Describe the typical workflow for adding, committing, and pushing changes to a Git repository.
* **Answer:** The workflow involves adding modified files to the staging area using **git add**, committing staged changes to the local repository using **git commit**, and pushing committed changes to a remote repository using **git push**, ensuring synchronization and collaboration in distributed development environments.

**AZURE DEVOPS PIPLEINE**

1. \*\*Explain the benefits of CI/CD pipelines:\*\*

- \*\*Answer:\*\* CI/CD pipelines automate the process of building, testing, and deploying code changes, leading to shorter development cycles, faster time to market, improved code quality, and reduced manual errors. They promote collaboration, consistency, and repeatability in software delivery, enabling teams to deliver high-quality software with greater efficiency and reliability.

2. \*\*Steps involved in Build and Release:\*\*

- \*\*Answer:\*\* The steps involved in build and release typically include:

1. \*\*Build:\*\* Compilation of source code, running unit tests, and generating artifacts.

2. \*\*Test:\*\* Execution of automated tests to ensure code quality and functionality.

3. \*\*Package:\*\* Creating deployable artifacts or packages from built code.

4. \*\*Release:\*\* Deploying packaged artifacts to target environments, such as staging or production.

5. \*\*Monitor:\*\* Monitoring application performance and collecting feedback for continuous improvement.

3. \*\*Basic things required to set up CI/CD pipelines:\*\*

- \*\*Answer:\*\* To set up CI/CD pipelines, you need:

- Source code repository (e.g., Git)

- CI/CD tool (e.g., Azure DevOps, Jenkins)

- Build and deployment scripts or configurations

- Testing frameworks and environments

- Target deployment environments (e.g., staging, production)

- Integration with version control, issue tracking, and monitoring tools.

4. \*\*Explain your pipeline strategy:\*\*

- \*\*Answer:\*\* My pipeline strategy focuses on automation, reliability, and scalability. It includes:

- Continuous integration to validate code changes and detect issues early.

- Automated testing to ensure code quality and prevent regressions.

- Continuous delivery to deploy changes consistently across environments.

- Infrastructure as code to manage and provision infrastructure in a repeatable and reliable manner.

- Monitoring and feedback loops for continuous improvement and optimization.

5. \*\*How do you create a Terraform pipeline, explain the stages and all:\*\*

- \*\*Answer:\*\* To create a Terraform pipeline, I define stages such as:

- \*\*Init:\*\* Initialize Terraform environment and install dependencies.

- \*\*Plan:\*\* Generate an execution plan to preview changes before applying.

- \*\*Validate:\*\* Validate Terraform configurations for syntax and compliance.

- \*\*Apply:\*\* Apply changes to infrastructure based on the execution plan.

- \*\*Destroy:\*\* Optionally, destroy infrastructure resources after testing or deployment.

Each stage is automated using Terraform commands and integrated into the CI/CD pipeline.

6. \*\*What are variable groups in CI/CD pipelines?\*\*

- \*\*Answer:\*\* Variable groups in CI/CD pipelines are sets of variables or secrets shared across multiple pipeline jobs or stages. They enable centralized management and secure storage of sensitive information such as API keys, passwords, or connection strings, reducing duplication and ensuring consistent configuration across pipelines.

7. \*\*What are the agents in CI/CD Pipelines?\*\*

- \*\*Answer:\*\* Agents in CI/CD pipelines are computing resources responsible for executing pipeline jobs or tasks. They run build, test, and deployment processes on target environments, either as self-hosted agents (on-premises or virtual machines) or Azure-hosted agents (provided by the CI/CD service).

8. \*\*Difference between Self-hosted and Azure-hosted Pipelines:\*\*

- \*\*Answer:\*\* Self-hosted pipelines use agents deployed and managed by the user on their own infrastructure, offering greater control and customization but requiring maintenance and resource provisioning. Azure-hosted pipelines use agents provided by the CI/CD service, offering scalability, reliability, and managed infrastructure but with fewer customization options and potential network latency.

9. \*\*Do you know about agents?\*\*

- \*\*Answer:\*\* Agents in CI/CD pipelines are responsible for executing pipeline jobs or tasks. They run build, test, and deployment processes on target environments. Agents can be self-hosted (deployed and managed by the user) or Azure-hosted (provided by the CI/CD service), and they play a crucial role in the execution and automation of CI/CD workflows.