**1. Technology Choices:**

AWS Lambda

* **Reason for Choice**: AWS Lambda is chosen for its serverless architecture, which allows you to run code without provisioning or managing servers. It's cost-effective (pay-per-use pricing model) and scales automatically.
* **Use Case**: It's ideal for handling backend operations of web applications, especially when workload varies.

Amazon S3

* **Reason for Choice**: Amazon S3 is used for its robust, secure, and scalable object storage. It's a standard choice for storing files and static content.
* **Use Case**: In this case, it's used for storing the zipped Lambda function code.

AWS API Gateway

* **Reason for Choice**: AWS API Gateway serves as a fully managed service that makes it easy to create, publish, maintain, monitor, and secure APIs.
* **Use Case**: It acts as a front door to access data, business logic, or functionality from the Lambda function.

AWS CloudWatch

* **Reason for Choice**: CloudWatch is utilized for its powerful monitoring and management service that provides data and actionable insights.
* **Use Case**: It's used for monitoring Lambda executions and API Gateway, and for triggering alerts.

AWS IAM

* **Reason for Choice**: IAM (Identity and Access Management) is crucial for securely controlling access to AWS services.
* **Use Case**: It's used to manage permissions, ensuring that Lambda functions and other services have the necessary rights to run.

AWS SNS

* **Reason for Choice**: SNS (Simple Notification Service) is a flexible, fully managed pub/sub messaging and mobile notifications service.
* **Use Case**: It's used for sending notifications (like email alerts) when CloudWatch alarms are triggered.

Terraform Cloud Integration

* **Terraform Cloud**: It's a cloud-based infrastructure as code (IaC) service that helps in automating the deployment and management of infrastructure. Terraform Cloud provides features like version control, automated plan and apply phases, state management, drift detection, and team collaboration.

**CI/CD Workflow – Terraform Cloud**

1. **Version Control Integration**: Connect your Terraform Cloud workspace to your version control system, where your Terraform scripts are stored.
2. **Automated Plan and Apply**: When changes are committed to the repository, Terraform Cloud automatically triggers a plan to show what will happen when the changes are applied. Once reviewed and approved, it applies the changes to deploy the updated infrastructure.
3. **Remote State Management**: Terraform Cloud stores the state of your infrastructure remotely, ensuring consistency and enabling collaboration among team members.
4. **Workspaces for Environments**: Use different workspaces in Terraform Cloud for managing different environments (e.g., development, staging, production).

**Process:**

1. **Local Testing:**
   * Initially, I’ve run the Terraform scripts locally. This is a crucial first step, allowing me to catch syntax errors, configuration issues, and ensure basic functionality.
   * During local testing, **terraform plan** is used to preview changes without applying them, and **terraform apply** is executed to create resources. This helps in validating the script's logic and resource configurations.
2. **Version Control and Terraform Cloud Integration:**
   * After local testing, the code is committed to a version control system (GitHub).
   * Terraform is configured to run tests like: terraform validate, terraform fmt, tflint, terraform-compliance. For unit testing, terratest can also be implemented in the pipeline.
   * Terraform Cloud is configured to automatically trigger hooks(pre-apply, post-apply), and runs upon code commits and test validation. This integration ensures that any changes in the repository automatically initiate planning and applying phases, as well as hooks, in Terraform Cloud.
3. **Remote Execution in Terraform Cloud:**
   * Terraform Cloud executes runs in a consistent remote environment. This includes running **terraform plan** to create an execution plan and **terraform apply** to apply the changes.
   * The remote execution environment provides consistency and avoids issues that might arise from different local environments.
4. **Review and Approve Changes:**
   * In Terraform Cloud, the plan generated for each run can be reviewed. For critical environments like production, you can enforce manual approval before changes are applied.
   * This step ensures that any potential impact of changes is reviewed and approved by a responsible team member, adding a layer of safety.
5. **Monitoring and Feedback:**
   * Use AWS CloudWatch or other monitoring tools to observe the behavior and performance of the deployed resources.
   * Monitor the Terraform Cloud run outputs and state changes for any anomalies or unexpected behavior.

**Deployment Process**

1. **Push to Version Control System:**
   * Finalized and tested Terraform configurations are committed and pushed to the VCS repository.
2. **Automatic Trigger in Terraform Cloud:**
   * The push to the VCS triggers a run in Terraform Cloud. Terraform Cloud fetches the latest code and initiates the planning phase.
3. **Review and Apply Changes:**
   * For each triggered run, review the plan in Terraform Cloud. Ensure all changes are as expected.
   * For non-production environments, the changes can be auto-applied. For production, enforce manual approval to apply the changes.
4. **Ongoing Updates and Maintenance:**
   * Subsequent updates to the Terraform code are tested locally, pushed to the VCS, and processed through Terraform Cloud, following the same review and approval process.
   * Continuous integration is achieved by regularly pushing updates to the repository, with Terraform Cloud handling the continuous deployment aspect.

By following this process, you can ensure that the solution is not only thoroughly tested but also that its deployment is controlled, predictable, and repeatable. Terraform Cloud acts as a central point of execution, providing an audit trail and a consistent environment for Terraform runs.