

Designing a Multi-Tenant SaaS Architecture on AWS

Objective:

This technical challenge is designed to assess your ability to design a scalable, secure, and cost-effective cloud architecture for a multi-tenant Software-as-a-Service (SaaS) application on AWS. Additionally, you will be expected to outline a corresponding CI/CD plan and implementation of a single module using Terraform.

Scenario:

As the Cloud Architect for a new SaaS product, you are responsible for designing a system that supports multiple tenants within a single, consolidated application deployment. The application consists of a traditional web app, a backend API, and a database.

- A **React single-page application** that serves as the user interface.
- A **Python-based backend** that provides a RESTful API for the frontend.
- A relational or NoSQL database.

A key requirement is that the application components **must be built and deployed as Docker containers**.

To ensure a successful launch and future scalability, we require a well-designed cloud architecture and a robust CI/CD pipeline. For this challenge, please keep the design generic, focusing on core principles and AWS best practices.

The Challenge

This technical challenge is designed to assess your skills in three key areas:

1. **High-Level Architectural Design:** Propose a cloud architecture for the multi-tenant SaaS application on AWS.
2. **CI/CD Blueprint:** Design a CI/CD pipeline to automate the build, test, and deployment of the container images.
3. **Hands-On Implementation:** Writing secure and reusable Infrastructure as Code (IaC).

No separate written documents are required for Parts 1 and 2; your diagrams must be detailed enough to be self-explanatory.

Part 1: SaaS Architecture

- **Architectural Diagram:** Create a clear, detailed architectural diagram. This diagram must show the container orchestration layer, a container registry, how the React frontend is delivered to users, how it communicates with the Python API, and how all underlying AWS services interact.
- **Architectural Decisions and Justifications:**
 - **Frontend Hosting & Delivery:** While the build process is containerized, how would you host and distribute the static assets of the React application (e.g., HTML, CSS, JavaScript files) to ensure low latency for global users?
 - **Multi-Tenancy Approach:** Explain how your architecture will handle multi-tenancy. Describe your strategies for isolating tenant data and compute resources to prevent "noisy neighbor" problems.
 - **Compute Layer:** Describe the AWS compute services you would use and explain why they are a good fit.
 - **Database Layer:** Detail your choice of database and the strategy for data isolation between tenants. Explain how you would ensure one tenant's data is not accessible to another.
 - **Networking Layer:** Detail the network foundation for your application.
 - **VPC Design & Subnet Strategy:** Describe your Virtual Private Cloud (VPC) layout, subnet structure, and multi-AZ strategy.
 - **Traffic Routing & Security:** Explain how you will manage traffic flow and secure your network, paying special attention to how traffic reaches your containers (e.g., Application Load Balancer).
 - **Security, Identity, and Compliance:** Outline your end-to-end security strategy and describe how you will handle:
 - **Identity and Access Management (IAM):** How will the application and your team access AWS resources securely?
 - **Data Protection:** How will you enforce encryption at rest and in transit?

- **Secrets Management:** Where will you store sensitive information like database credentials and API keys?
 - **Threat Detection:** Which services or strategies would you use to monitor for malicious activity?
- **Billing, Metering, and Cost Management:** Describe the architectural components and strategy you would implement to track tenant-specific usage for billing purposes.
 - How would you collect tenant consumption data (e.g., API calls, storage used)?
- **Observability Layer:** Describe your strategy for achieving comprehensive observability (monitoring, logging, and tracing). Which AWS services would you use, and how would you filter metrics and logs on a per-tenant basis?
- **Scalability and High Availability:** Explain how your architecture will scale to accommodate a growing number of tenants and how you will ensure high availability.

Deliverable for Part 1:

- Submit a single document containing a detailed architectural diagram. This diagram must visually communicate your entire design by showing the end-to-end flow and including all layers.

Part 2: CI/CD Blueprint

- **CI/CD Pipeline Diagram:** Create a diagram that visualizes the stages of your CI/CD pipeline, from code commit to production deployment.
- **CI/CD Pipeline Design and Justification:**
 - **Tools and Services:** Specify the tools & services you would use to build your CI/CD pipeline and explain the role of each.
 - **Pipeline Stages:** Describe the key stages of your pipeline (e.g., Source, Build, Test, Deploy).
 - **Testing Strategy:** Briefly outline your automated testing strategy (e.g., unit tests, integration tests, security scans).

- **Deployment Strategy:** Describe your chosen deployment strategy (e.g., blue/green, canary) and justify its suitability for this SaaS application.

Deliverable for Part 2:

- Submit a single, detailed CI/CD process flow diagram. The diagram must visually map the entire process from a git push to a live deployment, showing distinct, parallel pipelines for the React frontend and Python backend.

Evaluation Criteria

Your submission will be evaluated based on the following:

- **Clarity and Completeness:** Provide clear diagrams and thorough explanations that cover all relevant aspects of your solution.
- **Technical Soundness:** Ensure your architecture is robust, feasible, and suitable for the application's requirements.
- **Justification of Decisions:** Clearly explain your selections of technologies, deployment methods, and design choices.
- **Best Practices:** Your application of AWS best practices for security, scalability, cost-optimization, and operations in a multi-tenant environment.
- **Simplicity and Practicality:** Deliver a solution that avoids unnecessary complexity and can be efficiently implemented.

Part 3: Hands-On Terraform Challenge

This part of the challenge tests your practical IaC skills and your ability to apply security best practices.

Task:

Develop a reusable **Terraform module** that provisions an AWS Lambda function running from a **container image**.

Requirements for the Terraform Module:

Your module should be production-ready and adhere strictly to AWS security best practices.

1. **Module Structure:** The code must be structured as a proper Terraform module (e.g., using main.tf, variables.tf, outputs.tf).

2. **IAM Role (Least Privilege):**

- The module must create a dedicated IAM Role for the Lambda function.
- The associated IAM Policy must follow the principle of least privilege. For this challenge, assume the Lambda only needs permission to write logs to CloudWatch Logs. **Do not use AWS managed policies like AWSLambdaBasicExecutionRole.** You must create your own policy.

3. **Container Image Source:**

- The Lambda function must be configured to use a container image as its source.
- You can use a placeholder URI for the ECR image (e.g., 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-app:latest). This should be a configurable variable.

4. **Networking:**

- The Lambda function must be deployed into a VPC.
- It must be placed in **private subnets**. The IDs of the VPC and subnets should be passed as variables to the module.
- The module must create and attach a dedicated **Security Group** for the Lambda function. This security group should have no ingress rules and a locked-down egress rule (e.g., allowing outbound traffic only on port 443 to 0.0.0.0/0 for accessing AWS APIs).

5. **Logging and Configuration:**

- Ensure the Lambda's logs are captured in an Amazon CloudWatch Log Group.
- Use environment variables for any non-sensitive configuration values.

6. **Tagging:**

- All resources created by the module (Lambda, IAM Role, Security Group, etc.) must be properly tagged for identification and cost management.

Deliverable for Part 3:

- Submit your complete Terraform module as a .zip archive or a link to a public Git repository. The code should be well-commented and include a README.md file explaining how to use the module.

Evaluation Criteria

Your submission will be evaluated based on the criteria from the previous versions, with the addition of:

- **Terraform Proficiency:** The quality, correctness, and reusability of your Terraform code.
- **Security Implementation:** Your practical application of the principle of least privilege and other security best practices in the Terraform module.
- **Adherence to Best Practices:** Your use of standard IaC patterns for variables, outputs, and module structure.