Prefect ECS Assignment Report

1. Why Choose Terraform Over CloudFormation?

1.1 Leverage Existing Expertise

Given my prior experience with Terraform, it offers several advantages:

- **Accelerated Development and Deployment**: Familiarity with Terraform's syntax and workflows enables quicker infrastructure provisioning.

- **Reduced Learning Curve**: Avoids the need to learn CloudFormation's distinct syntax and tooling.

1.2 Multi-Cloud and Provider-Agnostic Support

Terraform supports multiple cloud providers (e.g., AWS, Azure, Google Cloud) and services like GitHub and Datadog. This flexibility is beneficial for managing resources across different platforms or adopting a multi-cloud strategy in the future.

1.3 Modular and Reusable Configurations

Terraform's module system promotes code reuse, enabling:

- **Standardized Infrastructure Components**: Facilitates consistency across deployments.

- **Simplified Complex Deployments**: Breaks down infrastructure into manageable modules.

- **Enhanced Team Collaboration**: Encourages sharing and collaboration through reusable modules.

1.4 Extensive Community and Ecosystem

Terraform boasts a vibrant community and a vast ecosystem of providers and modules, offering:

- **Access to Pre-Built Modules**: Accelerates development by leveraging existing solutions.

- **Community Support**: Facilitates troubleshooting and best practices through community engagement.
- **Continuous Contributions**: Ensures the tool remains up-to-date with industry trends.

---

2. Key Learnings About IaC, ECS, and Prefect

2.1 Infrastructure as Code (IaC) and ECS

While I had prior knowledge of Amazon ECS through the AWS Console, setting up ECS using Terraform was a new experience. This approach provided insights into automating infrastructure provisioning and managing ECS resources programmatically. Additionally, I learned about service discovery through private DNS namespaces, enhancing my understanding of ECS networking.

2.2 Prefect

Prefect was entirely new to me. I began by exploring its use cases, setting it up locally, and running demo flows on Prefect Cloud. This process helped me grasp fundamental concepts such as Flows, Deployments, and Work Pools. I also discovered various techniques for executing workflows, broadening my knowledge of workflow orchestration.

---

3. Challenges Faced and Resolutions

3.1 Docker Container Exiting Immediately

Issue: When running Prefect locally using a Docker image, the container exited immediately after creation.

Resolution: I realized that the container required a command to initiate the Prefect worker or agent upon startup. By specifying the appropriate command during container initialization, the issue was resolved.

3.2 Prefect API Key Not Passing Correctly

Issue: Despite providing the secret ARN in the module configuration, the Prefect API key was not being passed correctly.

Resolution: I had created the secret as a JSON object. In the ECS task definition's Secrets block, I needed to specify the specific key within the JSON to retrieve the correct value. Adjusting the configuration accordingly resolved the issue.

3.3 Inability to Create Work Pool on Prefect Cloud

Issue: I was unable to create a work pool on Prefect Cloud via the CLI or console.

Resolution: After extensive troubleshooting, I discovered that creating push-based or hybrid work pools requires a paid Prefect Cloud plan. Without the appropriate plan, ECS tasks would fail upon container startup due to the absence of a valid work pool.

---

4. Demo Video

I have documented the entire implementation process in the following recorded video:

👉 Watch the Demo

---

5. Suggestions for Improving the Setup

- **Service Auto Scaling:** Configure ECS Service Auto Scaling to adjust your worker count based on real-time metrics (e.g. CPU, memory, or custom Prefect queue depth). This ensures capacity scales up during spikes and down in idle periods, optimizing performance and cost.

- **Container & Image Security:** Store your own Prefect-based worker image in ECR and enable ECR Image Scanning to catch vulnerabilities & Enforce an image-build pipeline that runs `docker scan` (or Trivy) and rejects insecure builds.

- **Disaster Recovery & Backup:** Store Terraform state remotely (S3 + DynamoDB locking) with versioning. Back up CloudWatch Logs and periodically snapshot critical resources. Design multi-region ECS failover for critical workflows.
