Blossom Deployment Test – Design Decisions & Implementation Report

As a starting point for the project, I began with a local deployment to test and visualize the interactivity of the fully containerized application. I initially encountered a small issue on my Windows machine, which was my main workstation for this project, it wouldn't run the `docker-compose up --build` command due to a folder path issue. To avoid wasting time, I switched to a Mac, where everything ran smoothly, allowing me to visualize the app and proceed.

With a better understanding of the application, I was able to start making architecture decisions that were cost-effective, secure, and highly scalable.

The first big step was migrating the frontend from **Create React App** to **Vite**, which made the deployment to **AWS Amplify** much smoother. The decision to use **Amplify** came from its seamless integration with GitHub, built-in CI/CD capabilities, and native support for modern frontend frameworks. This migration led to **faster builds**, **easier configuration**, and an overall better **developer experience**, while keeping infrastructure overhead minimal, which was especially useful given the time constraints.

Next, I noticed the database was initially running inside a Docker container. While that setup works, I knew that for long-term scalability and resilience, switching to a managed database like **Amazon RDS** was the better choice. With RDS, I gained access to powerful features like automated backups, Multi-AZ replication for high availability, and built-in monitoring. To improve security, I used **Secrets Manager** to generate and store the database credentials securely, and I kept the RDS instance **fully isolated from the internet**, limiting access to only internal ECS tasks.

For the backend API, I kept the containerized approach but opted not to use traditional EC2-based containers. Instead, I deployed it using **ECS with Fargate**, a serverless container engine that automatically handles provisioning, scaling, and infrastructure management. This allowed me to scale the application based on demand and traffic received via the **Application Load Balancer (ALB)**, ensuring flexibility during peak usage and minimizing costs during low traffic periods.

To provision the infrastructure, I chose **CloudFormation** due to my familiarity with it and its extensive documentation. While I would have preferred using CDK, it required extra setup that I couldn’t accommodate given the time limitations. I organized the stack into multiple **nested stacks**, which made debugging much easier and allowed me to deploy individual parts of the infrastructure like networking or ECS independently, saving time and avoiding unnecessary redeployments.

For CI/CD, I used **GitHub Actions** because I knew I could quickly and securely integrate it with my AWS account using **OIDC**, removing the need to store AWS credentials in the repository. This also allowed me to keep a monolithic repository without needing to split the frontend, backend, and infrastructure into separate repos. While separating them might be a better long-term strategy, I chose to keep it unified for simplicity in this demo.

From a security standpoint, the IAM role used for deployments through GitHub Actions was configured with **least privilege policies**, and thanks to the OIDC setup, it's tightly scoped to a single GitHub repository, making it difficult for external actors to assume the role or misuse its permissions.

Along the way, I ran into a few issues. One of them was during the deployment to ECS, I had to step away after triggering the infrastructure deployment, and when I returned, the ECS stack was still in progress. After some investigation, I realized that ECS couldn't find the container image because it hadn't been uploaded yet. This meant I had to manually **push the image to ECR** from my local machine to allow the deployment to continue, a reminder that image upload timing is critical when coordinating infrastructure and application deployments.

Finally, the last issue I ran into, which I couldn't fully resolve due to budget constraints, was configuring **SSL certificates** for the **Application Load Balancer (ALB)**. Since I didn't have a custom domain available for this demo, I couldn't request a certificate through AWS Certificate Manager (ACM), which made it impossible to serve traffic over HTTPS. This resulted in a **Mixed Content** error when trying to interact with the backend from the Amplify frontend, since Amplify enforces HTTPS. As a workaround, I used the static **S3-hosted version of the frontend**, which runs over HTTP and allowed me to test the full integration without SSL, verifying that everything works as expected.