Cloud Computing Final Report

Spundun Gusain and Safa Abdalah

Our original goal was to deploy a version of Solitaire to the cloud that anyone could access from their browser. We knew we wanted to use Docker to containerize the game, Terraform to set up the AWS infrastructure, and GitHub Actions to automate the CI/CD pipeline. We also planned to explore the possibility of integrating Amazon S3 for persistent storage of user stats. Over the course of the project, we went through several iterations and ran into multiple roadblocks that forced us to adapt, reevaluate, and eventually reshape the project into something simpler and more realistic, but still valuable.

We started with a proposal to deploy **PySolFC**, a Python-based GUI solitaire game, onto an EC2 instance using Docker. We were hoping to expose the GUI through the browser using Guacamole or X11 forwarding. But we quickly realized that GUI applications are not designed to run in containers without a lot of extra layers. Docker wasn't going to be a good fit for this kind of desktop application, at least not without significantly increasing the complexity of the project. It didn't make sense to spend weeks trying to set up a GUI stack in Docker when the goal was to learn about deployment, automation, and cloud architecture—not desktop virtualization.

So we pivoted to **solitaire.gg**, a browser-based Solitaire game written in Scala. This seemed like a better fit at first, but the project had its own set of problems. It relied on outdated dependencies, including a version of SBT from 2016, and upgrading it caused compatibility issues with multiple plugins. We tried to resolve them, but every time we fixed one versioning issue, another one popped up. We realized that most of our time was being spent updating and debugging dependencies, not on the cloud or deployment side of things. That's when we moved to our final project direction using **Sawayama Solitaire**, which was a browser-based solitaire game written in modern JavaScript with TypeScript, PixiJS, and Anime.js. This choice aligned better with our project goals and let us focus on the infrastructure and deployment aspects instead of fighting the application itself.

Once we settled on Sawayama Solitaire, we containerized the app using Docker. The application listens on port 3000, so Safa created a Dockerfile based on the official Node.js image, installed all necessary dependencies using npm, added esbuild for TypeScript compilation, and then ran the app using npm run watch. We exposed port 3000 in the Docker container and tested that it worked locally. There were some issues initially with global dependencies and missing esbuild, but those were resolved through trial, error, and testing until the image built and ran reliably.

On the infrastructure side, we used **Terraform** to provision the AWS environment. Initially, we tried to define our own VPC and subnet configurations, but we ran

into problems getting internet access to work. Specifically, EC2 instances were being launched into subnets that weren't properly routed to an Internet Gateway, which meant they had no public IP and weren't reachable. After troubleshooting that for a while, we decided to simplify things by launching the EC2 instance into the **default VPC**, which we knew was correctly configured with public subnets and routing. This immediately solved the reachability issue, and from there we were able to assign a public IP and SSH into the instance as expected.

We also had to update our Docker installation steps on EC2. We were using an Amazon Linux 2023 AMI, and the traditional amazon-linux-extras command no longer worked because it had been deprecated. So we replaced it with a direct call to yum install docker -y, followed by starting and enabling the Docker service. This worked as expected and gave us a stable Docker environment to pull and run our container.

For the CI/CD pipeline, we used **GitHub Actions**. The workflow checks out the repo, logs in to Docker Hub using secrets, builds and pushes the Docker image, and then SSHs into the EC2 instance to stop and remove any existing containers and run the new one. Initially, the pipeline used a hardcoded public IP for the EC2 instance, which worked only sometimes and failed when the IP changed or wasn't reachable yet. We later added logic to dynamically fetch the public IP using AWS CLI commands, store it as an environment variable in the pipeline, and retry the SSH and Docker commands until the instance was confirmed to be

reachable. This included adding sleep and retry loops to account for EC2 startup time and DNS propagation.

We also ran into issues with the SSH key. We realized that the PEM key we stored in GitHub Secrets was missing the -----BEGIN RSA PRIVATE KEY----- and -----END RSA PRIVATE KEY----- headers. Without these, the key would be rejected during authentication. We fixed this by pasting the full key including the headers into GitHub Secrets under EC2_KEY. We also added an ssh-keyscan step to our pipeline to make sure the EC2 host was in known_hosts, and set the correct permissions using chmod 600 on the private key to avoid SSH warnings. After this, our SSH commands became reliable.

The last big technical hurdle was ensuring Docker ran on the EC2 instance without needing to reconfigure it manually every time. We included Docker installation in the Terraform user_data, but ultimately chose to let GitHub Actions handle the image pulling and running. This made the flow easier to troubleshoot and gave us better visibility into logs, errors, and step-by-step deployment.

There were also merge conflicts and branch management issues, mostly due to the fact that we were working in parallel. Safa focused on getting the Docker image to build and run correctly, while Spundun handled the Terraform infrastructure and CI/CD automation. We had separate branches for testing changes (sabdalah-branch and sgusain-branch), and sometimes these diverged far enough that Git merge required manual intervention. We used rebase, conflict

resolution, and careful coordination to keep our branches in sync and make sure everything was integrated back into main for deployment.

At this point, the project is complete. The EC2 instance is provisioned via Terraform. Docker is installed and running. GitHub Actions builds and pushes the image, then SSHs into the instance to deploy the latest version. The app is running on port 3000, accessible via the public IP of the EC2 instance. We verified the connection in the browser and confirmed that the container runs correctly after each update.

Looking ahead, we planned to store user statistics in **Amazon S3**, but we made the decision to push that to a future iteration. Our focus this semester was making sure the application deployed cleanly, could be updated automatically, and remained accessible. All of those goals were met.

In terms of contributions, Spundun worked on the infrastructure, Terraform setup, and CI/CD integration. Safa worked on Docker containerization, CI/CD debugging, and SSH configuration. Both of us worked together to test the deployment, debug runtime errors, and finalize the solution.

We ran into a lot of problems, but we solved them by reading documentation, experimenting, and taking the time to understand what wasn't working. We didn't just get the app working—we learned how to deal with real-world cloud issues, from security groups to dynamic IPs to automation and configuration drift. The

project may have looked different from the proposal on paper, but it taught us exactly what we hoped to learn when we started.