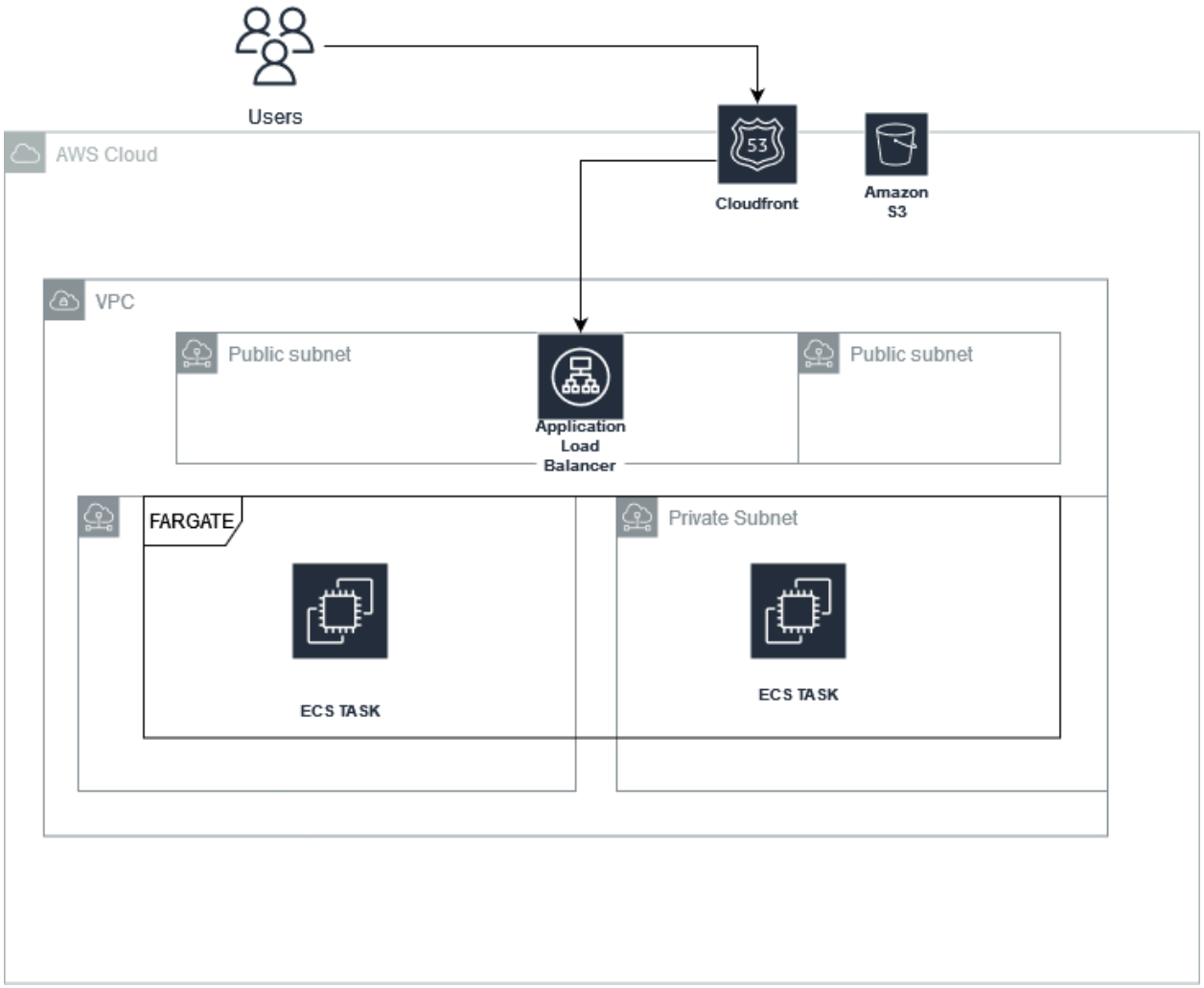
Building and Operating the Underlying Infrastructure

To support the development and deployment of the Java backend and React Single Page Application (SPA) in a scalable and highly available manner, the following services have been selected for the foundational infrastructure:

1. **Virtual Private Cloud (VPC):**
    - **Configuration:** One region with two Availability Zones (AZs), each containing one public and one private subnet, along with a NAT Gateway.
    - **Purpose:** The VPC isolates applications from the public internet, enabling fine-grained access control. Multiple AZs ensure high availability by preventing a single point of failure.
2. **Amazon S3 and CloudFront:**
    - **S3:** Stores the SPA assets (JS, CSS, HTML, Images) as a static website.
    - **CloudFront:** Acts as a global Content Delivery Network (CDN), ensuring efficient delivery and caching of static assets to users worldwide. This setup reduces latency and improves load times, enhancing user experience.
3. **Load Balancer:**
    - **Purpose:** Ensures high availability by distributing incoming traffic across multiple instances of the backend application, preventing any single instance from becoming a bottleneck.
4. **Amazon ECS:**
    - **Purpose:** Deploys the containerized backend application, leveraging the 12-factor app principles. ECS handles the orchestration of Docker containers, ensuring efficient resource usage and scaling.
    - **Auto Scaling:** With Fargate, we don't need to manage the underlying servers, it automatically provisions and scales the compute resource needed to run our containerized tasks.
5. **Amazon ECR:**
    - **Purpose:** Stores Docker images of the backend application, facilitating consistent and repeatable deployments.
6. **Terraform:**
    - **Purpose:** Provisions and manages the infrastructure. The Terraform configuration is modularized, allowing components to be included or excluded based on specific needs. Variables are defined in a configuration file, enabling easy adjustments and redeployments.



Java Backend Deployment Strategy

To meet the requirements of scalability, high availability, and frequent rollouts, the following steps and strategies are implemented:

1. **Containerization:**
    - The backend Java application is containerized using Docker. A multi-stage build process is employed to create a small and secure final image, which is then stored in Amazon ECR.
2. **CI/CD Pipeline:**
    - **CI Pipeline:** Triggered automatically upon a push to the backend repository. The pipeline builds the application, runs tests, and pushes the Docker image to ECR.
    - **Manual Deployment Pipeline:** Allows managers to control when new versions are deployed to production. This can be automated in the future if manual intervention is deemed unnecessary.

3. **Deployment Flow:**
    - Developers create pull requests on GitHub. Upon approval, the code is merged, triggering the CI pipeline.
    - Managers can then initiate the deployment pipeline, which updates the ECS service, ensuring that the new version is rolled out seamlessly.

Single Page Application (SPA) Deployment Strategy

The SPA is developed in React and is deployed using Amazon S3 and CloudFront for the following reasons:

1. **Amazon S3:**
    - **Purpose:** Serves as a cost-effective and scalable storage solution for static assets. S3 is designed for high availability and durability, ensuring that the assets are reliably stored and served.
2. **Amazon CloudFront:**
    - **Purpose:** Acts as a global CDN, caching the static assets in edge locations around the world. This setup significantly reduces latency and load times, providing a fast and responsive user experience regardless of the user's geographic location.
3. **CI/CD Pipeline:** automates the software development lifecycle. Whenever a change is committed to the code repository, the pipeline triggers a build process. This build process constructs the application and then synchronizes the latest version of the application with the S3 bucket for deployment.

By leveraging these AWS services, the infrastructure can support the expected growth in development teams and application components, ensuring that the deployment process remains efficient and that the applications remain highly available and scalable.

Introducing organizational constructs:

As the solution is providing based on AWS, therefore we can introduce organizational constructs based on :

Single AWS Account (Suitable for smaller deployments or limited teams):

To design a robust AWS IAM structure to support the different user groups and their respective responsibilities, we need to create a set of IAM policies and roles. Each policy will grant the necessary permissions for the user groups (infra, dev, and spa) to perform their tasks.

1. Define IAM policies for Each user group:

    - Infra user group: This group needs permissions to create and manage VPCs, S3 buckets, CloudFront distributions, ECR repositories, ECS clusters, and Application Load Balancers (ALBs).
    - Dev User group: This group needs permissions to deploy new tasks on given ECS clusters and create ECR repositories.
    - SPA User group: This group needs permissions to upload and delete objects in a given S3 bucket and add CloudFront behaviors to a particular distribution.
2. Create IAM Roles and Attach Policies:
    - Need to create IAM Roles for each user group and attach the policy to each role respectively.
3. Create IAM Groups and Attach role:
    - That will create IAM groups for each role and attach the role accordingly.
4. Lastly we add individual users to the group and get access keys and secret access keys, which we can use as secret values for our github repositories to trigger the pipeline.

Multiple AWS Accounts (For larger deployment and multiple teams):

We can utilize AWS Organizations to create a hierarchy of accounts such as a Central IAM Account for user management and Resource accounts for resource management.

Central IAM Account:

- Create roles (InfraUserRole, DevUserRole, SpaUserRole) with trust policies allowing resource accounts to assume them.
- Attach respective policies (InfraUserPolicy, DevUserPolicy, SpaUserPolicy) to these roles.
- Create user groups and add users.

Resource Accounts:

- Create roles with trust relationships allowing central IAM account roles to assume them.
- Attach policies to these roles that allow resource management.

Access Resources from Central IAM Account:

- Users in the central IAM account can assume roles in the resource accounts to manage resources
- The assume-role command returns temporary security credentials which can be used to manage resources in the resource account or use in github actions pipelines.

By implementing these organizational constructs with either a Single Account or Multi-Account strategy, you can empower development teams to manage their applications while maintaining security and control over the underlying infrastructure on AWS.