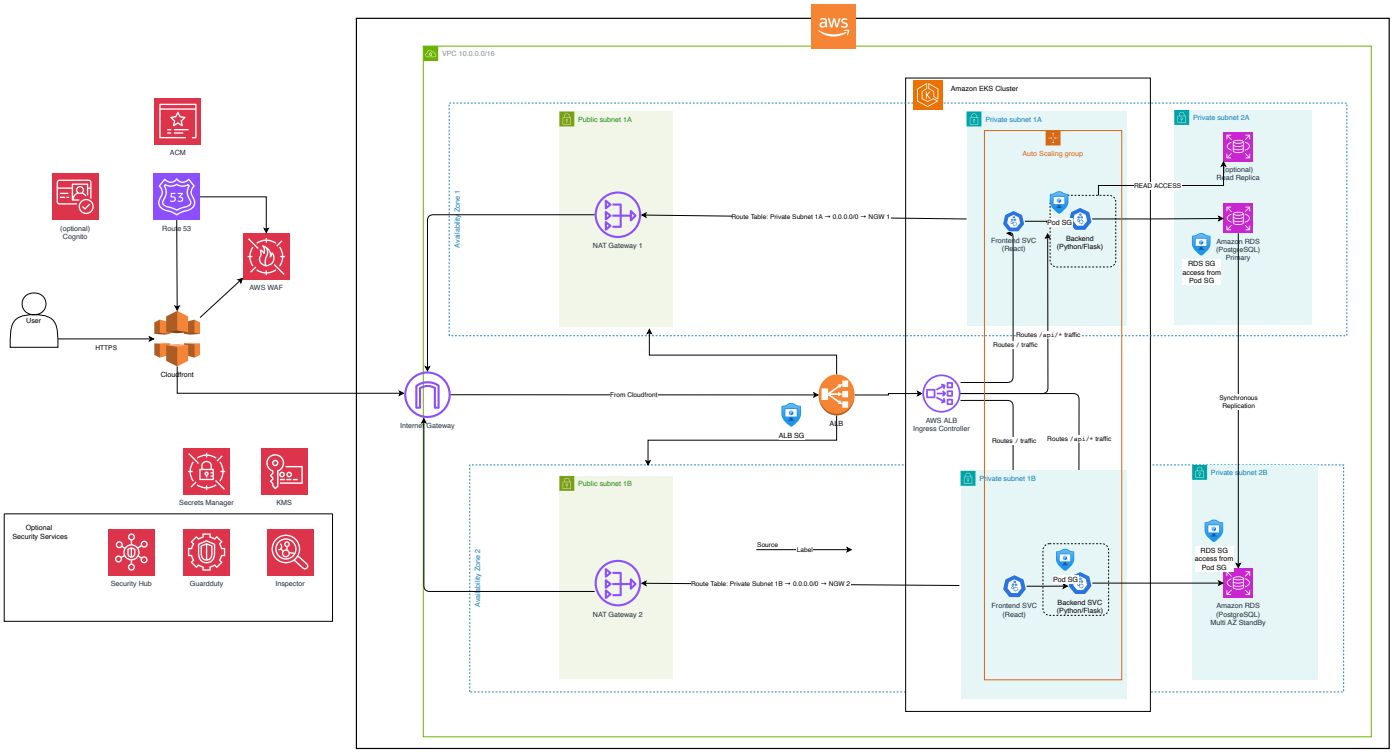
Innovate Inc.'s Cloud infrastructure Architecture

Requirements

Innovate Inc. is building a web app with a Python/Flask REST API server-side, a React single-page app (SPA) client-side, and a PostgreSQL database. Detailed requirements are in the file Requirements.md

Architecture diagram



AWS Accounts

According the AWS best security practices, the following AWS account structure is recommended:

- Development Account: Dev and staging/testing environments
- Production Account: Production deployment of the live application.
- Management / Shared Services Account: Logging, monitoring, and IAM roles, ECR repositories and CI/CD tools.

Note: If the Innovate does not yet use AWS SSO (IAM Identity Centre) for managing their AWS account, it is higly recommended to use it and manage all users in there, providing access to Dev, Prod accounts by assuming fine tuned defined roles with minimal permissions with short-term self expiring credentials

VPC Network Design

VPC Architecture: A single dedicated Amazon VPC with public and private subnets across at least two availability zones for availability (best practice is to use 3 AZ for production). For cost savings, the DEV and Staging clusters can use 2 AZ.

VPC will contain (for each AZ) a public subnet (load balancer, NAT gateway for outbound traffic from private subnets), a private subnet for EKS loads and separate private subnet for RDS deployment. EKS will be deployed using 1 public and one private subnet per AZ.

Security

The architecture is using Security Groups (optionally also Network ACLs) to restrict access.

All traffic from the internet if coming through Cloudfront (the DNS is configured to Cloudfront distribution).

The SG for the ALB only allows access from Cloudfront using Origin Access Control (OAC) and forwards traffic to pods using Ingress controller.

There is a separate SG that will be assigned to pods and will allow ingress access only from the ALB SG (plus the internal EKS security group allowing the management layer access of course).

The third SG assigned to the RDS, will allow access ONLY from the Pods SG, restricted on the database port.

This way the access to database will be limited to backend pods only.

Optionally (adds aditional complexity) we could deploy network policies inside EKS cluster to limit the access between pods and services.

Architecture recommends using AWS WAF to protect against common web threats (if needed) and Cloudfront to cache static resources.
NOTE: There may be cost implications for WAF, depending on the traffic and number of rules. The AWS Shield (DDoS protection) Basic is
included for free. Unless the site is frequent victim of DDoS, the Shield Advanced is probably an overkill - but can be easily enabled later on.

Computing Platform

Kubernetes Cluster (Amazon EKS), using HPA for pod scaling (needs Metrics server) and Karpenter to provide node level autoscaling.
EKS is using different node groups for front end and backend as there may be different scaling requirements (depends on the app).

NOTE: I would need more details on clients business objective / cost sensitivity / acceptable management overhead to go to more details here.

There are multiple options on how to design node groups (among OnDemand, spot, Fargate) and which type/size of EC2 for OnDemand is optimal. For some utilization patterns, even Fargate can be best option, but usually some combination of OnDemand node group and Spot for handling peeks / cost savings.

Also, the NodeGroups can be created by EKS (in essence a managed ASG) or by Karpenter. Using EKS nodegroups is simpler and allows specialized IAM roles, Karpenter needs more config but is faster and better for dynamic apps. More input from client would be needed.

Containerization Approach

- create Python/Flask backend and React frontend images in Docker
- save container images in Amazon Elastic Container Registry (ECR), placed in management account, using cross-account roles for secure access
- for CI/CD automation, we can use either AWS CodePipeline or (my personal preference) the Github Actions / Gitlab to create, tag and push the images

As an alternative, using 3rd party Image repository is possible (Dockerhub, Github) with comparable cost - this eliminates the ECR region cost for the DR environment, but needs extra user management outside of the AWS.

Database

Use Amazon RDS for PostgreSQL: managed database services, enable autoscaling and backups.

- Deploy a Multi-AZ RDS instance for high availability
- Consider using read replicas (requires code adaptation for different R/O and R/W endpoints in API if replication lag is OK)
- Configure automated backups and snapshots
- Encrypt data at rest using KMS managed key
- Use AWS Secrets Manager to store database passwords and to rotate them
- Use AWS Backup for scheduled snapshots and long-term retention. Encrypt backups

If Backups & Disaster Recovery is required (NOTE: cost implication, there is no mention in requirements but often comes up in real life):
Implement cross-region replication for disaster recovery of database backups and (optionally) ECR images.

NOTE: This is dependent on RTO / RPO / MTTR, and can double the cost of PROD environment (by running a different PROD in other region plus regular data transfer cost between regions, can become expensive quickly). Also it is important to consider data governance rules for countries with single AWS region (most European countries).

If performance is high on priority list, AWS offers Aurora database compatible with PostgreSQL with better performance, faster scaling and failover and serverless option for very variable traffic patterns. A detailed price computation would be required based on actual data size information and traffic patterns.

Security Best Practices

- Always Implement least privilege access with all IAM roles and policies.
- with AWS IAM Identity Center (SSO), only users managed in central account can assume role in Dev / Prod account, use multi-factor authentication (MFA) for all AWS-SSO IAM users. No local IAM users in PROD or DEV accounts
- use TLS for service-to-service communication
- use AWS ACM for TLS certificate management / renewal
- enable database encryption with RDS native encryption

Consider / propose to client using value-adding security services (cost implications):

- AWS Inspector for automated security vulnerability assessments (
  scans EC2 instances and container images for software vulnerabilities and unintended network exposure)
- consider Amazon GuardDuty (threat detection service that continuously monitors for malicious activity and unauthorized behavior)

Monitoring and Logging

- Use Amazon CloudWatch for application monitoring.
- Store logs in Amazon S3 with lifecycle policies for cost savings
- use Cloudtrail to store all trails of user activity in the management account
- use AWS Secrets Manager for safe credential storage (specially RDS passwords) and password rotation

If Inspector / GuardDuty is used, consider AWS Security Hub for a centralized security monitor.

CI/CD Pipeline

I would recommend cloud-native CI/CD tool (depending whether source code is on Github or Gitlab)

The high level phases (details depends on the CI/CD provider):

- Build: Package and build the application, create and push the DEV image. Triggered by merge request to develop
- Test: deploy latest images to DEV environment, run unit tests, sanity tests and integration tests
- Stage: deploy the application to Staging env, triggered by merge request to main
- Prod: triggered by tagging the images for PROD or by push to Helm chart repository (if using Argo)

As deployment tool I recommend using Helm charts / Kubernetes manifests, or as an alternative ArgoCD. This needs to be discussed with
the actual OPS team.

Additional considerations

I would like to make three additional suggestions:

First - the requirements does not specify how exactly the user identity / login / registration will be handled. One option to consider in AWS
would be Cognito service allowing registration, login, password reset (including login via social network accounts). In addition to that, it would allow finer-grain security when accessing the AWS resources.

Second, it is not clear where is the DNS for application deployed. AWS Route 53 would be a good option if needed. It integrates well with AWS Certificate Manager and other security services.

Third, (it may be an overkill for this case) an AWS API Gateway could be a useful part of the architecture if the application / developers need the OpenAPI support, additional authorization and request validation, API throttling and rate limiting to prevent abuse. It provides great integration with WAF, Cognito, routing and versioning of the APIs, canary deployments + blue/green deployments.

Cost Optimization consideration

- use EC2 Savings Plans / Reserved Instances for predictable workloads (on EKS side and definitely on RDS side)
- use AWS Cost Explorer (or tools like CloudCheckr) for cost monitoring.
- we may or may not be able to leverage AWS Spot Instances (depends on the application details), We also may want consider tool like Spot Ocean for rightsizing
- It is recommended to configure AWS Budgets, AWS CUR, Trusted Advisor and Alerts to track and manage spending
- It is necessary to make sure the consistent tagging strategy is in place to allow proper cost allocation and visibility (something to implement in Terraforming the infra)