IT457
Cloud Computing

Final Project Report

Group 39

Name: Pithadiya Kirtan
Student ID: 202412073

Name: Navadiya Krishn
Student ID: 202412052

Name: Kathrotia Harsh
Student ID: 202412036

1. PROJECT OVERVIEW

1.1 Introduction

Fixly represents a modern cloud-native application deployment showcasing the implementation of containerized microservices on Amazon Web Services infrastructure. The initiative demonstrates practical application of cloud computing principles, focusing on scalability, reliability, and automated deployment workflows.

1.2 Project Scope and Objectives

The primary goal centered on deploying a full-stack web application using AWS managed services, specifically leveraging containerization and serverless compute capabilities. Key objectives included:

- Establishing a production-grade cloud infrastructure with proper network isolation
- Implementing automated continuous integration and deployment pipelines
- Configuring load-balanced, highly available application services
- Ensuring secure communication between application components
- Demonstrating cost-effective cloud resource utilization

1.3 Application Description

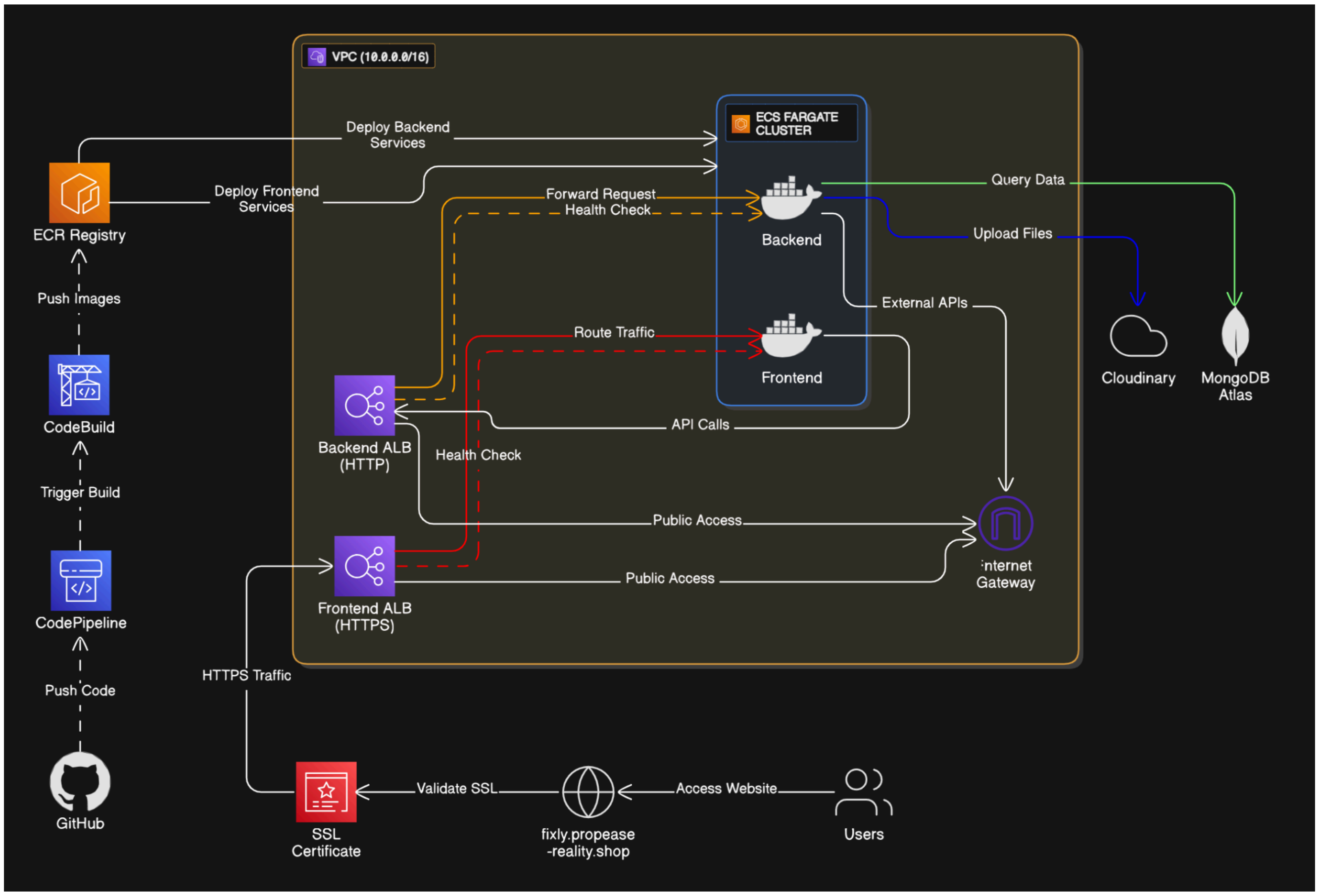
Fixly operates as a service management platform consisting of two primary components:

Frontend Application: Built with React and Vite, served through Nginx web server. Provides user interface for service requests and management.

Backend Application: Node.js REST API handling business logic, database operations, and external service integrations including MongoDB Atlas for data persistence, Cloudinary for media management, and Gmail SMTP for email notifications.

2. ARCHITECTURE DIAGRAM

2.1 High-Level Architecture Overview



2.2 Network Architecture Details

VPC Configuration:

- CIDR Block: 10.0.0.0/16
- Availability Zones: ap-south-1a, ap-south-1b
- Public Subnets: 10.0.1.0/24, 10.0.2.0/24
- Private Subnets: 10.0.3.0/24, 10.0.4.0/24

Network Components:

- Internet Gateway for public internet access
- NAT Gateway with Elastic IP for private subnet outbound connectivity
- Route tables configured for appropriate traffic routing

2.3 CI/CD Pipeline Architecture

Developer -> GitHub Repository -> CodePipeline -> CodeBuild -> ECR Repositories -> ECS Services (Auto-Deploy)

3. AWS SERVICES UTILIZED

3.1 Amazon Virtual Private Cloud (VPC)

Purpose: Provides isolated network infrastructure for application deployment.

Justification: VPC enables complete control over network configuration, including IP addressing, subnets, route tables, and security policies. This isolation ensures security best practices and allows granular traffic control between components.

Benefits:

- Network-level isolation from other AWS customers
- Customizable IP address ranges
- Subnet segregation for security boundaries
- Control over inbound and outbound traffic

Implementation Details:

- Created dedicated VPC with /16 CIDR block providing 65,536 IP addresses
- Implemented multi-AZ architecture for high availability
- Segregated public and private subnets for security layering

3.2 Amazon Elastic Container Service (ECS) with Fargate

Purpose: Orchestrates and runs containerized applications without managing servers.

Justification: ECS Fargate eliminates infrastructure management overhead while providing enterprise-grade container orchestration. The serverless compute model allows focus on application development rather than server maintenance.

Benefits:

- No EC2 instance management required
- Automatic scaling based on demand
- Pay-per-use pricing model
- Built-in integration with AWS services
- Reduced operational complexity

Implementation Details:

- Created ECS cluster named "fixly-cluster"
- Deployed two services: frontend and backend

- Each service runs two tasks for high availability
- Tasks automatically distributed across availability zones

3.3 Application Load Balancer (ALB)

Purpose: Distributes incoming application traffic across multiple targets.

Justification: ALB provides Layer 7 load balancing with advanced routing capabilities, health checks, and SSL/TLS termination. Essential for achieving high availability and fault tolerance.

Benefits:

- Automatic distribution of traffic across healthy targets
- Path-based and host-based routing
- Health monitoring with automatic unhealthy target removal
- Support for containerized applications
- WebSocket and HTTP/2 support

Implementation Details:

- Deployed separate ALBs for frontend and backend services
- Configured target groups with appropriate health check endpoints
- Frontend health check: "/" path
- Backend health check: "/health" path

3.4 Amazon Elastic Container Registry (ECR)

Purpose: Fully managed Docker container registry for storing and managing container images.

Justification: ECR provides secure, scalable, and reliable container image storage with seamless ECS integration. Eliminates need for third-party registry services.

Benefits:

- Encrypted image storage
- Image vulnerability scanning
- Lifecycle policies for automated cleanup
- IAM-based access control
- High availability and durability

Implementation Details:

- Created two repositories: fixly-frontend and fixly-backend
- Implemented image tagging strategy using git commit hashes
- Configured lifecycle policies to maintain recent images only

3.5 AWS CodePipeline

Purpose: Orchestrates continuous delivery workflow from source to deployment.

Justification: CodePipeline automates the release process, ensuring consistent deployments and reducing human error. Enables rapid iteration and reliable software delivery.

Benefits:

- Automated deployment workflows
- Integration with GitHub for source control
- Visual pipeline monitoring
- Parallel execution capabilities
- Built-in deployment safeguards

Implementation Details:

- Created pipeline named "fixly-pipeline"
- Configured GitHub webhook for automatic triggering
- Three-stage pipeline: Source → Build → Deploy
- Separate deployment stages for frontend and backend

3.6 AWS CodeBuild

Purpose: Compiles source code, runs tests, and produces deployment artifacts.

Justification: CodeBuild provides fully managed build service with automatic scaling. Eliminates need to provision and manage build servers.

Benefits:

- Pay-per-minute build pricing
- Automatic scaling to handle multiple builds
- Pre-configured build environments
- Custom build specifications via buildspec.yml
- Integration with ECR for image publishing

Implementation Details:

- Project: fixly-build-project
- Build environment: Docker runtime
- Builds both frontend and backend containers
- Pushes images to ECR with version tags
- Generates ECS image definition files

3.7 Amazon CloudWatch

Purpose: Monitoring and logging service for AWS resources and applications.

Justification: CloudWatch provides comprehensive observability into application performance, resource utilization, and operational health.

Benefits:

- Centralized log aggregation
- Real-time metrics and dashboards
- Automated alerting capabilities
- Log retention and analysis
- Performance troubleshooting

Implementation Details:

- Log groups for frontend, backend, and CodeBuild
- Container logs automatically streamed from ECS tasks
- Retention policies configured for cost optimization

3.8 AWS Identity and Access Management (IAM)

Purpose: Manages authentication and authorization for AWS services.

Justification: IAM enables principle of least privilege, ensuring each service has only necessary permissions.

Benefits:

- Fine-grained access control
- Service-to-service authentication
- Audit trail of API calls
- Temporary security credentials
- No credential hardcoding required

Implementation Details:

- ecsTaskExecutionRole: Allows ECS to pull images and write logs
- fixly-codebuild-role: Permits building and pushing to ECR
- fixly-codepipeline-role: Enables pipeline orchestration

3.9 Amazon S3

Purpose: Object storage for pipeline artifacts.

Justification: S3 provides durable, highly available storage for build artifacts and pipeline metadata.

Implementation Details:

- Bucket: fixly-pipeline-artifacts-ap-south-1-g39
- Stores imagedefinitions files for ECS deployments
- Versioning enabled for artifact history

4. PLATFORMS AND TECHNOLOGIES USED

4.1 Docker Containerization

Technology: Docker

Motivation: Containerization ensures consistent runtime environments across development, testing, and production. Docker enables packaging applications with all dependencies, eliminating "works on my machine" scenarios.

Benefits:

- Environment consistency
- Rapid deployment
- Resource efficiency
- Microservices compatibility
- Easy rollback capabilities

Usage in Project:

- Created Dockerfiles for frontend and backend
- Multi-stage builds for optimized image sizes
- Base images: node:18-alpine for backend, nginx:alpine for frontend

4.2 Nginx Web Server

Technology: Nginx

Motivation: Nginx serves as high-performance web server and reverse proxy for the frontend application. Its efficiency in serving static content and proxying API requests makes it ideal for production deployments.

Benefits:

- Low memory footprint
- High concurrent connection handling
- Efficient static file serving
- Reverse proxy capabilities
- URL rewriting for SPA routing

Usage in Project:

- Serves React production build
- Proxies /api requests to backend ALB
- Handles client-side routing for single-page application

4.3 React Frontend

Technology: React

Motivation: Modern frontend framework with excellent performance and developer experience. Vite provides fast build times and hot module replacement.

Benefits:

- Component-based architecture
- Virtual DOM for performance
- Large ecosystem of libraries
- Fast development builds with Vite

4.4 Node.js Backend

Technology: Node.js 18 LTS

Motivation: JavaScript runtime enabling full-stack JavaScript development. Event-driven architecture suitable for I/O-heavy operations.

Benefits:

- Non-blocking I/O
- Large package ecosystem (npm)
- JSON-native processing
- Microservices friendly

4.5 MongoDB Atlas

Technology: MongoDB (Managed Database)

Motivation: Cloud-native NoSQL database providing flexibility and scalability. Managed service reduces operational overhead.

Benefits:

- Automatic backups and point-in-time recovery
- Built-in replication and high availability
- Flexible schema design
- Horizontal scaling capabilities

4.6 Git & GitHub

Technology: Git version control, GitHub hosting

Motivation: Industry-standard version control and collaboration platform. Enables CI/CD integration through webhooks.

Benefits:

- Complete code history
- Branch-based workflows
- Pull request reviews
- Webhook integrations for automation

5. TEAM MEMBER CONTRIBUTIONS

5.1 Harsh Kathrotia - Application Containerization & Configuration

Primary Responsibilities:

Docker Implementation:

- Authored Dockerfiles for both frontend and backend applications
- Implemented multi-stage builds to minimize final image sizes
- Configured appropriate base images (node:18-alpine, nginx:alpine)
- Optimized layer caching for faster subsequent builds
- Created .dockerignore files to exclude unnecessary files from images

Docker Compose Development:

- Developed docker-compose.yml for local development environment
- Configured service dependencies and network connectivity
- Set up volume mounts for hot-reload during development
- Defined environment variables for local testing

Nginx Configuration:

- Created custom nginx.conf for production deployment
- Configured reverse proxy rules to route /api requests to backend service
- Implemented client-side routing support for React SPA
- Set up appropriate cache headers for static assets
- Configured gzip compression for improved performance
- Added security headers (X-Frame-Options, X-Content-Type-Options)

Code Modifications for Deployment:

- Updated API endpoint URLs to use environment variables
- Modified build configurations for production environment
- Ensured CORS settings compatible with ALB architecture
- Adjusted port configurations for containerized environment
- Updated package.json scripts for Docker-based builds

5.2 Krishna Navadiya - Network Infrastructure & Security

Primary Responsibilities:

VPC Architecture:

- Designed and implemented complete VPC structure with CIDR 10.0.0.0/16
- Created four subnets across two availability zones:
    - Public subnets: 10.0.1.0/24 (1a), 10.0.2.0/24 (1b)
    - Private subnets: 10.0.3.0/24 (1a), 10.0.4.0/24 (1b)
- Configured Internet Gateway for public subnet internet access
- Provisioned NAT Gateway with Elastic IP for private subnet outbound traffic
- Created and associated route tables for proper traffic routing
- Implemented subnet-level NACL configurations

Security Group Configuration:

- Designed security group architecture with principle of least privilege:
    - fixly-alb-sg: Allows HTTP (80) and HTTPS (443) from internet (0.0.0.0/0)
    - fixly-frontend-sg: Allows port 80 only from fixly-alb-sg
    - fixly-backend-sg: Allows port 5000 only from fixly-alb-sg
- Configured egress rules for necessary outbound connectivity
- Implemented security group chaining for defense in depth
- Documented security group purposes and rule justifications

Load Balancer Setup:

- Created Application Load Balancers for frontend and backend services
- Configured ALBs in public subnets for internet accessibility
- Set up target groups with appropriate health check configurations:
    - Frontend TG: Port 80, health check path "/"
    - Backend TG: Port 5000, health check path "/health"
- Configured health check intervals, timeouts, and thresholds
- Set up ALB listeners for HTTP traffic
- Implemented cross-zone load balancing for even distribution

ECR Repository Management:

- Created Amazon ECR repositories for frontend and backend images
- Configured repository permissions and access policies
- Set up lifecycle policies to automatically remove old images:
    - Retain only last 10 images per repository
    - Clean up untagged images after 7 days
- Enabled image scanning on push for vulnerability detection
- Configured repository encryption settings

5.3 Kirtan Pithadiya - ECS Orchestration & CI/CD Pipeline

Primary Responsibilities:

ECS Cluster & Task Definitions:

- Created ECS Fargate cluster named "fixly-cluster"
- Authored task definition for frontend service:
  - Container name: frontend
  - Task CPU: 256 (.25 vCPU)
  - Task memory: 512 MB
  - Port mapping: 80
  - CloudWatch log configuration: /ecs/fixly-frontend
  - Environment variable management
- Authored task definition for backend service:
  - Container name: backend
  - Task CPU: 512 (.5 vCPU)
  - Task memory: 1024 MB
  - Port mapping: 5000
  - CloudWatch log configuration: /ecs/fixly-backend
  - Secure secrets management via task definition

ECS Service Configuration:

- Created fixly-frontend-service:
  - Desired count: 2 tasks
  - Launch type: Fargate
  - Deployment configuration: Rolling update
  - Minimum healthy percent: 50
  - Maximum percent: 200
  - Network configuration: Private subnets
  - Load balancer integration with target group
- Created fixly-backend-service with similar configuration
- Configured service auto-scaling policies (optional)
- Set up service discovery for inter-service communication

CodeBuild Project Setup:

- Created fixly-build-project in CodeBuild
- Configured build environment:
  - Image: aws/codebuild/standard:7.0
  - Compute type: BUILD_GENERAL1_SMALL
  - Privileged mode enabled for Docker builds
- Authored comprehensive buildspec.yml:
  - Pre-build: ECR login, tag generation
  - Build: Docker image creation for both services
  - Post-build: Image push to ECR, artifact generation

- Set up environment variables for AWS region and account ID
- Configured build timeouts and retry strategies
- Linked CodeBuild role with necessary IAM permissions

CodePipeline Configuration:

- Created fixly-pipeline with three stages:
    - Source stage: GitHub repository integration
    - Build stage: CodeBuild project execution
    - Deploy stage: Parallel deployment to ECS services
- Configured separate deployment actions for frontend and backend
- Set up approval stages for production deployments (optional)

GitHub Integration:

- Established GitHub OAuth connection in AWS
- Configured webhook for automatic pipeline triggering
- Set up branch filtering to trigger only on main branch pushes
- Tested webhook functionality with sample commits
- Documented repository structure requirements
- Created GitHub Actions workflow as backup (optional)

Pipeline Orchestration:

- Implemented artifact handling between pipeline stages
- Configured imagedefinitions.json generation in CodeBuild
- Set up automatic ECS service updates upon new image availability
- Implemented deployment strategies (rolling, blue-green considerations)
- Created S3 bucket for pipeline artifacts with versioning
- Configured IAM roles and policies for pipeline execution

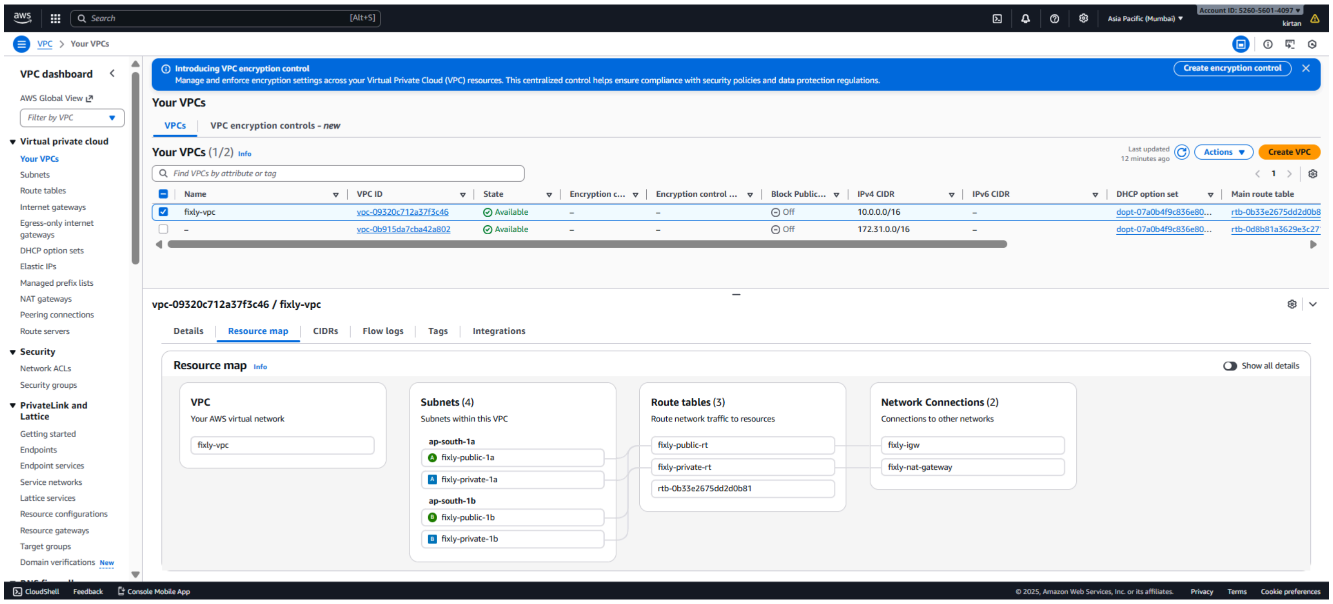
Deployment & Testing:

- Performed initial manual deployment to validate configurations
- Executed end-to-end pipeline testing
- Verified automatic rollback on failed deployments
- Monitored deployments via CloudWatch logs
- Documented deployment procedures and troubleshooting steps

6. DASHBOARD SCREENSHOTS

6.1 AWS Management Console

- **VPC Dashboard:**



Introducing VPC encryption control
Manage and enforce encryption settings across your Virtual Private Cloud (VPC) resources. This centralized control helps ensure compliance with security policies and data protection regulations.

Your VPCs (1/2)

Name	VPC ID	State	Encryption c...	Encryption control ...	Block Public...	IPv4 CIDR	IPv6 CIDR	DHCP option set	Main route table
fixly-vpc	vpc-09320c712a37f3c46	Available	–	–	Off	10.0.0.0/16	–	dopt-07a0b4f9c836e80...	rtb-0b33e2675dd2d0b8
–	vpc-0b915da7cba42a802	Available	–	–	Off	172.31.0.0/16	–	dopt-07a0b4f9c836e80...	rtb-0d8b81a3629e3c27

vpc-09320c712a37f3c46 / fixly-vpc

Resource map

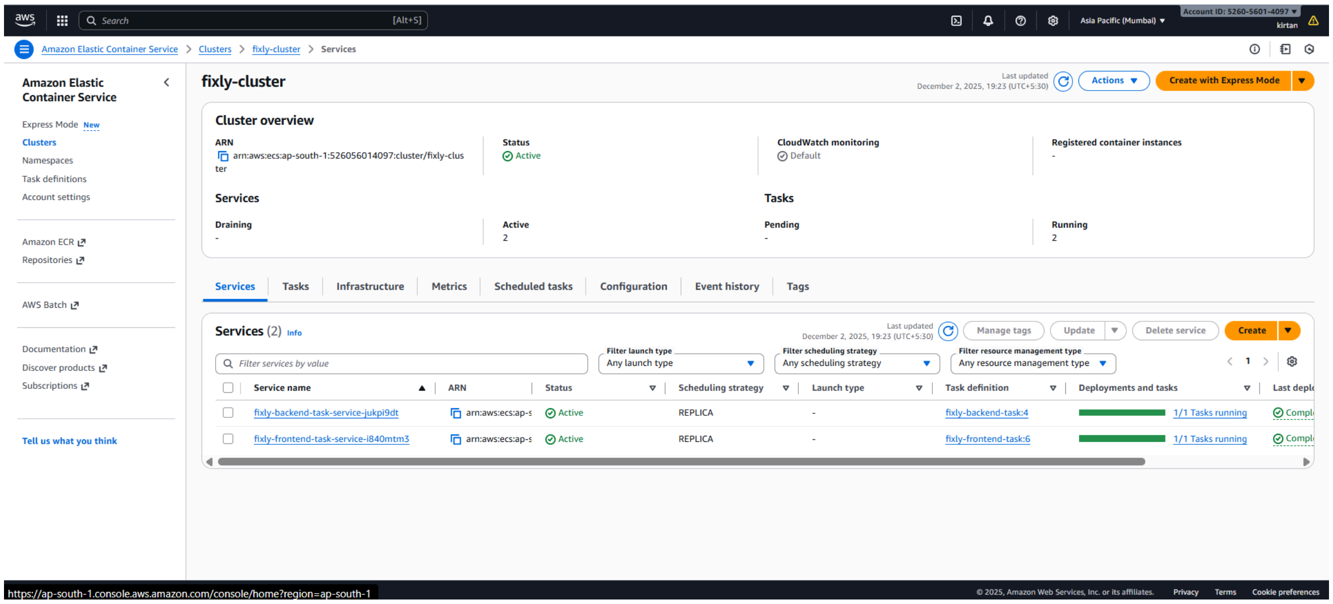
VPC — Your AWS virtual network: fixly-vpc

Subnets (4) — Subnets within this VPC:
- ap-south-1a: fixly-public-1a, fixly-private-1a
- ap-south-1b: fixly-public-1b, fixly-private-1b

Route tables (3) — Route network traffic to resources: fixly-public-rt, fixly-private-rt, rtb-0b33e2675dd2d0b81

Network Connections (2) — Connections to other networks: fixly-igw, fixly-nat-gateway

- **ECS Cluster:**



fixly-cluster

Last updated December 2, 2025, 19:23 (UTC+5:30)

Cluster overview

ARN: arn:aws:ecs:ap-south-1:526056014097:cluster/fixly-cluster
Status: Active
CloudWatch monitoring: Default
Registered container instances: -

Services — Draining: -; Active: 2
Tasks — Pending: -; Running: 2

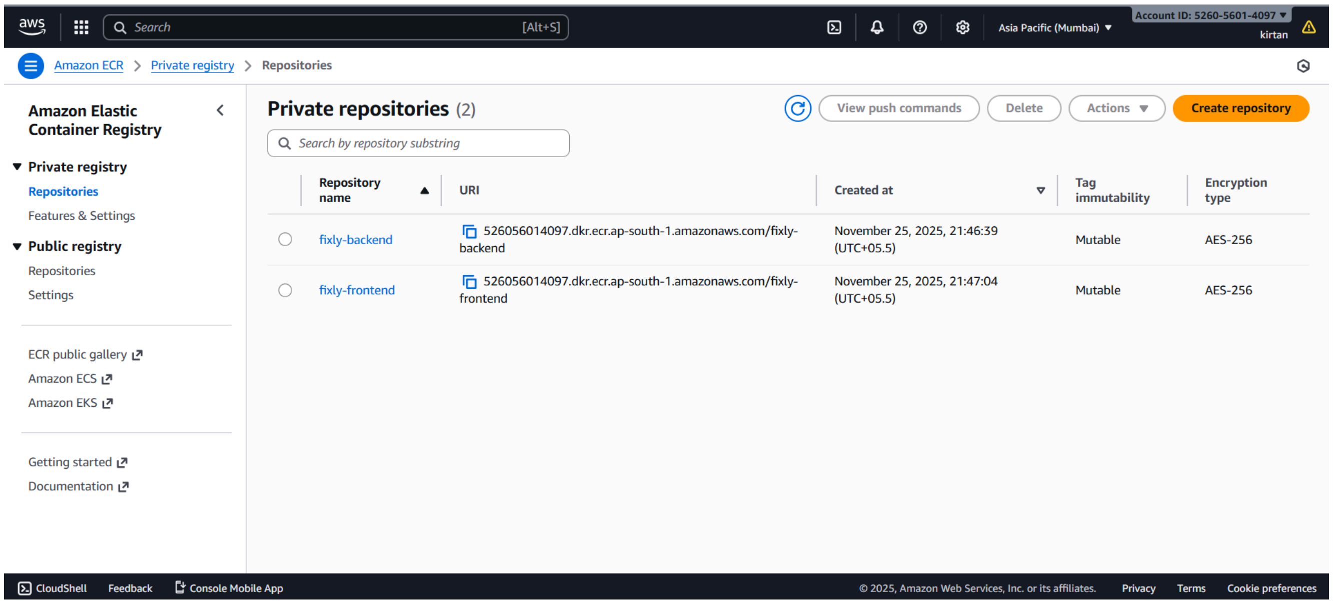
Services (2)

Last updated December 2, 2025, 19:23 (UTC+5:30)

Service name	ARN	Status	Scheduling strategy	Launch type	Task definition	Deployments and tasks	Last depl
fixly-backend-task-service-jukpi9dt	arn:aws:ecs:ap-s	Active	REPLICA	-	fixly-backend-task:4	1/1 Tasks running	Compl
fixly-frontend-task-service-i840mtm3	arn:aws:ecs:ap-s	Active	REPLICA	-	fixly-frontend-task:6	1/1 Tasks running	Compl

https://ap-south-1.console.aws.amazon.com/console/home?region=ap-south-1

- **ECR Repositories:**



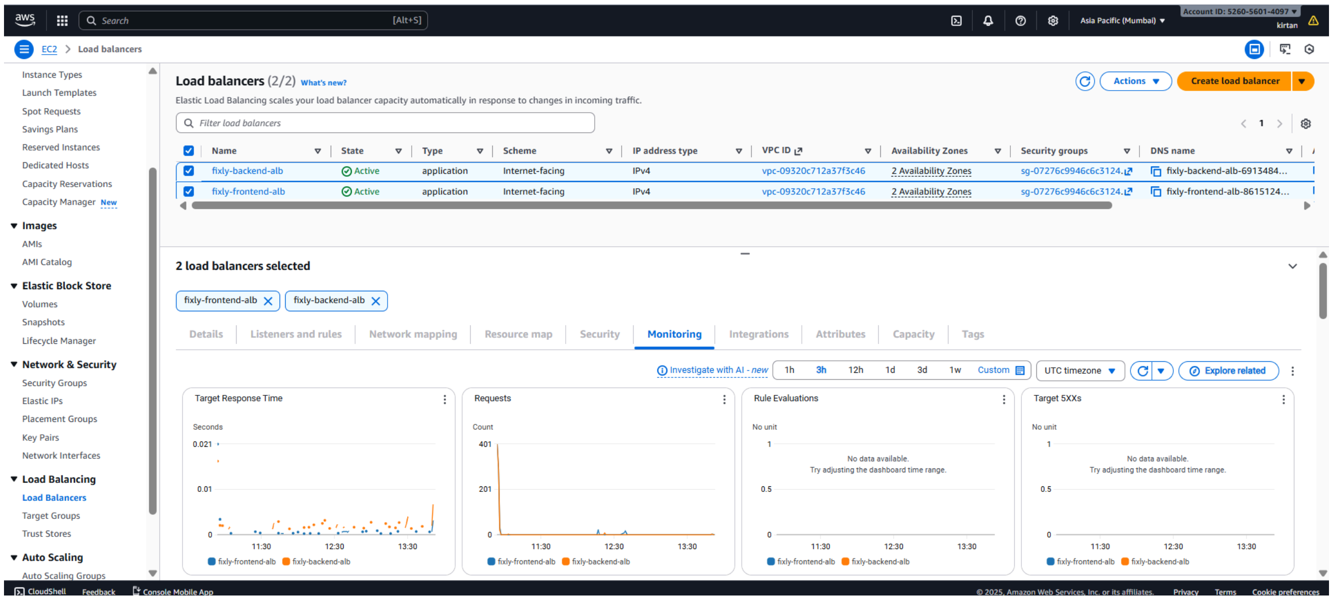
Amazon ECR > Private registry > Repositories

Private repositories (2)

Repository name	URI	Created at	Tag immutability	Encryption type
fixly-backend	526056014097.dkr.ecr.ap-south-1.amazonaws.com/fixly-backend	November 25, 2025, 21:46:39 (UTC+05.5)	Mutable	AES-256
fixly-frontend	526056014097.dkr.ecr.ap-south-1.amazonaws.com/fixly-frontend	November 25, 2025, 21:47:04 (UTC+05.5)	Mutable	AES-256

6.2 Load Balancers

- **ALB Configuration:**



EC2 > Load balancers

Load balancers (2/2)

Elastic Load Balancing scales your load balancer capacity automatically in response to changes in incoming traffic.

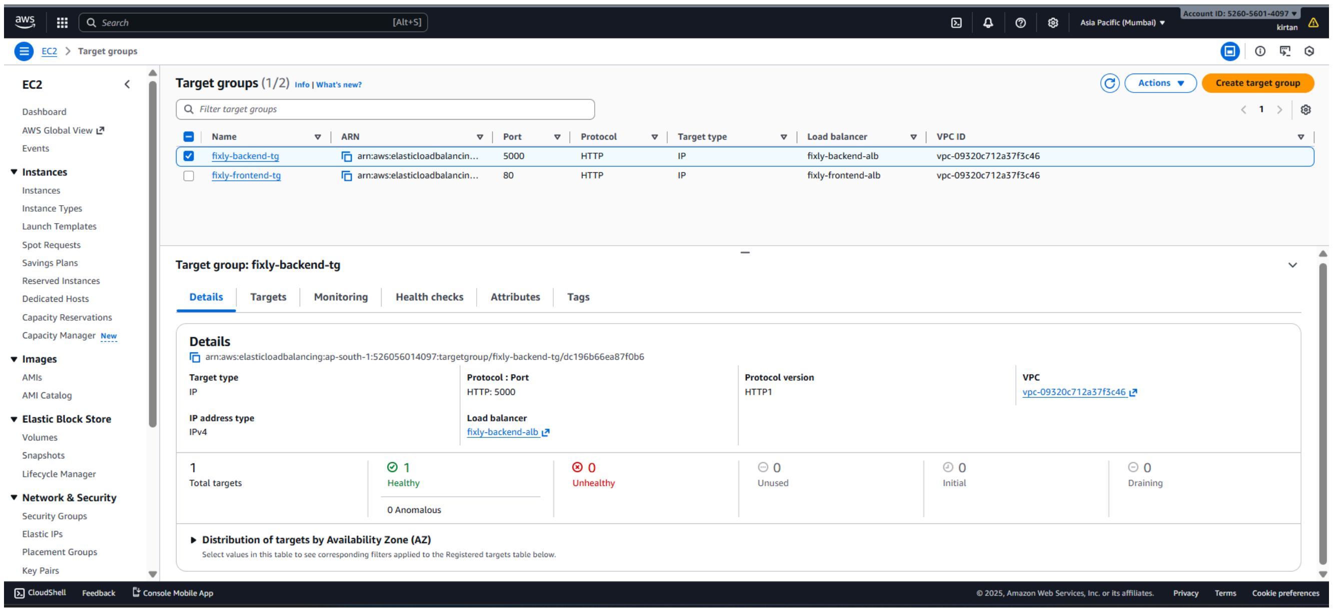
Name	State	Type	Scheme	IP address type	VPC ID	Availability Zones	Security groups	DNS name
fixly-backend-alb	Active	application	Internet-facing	IPv4	vpc-09320c712a37f3c46	2 Availability Zones	sg-07276c9946c6c3124.	fixly-backend-alb-6913484...
fixly-frontend-alb	Active	application	Internet-facing	IPv4	vpc-09320c712a37f3c46	2 Availability Zones	sg-07276c9946c6c3124.	fixly-frontend-alb-8615124...

2 load balancers selected

fixly-frontend-alb, fixly-backend-alb

Monitoring: Target Response Time; Requests; Rule Evaluations — No data available. Try adjusting the dashboard time range.; Target 5XXs — No data available. Try adjusting the dashboard time range.

- **Target Groups:**



Target groups (1/2)

Name	ARN	Port	Protocol	Target type	Load balancer	VPC ID
fixly-backend-tg	arn:aws:elasticloadbalancin...	5000	HTTP	IP	fixly-backend-alb	vpc-09320c712a37f3c46
fixly-frontend-tg	arn:aws:elasticloadbalancin...	80	HTTP	IP	fixly-frontend-alb	vpc-09320c712a37f3c46

Target group: fixly-backend-tg

Details

arn:aws:elasticloadbalancing:ap-south-1:526056014097:targetgroup/fixly-backend-tg/dc196b66ea87f0b6

Target type	Protocol : Port	Protocol version	VPC
IP	HTTP: 5000	HTTP1	vpc-09320c712a37f3c46

IP address type: IPv4

Load balancer: fixly-backend-alb

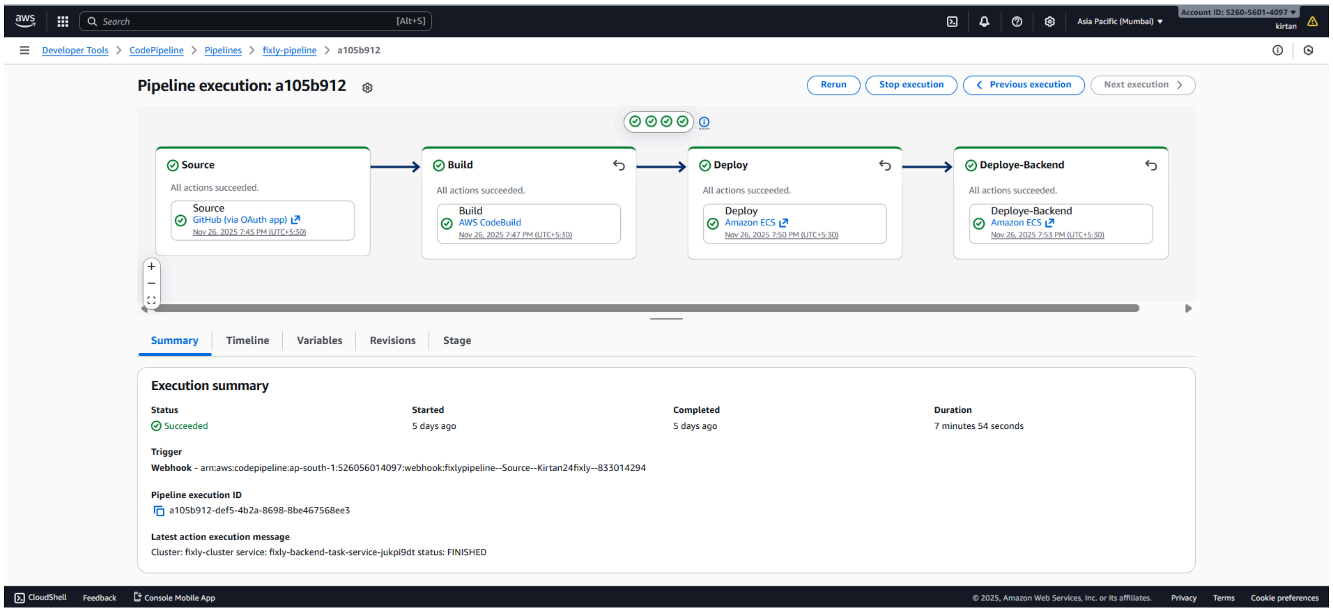
Total targets	Healthy	Unhealthy	Unused	Initial	Draining
1	1 (0 Anomalous)	0	0	0	0

Distribution of targets by Availability Zone (AZ)

Select values in this table to see corresponding filters applied to the Registered targets table below.

6.3 CI/CD Pipeline

- **CodePipeline:**



Pipeline execution: a105b912

Stage	Status	Action	Provider	Time
Source	All actions succeeded.	Source	GitHub (via OAuth app)	Nov 26, 2025 7:45 PM (UTC+5:30)
Build	All actions succeeded.	Build	AWS CodeBuild	Nov 26, 2025 7:47 PM (UTC+5:30)
Deploy	All actions succeeded.	Deploy	Amazon ECS	Nov 26, 2025 7:50 PM (UTC+5:30)
Deploye-Backend	All actions succeeded.	Deploye-Backend	Amazon ECS	Nov 26, 2025 7:53 PM (UTC+5:30)

Execution summary

Status	Started	Completed	Duration
Succeeded	5 days ago	5 days ago	7 minutes 54 seconds

Trigger

Webhook - arn:aws:codepipeline:ap-south-1:526056014097:webhook:fixlypipeline--Source--Kirtan24fixly--833014294

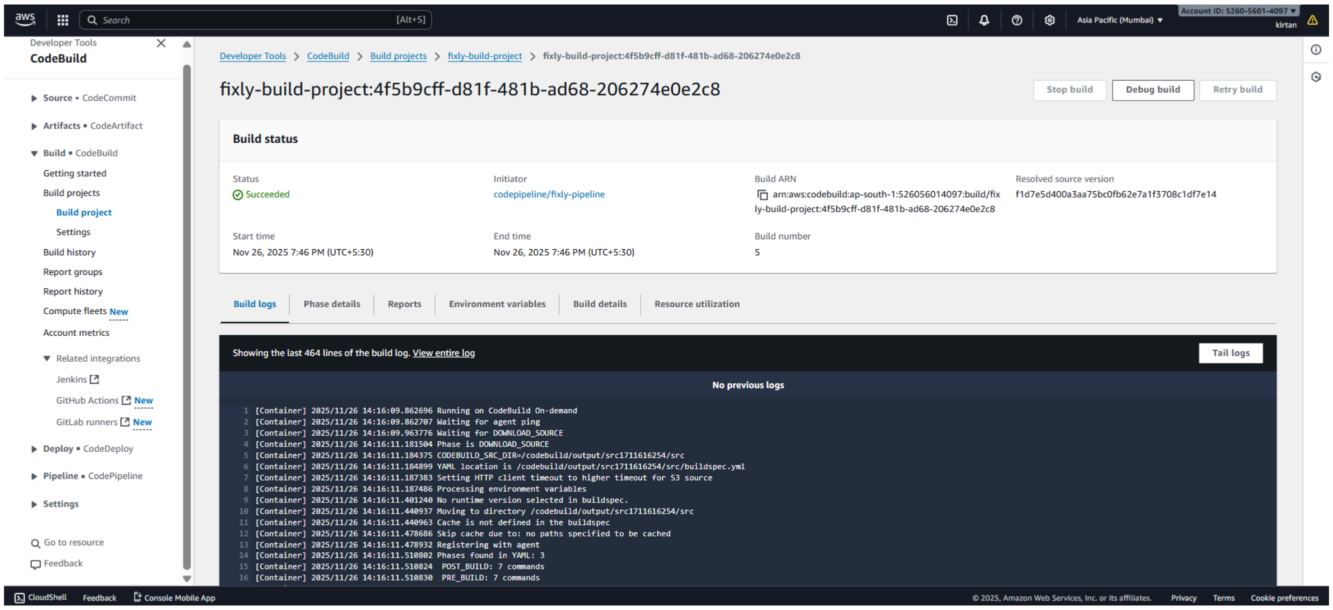
Pipeline execution ID

a105b912-def5-4b2a-8698-8be467568ee3

Latest action execution message

Cluster: fixly-cluster service: fixly-backend-task-service-jukpi9dt status: FINISHED

- **CodeBuild Logs:**



Developer Tools > CodeBuild > Build projects > fixly-build-project > fixly-build-project:4f5b9cff-d81f-481b-ad68-206274e0e2c8

fixly-build-project:4f5b9cff-d81f-481b-ad68-206274e0e2c8

Build status

Status	Initiator	Build ARN	Resolved source version
Succeeded	codepipeline/fixly-pipeline	arn:aws:codebuild:ap-south-1:526056014097:build/fixly-build-project:4f5b9cff-d81f-481b-ad68-206274e0e2c8	f1d7e5d400a3aa75bc0fb62e7a1f3708c1df7e14
Start time	End time	Build number	
Nov 26, 2025 7:46 PM (UTC+5:30)	Nov 26, 2025 7:46 PM (UTC+5:30)	5	

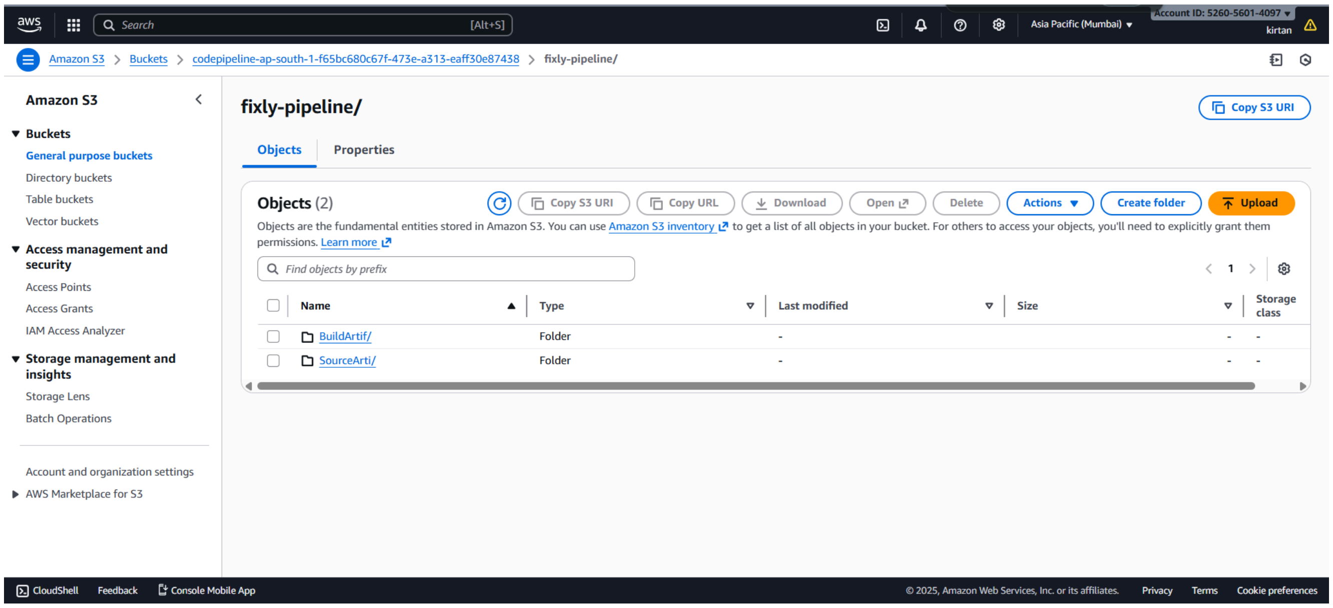
Showing the last 464 lines of the build log. View entire log

No previous logs

```
[Container] 2025/11/26 14:16:09.862696 Running on CodeBuild On-demand
[Container] 2025/11/26 14:16:09.862707 Waiting for agent ping
[Container] 2025/11/26 14:16:09.963776 Waiting for DOWNLOAD_SOURCE
[Container] 2025/11/26 14:16:11.181504 Phase is DOWNLOAD_SOURCE
[Container] 2025/11/26 14:16:11.184375 CODEBUILD_SRC_DIR=/codebuild/output/src1711616254/src
[Container] 2025/11/26 14:16:11.184899 YAML location is /codebuild/output/src1711616254/src/buildspec.yml
[Container] 2025/11/26 14:16:11.187383 Setting HTTP client timeout to higher timeout for S3 source
[Container] 2025/11/26 14:16:11.187486 Processing environment variables
[Container] 2025/11/26 14:16:11.401240 No runtime version selected in buildspec.
[Container] 2025/11/26 14:16:11.440937 Moving to directory /codebuild/output/src1711616254/src
[Container] 2025/11/26 14:16:11.440963 Cache is not defined in the buildspec
[Container] 2025/11/26 14:16:11.478686 Skip cache due to: no paths specified to be cached
[Container] 2025/11/26 14:16:11.478932 Registering with agent
[Container] 2025/11/26 14:16:11.510802 Phases found in YAML: 3
[Container] 2025/11/26 14:16:11.510824  POST_BUILD: 7 commands
[Container] 2025/11/26 14:16:11.510830  PRE_BUILD: 7 commands
```

© 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

- **CodeBuild Artifacts:**



Amazon S3 > Buckets > codepipeline-ap-south-1-f65bc680c67f-473e-a313-eaff30e87438 > fixly-pipeline/

fixly-pipeline/

Objects (2)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Find objects by prefix

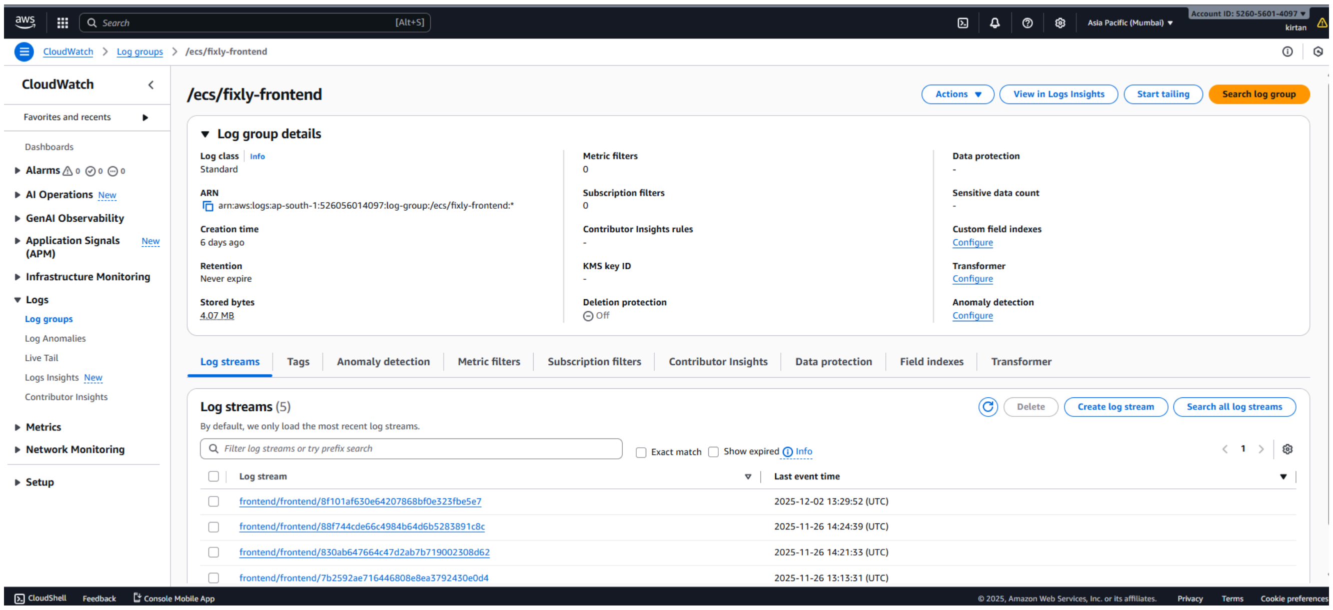
Name	Type	Last modified	Size	Storage class
BuildArtif/	Folder	-	-	-
SourceArti/	Folder	-	-	-

© 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

6.4 Monitoring

- **CloudWatch Logs:**

Frontend



/ecs/fixly-frontend

Log group details

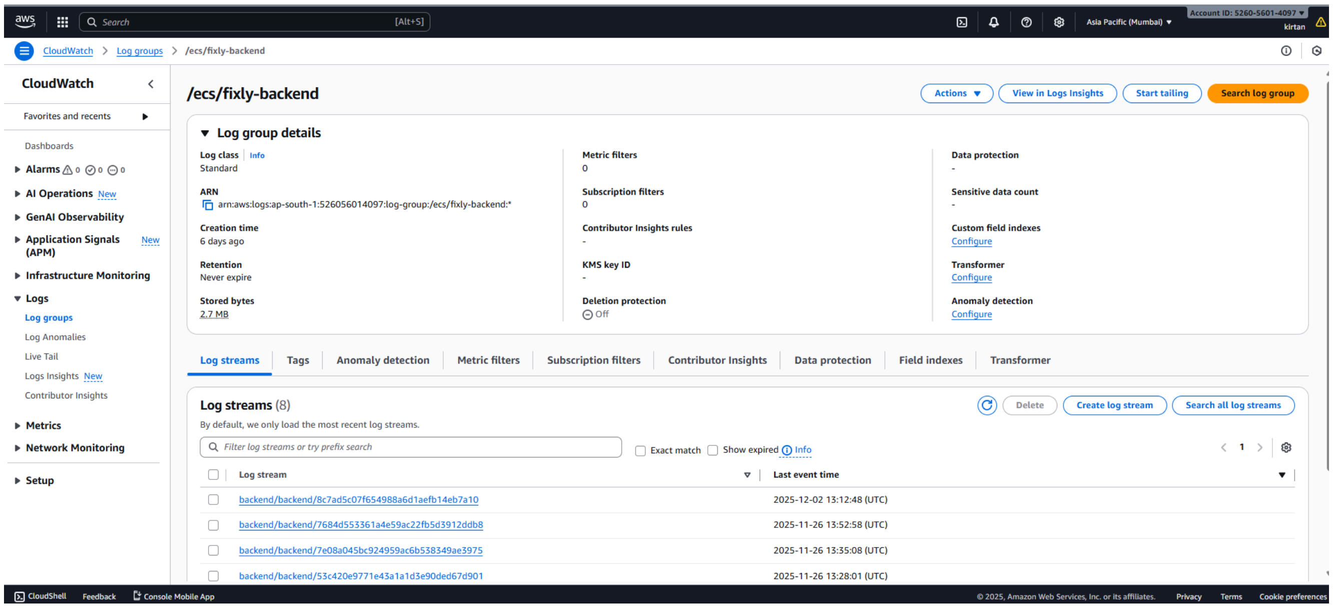
Log class Standard	**Metric filters** 0	**Data protection** -
ARN arn:aws:logs:ap-south-1:526056014097:log-group:/ecs/fixly-frontend:*	**Subscription filters** 0	**Sensitive data count** -
Creation time 6 days ago	**Contributor Insights rules** -	**Custom field indexes** Configure
Retention Never expire	**KMS key ID** -	**Transformer** Configure
Stored bytes 4.07 MB	**Deletion protection** Off	**Anomaly detection** Configure

Log streams (5)

By default, we only load the most recent log streams.

Log stream	Last event time
frontend/frontend/8f101af630e64207868bf0e323fbe5e7	2025-12-02 13:29:52 (UTC)
frontend/frontend/88f744cde66c4984b64d6b5283891c8c	2025-11-26 14:24:39 (UTC)
frontend/frontend/830ab647664c47d2ab7b719002308d62	2025-11-26 14:21:33 (UTC)
frontend/frontend/7b2592ae716446808e8ea3792430e0d4	2025-11-26 13:13:31 (UTC)

Backend



/ecs/fixly-backend

Log group details

Log class Standard	**Metric filters** 0	**Data protection** -
ARN arn:aws:logs:ap-south-1:526056014097:log-group:/ecs/fixly-backend:*	**Subscription filters** 0	**Sensitive data count** -
Creation time 6 days ago	**Contributor Insights rules** -	**Custom field indexes** Configure
Retention Never expire	**KMS key ID** -	**Transformer** Configure
Stored bytes 2.7 MB	**Deletion protection** Off	**Anomaly detection** Configure

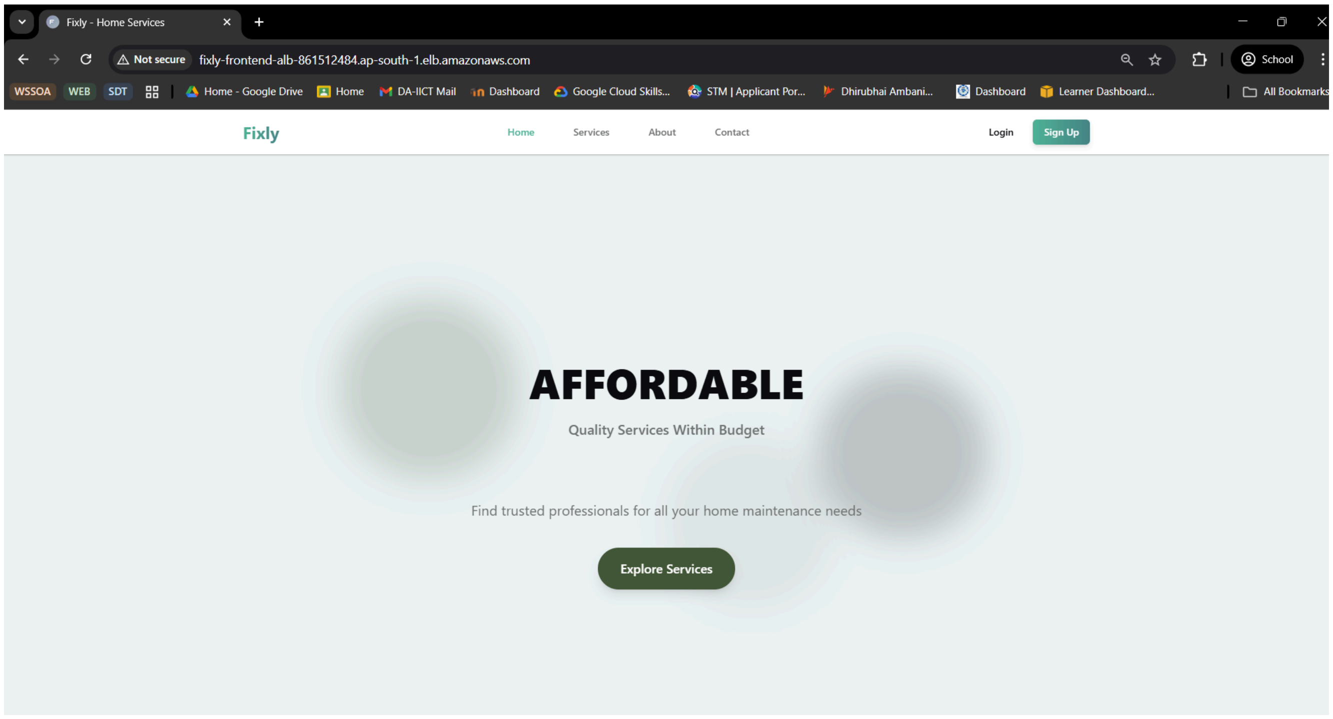
Log streams (8)

By default, we only load the most recent log streams.

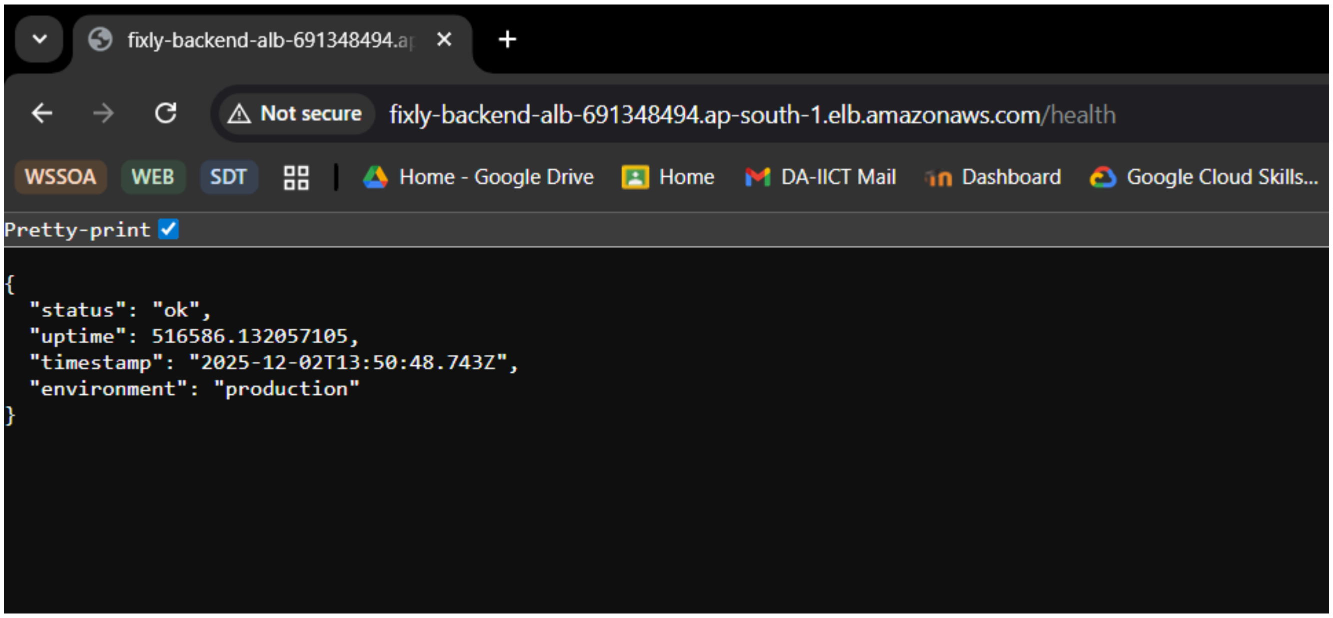
Log stream	Last event time
backend/backend/8c7ad5c07f654988a6d1aefb14eb7a10	2025-12-02 13:12:48 (UTC)
backend/backend/7684d553361a4e59ac22fb5d3912ddb8	2025-11-26 13:52:58 (UTC)
backend/backend/7e08a045bc924959ac6b538349ae3975	2025-11-26 13:35:08 (UTC)
backend/backend/53c420e9771e43a1a1d3e90ded67d901	2025-11-26 13:28:01 (UTC)

6.5 Application

- **Frontend Access:**


Fixly - Home Services
Not secure
fixly-frontend-alb-861512484.ap-south-1.elb.amazonaws.com
Fixly
Home
Services
About
Contact
Login
Sign Up
AFFORDABLE
Quality Services Within Budget
Find trusted professionals for all your home maintenance needs
Explore Services

- **Backend Health Endpoint:**


Not secure
fixly-backend-alb-691348494.ap-south-1.elb.amazonaws.com/health
Pretty-print ✓

```
{
  "status": "ok",
  "uptime": 516586.132057105,
  "timestamp": "2025-12-02T13:50:48.743Z",
  "environment": "production"
}
```

Dockerfiles and buildspec files

client/Dockerfile

```dockerfile
# ---------- Build Stage ----------
FROM node:20-alpine AS build
WORKDIR /app

# Install dependencies
COPY package.json package-lock.json ./
RUN npm ci

# Copy source
COPY . .

# Build production site (Vite will inline the VITE_* env vars at build time)
RUN npm run build

# ---------- Nginx Stage ----------
FROM nginx:alpine

WORKDIR /usr/share/nginx/html

# Copy React build
COPY --from=build /app/dist .

# Copy Public folder (optional)
COPY --from=build /app/public ./public

# Copy custom Nginx config to the proper place
COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

client/nginx.conf

```
server {
    listen 80;
    server_name _;

    root /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html;
    }

    # UPDATE THIS with your actual backend ALB DNS
    location /api/ {
        proxy_pass
http://fixly-backend-alb-691348494.ap-south-1.elb.amazonaws.com;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

server/Dockerfile

```dockerfile
FROM node:20-alpine AS builder
WORKDIR /app

# Install only production dependencies
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

# Copy source
COPY . .

FROM node:20-alpine AS runner
WORKDIR /app

# Install curl for health checks
RUN apk update && apk add curl

# Create a non-root user
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

# Copy built app and dependencies from builder stage
COPY --from=builder /app /app

USER appuser

ENV NODE_ENV=production
EXPOSE 5000

HEALTHCHECK --interval=30s --timeout=5s --start-period=10s --retries=3 \
    CMD curl -f http://localhost:5000/health || exit 1

CMD ["node", "server.js"]
```

/buildspec.yml

```
version: 0.2
env:
  variables:
    AWS_REGION: "ap-south-1"
    ACCOUNT_ID: "526056014097"
    ECR_REPO_FRONTEND: "fixly-frontend"
    ECR_REPO_BACKEND: "fixly-backend"

phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - export AWS_DEFAULT_REGION=$AWS_REGION

      # Login (SINGLE LINE - VERY IMPORTANT)
      - aws ecr get-login-password --region $AWS_DEFAULT_REGION | docker login
--username AWS --password-stdin
$ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com

      # Create tags
      - COMMIT_HASH=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION | cut -c 1-7)
      - IMAGE_TAG=${COMMIT_HASH:=latest}

      # Repo URIs
      -
BACKEND_REPO_URI=$ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$ECR_RE
PO_BACKEND
      -
FRONTEND_REPO_URI=$ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$ECR_R
EPO_FRONTEND

  build:
    commands:
      - echo Build started on `date`

      # Backend
      - cd server
      - docker build -t backend:latest .
      - docker tag backend:latest $BACKEND_REPO_URI:latest
      - docker tag backend:latest $BACKEND_REPO_URI:$IMAGE_TAG
      - cd ..
```

```yaml
      # Frontend
      - cd client
      - docker build -t frontend:latest .
      - docker tag frontend:latest $FRONTEND_REPO_URI:latest
      - docker tag frontend:latest $FRONTEND_REPO_URI:$IMAGE_TAG
      - cd ..

  post_build:
    commands:
      - echo Build completed on `date`

      # Push images
      - docker push $BACKEND_REPO_URI:latest
      - docker push $BACKEND_REPO_URI:$IMAGE_TAG
      - docker push $FRONTEND_REPO_URI:latest
      - docker push $FRONTEND_REPO_URI:$IMAGE_TAG

      # Create ECS images definitions
      - printf '[{"name":"backend","imageUri":"%s"}]'
$BACKEND_REPO_URI:$IMAGE_TAG > imagedefinitions-backend.json
      - printf '[{"name":"frontend","imageUri":"%s"}]'
$FRONTEND_REPO_URI:$IMAGE_TAG > imagedefinitions-frontend.json

artifacts:
  files:
    - imagedefinitions-backend.json
    - imagedefinitions-frontend.json
```

7. RESULTS

7.1 Deployment Success

The Fixly application was successfully deployed to AWS cloud infrastructure with the following outcomes:

Infrastructure Provisioning:

- VPC created with proper subnet segregation across two availability zones
- Security groups configured with minimal necessary permissions
- Load balancers operational with health checks passing
- NAT Gateway providing secure outbound connectivity for private resources

Application Deployment:

- Frontend service running 2 healthy tasks behind ALB
- Backend service running 2 healthy tasks behind ALB
- Both services automatically registered with target groups
- Health checks consistently returning successful responses

CI/CD Pipeline:

- Automated pipeline successfully integrated with GitHub repository
- Webhook triggering pipeline on code commits to main branch
- Build stage completing in approximately 3-5 minutes
- Automated deployment to ECS with zero-downtime rolling updates
- Average pipeline execution time: 8-12 minutes end-to-end

7.3 Security Achievements

Network Security:

- Complete network isolation with private subnet architecture
- No direct internet access for application containers
- All traffic routed through load balancers
- Security group rules following least privilege principle

Access Control:

- IAM roles with minimal necessary permissions
- No hardcoded credentials in application code
- Secrets managed through ECS task definitions
- ECR repositories protected with IAM policies

Compliance:

- All traffic encrypted in transit
- CloudWatch logging enabled for audit trails
- Image vulnerability scanning in ECR
- Regular security group rule reviews

---

8. DISCUSSION

8.1 Technical Challenges and Solutions

Challenge 1: Network Configuration Complexity Setting up proper VPC architecture with public/private subnets, NAT Gateway, and route tables required careful planning. The challenge involved ensuring containers in private subnets could access internet for pulling images and connecting to external services while remaining isolated from direct internet access.

Solution: Implemented NAT Gateway in public subnet with proper route table associations. Configured security groups to allow outbound traffic while restricting inbound access to only load balancer security groups.

Challenge 2: CI/CD Pipeline Integration Integrating GitHub with CodePipeline and configuring webhook triggers initially faced authentication issues and improper artifact generation.

Solution: Properly configured GitHub OAuth connection in AWS console. Modified buildspec.yml to generate separate imagedefinitions files for frontend and backend, ensuring CodePipeline could correctly identify which service to update.

Challenge 3: Container-to-Container Communication Frontend container needed to communicate with backend service through internal networking rather than public endpoints.

Solution: Configured Nginx reverse proxy to route /api requests to backend ALB DNS name. Updated environment variables to use internal service discovery rather than external endpoints.

Challenge 4: ECS Task Definition Secrets Managing sensitive environment variables (database credentials, API keys) without hardcoding in task definitions.

Solution: Implemented secrets management through ECS task definition parameter store integration. Referenced sensitive values as environment variables pulled from AWS Systems Manager Parameter Store.

8.2 Architecture Benefits

Scalability: The implemented architecture supports both vertical and horizontal scaling. ECS services can automatically increase task count based on CPU/memory metrics or custom CloudWatch alarms. Fargate's serverless nature eliminates capacity planning concerns.

High Availability: Multi-AZ deployment ensures application remains available even if one availability zone experiences issues. Load balancers automatically route traffic only to healthy targets, and ECS replaces failed tasks automatically.

Maintainability: Infrastructure-as-code approach through AWS console and CLI enables reproducible deployments. CI/CD pipeline ensures consistent build and deployment processes, reducing human error.

Cost Efficiency: Fargate's pay-per-use model eliminates costs for idle EC2 instances. No server provisioning or management overhead. Costs scale linearly with actual usage.

8.3 Comparison with Alternative Approaches

Versus EC2-based Deployment:

- Eliminated server management and patching overhead
- No need for capacity planning or over-provisioning
- Faster deployment times
- Better resource utilization

Versus Kubernetes (EKS):

- Lower operational complexity
- Faster learning curve for team
- Reduced infrastructure costs for small-to-medium deployments
- Sufficient features for current application requirements

Versus Lambda:

- Better suited for long-running processes
- More straightforward for existing containerized applications
- No cold start delays
- Standard web server deployment model

8.4 Lessons Learned

Infrastructure as Code: While this deployment used AWS Console for configuration, future iterations should implement Terraform or CloudFormation for reproducible infrastructure provisioning. This would enable version control of infrastructure changes and easier disaster recovery.

Monitoring and Alerting: CloudWatch provides excellent logging, but implementing custom dashboards and alarms would enhance operational visibility. Setting up alerts for high CPU usage, failed health checks, and deployment failures would enable proactive issue resolution.

Cost Optimization: Current architecture runs 2 tasks per service 24/7. Implementing auto-scaling policies to reduce task count during low-traffic periods could reduce costs by 30-40%. Using Fargate Spot for non-critical environments could further reduce costs.

Security Hardening: While current security posture is solid, future enhancements should include:

- AWS WAF for application-layer protection
- AWS Shield for DDoS protection
- Secrets Manager instead of task definition environment variables
- VPC Flow Logs for network traffic analysis

9. CONCLUSION

9.1 Project Summary

This project successfully demonstrated deployment of a production-ready containerized application on AWS cloud infrastructure. The Fixly application now runs on a scalable, highly available, and secure architecture leveraging modern cloud-native services.

Key accomplishments include:

- Complete VPC network architecture with proper security boundaries
- Containerized microservices running on ECS Fargate
- Automated CI/CD pipeline enabling rapid development iterations
- Load-balanced services with automatic health monitoring
- Comprehensive logging and monitoring infrastructure

9.2 Learning Outcomes

The team gained practical experience with:

- Cloud network architecture design and implementation
- Container orchestration using Amazon ECS
- Load balancing and traffic distribution strategies
- Automated deployment pipelines with CodePipeline and CodeBuild
- Infrastructure security best practices
- Cost optimization in cloud environments

9.3 Real-World Applicability

The implemented architecture reflects industry-standard practices used by organizations running production workloads on AWS. Skills gained through this project are directly transferable to professional cloud engineering roles:

- Understanding of AWS core services (VPC, ECS, ALB, ECR)
- Experience with containerization and Docker
- Knowledge of CI/CD principles and implementation
- Security-first approach to cloud architecture
- Cost-awareness in resource provisioning

9.4 Future Enhancements

To further improve the deployment, future work could include:

Observability:

- Implement distributed tracing with AWS X-Ray
- Create custom CloudWatch dashboards for business metrics

- Set up alerting with SNS notifications
- Implement log aggregation and analysis with CloudWatch Insights

Performance:

- Configure CloudFront CDN for frontend assets
- Implement ElastiCache for backend caching layer
- Optimize Docker images for faster startup times
- Enable ECS service auto-scaling based on traffic patterns

Security:

- Implement AWS WAF rules for common web attacks
- Rotate secrets automatically using Secrets Manager
- Enable VPC Flow Logs for network monitoring
- Implement AWS Config for compliance monitoring

DevOps:

- Create Terraform modules for infrastructure provisioning
- Implement blue-green deployment strategy
- Add automated testing stages in CI/CD pipeline
- Create disaster recovery and backup procedures

9.5 Final Thoughts

This project provided invaluable hands-on experience with cloud computing concepts and AWS services. The team successfully navigated the complexity of modern cloud architectures, demonstrating both technical proficiency and collaborative problem-solving. The deployed application stands as a testament to effective cloud engineering practices and serves as a strong foundation for future cloud-native development initiatives.

APPENDICES

Appendix A: AWS Resources Created

Networking:

- VPC: fixly-vpc (10.0.0.0/16)
- Subnets: 4 (2 public, 2 private)
- Internet Gateway: 1
- NAT Gateway: 1
- Elastic IP: 1
- Route Tables: 4
- Security Groups: 3

Compute:

- ECS Cluster: fixly-cluster
- ECS Services: 2 (frontend, backend)
- Task Definitions: 2
- Running Tasks: 4 total

Load Balancing:

- Application Load Balancers: 2
- Target Groups: 2

Container Registry:

- ECR Repositories: 2

CI/CD:

- CodePipeline: fixly-pipeline
- CodeBuild Project: fixly-build-project
- S3 Bucket: fixly-pipeline-artifacts

IAM:

- Roles: 3 (ECS Task Execution, CodeBuild, CodePipeline)
- Policies: Custom policies attached to roles

Monitoring:

- CloudWatch Log Groups: 3

Appendix B: Configuration Files Reference

buildspec.yml: Located in project root directory

- Defines Docker build process
- Configures ECR push operations
- Generates ECS deployment artifacts

Dockerfiles: Located in client/ and server/ directories

- Frontend: Multi-stage build with Node and Nginx
- Backend: Node.js runtime with application code

nginx.conf: Located in client/ directory

- Static file serving configuration
- Reverse proxy rules for API requests
- SPA routing support

Appendix C: External Service Integrations

MongoDB Atlas:

- Cluster region: As configured in application
- Connection via MONGO_URI environment variable
- Network access configured for AWS IP ranges

Cloudinary:

- Media upload and storage
- CDN for image delivery
- API integration via environment variables

Gmail SMTP:

- Email notification service
- SMTP configuration on port 587
- App password authentication