

Learning Report

Case Study 3



Employee Lifecycle Management Platform

Kubernetes-Based Automation with AWS Integration

Student name: Sonny He

Student number: 5319587

Class: MA-NCA1

Semester coach: Erik Aerdt

Version: 1.0

Date: December 12, 2025

Table of Contents

1. INTRODUCTION	5
1.1 Project Goal.....	5
1.2 Key Technologies.....	5
1.3 System Architecture Overview.....	6
2. PHASE 1: ANALYSIS, DESIGN & PLANNING	8
2.1 Requirements Analysis	8
2.1.1 Functional Requirements.....	8
2.1.2 Non-Functional Requirements	9
2.2 Technology Choices & Justification	10
2.2.1 Container Orchestration Platform	10
2.2.2 Identity Provider	12
2.2.3 Active Directory Solution	14
2.2.4 Database Solution.....	16
2.2.5 Software Deployment: Group Policy (GPO) vs. AWS Systems Manager (SSM)	17
2.3 System Architecture Design	18
2.3.1 Network Architecture.....	18
2.3.2 Kubernetes Architecture	20
2.3.3 Evolutionary Architecture Pattern	21
2.3.4 Zero Trust Security Design.....	22
2.4 DevOps & IaC Setup	23
2.4.1 Terraform Structure	23
2.4.2 CI/CD Pipeline	25
3. PHASE 2: EMPLOYEE LIFECYCLE AUTOMATION	27
3.1 User creation (onboarding)	27
3.1.1 Admin-only employee creation interface.....	27
3.1.2 Successful onboarding response from backend	27
3.1.3 Newly created employee visible in portal database.....	28
3.1.4 Cognito user account created and enabled	28
3.1.5 Cognito group assignment for role-based access	28
3.1.6 Active Directory user created in correct organizational unit	29

3.1.7	AWS WorkSpace provisioning for the new employee	29
3.1.8	GPO linked to Developers OU.....	30
3.1.9	Portal “Sync WorkSpaces to AD” success message.....	31
3.1.10	Proof software applied on the WorkSpace	32
3.2	User termination (offboarding)	33
3.2.1	Secure offboarding and access revocation (Identity, directory, and WorkSpace cleanup)	33
3.2.2	Admin-initiated termination action	33
3.2.3	Portal confirmation of successful offboarding	34
3.2.4	Employee removed from active portal view	34
3.2.5	Cognito account disabled	35
3.2.6	Active Directory user disabled	35
3.2.7	Amazon WorkSpace termination	36
3.3	Identity Provider Implementation	37
3.3.1	Cognito User Pool Configuration.....	37
3.3.2	Authentication Flow	38
3.4	Active Directory Integration	39
3.4.1	AWS Managed AD Setup	39
3.4.2	Hybrid LDAP/API Approach.....	41
3.5	Automation Scripts Development.....	43
3.5.1	Backend API Architecture	43
3.5.2	Onboarding Workflow	45
3.5.3	Offboarding Workflow (terminate-user)	50
3.6	RBAC & Access Control	54
3.6.1	IRSA for Employee Portal pod permissions	54
3.6.2	Application-Level Authorization.....	57
4.	PHASE 3: KUBERNETES DEPLOYMENT & PORTAL	62
4.1	Phase goal	62
4.2	EKS Cluster Configuration.....	62
4.2.1	Cluster Setup.....	62
4.3	Containerization	63
4.3.1	Backend Dockerfile	63

4.3.2	Frontend Dockerfile	63
4.3.3	Container Registry (ECR).....	64
4.4	Kubernetes Deployment	65
4.4.1	Deployment Verification	66
4.5	Database Persistence.....	67
4.6	Portal Accessibility	69
4.7	Logging.....	70
4.8	CI/CD Deployment Process	71
5.	PHASE 4: VALIDATION, MONITORING & OPERATION	72
5.1	Functional Validation.....	72
5.1.1	Onboarding Validation (Reference to Phase 2)	72
5.1.2	Offboarding validation (reference)	73
5.2	Kubernetes Failover Testing.....	73
5.3	Kubernetes Network Security Validation	74
5.3.1	Active NetworkPolicies in production.....	74
5.4	Kubernetes Runtime Monitoring	75
5.4.1	EKS cluster monitoring (CloudWatch)	75
5.4.2	Amazon WorkSpaces Monitoring	76
5.5	CI/CD Deployment Verification	77
5.6	Cost Management & Optimization	78
5.7	Summary of Phase 4 validation	79
6.	Evaluation & Reflection	80
6.1	Evaluation of the final Solution	80
6.2	Reflection on design decisions.....	80
6.3	Comparison with previous Case Studies	81
6.4	Challenges, limitations, and improvements	81
6.5	Teacher Feedback & Iterative Improvement	82
6.6	Conclusion	83

1. INTRODUCTION

1.1 Project Goal

The goal of Case Study 3 was to design and implement an automated employee lifecycle management platform for Innovatech Solutions. This system automates employee onboarding and offboarding processes through a self-service portal deployed on Kubernetes, building upon the infrastructure established in Case Studies 1 and 2.

The platform addresses key business requirements:

- Automated user provisioning across multiple systems (Cognito, Active Directory, WorkSpaces)
- Self-service portal for employee access requests
- Role-based access control with least-privilege principles
- Integration with existing monitoring infrastructure
- Cost-optimized cloud resources

1.2 Key Technologies

The implementation leverages several AWS services and open-source technologies:

Infrastructure & Orchestration:

- Amazon EKS (Elastic Kubernetes Service) for container orchestration
- Terraform for Infrastructure as Code
- GitHub Actions for CI/CD pipeline

Identity & Access Management:

- AWS Cognito for application authentication
- AWS Managed Active Directory for enterprise identity
- AWS WorkSpaces for virtual desktop infrastructure

Application Stack:

- Python Flask backend for API and automation
- Nginx for frontend serving
- PostgreSQL StatefulSet for data persistence

Integration with Case Studies 1 & 2:

- VPC and networking from CS1

- OpenVPN for secure access from CS1
- Monitoring infrastructure (Grafana/Prometheus) from CS1
- Event-driven patterns from CS2

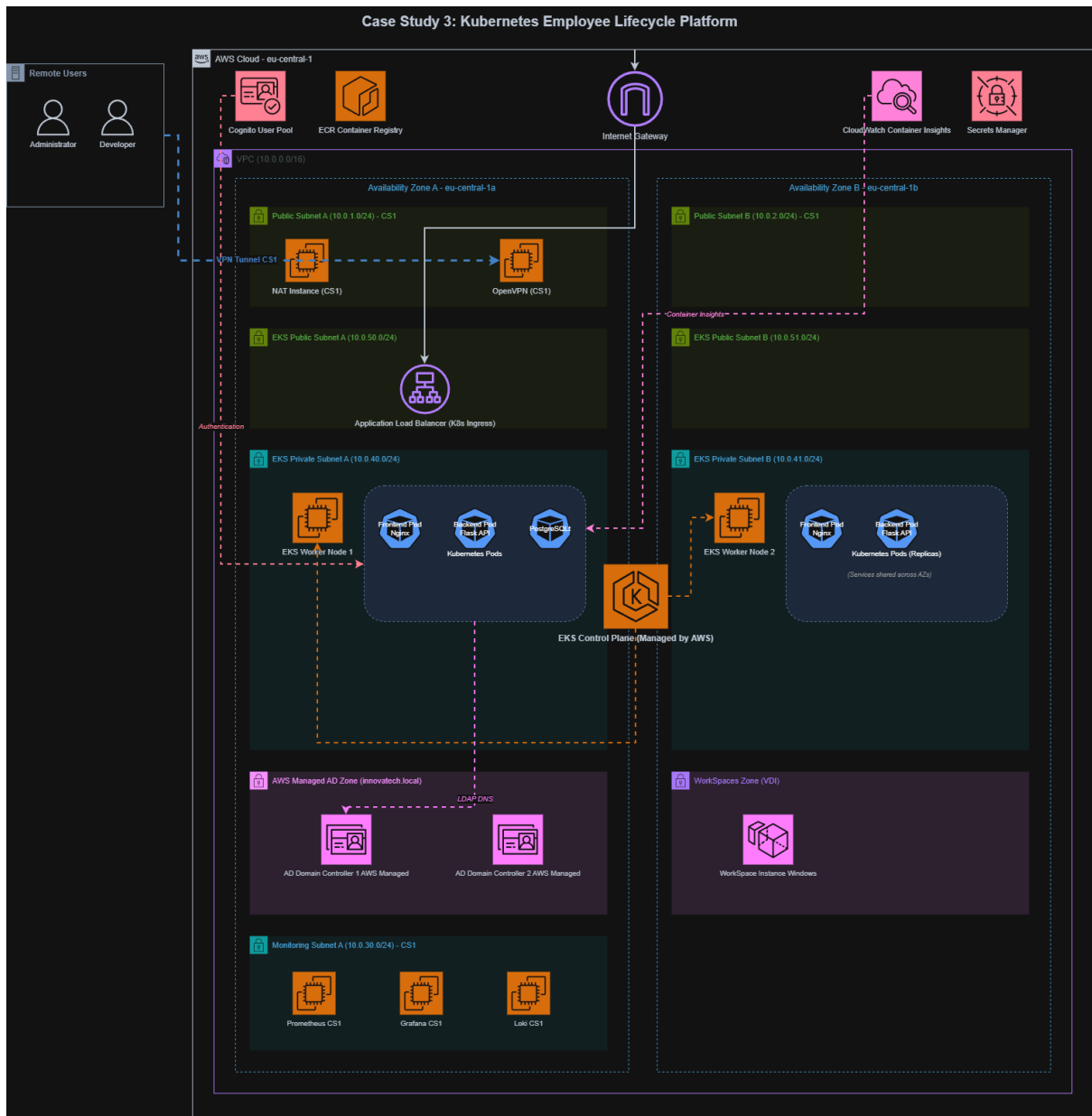
1.3 System Architecture Overview

The system follows a microservices architecture deployed on Kubernetes with three main components:

1. **Employee Portal (Frontend):** Nginx-served web interface for user interactions
2. **Backend API:** Flask application handling automation logic and database operations
3. **PostgreSQL Database:** StatefulSet providing persistent storage for employee records

The architecture integrates with AWS Managed Services:

- Cognito User Pool for authentication
- Active Directory for enterprise identity management
- WorkSpaces for provisioning virtual desktops
- GPO for software deployment



2. PHASE 1: ANALYSIS, DESIGN & PLANNING

2.1 Requirements Analysis

2.1.1 Functional Requirements

The system must support complete employee lifecycle automation:

Onboarding Process:

- Create user accounts in Cognito User Pool
- Provision Active Directory user with appropriate OU placement
- Deploy AWS WorkSpace with role-specific software
- Generate temporary credentials
- Store employee data in persistent database

Offboarding Process:

- Disable Cognito user account
- Update employee status in database
- Revoke active access requests
- Maintain audit trail

Self-Service Portal:

- User authentication via Cognito
- Role-based dashboard views
- Access request submission and tracking
- Admin approval workflow

2.1.2 Non-Functional Requirements

Scalability: System must handle at least 100 concurrent users with ability to scale horizontally

Availability: Target 99.5% uptime with graceful degradation during component failures

Security:

- Zero-trust principles with least-privilege access
- Encrypted data at rest and in transit
- Network micro-segmentation
- Comprehensive audit logging

Cost Optimization: Stay within \$200/month budget using free-tier resources where possible

Integration: Seamless integration with existing CS1 infrastructure without modifying proven components

2.2 Technology Choices & Justification

This section details the technology selection process, alternatives considered, and justification for final choices. Per feedback from CS2, I provide deeper analysis of alternatives and decision rationale.

2.2.1 Container Orchestration Platform

Requirement: Host multi-container application with auto-scaling, health checks, and persistent storage

Alternatives Evaluated:

Option	Pros	Cons	Student Suitability
AWS EKS	Industry-standard, extensive AWS integration, managed control plane, Container Insights built-in	Higher cost (\$0.10/hr cluster + nodes), steeper learning curve	Best for learning enterprise patterns
Docker Swarm	Simpler setup, lower overhead, integrated with Docker, no separate control plane cost	Limited AWS integration, smaller ecosystem, fewer features	Good for basic orchestration learning
ECS Fargate	Serverless, no node management, simpler than EKS, used successfully in CS2	Less portable, vendor lock-in, limited Kubernetes learning	Good but less educational value
EC2 Self-Managed K8s	Full control, cheaper than EKS, highly customizable	Significant management overhead, security responsibility, no managed upgrades	Too much operational burden

Decision: AWS EKS

Rationale:

1. **Learning Value:** EKS provides hands-on experience with Kubernetes, the industry-standard orchestration platform
2. **AWS Integration:** Native integration with IAM, VPC, CloudWatch, and other AWS services used in CS1/CS2

3. **Managed Control Plane:** Reduces operational overhead while maintaining full Kubernetes API access
4. **Future-Proof:** Skills directly transferable to any Kubernetes environment (GKE, AKS, on-premises)
5. **Monitoring:** Container Insights provides production-grade observability out of the box

Cost Mitigation: Using t3.medium nodes with cluster autoscaling and free-tier WorkSpaces to stay within budget

2.2.2 Identity Provider

Requirement: Application-level authentication with user management, password policies, and group-based authorization

Alternatives Evaluated:

Option	Pros	Cons	Decision Factors
AWS Cognito	Managed service, built-in MFA, OAuth/OIDC support, no server management, free tier (50k MAU)	Basic UI, limited customization, AWS-specific	Best for projects such as these
AWS IAM Identity Center	Enterprise SSO, SAML support, AWS-native integration	Blocked by AWS Academy Service Control Policy , requires organization-level setup	Not available in student accounts
Keycloak	Full-featured, highly customizable, open-source, self-hosted control	Requires infrastructure (EC2/EKS), operational overhead, database dependency	Too complex for timeline
Auth0	Excellent developer experience, extensive features, good documentation	Commercial service, costs increase with users, external dependency	Budget concerns

Decision: AWS Cognito

Rationale:

1. **AWS Academy Constraint:** IAM Identity Center blocked by Service Control Policies in student accounts
2. **Managed Service:** No infrastructure overhead for authentication system
3. **AWS Integration:** Direct integration with API Gateway, Lambda, and application workloads
4. **Cost:** Free tier covers project needs (50,000 monthly active users)
5. **Timeline:** Quick setup allows focus on core automation logic rather than identity infrastructure
6. **Security:** Built-in MFA, password policies, and token management

Trade-offs Accepted:

- Less customizable UI than self-hosted solutions
- AWS vendor lock-in for authentication layer
- Migration path exists if needed (federation with external IdP)

2.2.3 Active Directory Solution

Requirement: Enterprise directory service for WorkSpaces integration and Windows authentication

Alternatives Evaluated:

Option	Pros	Cons	Integration Complexity
AWS Managed Microsoft AD	Fully managed, WorkSpaces integration built-in, multi-AZ HA, automatic backups	Limited admin access, higher cost (\$162.84/month for Standard), no service account creation without AWS Support	Acceptable with workarounds
AD Connector	Lower cost (\$57.75/month), connects to on-premises AD	Requires existing AD infrastructure, single point of failure, not suitable for greenfield	No on-premises AD available
Self-Managed AD (EC2)	Full control, customizable, cheaper compute	High operational burden, backup responsibility, no managed HA, security hardening required	Too much management overhead
Simple AD	Cheaper option (\$49.30/month), Samba-based	Not compatible with WorkSpaces , limited AD features, no trust relationships	Incompatible with requirements

Decision: AWS Managed Microsoft AD (Standard Edition)

Rationale:

1. **WorkSpaces Requirement:** Only Managed AD supports WorkSpaces integration natively
2. **Managed Service:** Automatic patching, backups, and multi-AZ high availability
3. **Learning Value:** Experience with AWS-specific AD limitations prepares for real-world constraints
4. **Time Efficiency:** Managed service allows focus on automation rather than AD infrastructure

Architectural Adaptations Due to Limitations:

- **Default Admin account issue:** Default Admin account has permission limitations, which doesn't allow the creation of AD Users in OUs
 - **Workaround:** Create and use a Service account in the AD with delegated permissions for the automation(AD user creation in Ous)
 - **Learning:** Documented limitation for future enterprise implementations
- **API Hybrid Approach:**
 - Plain LDAP (port 389) for user/group queries within VPC
 - AWS Directory Service API for password operations via boto3
 - **Justification:** Avoids SSL/TLS certificate complexity while maintaining security via VPC isolation

Cost Analysis:

Monthly Cost Breakdown:

- Managed AD Standard: \$162.84/month
- Alternative (EC2 AD):
 - * t3.medium × 2 (HA): ~\$60/month
 - * EBS volumes: ~\$10/month
 - * Backup storage: ~\$5/month
 - * Operational time cost: ~40 hours setup + ongoing maintenance

Decision: Pay \$87.84 extra/month to save 40+ hours of setup and avoid ongoing operations

2.2.4 Database Solution

Requirement: Persistent storage for employee records with relational data model

Alternatives Evaluated:

Option	Pros	Cons	Suitability
PostgreSQL StatefulSet	In-cluster, no external dependency, fast access, cost-free, full control	Requires manual backup strategy, single point of failure, limited to node storage	Ideal for this project project
Amazon RDS PostgreSQL	Fully managed, automated backups, multi-AZ HA, read replicas	Blocked by AWS Academy Service Control Policy, minimum \$25/month	Not available in student accounts
Amazon DynamoDB	Serverless, auto-scaling, free tier available, no servers to manage	NoSQL model doesn't fit relational requirements well, more complex queries	Not ideal for relational data
Aurora Serverless v2	Auto-scaling, PostgreSQL compatible, only pay for usage	More expensive than RDS, complex pricing model, learning curve	Cost concerns

Decision: PostgreSQL StatefulSet on EKS

Rationale:

1. **AWS Academy Constraint:** RDS blocked by Service Control Policies
2. **Cost:** Zero additional cost using EKS node storage
3. **Performance:** Local access within cluster provides low latency
4. **Simplicity:** No cross-service IAM complexity
5. **Educational Value:** Experience with StatefulSets and persistent volumes in Kubernetes

Backup Strategy:

- Manual backup script using kubectl exec and pg_dump
- Storage configured with EBS volume for persistence across pod restarts
- Accept single-point-of-failure risk for student project timeline

Production Considerations Documented:

- Recommend RDS PostgreSQL for production deployment
- Multi-AZ configuration for high availability
- Automated backup with point-in-time recovery
- This is noted in final documentation as future improvement

2.2.5 Software Deployment: Group Policy (GPO) vs. AWS Systems Manager (SSM)

To install software (Node.js, PuTTY) on the WorkSpaces (REQ-P3-03) I evaluated two methods.

Criteria	AWS Systems Manager (SSM)	Group Policy Objects (GPO)	Analysis
Infrastructure	Requires SSM Agent + VPC Endpoints (4+ endpoints needed for private subnets).	Requires Active Directory Domain Controllers (already deployed for WorkSpaces).	Analysis: SSM requires complex networking (VPC Endpoints) to work in the isolated private subnets where WorkSpaces live. GPO traffic flows naturally to the AD Controller.
Permissions	Runs commands as SYSTEM.	"Computer Configuration" policies run as SYSTEM.	Analysis: Both provide the necessary Admin rights. However SSM associations are "eventually consistent" and can be delayed.
Reliability	Medium. Agent startup can lag behind boot causing software to appear "late".	High. Computer Policies apply synchronously during the boot sequence ensuring software is ready <i>before</i> login.	Decision: GPO is the industry standard for Windows VDI.

Decision: I chose **GPO (Computer Configuration)**. While SSM is "Cloud Native" GPO proved more reliable for ensuring the software was ready the instant the user logged in which was critical for the "Day 1" employee experience.

2.3 System Architecture Design

2.3.1 Network Architecture

The system extends the VPC infrastructure from Case Study 1:

Subnet Strategy:

- **Private Subnets (10.0.32.0/19, 10.0.64.0/19):** EKS worker nodes
- **Private Subnets (10.0.40.0/22, 10.0.44.0/22):** AWS Managed AD
- **Public Subnets:** Load balancers for portal access
- **VPN Subnet:** OpenVPN for administrative access (from CS1)

Security Group Architecture:

EKS Nodes Security Group:

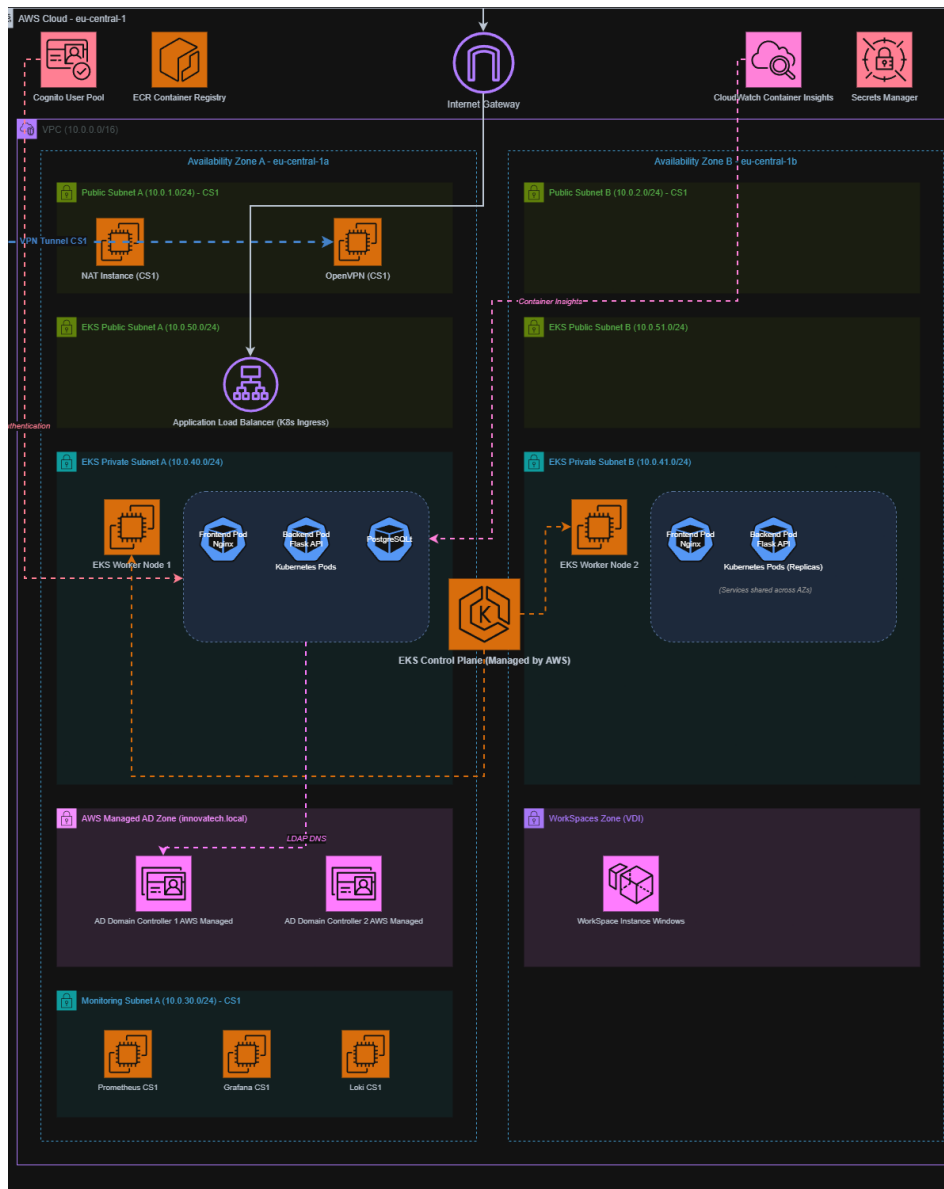
- └─ Inbound: Allow 443 from EKS Control Plane
- └─ Inbound: Allow 10250 from EKS Control Plane (kubelet)
- └─ Outbound: Allow all to AD security group (LDAP, Kerberos, DNS)
- └─ Outbound: Allow HTTPS to internet (for pulling images)

Active Directory Security Group:

- └─ Inbound: Allow 389 (LDAP) from EKS nodes and cluster SG
- └─ Inbound: Allow 636 (LDAPS) from EKS nodes and cluster SG
- └─ Inbound: Allow 88 (Kerberos) from EKS nodes and cluster SG
- └─ Inbound: Allow 53 (DNS) from EKS nodes and cluster SG
- └─ Inbound: Allow 3269 (Global Catalog SSL) from EKS nodes and cluster SG

WorkSpaces Security Group:

- └─ Inbound: Allow RDP from specific IP ranges
- └─ Outbound: Allow all (for internet access)



2.3.2 Kubernetes Architecture

Cluster Configuration:

- **Control Plane:** AWS-managed, multi-AZ for high availability
- **Node Group:** 2× t3.medium instances (2 vCPU, 4GB RAM each)
- **Networking:** VPC-CNI plugin with ENI-based pod networking
- **Version:** 1.31 (latest stable at deployment time)

Application Namespace:

employee-services/

└─ employee-portal (Deployment, 2 replicas)

| └─ Container: Flask backend (port 5000)

| └─ ServiceAccount: employee-portal-sa (IRSA for AWS API access)

└─ employee-portal-frontend (Deployment, 2 replicas)

| └─ Container: Nginx serving static files

└─ postgres (StatefulSet, 1 replica)

| └─ PersistentVolumeClaim: 20GB EBS volume

| └─ Service: ClusterIP for internal access

└─ Services:

| └─ employee-portal (ClusterIP for internal routing)

| └─ employee-portal-frontend (LoadBalancer for external access)

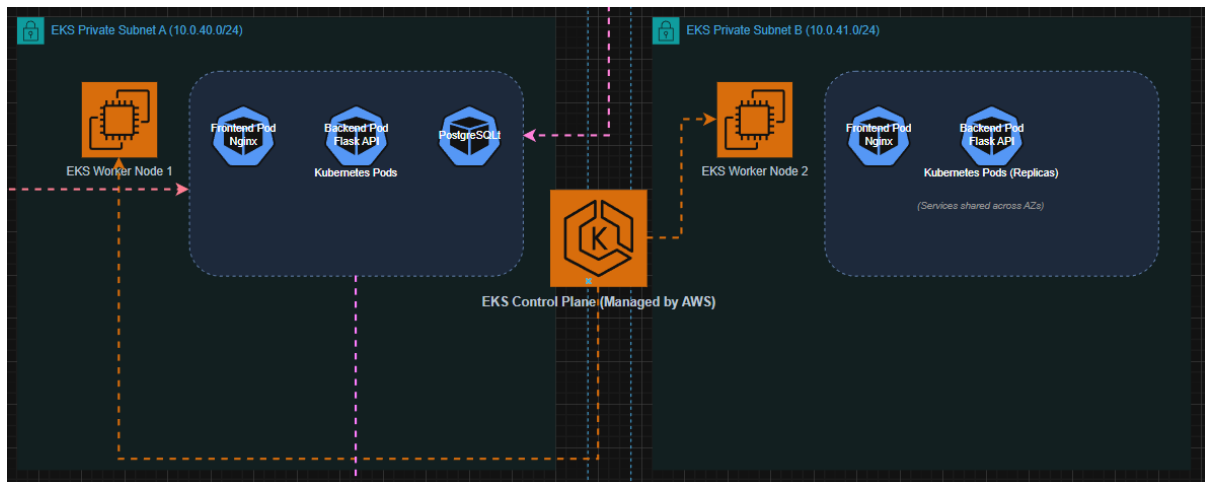
| └─ postgres (ClusterIP for database)

└─ ConfigMaps & Secrets:

└─ cognito-config (auto-synced from Terraform)

└─ ad-config (auto-synced from Terraform)

└─ employee-db-credentials (database password)



2.3.3 Evolutionary Architecture Pattern

Following the principle of evolutionary architecture from CS1 and CS2, CS3 extends existing infrastructure rather than rebuilding:

Benefits of Evolutionary Approach:

1. **Reduced Risk:** Proven infrastructure remains unchanged and operational
2. **Time Savings:** No need to recreate VPC, monitoring, or VPN infrastructure
3. **Consistency:** Same networking patterns and security controls as CS1/CS2
4. **Incremental Value:** Each case study adds capability without breaking existing functionality

2.3.4 Zero Trust Security Design

The system implements Zero Trust principles:

Identity-Centric Access:

- Every API request requires valid Cognito JWT token
- Kubernetes ServiceAccount with IRSA for AWS API access
- No long-lived credentials stored in application code

Micro-Segmentation:

- Kubernetes Network Policies restrict pod-to-pod communication
- Security groups enforce least-privilege network access
- VPC isolation for Active Directory

Encryption:

- TLS for all external communication (LoadBalancer)
- Secrets stored in Kubernetes Secrets (encrypted at rest by EKS)
- AD passwords stored in AWS Secrets Manager

Least Privilege:

- IAM roles with minimal required permissions
- RBAC policies in Kubernetes namespace-scoped
- Cognito groups for role-based portal access (admin, developer, employee)

2.4 DevOps & IaC Setup

2.4.1 Terraform Structure

The CS3 infrastructure is organized into modular Terraform files:

cs3-terraform/

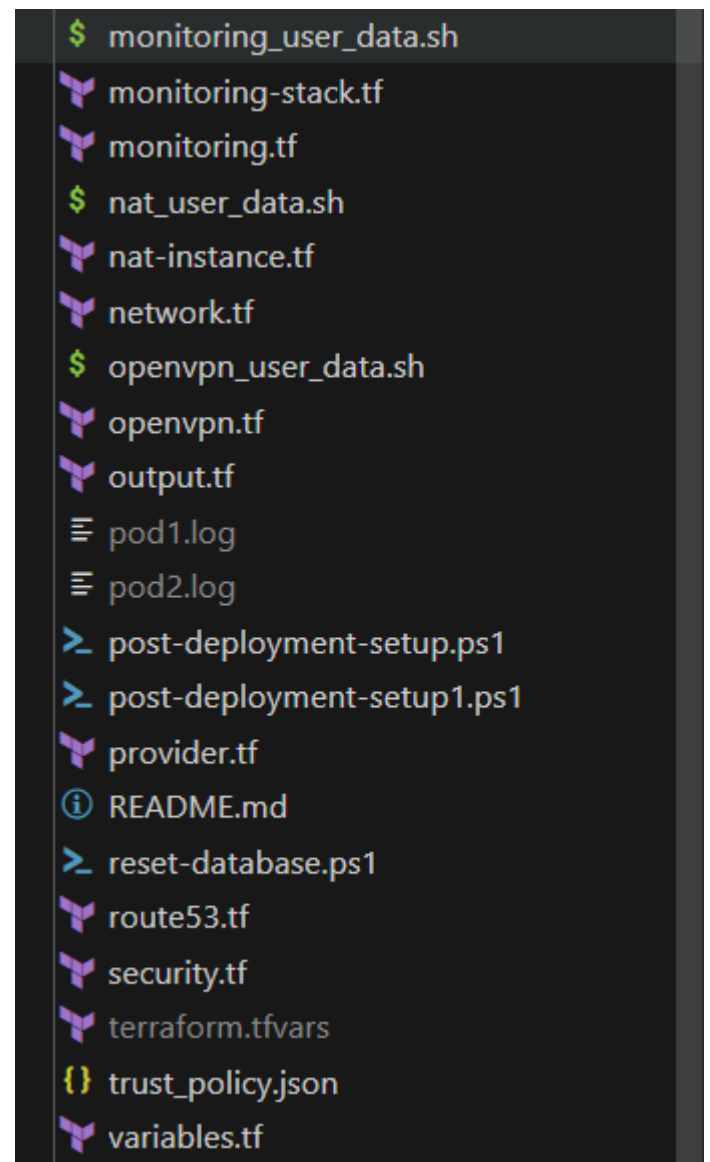
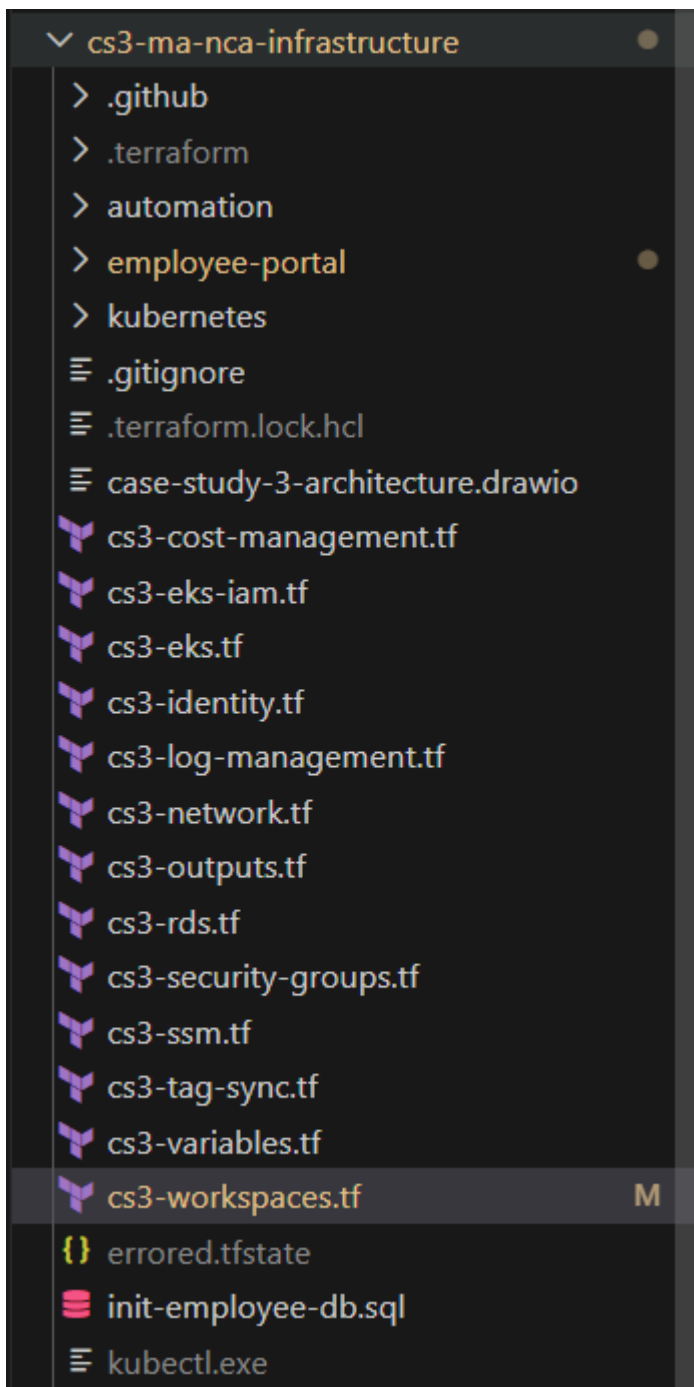
- └─ cs3-eks.tf # EKS cluster and node group
- └─ cs3-eks-iam.tf # IAM roles, IRSA, service account policies
- └─ cs3-cognito.tf # Cognito User Pool, groups, app client
- └─ cs3-workspaces.tf # Managed AD, WorkSpaces directory, admin workspace
- └─ cs3-monitoring.tf # CloudWatch Container Insights, alarms
- └─ cs3-kubernetes.tf # ConfigMaps auto-sync (Cognito, AD)
- └─ cs3-variables.tf # Configurable parameters
- └─ cs3-outputs.tf # Useful outputs (cluster name, endpoints)
- └─ backend.tf # S3 backend for state management

Key Design Patterns:

Auto-Sync ConfigMaps: Terraform creates Kubernetes ConfigMaps with values that Kubernetes manifests reference:

```
116 # Auto-create Kubernetes ConfigMap with Cognito values
117 resource "kubernetes_config_map" "cognito_config" {
118   metadata {
119     name      = "cognito-config"
120     namespace = "employee-services"
121   }
122
123   data = {
124     user_pool_id = aws_cognito_user_pool.employees.id
125     client_id    = aws_cognito_user_pool_client.employee_portal.id
126     region       = var.aws_region
127     domain       = aws_cognito_user_pool_domain.employees.domain
128   }
129
130   depends_on = [
131     kubernetes_namespace.employee_services
132   ]
133 }
```

This eliminates manual updates to Kubernetes manifests when AWS resources change.



2.4.2 CI/CD Pipeline

GitHub Actions workflow automates deployment:

Workflow Stages:

1. **Terraform Validation:** Format check, syntax validation, Trivy security scan
2. **Terraform Plan:** Preview infrastructure changes
3. **Manual Approval:** Required for applying changes
4. **Terraform Apply:** Deploy infrastructure changes
5. **Post-Deployment:** Run PowerShell script to configure K8s namespace, database schema, sample data

OIDC Authentication: GitHub Actions uses OpenID Connect to authenticate with AWS without long-lived credentials:

```
29  - name: Configure AWS Credentials
30    uses: aws-actions/configure-aws-credentials@v4
31  with:
32    role-to-assume: arn:aws:iam::098347675427:role/GitHubActionsRole
33    aws-region: ${{ env.AWS_REGION }}
```

Security Benefits:

- No AWS access keys in GitHub Secrets
- Automatic credential rotation
- Short-lived session tokens (1 hour)
- CloudTrail audit trail of all actions

Terraform

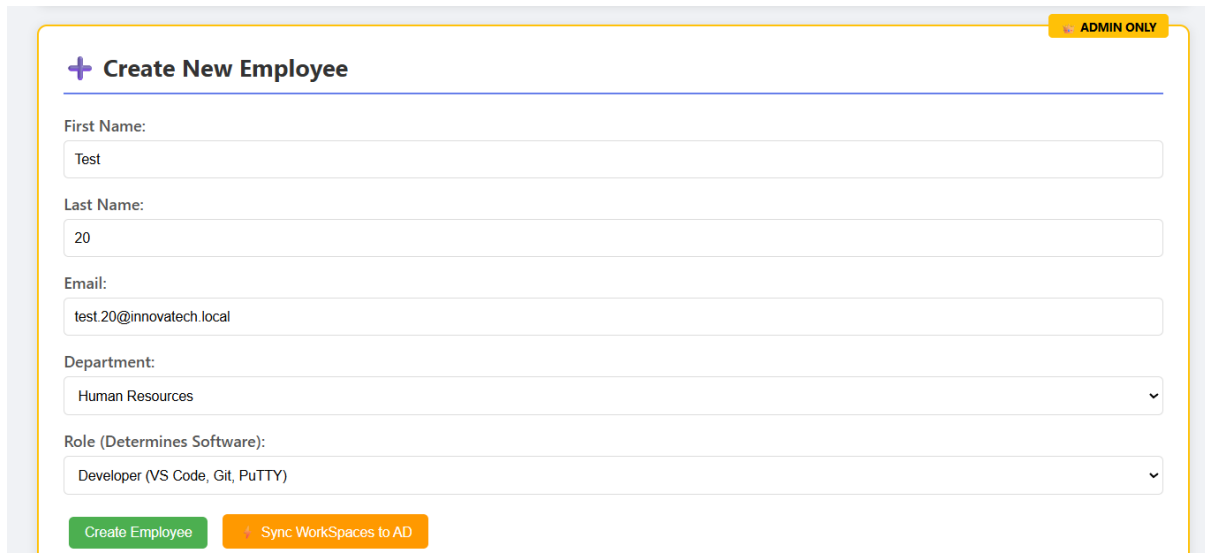
succeeded yesterday in 1m 39s

- > ☒ Set up job
- > ☒ Checkout
- > ☒ Configure AWS Credentials
- > ☒ Setup Terraform
- > ☒ Test AWS Access
- > ☒ Terraform Format Check
- > ☒ Terraform Init
- > ☒ Terraform Validate
- ☐ Terraform Plan
- ☐ Update Pull Request
- ☐ Terraform Plan Status
- > ☒ Terraform Apply
- > ☒ Configure kubectl
- > ☒ Deploy AWS Load Balancer Controller
- > ☒ Deploy Employee Portal
- > ☒ Post Configure AWS Credentials
- > ☒ Post Checkout
- > ☒ Complete job

3. PHASE 2: EMPLOYEE LIFECYCLE AUTOMATION

3.1 User creation (onboarding)

3.1.1 Admin-only employee creation interface



The image shows a web form titled '+ Create New Employee' with a yellow 'ADMIN ONLY' badge in the top right corner. The form contains the following fields: 'First Name' with the value 'Test', 'Last Name' with the value '20', 'Email' with the value 'test.20@innovatech.local', 'Department' as a dropdown menu showing 'Human Resources', and 'Role (Determines Software)' as a dropdown menu showing 'Developer (VS Code, Git, PuTTY)'. At the bottom of the form are two buttons: a green 'Create Employee' button and an orange 'Sync WorkSpaces to AD' button.

Admin-only employee creation form in the Employee Portal

Admin-only employee creation form in the Employee Portal. Only users with the admin role can access the employee onboarding form, enforcing role-based access control at the UI level.

3.1.2 Successful onboarding response from backend



The image shows a confirmation message within a form container. At the top are two buttons: a green 'Create Employee' button and an orange 'Sync WorkSpaces to AD' button. Below the buttons is a green checkmark icon followed by the text 'User Test 20 created successfully'. Underneath this, the following status information is listed: 'Employee ID: 30', 'Cognito: created', 'Database: created', and 'Workspace: provisioning'.

Backend confirmation of successful employee onboarding across systems

After submitting the onboarding request, the backend confirms successful creation in Cognito, the database, and the initiation of Workspace provisioning.

3.1.3 Newly created employee visible in portal database

30	Test 20	test.20@innovatech.local	Human Resources	Developer	active	Term
----	---------	--------------------------	-----------------	-----------	--------	------

Newly onboarded employee visible in the portal employee directory

The employee appears in the portal immediately after onboarding, confirming that the database record was created successfully.

3.1.4 Cognito user account created and enabled

User: test.20@innovatech.local info

Actions

User information

User ID (Sub)
93a49832-60a1-70f3-06c5-95a007cfd180

Alias attributes used to sign in
User name

MFA setting
MFA inactive

MFA methods
-

Account status
Enabled

Confirmation status
Force change password

Created time
December 14, 2025 at 21:48 GMT+1

Last updated time
December 14, 2025 at 21:48 GMT+1

Cognito user account created and enabled for the new employee

The onboarding process creates a Cognito user account in an enabled state, allowing the employee to authenticate using the centralized identity provider.

3.1.5 Cognito group assignment for role-based access

Group: employees

Delete

Group information

Group name
employees

IAM Role ARN
-

Description
Standard employee access

Precedence
3

Created time
December 10, 2025 at 18:45 GMT+1

Last updated time
December 10, 2025 at 18:45 GMT+1

Group members (25) info

Remove user from group

Add user to group

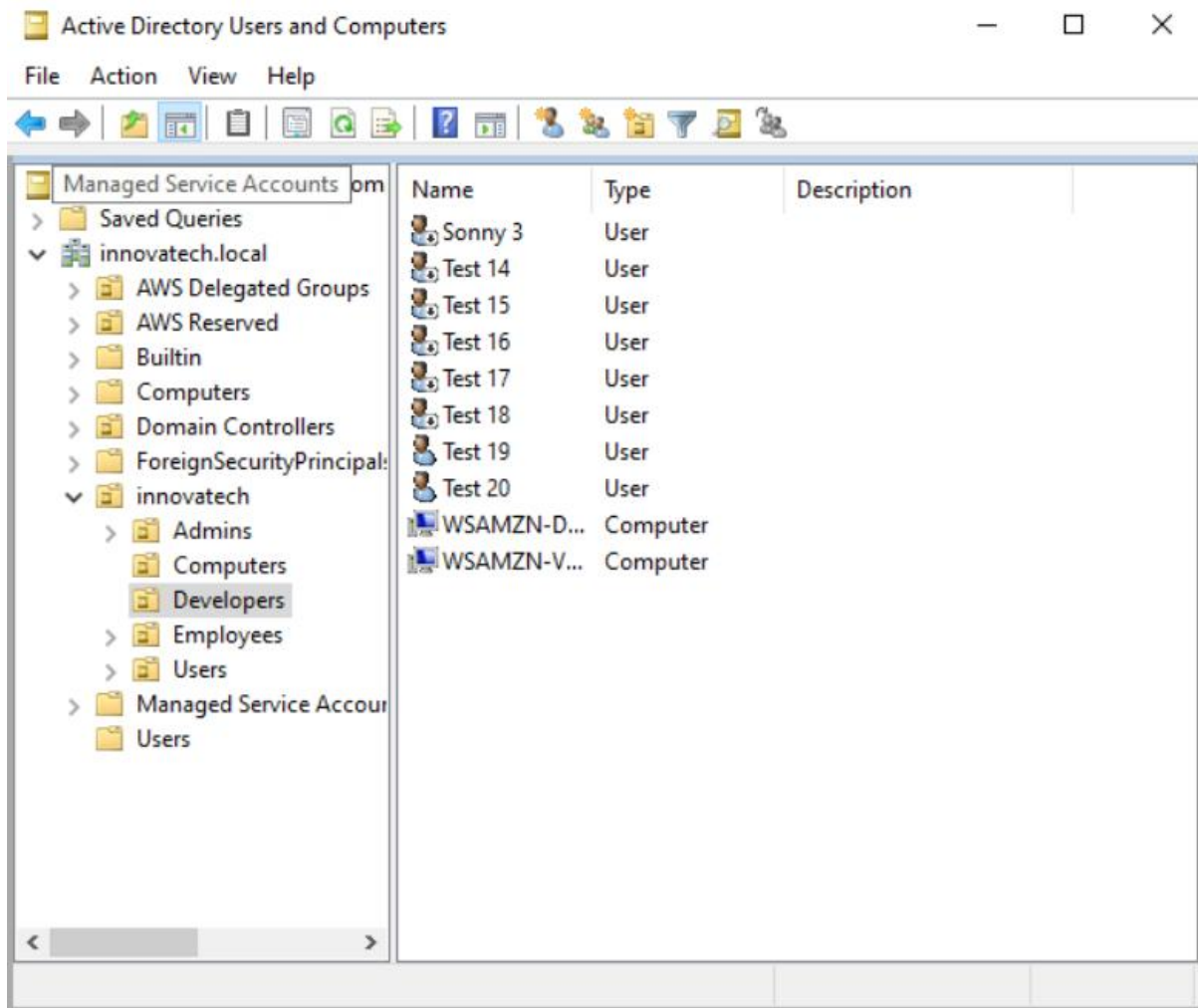
< 1 2 3 >

User name	Email address	Email verified	Confirmation status	Status
test.14@innovatech.local	test.14@innovatech.local	No	Force change password	Disabled
test.19@innovatech.local	test.19@innovatech.local	No	Confirmed	Enabled
test.16@innovatech.local	test.16@innovatech.local	No	Force change password	Enabled
test.20@innovatech.local	test.20@innovatech.local	No	Force change password	Enabled
test.4@innovatech.local	test.4@innovatech.local	No	Confirmed	Disabled

Cognito group membership assigned during onboarding

The user is automatically added to the appropriate Cognito group, enabling role-based authorization within the Employee Portal.

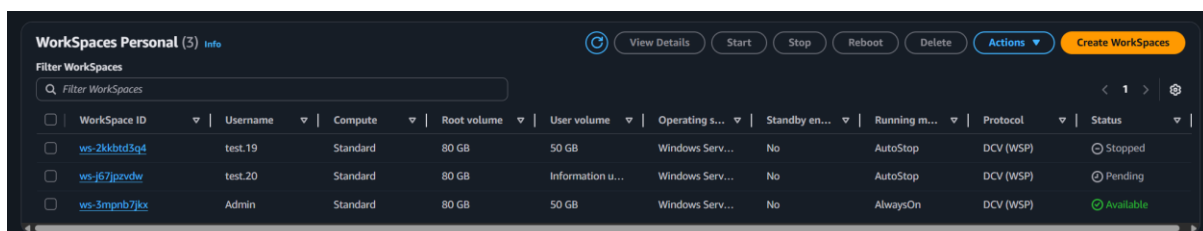
3.1.6 Active Directory user created in correct organizational unit



Active Directory user created in the correct organizational unit

The employee account is created in Active Directory under the appropriate OU, aligning with role-based access control and group policy management.

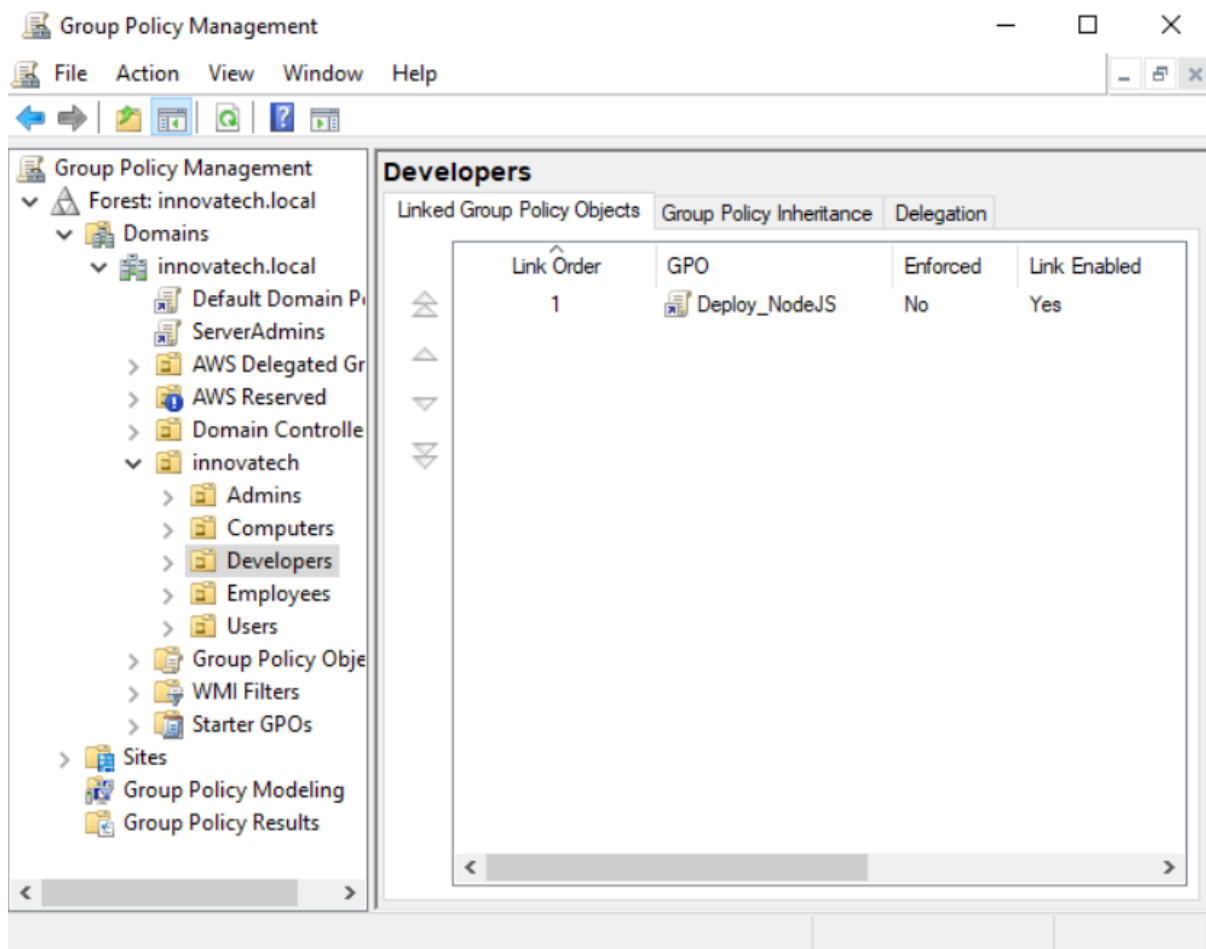
3.1.7 AWS WorkSpace provisioning for the new employee



AWS WorkSpace provisioning initiated for the onboarded employee

An AWS WorkSpace is automatically provisioned for the employee, providing a managed desktop environment linked to the Active Directory account.

3.1.8 GPO linked to Developers OU



Computer-based developer GPO linked to the Developers OU

After the WorkSpace is provisioned and joined to Active Directory, role-based software configuration is applied using **computer-based Group Policy Objects**. These GPOs are linked to role-specific organizational units and apply based on the **location of the WorkSpace computer object**, not the user account. Once the WorkSpace computer is moved into the appropriate OU (for example Developers), the linked GPO applies automatically and installs the required developer tooling without additional cloud-side provisioning logic.

3.1.9 Portal “Sync WorkSpaces to AD” success message



Create Employee Sync WorkSpaces to AD

Sync complete. Moved 1 computers.

Successful “Sync WorkSpaces to AD” operation moving Workspace computer objects into role-based Ous

After Workspace provisioning, the portal sync moves the computer object based on the Workspace Role tag so the correct GPO is applied.

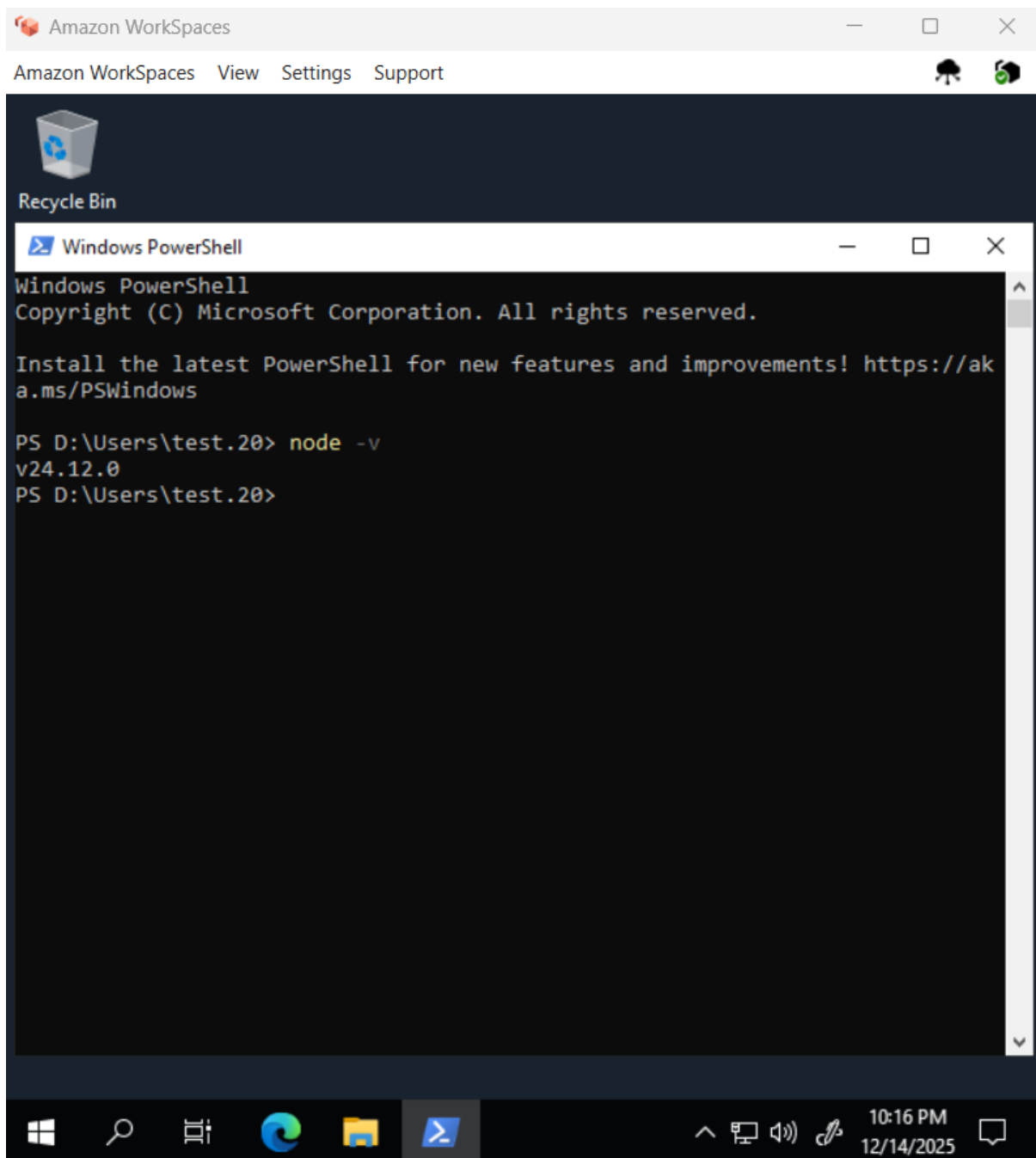
Role-based software configuration (Computer-based GPO via Workspace OU placement)

Role-specific software is deployed using **computer-based Group Policy Objects (GPOs)** linked to role-specific organizational units (for example *Developers*). Because these policies apply to the **Workspace computer object rather than the user account**, a post-provisioning synchronization step is required.

After a Workspace is provisioned and reaches the **AVAILABLE** state, the portal’s *Sync WorkSpaces to AD* function moves the corresponding computer account into the correct role-based OU. Once the computer object is placed in the appropriate OU, the linked GPO is applied automatically, resulting in the installation of the expected role-specific tooling (such as developer software) on the Workspace.

Because Group Policy is applied using computer-based configuration, an automated synchronization step ensures that newly provisioned Workspace computer objects are moved into the correct role-based OU after becoming available. The synchronization logic is implemented in the backend and exposed through an administrator-only maintenance endpoint.

3.1.10 Proof software applied on the WorkSpace



Developer tooling available after Group Policy application on the WorkSpace

NodeJS is available on the WorkSpace, confirming that the computer-based developer GPO was applied successfully.

3.2 User termination (offboarding)

3.2.1 Secure offboarding and access revocation (Identity, directory, and WorkSpace cleanup)

After an administrator initiates the termination process through the employee portal, the backend executes a coordinated offboarding workflow to revoke access across all integrated systems. The employee's account is immediately disabled in Amazon Cognito to prevent further authentication attempts.

At the same time, the employee record in the internal database is updated to reflect a terminated status, ensuring that the user is no longer treated as active by the application. This causes the employee to be removed from the active employee directory in the portal interface.

The corresponding Active Directory user account is then disabled, preventing any further domain-based authentication or access to corporate resources. This ensures that both cloud-based and directory-based access are revoked consistently.

Finally, the Amazon WorkSpace associated with the employee is terminated. Once the WorkSpace enters the terminating state, the virtual desktop becomes unavailable, completing the offboarding process and ensuring that no compute resources remain assigned to the former employee.

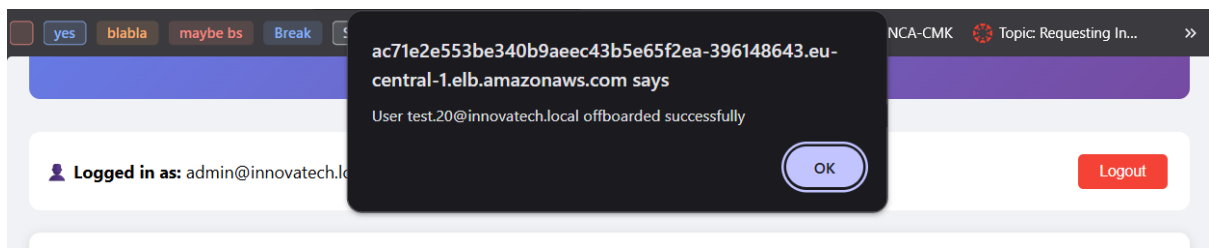
3.2.2 Admin-initiated termination action

30	Test 20	test.20@innovatech.local	Human Resources	Developer	active	Term
----	---------	--------------------------	-----------------	-----------	--------	------

Administrator-initiated user termination with identity, access, and WorkSpace deprovisioning

The administrator selects the termination action for the employee, triggering the centralized offboarding workflow.

3.2.3 Portal confirmation of successful offboarding



Portal confirmation message indicating that the employee was offboarded successfully

After the termination request is processed, the portal confirms that the offboarding workflow completed without errors.

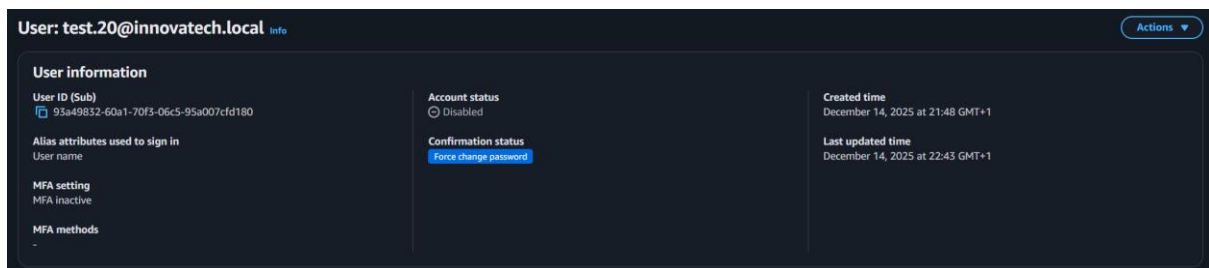
3.2.4 Employee removed from active portal view

Employee Directory						
Refresh						
ID	Name	Email	Department	Position	Status	Actions
1	John Doe	john.doe@innovatech.local	Engineering	Senior Developer	active	Term
2	Jane Smith	jane.smith@innovatech.local	Human Resources	HR Manager	active	Term
3	Bob Johnson	bob.johnson@innovatech.local	Engineering	DevOps Engineer	active	Term
4	Alice Williams	alice.williams@innovatech.local	Finance	Financial Analyst	active	Term
5	Admin User	admin@innovatech.local	IT	Administrator	active	Term
11	Sonny 3	sonny.3@innovatech.local	Engineering	Developer	active	Term
26	Test 16	test.16@innovatech.local	Engineering	Developer	active	Term
27	Test 17	test.17@innovatech.local	Engineering	Developer	active	Term
29	Test 19	test.19@innovatech.local	Engineering	Developer	active	Term

Terminated employee no longer visible in the active employee directory

Once the database status is updated, the terminated employee is removed from the active employee list in the portal.

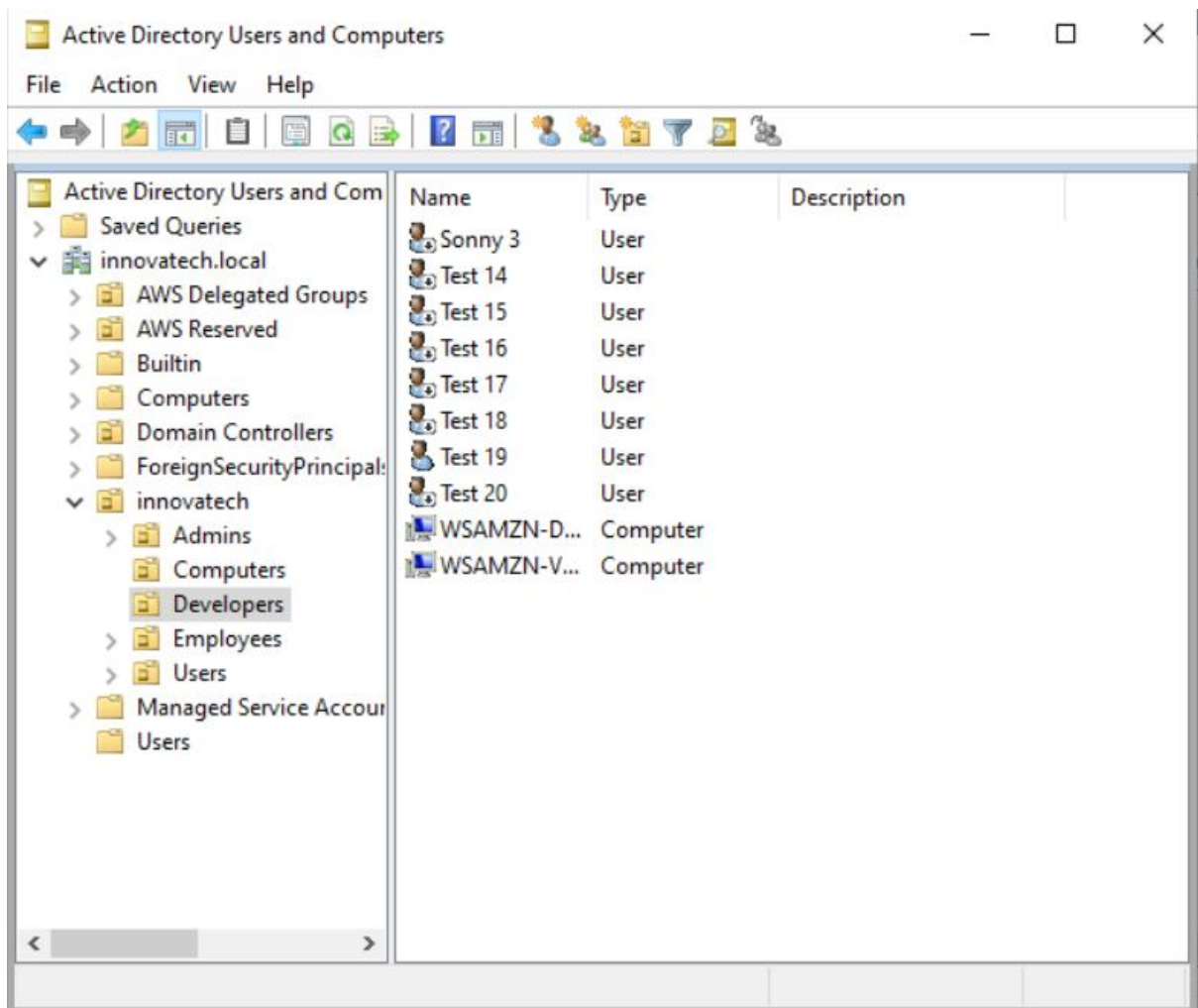
3.2.5 Cognito account disabled



Cognito user account disabled to prevent further authentication

The employee's Cognito account is disabled, immediately blocking access to the portal and all Cognito-protected resources.

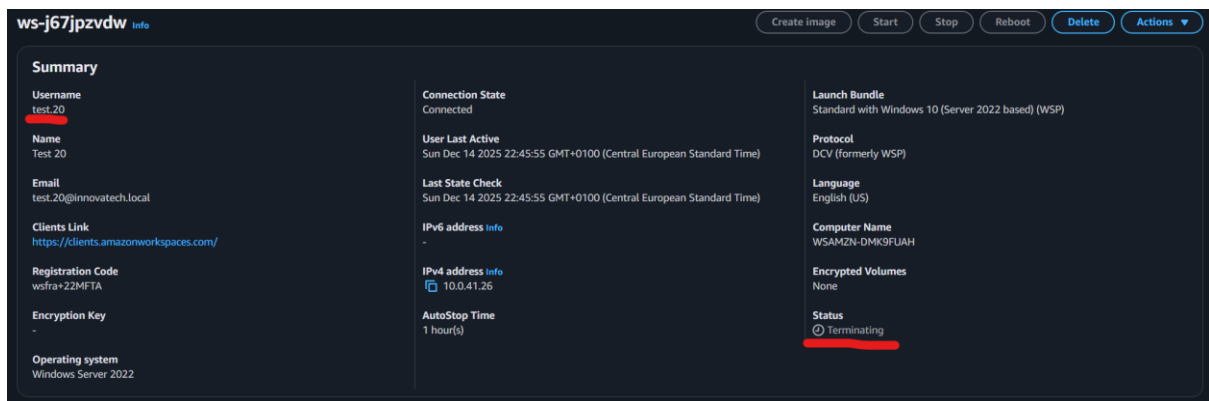
3.2.6 Active Directory user disabled



Active Directory user account disabled as part of the offboarding process

The corresponding Active Directory user account is disabled to revoke domain-based authentication and access.

3.2.7 Amazon WorkSpace termination

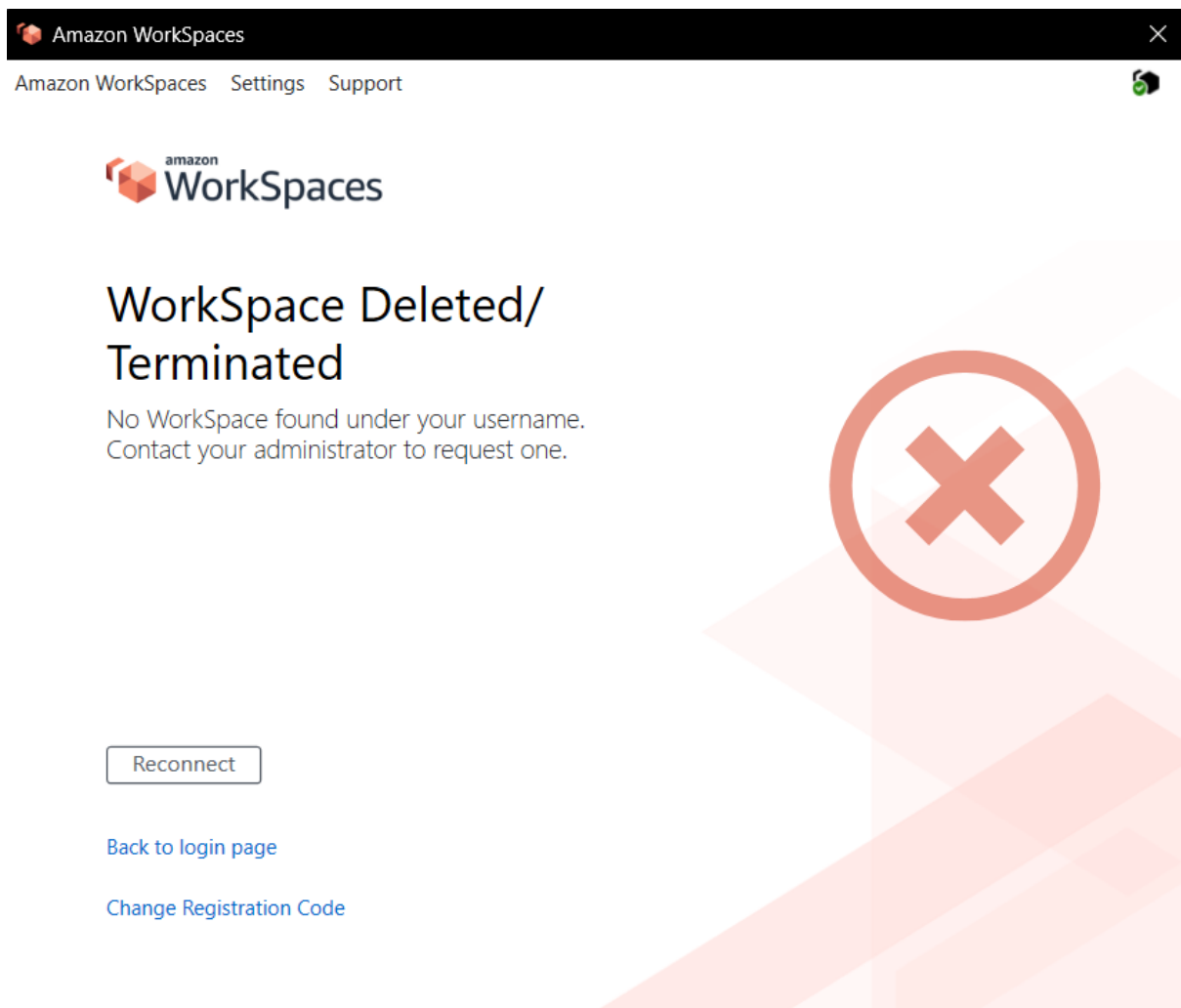


The screenshot shows the Amazon WorkSpace console for a workspace named 'ws-j67jpzvdw'. The workspace is in a 'Terminating' state, indicated by a red circle around the status label. The console displays various details about the workspace, including its configuration, connection state, and user information.

Summary		
Username test-20	Connection State Connected	Launch Bundle Standard with Windows 10 (Server 2022 based) (WSP)
Name Test 20	User Last Active Sun Dec 14 2025 22:45:55 GMT+0100 (Central European Standard Time)	Protocol DCV (formerly WSP)
Email test.20@innovatech.local	Last State Check Sun Dec 14 2025 22:45:55 GMT+0100 (Central European Standard Time)	Language English (US)
Clients Link https://clients.amazonworkspaces.com/	IPv6 address info -	Computer Name WSAMZN-DMK9FUAH
Registration Code wsfra+ZZMFTA	IPv4 address info 10.0.41.26	Encrypted Volumes None
Encryption Key -	AutoStop Time 1 hour(s)	Status Terminating
Operating system Windows Server 2022		

Amazon WorkSpace entering terminating state after employee offboarding

The employee's WorkSpace is terminated, ensuring that no virtual desktop resources remain assigned after offboarding.



The screenshot shows the Amazon WorkSpaces client interface. At the top, there is a header bar with the Amazon WorkSpaces logo and navigation links for 'Amazon WorkSpaces', 'Settings', and 'Support'. Below the header, the main content area displays a large red 'X' icon, indicating an error or termination. The text reads: 'Workspace Deleted/Terminated. No Workspace found under your username. Contact your administrator to request one.' Below this message, there is a 'Reconnect' button and two links: 'Back to login page' and 'Change Registration Code'.

3.3 Identity Provider Implementation

3.3.1 Cognito User Pool Configuration

The Cognito User Pool provides application-level authentication:

Configuration Details:

```
1 # Cognito User Pool for employee authentication
2 resource "aws_cognito_user_pool" "employees" {
3   name = "${var.project_name}-employees"
4
5   # Password policy
6   password_policy {
7     minimum_length      = 8
8     require_lowercase   = true
9     require_uppercase   = true
10    require_numbers     = true
11    require_symbols      = true
12    temporary_password_validity_days = 7
13  }
14
15  # Auto-verify email
16  auto_verified_attributes = ["email"]
17
18  # User attributes
19  schema {
20    name           = "email"
21    attribute_data_type = "String"
22    required       = true
23    mutable        = false
24  }
25
26  schema {
27    name           = "name"
28    attribute_data_type = "String"
29    required       = true
30    mutable        = true
31  }
32
33  schema {
34    name           = "department"
35    attribute_data_type = "String"
36    required       = false
37    mutable        = true
38    developer_only_attribute = false
39  }
40
41  # Account recovery
42  account_recovery_setting {
43    recovery_mechanism {
44      name     = "verified_email"
45      priority = 1
46    }
47  }
```

User Groups for RBAC:

- **admins:** Full portal access, approve requests, view all employees

- **developers:** Submit access requests, deploy WorkSpaces with dev tools
- **employees:** Basic access, personal information management

Groups info

Groups (3) Info

Configure groups and add users. Groups can be used to add permissions to the access token for multiple users.

Filter groups by name and description

Group name	Description	Precedence	Created time
admins	Administrator group with full access	1	3 days ago
developers	Developer group with limited access	2	3 days ago
employees	Standard employee access	3	3 days ago

3.3.2 Authentication Flow

The portal implements OAuth 2.0 Authorization Code flow:

1. **User Login Request:** Browser redirects to Cognito hosted UI
2. **Cognito Authentication:** User enters credentials, MFA if enabled
3. **Authorization Code:** Cognito redirects back with authorization code
4. **Token Exchange:** Backend exchanges code for JWT tokens (ID token, access token, refresh token)
5. **Session Management:** Backend validates JWT on each API request

Token Validation in Backend:

```

226 # --- SECURITY: TOKEN VERIFICATION ---
227 def verify_token(token):
228     """Verify Cognito JWT token"""
229     try:
230         keys_url = f"{COGNITO_ISSUER}/.well-known/jwks.json"
231         keys = requests.get(keys_url).json()['keys']
232         header = jwt.get_unverified_header(token)
233         key = next((k for k in keys if k['kid'] == header['kid']), None)
234         if not key:
235             return None
236         return jwt.decode(token, key, algorithms=['RS256'], audience=COGNITO_CLIENT_ID, issuer=COGNITO_ISSUER)
237     except Exception as e:
238         print(f"Token verification error: {e}")
239     return None

```

3.4 Active Directory Integration

3.4.1 AWS Managed AD Setup

The Managed AD provides enterprise directory services:

Directory Configuration:

- **Edition:** Standard (sufficient for project, 30,000 object limit)
- **Domain:** innovatech.local
- **DNS IPs:** 10.0.41.73, 10.0.45.201 (automatically assigned)
- **Subnets:** Two private subnets in different AZs for HA
- **VPC:** Same VPC as EKS cluster

OU Structure:

innovatech.local

```
└─ OU=innovatech
    ├── OU=Employees
    ├── OU=Developers
    └─ OU=Admins
```

Security Groups:

CN=EmployeesGroup # Standard employees

CN=DevelopersGroup # Software developers with additional software

CN=AdminsGroup # IT administrators

Active Directory Users and Com

>

Saved Queries

▼

innovatech.local

>

AWS Delegated Groups

>

AWS Reserved

>

Builtin

>

Computers

>

Domain Controllers

>

ForeignSecurityPrincipal:

▼

innovatech

Admins

Computers

Developers

>

Employees

>

Users

Managed Service Accour

Users

Name	Type	Description
Admins	Organizational...	IT administrators
Computers	Organizational...	
Developers	Organizational...	Software developers
Employees	Organizational...	Standard employees
Users	Organizational...	
AdminsGroup	Security Group...	IT Administrators
DevelopersG...	Security Group...	Developers with code ac...
EmployeesG...	Security Group...	Standard employees
svc-employ...	User	

3.4.2 Hybrid LDAP/API Approach

Due to AWS Managed AD limitations, I implemented a hybrid approach:

Plain LDAP for Queries (Port 389):

```
84 def get_ad_connection():
85     """Plain LDAP connection (no SSL) - acceptable for VPC-internal communication"""
86     creds = get_ad_service_credentials()
87     if not creds or not creds.get('password'):
88         print("✗ No AD credentials available")
89         return None
90
91     try:
92         ad_server = '10.0.41.73'
93
94         server = Server(
95             ad_server,
96             port=389,
97             use_ssl=False, # Plain LDAP
98             get_info=NONE,
99             connect_timeout=5
100         )
101
102         conn = Connection(
103             server,
104             user=f'INNOVATECH\\{creds["username"]}',
105             password=creds['password'],
106             authentication=NTLM,
107             auto_bind=True,
108             raise_exceptions=True
109         )
110
111         print(f"✅ Connected to AD (LDAP) at {ad_server}:389")
112         return conn
113
114     except Exception as e:
115         print(f"✗ AD connection failed: {e}")
116         import traceback
117         traceback.print_exc()
118         return None
```

AWS Directory Service API for Password Operations:

```
1 # 3. Set password using AWS DS (Global Client + Retry Logic)
2 password_set = False
3 password = 'TempPass123!'
4 max_retries = 15
5
6 for attempt in range(max_retries):
7     try:
8         ds_client.reset_user_password(
9             DirectoryId=DIRECTORY_ID,
10            Username=username,
11            NewPassword=password
12        )
13        print("Password set via AWS DS API")
14        password_set = True
15        break
16    except ClientError as e:
17        error_code = e.response['Error']['Code']
18        if error_code == 'UserDoesNotExistException' and attempt < max_retries - 1:
19            print(f"AWS DS hasn't seen the user yet. Retrying in 3s... (Attempt {attempt + 1}/{max_retries})")
20            time.sleep(3)
21            continue
22        else:
23            print(f"AWS DS password reset failed: {e}")
24            conn.unbind()
25            return False
26
27 if not password_set:
28     print("Failed to set password after retries")
29     conn.unbind()
30     return False
```

Retry logic for Active Directory password provisioning via AWS Directory Service

Justification for Plain LDAP:

1. **VPC Isolation:** All communication stays within private VPC
2. **No Internet Exposure:** LDAP traffic never leaves VPC
3. **Simplicity:** Avoids SSL/TLS certificate management complexity
4. **Security Trade-off:** Acceptable for student project within isolated network
5. **Production Note:** Would use LDAPS (636) or Global Catalog SSL (3269) in production

3.5 Automation Scripts Development

3.5.1 Backend API Architecture

The Flask backend consolidates all automation logic:

Application Structure:

app.py (unified backend)

└─ Database operations (PostgreSQL)

└─ Cognito user management (boto3)

└─ Active Directory operations (ldap3)

└─ WorkSpaces provisioning (boto3)

└─ API endpoints for frontend

Design Decision: Consolidated vs Separate Scripts

Initial Approach (Separate Scripts):

automation/

└─ onboard-employee.py # CLI script

└─ offboard-employee.py # CLI script

└─ requirements.txt

Final Approach (Consolidated Backend):

employee-portal/backend/

└─ app.py # Single Flask app with all logic

Rationale for Consolidation:

1. **Shared Dependencies:** All operations need database, Cognito, and AD access
2. **API Exposure:** Portal needs endpoints, not CLI scripts
3. **Simpler Deployment:** One container image instead of multiple Lambda functions
4. **Easier Testing:** Single application to test and debug
5. **Resource Efficiency:** Shared connection pools and AWS clients

```

1  from flask import Flask, request, jsonify
2  from flask_cors import CORS
3  from jose import jwt
4  from botocore.exceptions import ClientError
5  from datetime import datetime
6  from functools import wraps
7  from ldap3.core.exceptions import LDAPException
8  import boto3
9  import psycpg2
10 import os
11 import requests
12 import json
13 from ldap3 import Server, Connection, ALL, NONE, NTLM, MODIFY_REPLACE, MODIFY_ADD, Tls
14 import time
15
16 app = Flask(__name__)
17 CORS(app)
18
19 # --- CONFIGURATION ---
20 DB_HOST = os.environ.get('DB_HOST', 'localhost')
21 DB_NAME = os.environ.get('DB_NAME', 'employees')
22 DB_USER = os.environ.get('DB_USER', 'admin')
23 DB_PASSWORD = os.environ.get('DB_PASSWORD', '')
24 AWS_REGION = os.environ.get('AWS_REGION', 'eu-central-1')
25 USER_POOL_ID = os.environ.get('COGNITO_USER_POOL_ID', '')
26 COGNITO_ISSUER = f"https://cognito-idp.{AWS_REGION}.amazonaws.com/{USER_POOL_ID}"
27 COGNITO_CLIENT_ID = os.environ.get('COGNITO_CLIENT_ID')
28
29 # WorkSpaces & AD Configuration
30 AD_HOST = os.environ.get('AD_HOST', 'innovatech.local')
31 DIRECTORY_ID = os.environ.get('AD_DIRECTORY_ID', '')
32 BUNDLE_ID = os.environ.get('AD_BUNDLE_ID', '')
33
34 # Initialize AWS Clients
35 cognito = boto3.client('cognito-idp', region_name=AWS_REGION)
36 workspaces = boto3.client('workspaces', region_name=AWS_REGION)
37 secretsmanager = boto3.client('secretsmanager', region_name=AWS_REGION)
38 ds_client = boto3.client('ds', region_name=AWS_REGION)
39
40 # Cache service account credentials
41 _ad_service_creds = None

```

Consolidated Flask backend structure (single app for DB, Cognito, AD, WorkSpaces)

3.5.2 Onboarding Workflow

Complete Onboarding Process:

1. Create Cognito User:

```
602 # Step 1: Create Cognito user
603 try:
604     print(f"Creating Cognito user: {email}")
605
606     cognito.admin_create_user(
607         UserPoolId=USER_POOL_ID,
608         Username=email,
609         UserAttributes=[
610             {'Name': 'email', 'Value': email},
611             {'Name': 'name', 'Value': f'{first_name} {last_name}'}
612         ],
613         TemporaryPassword='TempPass123!',
614         MessageAction='SUPPRESS'
615     )
616
617     group_name = 'admins' if position.lower() == 'admin' else 'employees'
618     cognito.admin_add_user_to_group(
619         UserPoolId=USER_POOL_ID,
620         Username=email,
621         GroupName=group_name
622     )
623
624     print(f"Cognito user created: {email} (group: {group_name})")
625
626 except ClientError as e:
627     error_code = e.response['Error']['Code']
628     if error_code == 'UsernameExistsException':
629         return jsonify({'error': 'User already exists in Cognito'}), 400
630     return jsonify({'error': f"Cognito error: {error_code}"}), 500
```

Creating a Cognito user and assigning role-based access

This snippet shows how the backend creates a new user in Amazon Cognito and assigns the user to the correct group based on their role. This establishes the user's cloud identity and role-based access as the first step in the onboarding workflow.

2. Create Database Record:

```
632 # Step 2: Create database record
633 try:
634     print(f"Creating database record...")
635
636     conn = get_db()
637     cur = conn.cursor()
638
639     cur.execute("""
640         INSERT INTO employees
641         (first_name, last_name, email, department, position, status, hire_date)
642         VALUES (%s, %s, %s, %s, %s, 'active', CURRENT_DATE)
643         RETURNING employee_id
644     """, (first_name, last_name, email, department, position))
645
646     employee_id = cur.fetchone()[0]
647
648     conn.commit()
649     cur.close()
650     conn.close()
651
652     print(f"Database record created: Employee ID = {employee_id}")
653
654 except Exception as e:
655     print(f"Database error: {str(e)}")
656
657     # Rollback: Delete Cognito user
658     try:
659         cognito.admin_delete_user(UserPoolId=USER_POOL_ID, Username=email)
660         print(f"Rolled back Cognito user: {email}")
661     except:
662         pass
663
664     return jsonify({'error': 'Failed to create database record'}), 500
665
```

Creating the employee database record with rollback on failure

This snippet inserts a new employee into PostgreSQL. If the database step fails, the backend catches the exception, logs the error, and performs a compensating action by deleting the previously created Cognito user to avoid an inconsistent onboarding state.

3. Provision Active Directory User (LDAP + AWS DS password + RBAC groups):

```
1  # --- ACTIVE DIRECTORY FUNCTIONS ---
2  def create_ad_user(username, first_name, last_name, email, role="Employee"):
3      conn = get_ad_connection()
4      if not conn:
5          return False
6
7      try:
8          # 1) Role-based OU placement
9          ou_map = {
10             "Developer": "OU=Developers,OU=innovatech,DC=innovatech,DC=local",
11             "Admin": "OU=Admins,OU=innovatech,DC=innovatech,DC=local",
12             "Employee": "OU=Employees,OU=innovatech,DC=innovatech,DC=local",
13         }
14         target_ou = ou_map.get(role, ou_map["Employee"])
15         user_dn = f"CN={first_name} {last_name},{target_ou}"
16
17         # 2) Create user disabled initially
18         attributes = {
19             "objectClass": ["top", "person", "organizationalPerson", "user"],
20             "sAMAccountName": username,
21             "userPrincipalName": f"{username}@innovatech.local",
22             "givenName": first_name,
23             "sn": last_name,
24             "displayName": f"{first_name} {last_name}",
25             "mail": email,
26             "userAccountControl": 514, # disabled
27         }
28
29         if not conn.add(user_dn, attributes=attributes):
30             conn.unbind()
31             return False
32
33         # 3) Set password via AWS Directory Service with retry (eventual consistency)
34         password = "TempPass123!"
35         for attempt in range(15):
36             try:
37                 ds_client.reset_user_password(
38                     DirectoryId=DIRECTORY_ID,
39                     UserName=username,
40                     NewPassword=password,
41                 )
42                 break
43             except ClientError as e:
44                 code = e.response["Error"]["Code"]
45                 if code == "UserDoesNotExistException" and attempt < 14:
46                     time.sleep(3)
47                     continue
48                 conn.unbind()
49                 return False
50
51         # 4) Enable account after password replication
52         time.sleep(5)
53         conn.modify(user_dn, {"userAccountControl": [(MODIFY_REPLACE, [512])])})
54
55         # 5) Add to role-based AD security group (RBAC)
56         group_base_dn = "OU=innovatech,DC=innovatech,DC=local"
57         group_map = {
58             "Developer": f"CN=DevelopersGroup,{group_base_dn}",
59             "Admin": f"CN=AdminsGroup,{group_base_dn}",
60             "Employee": f"CN=EmployeesGroup,{group_base_dn}",
61         }
62         target_group = group_map.get(role, group_map["Employee"])
63         conn.modify(target_group, {"member": [(MODIFY_ADD, [user_dn])])})
64
65         conn.unbind()
66         return True
67     except Exception:
68         return False
69
70
```

AD provisioning via LDAP with AWS DS password reset and RBAC group assignment

The backend creates an AD user in a role-based OU, sets the initial password via AWS Directory Service with retry logic to handle propagation delays, enables the account, and assigns the user to the correct AD security group.

4. Deploy Workspace:

```
1 # Step 3: Provision Workspace (if configured)
2 workspace_status = "not_configured"
3
4 if DIRECTORY_ID and BUNDLE_ID:
5     print(f"Provisioning Workspace for {username}...")
6     ad_success = create_ad_user(username, first_name, last_name, email, role)
7     if ad_success:
8         print(f"AD user created: {username}")
9         workspace_success = provision_workspace(username, role)
10        workspace_status = "provisioning" if workspace_success else "failed"
11        print(f"Workspace status: {workspace_status}")
12    else:
13        print(f"AD user creation failed")
14        workspace_status = "ad_failed"
15
16 print(f"User creation completed: {email}")
17
18 return jsonify({
19     'message': f'User {first_name} {last_name} created successfully',
20     'employee_id': employee_id,
21     'cognito': 'created',
22     'database': 'created',
23     'workspace': workspace_status
24 }, 201)
```

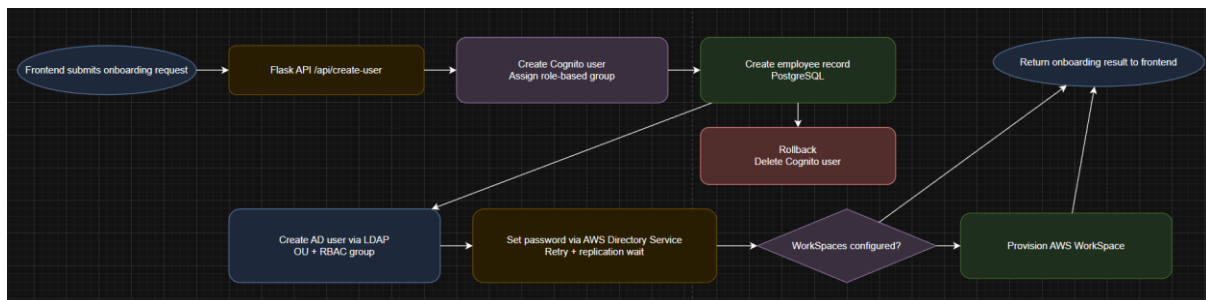
Conditional AWS Workspace provisioning with AD dependency

This snippet shows how the backend provisions an AWS Workspace only after successful Active Directory user creation. The provisioning logic is encapsulated in a dedicated function and returns a clear status to the frontend.

5. Software provisioning via Group Policy (GPO)

After the WorkSpace is domain-joined, software and configuration are applied through Active Directory **Group Policy**. This keeps software deployment centralized and consistent per role, without adding extra AWS Systems Manager automation in the backend. The backend's responsibility ends after creating the AD user, assigning the correct AD group, and provisioning the WorkSpace. The role-based policies are then enforced through GPO.

Role-based software and settings are applied by placing users in the correct AD security group, which is linked to the relevant GPO.



Complete onboarding workflow implemented in the consolidated Flask backend

This diagram shows the end-to-end onboarding process, including Cognito identity creation, database registration, Active Directory provisioning with password handling, conditional AWS WorkSpace deployment, and rollback handling to prevent inconsistent states.

3.5.3 Offboarding Workflow (terminate-user)

Secure Offboarding Process:

1. **Validate request and check user exists in DB (get first name, last name, workspace_id):**

```
697 @app.route('/api/terminate-user', methods=['POST'])
698 @admin_required
699 def terminate_user():
700     """Offboard an employee"""
701     try:
702         data = request.json
703         email = data.get('email')
704
705         if not email:
706             return jsonify({'error': 'Email is required'}), 400
707
708         print(f"Starting offboarding for: {email}")
709
710         conn = get_db()
711         cur = conn.cursor()
712
713         cur.execute("SELECT first_name, last_name, workspace_id FROM employees WHERE email = %s", (email,))
714         user = cur.fetchone()
715
716         if not user:
717             return jsonify({'error': 'User not found'}), 404
718
719         first_name, last_name, workspace_id = user
```

Offboarding validation and employee lookup (database check)

This snippet validates the offboarding request by requiring an email address and then queries the employee database to retrieve the user's identity and WorkSpace ID. If the employee does not exist, the API returns a 404 response and stops the offboarding process.

2. **Disable Cognito account:**

The backend disables the AD account using LDAP by resolving the user's Distinguished Name across the OU structure and updating the account control flags. This ensures access is revoked while preserving the account for auditing.

```
721 # Disable in Cognito
722 try:
723     cognito.admin_disable_user(UserPoolId=USER_POOL_ID, Username=email)
724     print(f"Cognito user disabled")
725 except Exception as e:
726     print(f"Cognito issue: {e}")
```

Disable Cognito user access

This snippet disables the user in Amazon Cognito to immediately revoke portal access. The operation is wrapped in a try/except block so offboarding can continue even if Cognito returns an error.

3. Disable the AD user account:

```
728 # Disable in Active Directory
729 disable_ad_user(first_name, last_name)
```

Trigger AD offboarding from the API endpoint

This snippet shows that the offboarding endpoint delegates Active Directory actions to a dedicated function, keeping the API logic clean and reusable.

```
361 def disable_ad_user(first_name, last_name):
362     """Disable user in Active Directory (Fixed to search sub-OU's)"""
363     print(f"Disabling AD user: {first_name} {last_name}")
364
365     conn = get_ad_connection()
366     if not conn:
367         return False
368
369     try:
370         # 1. Find the user's correct DN (Distinguished Name)
371         # We search the entire innovatech OU subtree
372         search_base = 'OU=innovatech,DC=innovatech,DC=local'
373         search_filter = f'(&(objectClass=user)(cn={first_name} {last_name}))'
374
375         conn.search(search_base, search_filter, attributes=['distinguishedName'])
376
377         if not conn.entries:
378             print(f"User not found in AD: {first_name} {last_name}")
379             conn.unbind()
380             return False
381
382         user_dn = conn.entries[0].distinguishedName.value
383         print(f"✓ Found user at: {user_dn}")
384
385         # 2. Disable the account
386         # 514 = Normal Account (512) + Disabled (2)
387         success = conn.modify(user_dn, {'userAccountControl': [(MODIFY_REPLACE, [514])])})
388
389         if success:
390             print(f"AD account disabled")
391         else:
392             print(f"Failed to disable: {conn.result}")
393
394         conn.unbind()
395         return success
396     except Exception as e:
397         print(f"Error disabling AD user: {e}")
398         return False
```

Disable Active Directory account during offboarding (LDAP)

This function locates the user across all Active Directory sub-OUs, resolves the Distinguished Name, and disables the account by updating userAccountControl. The account is preserved for audit purposes while access is fully revoked.

4. Terminate Workspace:

If the employee has a `workspace_id` stored in the database, the backend looks up the user's Workspace in the configured directory and sends a termination request. The API returns a clear status (`terminating`, `failed`, or `none`) so the frontend can show the result.

```
731 # Terminate Workspace
732 workspace_status = "none"
733 if workspace_id:
734     try:
735         username = email.split('@')[0]
736         ws_resp = workspaces.describe_workspaces(UserName=username, DirectoryId=DIRECTORY_ID)
737
738         if ws_resp['Workspaces']:
739             ws_id = ws_resp['Workspaces'][0]['WorkspaceId']
740             print(f"Terminating Workspace: {ws_id}")
741             workspaces.terminate_workspaces(TerminateWorkspaceRequests=[{'WorkspaceId': ws_id}])
742             workspace_status = "terminating"
743     except Exception as e:
744         print(f"Workspace error: {e}")
745         workspace_status = "failed"
```

Workspace termination during offboarding

This snippet terminates the employee's AWS Workspace when a `workspace_id` exists in the database. It retrieves the Workspace using `describe_workspaces` and submits a termination request, while capturing failures and returning a status to the frontend.

5. Update database and return result:

```
747     # Update database
748     cur.execute("""
749         UPDATE employees
750         SET status = 'terminated',
751             termination_date = CURRENT_DATE
752         WHERE email = %s
753     """, (email,))
754
755     conn.commit()
756     cur.close()
757     conn.close()
758
759     return jsonify({
760         'message': f'User {email} offboarded successfully',
761         'workspace': workspace_status
762     })
763
764     except Exception as e:
765         print(f"Error in terminate-user: {e}")
766         return jsonify({'error': str(e)}), 500
```

Security Considerations:

- Immediate access revocation via Cognito disable
- Audit trail maintained in database
- Workspace termination configurable (30-day grace period standard)
- AD account disabled but not deleted (compliance requirement)

3.6 RBAC & Access Control

3.6.1 IRSA for Employee Portal pod permissions

ServiceAccount with IRSA annotation:

```
8  ✓ apiVersion: v1
9    kind: ServiceAccount
10  ✓ metadata:
11    name: employee-portal-sa
12    namespace: employee-services
13  ✓ annotations:
14    eks.amazonaws.com/role-arn: arn:aws:iam::098347675427:role/cs1-ma-nca-employee-portal-sa
```

ServiceAccount annotation for IRSA (pod-to-IAM role link)

This ServiceAccount is annotated with an IAM role ARN. When the Employee Portal pod uses this ServiceAccount, it can request temporary AWS credentials for that role. This avoids storing AWS access keys inside the container or Kubernetes secrets.

```
16  apiVersion: apps/v1
17  kind: Deployment
18  metadata:
19    name: employee-portal
20    namespace: employee-services
21  spec:
22    replicas: 2
23    selector:
24      matchLabels:
25        app: employee-portal
26    template:
27      metadata:
28        labels:
29          app: employee-portal
30      spec:
31        serviceAccountName: employee-portal-sa
32        containers:
```

Deployment binds the Employee Portal pod to the IRSA ServiceAccount

This snippet shows that the Employee Portal Deployment explicitly uses **serviceAccountName: employee-portal-sa**. This is required for IRSA, because the pod only receives AWS permissions when it runs under the annotated ServiceAccount.

```

536 data "aws_iam_policy_document" "employee_portal_assume" {
537   statement {
538     effect = "Allow"
539     actions = ["sts:AssumeRoleWithWebIdentity"]
540
541     principals {
542       type       = "Federated"
543       identifiers = [aws_iam_openid_connect_provider.eks.arn]
544     }
545
546     condition {
547       test       = "StringEquals"
548       variable   = "${replace(aws_iam_openid_connect_provider.eks.url, "https://", "")}:sub"
549       values     = ["system:serviceaccount:employee-services:employee-portal-sa"]
550     }
551
552     condition {
553       test       = "StringEquals"
554       variable   = "${replace(aws_iam_openid_connect_provider.eks.url, "https://", "")}:aud"
555       values     = ["sts.amazonaws.com"]
556     }
557   }
558 }
559
560 resource "aws_iam_role" "employee_portal_sa" {
561   name = "${var.project_name}-employee-portal-sa"
562   assume_role_policy = data.aws_iam_policy_document.employee_portal_assume.json
563
564   tags = merge(
565     var.cs3_tags,
566     {
567       Name          = "${var.project_name}-employee-portal-sa-role"
568       Environment = var.environment
569     }
570   )
571 }

```

IRSA trust policy restricting role assumption to one ServiceAccount

This Terraform snippet defines the IAM role trust policy for ***sts:AssumeRoleWithWebIdentity***. It restricts role assumption to the exact Kubernetes identity ***system:serviceaccount:employee-services:employee-portal-sa***, preventing other pods or namespaces from using the same IAM role.

```

573  ✓ resource "aws_iam_role_policy" "employee_portal_cognito" {
574      name = "cognito-access"
575      role = aws_iam_role.employee_portal_sa.id
576
577  ✓  policy = jsonencode({
578      Version = "2012-10-17"
579  ✓      Statement = [
580  ✓          {
581              Effect = "Allow"
582  ✓          Action = [
583              "cognito-idp:InitiateAuth",
584              "cognito-idp:AdminInitiateAuth",
585              "cognito-idp:AdminCreateUser",
586              "cognito-idp:AdminDisableUser",
587              "cognito-idp:AdminGetUser",
588              "cognito-idp:AdminAddUserToGroup",
589              "cognito-idp:AdminRespondToAuthChallenge",
590              "cognito-idp:AdminSetUserPassword",
591              "cognito-idp:ListUsers",
592              "cognito-idp:GetUser"
593          ]
594          Resource = aws_cognito_user_pool.employees.arn
595      },
596  ✓      {
597          Effect = "Allow"
598  ✓          Action = [
599              "ds:DescribeDirectories",
600              "workspaces:DescribeWorkspaces",
601              "workspaces:CreateWorkspaces",
602              "workspaces:TerminateWorkspaces",
603              "workspaces:CreateTags",
604              "workspaces:DescribeTags"
605          ]
606          Resource = "*"
607      }
608  ]
609  })
610 }

```

Least-privilege IAM permissions for onboarding and offboarding actions on Cognito

This policy grants only the AWS actions required by the backend API for identity lifecycle management and WorkSpaces operations, such as creating and disabling Cognito users, assigning Cognito groups, and provisioning or terminating WorkSpaces.

```

635 resource "aws_iam_role_policy" "employee_portal_directory_service" {
636   name = "directory-service-access"
637   role = aws_iam_role.employee_portal_sa.id
638
639   policy = jsonencode({
640     Version = "2012-10-17"
641     Statement = [
642       {
643         Effect = "Allow"
644         Action = [
645           "ds:ResetUserPassword"
646         ]
647         Resource = "arn:aws:ds:${var.aws_region}:${data.aws_caller_identity.current.account_id}:directory/${aws_directory_service_directory.innovatech_ad.id}"
648       }
649     ]
650   })
651 }

```

Directory Service permission for AD password initialization

This policy allows **ds:ResetUserPassword** on the specific AWS Directory Service directory ARN. The backend uses this during onboarding to set the initial AD password after creating the AD user via LDAP.

3.6.2 Application-Level Authorization

Role-Based Dashboard Access:

Role	Capabilities	UI Elements
admin	View all employees, approve access requests, system configuration	Full dashboard, user management, approvals, logs
developer	Submit access requests, view own info, deploy dev WorkSpaces	Personal dashboard, request form, dev software selection
employee	View own info, basic access requests	Personal info, limited request form

Backend Authorization Check:

```

225 # --- SECURITY: TOKEN VERIFICATION ---
226 def verify_token(token):
227     """Verify Cognito JWT token"""
228     try:
229         keys_url = f"{COGNITO_ISSUER}/.well-known/jwks.json"
230         keys = requests.get(keys_url).json()['keys']
231         header = jwt.get_unverified_header(token)
232         key = next((k for k in keys if k['kid'] == header['kid']), None)
233         if not key:
234             return None
235         return jwt.decode(token, key, algorithms=['RS256'], audience=COGNITO_CLIENT_ID, issuer=COGNITO_ISSUER)
236     except Exception as e:
237         print(f"Token verification error: {e}")
238     return None

```

Cognito JWT verification in the backend

This function validates the Cognito JWT by fetching the JWKS keys, selecting the correct key using the token header, then decoding the token with the expected issuer and client ID. It returns the token claims which are later used for authorization.

```

240 ✓ def admin_required(f):
241     """Decorator to require admin group membership"""
242     @wraps(f)
243     def decorated_function(*args, **kwargs):
244         auth_header = request.headers.get('Authorization')
245         if not auth_header:
246             return jsonify({'error': 'No authorization token provided'}), 401
247
248         token = auth_header.split(" ")[1] if " " in auth_header else auth_header
249         claims = verify_token(token)
250
251         if not claims:
252             return jsonify({'error': 'Invalid or expired token'}), 401
253
254         groups = claims.get('cognito:groups', [])
255         if 'admins' not in groups:
256             return jsonify({'error': 'Access Denied: Administrator privileges required'}), 403
257
258         return f(*args, **kwargs)
259     return decorated_function

```

Application-level authorization using Cognito groups

This decorator extracts the bearer token from the request header, verifies the token, reads the user's **cognito:groups** claim, and blocks access with HTTP 403 if the user is not in the **admins** group. This enforces backend authorization even if someone bypasses the frontend UI.

```

581 @app.route('/api/create-user', methods=['POST'])
582 @admin_required
583 def create_user():
584     """Create new employee (Admin only)"""
585     try:

```

Admin-only API endpoint protected by the authorization decorator

This endpoint is restricted with **@admin_required**, meaning only users in the admin Cognito group can call it. This is the practical enforcement point of role-based API access control.

```
697 @app.route('/api/terminate-user', methods=['POST'])
698 @admin_required
699 def terminate_user():
700     """Offboard an employee"""
701     try:
```

Admin-only offboarding endpoint protected by role-based authorization

This endpoint is also restricted to administrators using the *@admin_required* decorator. As explained before, this function performs security-critical offboarding actions such as disabling Cognito access, disabling the Active Directory account, optionally terminating the WorkSpace, and updating the employee status in the database.

Role-based dashboard views (Admin vs Employee)

Logged in as: admin@innovatech.local

ADMIN

Logout

Employee Directory

Refresh

ID	Name	Email	Department	Position	Status	Actions
1	John Doe	john.doe@innovatech.local	Engineering	Senior Developer	active	Term
2	Jane Smith	jane.smith@innovatech.local	Human Resources	HR Manager	active	Term
3	Bob Johnson	bob.johnson@innovatech.local	Engineering	DevOps Engineer	active	Term
4	Alice Williams	alice.williams@innovatech.local	Finance	Financial Analyst	active	Term
5	Admin User	admin@innovatech.local	IT	Administrator	active	Term
11	Sonny 3	sonny.3@innovatech.local	Engineering	Developer	active	Term
26	Test 16	test.16@innovatech.local	Engineering	Developer	active	Term
27	Test 17	test.17@innovatech.local	Engineering	Developer	active	Term
29	Test 19	test.19@innovatech.local	Engineering	Developer	active	Term

Create New Employee

ADMIN ONLY

First Name:

John

Last Name:

Doe

Email:

john.doe@innovatech.local

Department:

Engineering

60 | Page

Employee Directory

Refresh

ID	Name	Email	Department	Position	Status	Actions
1	John Doe	john.doe@innovatech.local	Engineering	Senior Developer	active	
2	Jane Smith	jane.smith@innovatech.local	Human Resources	HR Manager	active	
3	Bob Johnson	bob.johnson@innovatech.local	Engineering	DevOps Engineer	active	
4	Alice Williams	alice.williams@innovatech.local	Finance	Financial Analyst	active	
5	Admin User	admin@innovatech.local	IT	Administrator	active	
11	Sonny 3	sonny.3@innovatech.local	Engineering	Developer	active	
26	Test 16	test.16@innovatech.local	Engineering	Developer	active	
27	Test 17	test.17@innovatech.local	Engineering	Developer	active	
29	Test 19	test.19@innovatech.local	Engineering	Developer	active	

Access Requests

Refresh

Request ID	Employee	Resource	Type	Status	Date
------------	----------	----------	------	--------	------

These screenshots show how the frontend dynamically renders different dashboard views based on the authenticated user's role. An administrator sees management features such as employee termination actions and the "Create New Employee" section, while a regular employee is limited to viewing the employee directory and their own information. All security-critical authorization is enforced by the backend API, with the frontend role-based rendering improving usability and preventing accidental access to restricted functionality.

4. PHASE 3: KUBERNETES DEPLOYMENT & PORTAL

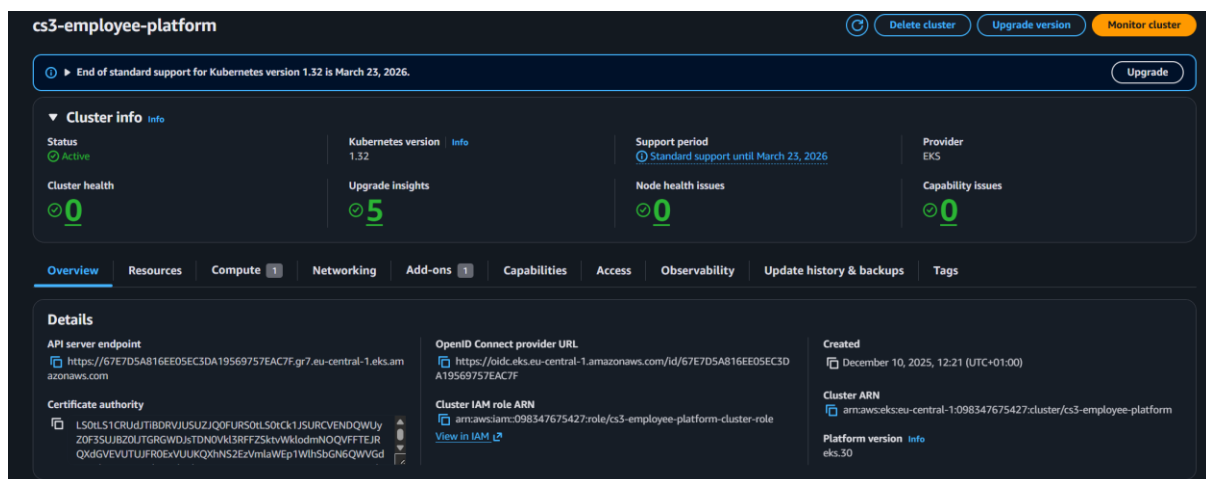
4.1 Phase goal

In this phase I deployed the Employee Portal as a 3-tier application on Amazon EKS. The portal runs as two containerized components (frontend and backend API) with a PostgreSQL database for persistent storage. This phase focuses on packaging, Kubernetes manifests, persistence, deployment verification, and logging.

4.2 EKS Cluster Configuration

4.2.1 Cluster Setup

I provisioned the EKS cluster and its IAM roles with Terraform. The cluster runs in private subnets and is used as the platform for the portal workloads. The main files are `terraform/cs3-eks.tf` and `terraform/cs3-eks-iam.tf`.



EKS cluster overview in AWS Console

This EKS cluster provides the managed Kubernetes platform on which the Employee Portal workloads are deployed.

4.3 Containerization

4.3.1 Backend Dockerfile

The backend is packaged as a Docker container. It exposes the API endpoints and contains the automation logic for Cognito, database, AD, and WorkSpaces.

```
1 FROM python:3.9-slim-bullseye
2
3 WORKDIR /app
4
5 RUN apt-get update && apt-get install -y postgresql-client && rm -rf /var/lib/apt/lists/*
6
7 COPY requirements.txt .
8
9 RUN pip install --no-cache-dir -r requirements.txt
10
11 COPY app.py .
12
13 EXPOSE 5000
14 EXPOSE 5140/udp
15 EXPOSE 8080/tcp
16
17 ENV PYTHONUNBUFFERED=1
18
19 CMD ["python", "app.py"]
```

Backend Dockerfile for the Flask API

This Dockerfile packages the Flask backend and its dependencies into a container image that can be deployed consistently on EKS.

4.3.2 Frontend Dockerfile

The frontend is also packaged as a Docker container so it can be deployed and scaled the same way as the backend. This supports the requirement to containerize both frontend and backend.

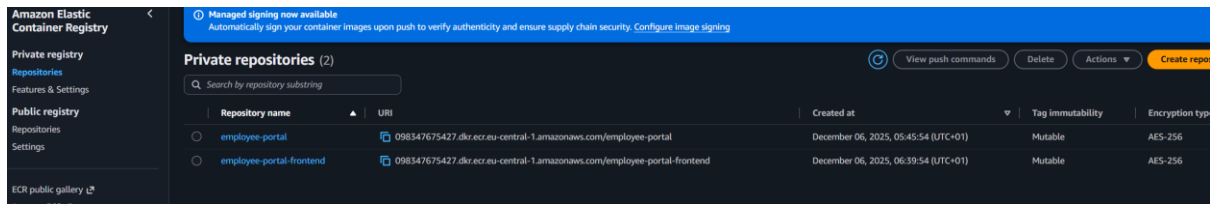
```
1 FROM nginx:alpine (last pushed 19 hours ago)
2
3 # Copy nginx configuration
4 COPY nginx.conf /etc/nginx/conf.d/default.conf
5
6 # Copy frontend HTML
7 COPY index.html /usr/share/nginx/html/
8
9 # Create custom error page
10 RUN echo ' <html><body><h1>Service Temporarily Unavailable</h1><p>The backend service is starting up. Please try again in a moment.</p></body></html>' > /usr/share/nginx/html/50x.html
11
12 EXPOSE 80
13
14 # Health check
15 HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
16   CMD wget --quiet --tries=1 --spider http://localhost/health || exit 1
17
18 # Run nginx
19 CMD ["nginx", "-g", "daemon off;"]
```

Frontend Dockerfile for the portal UI (Nginx)

The frontend is packaged as a lightweight Nginx container that serves the static UI and proxies API requests to the backend service.

4.3.3 Container Registry (ECR)

After building the images locally, I pushed both images to Amazon ECR so Kubernetes can pull them during deployment.



Backend and frontend container images stored in Amazon ECR

Both portal components are built as container images and stored in ECR for deployment to EKS.

4.4 Kubernetes Deployment

Application manifests The portal is deployed using Kubernetes manifests stored in the repository under /kubernetes. I applied them with kubectl during deployment.

```
1 apiVersion: v1
2 kind: Namespace
3 metadata:
4   name: employee-services
5   labels:
6     name: employee-services
7 ---
8 apiVersion: v1
9 kind: ServiceAccount
10 metadata:
11   name: employee-portal-sa
12 namespace: employee-services
13 annotations:
14   eks.amazonaws.com/role-arn: arn:aws:iam::098347675427:role/csi-ma-nca-employee-portal-sa
15 ---
16 apiVersion: apps/v1
17 kind: Deployment
18 metadata:
19   name: employee-portal
20 namespace: employee-services
21 spec:
22   replicas: 2
23   selector:
24     matchLabels:
25       app: employee-portal
26   template:
27     metadata:
28       labels:
29         app: employee-portal
30     spec:
31       serviceAccountName: employee-portal-sa
32       containers:
33         - name: portal
34           image: 098347675427.dkr.ecr.eu-central-1.amazonaws.com/employee-portal:latest
35           imagePullPolicy: Always
36           ports:
37             - containerPort: 5000
38             name: http
39           env:
40             - name: PYTHONUNBUFFERED
41               value: "1"
42             # Database Configuration
43             - name: DB_HOST
44               value: postgres:employee-services.svc.cluster.local
45             - name: DB_NAME
46               value: "employees"
47             - name: DB_USER
48               value: "admin"
49             - name: DB_PASSWORD
50               valueFrom:
51                 secretKeyRef:
52                   name: employee-db-credentials
53                   key: password
54             # AWS & Cognito Configuration (AUTO-SYNCD FROM TERRAFORM)
55             - name: AWS_REGION
56               valueFrom:
57                 configMapKeyRef:
58                   name: cognito-config
59                   key: region
60             - name: COGNITO_USER_POOL_ID
61               valueFrom:
62                 configMapKeyRef:
63                   name: cognito-config
64                   key: user_pool_id
65             - name: COGNITO_CLIENT_ID
66               valueFrom:
67                 configMapKeyRef:
68                   name: cognito-config
69                   key: client_id
70             # Active Directory Configuration
71             - name: AD_HOST
72               valueFrom:
73                 configMapKeyRef:
74                   name: ad-config
75                   key: domain
76             - name: AD_DNS_IP
77               value: "10.0.41.73"
78             # - name: AD_USER
79             #   value: "svc-automation"
80             # - name: AD_PASSWORD
81             #   value: "ServiceAuto123!@#"
82             - name: AD_DIRECTORY_ID
83               valueFrom:
84                 configMapKeyRef:
85                   name: ad-config
86                   key: directory_id
87             - name: AD_BUNDLE_ID
88               value: "wsb-93xk71ss4"
89             resources:
90               requests:
91                 memory: "256Mi"
92                 cpu: "250m"
93               limits:
94                 memory: "512Mi"
95                 cpu: "500m"
96             livenessProbe:
97               httpGet:
98                 path: /api/health
99                 port: 5000
100               initialDelaySeconds: 30
101               periodSeconds: 10
102             readinessProbe:
103               httpGet:
104                 path: /api/health
105                 port: 5000
106               initialDelaySeconds: 10
107               periodSeconds: 5
108 ---
109 apiVersion: v1
110 kind: Service
111 metadata:
112   name: employee-portal
113 namespace: employee-services
114 spec:
115   type: ClusterIP
116   selector:
117     app: employee-portal
118   ports:
119     - port: 5000
120     targetPort: 5000
121     protocol: TCP
```

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: employee-portal-frontend
5 namespace: employee-services
6 labels:
7   app: employee-portal-frontend
8   tier: frontend
9 spec:
10   replicas: 2
11   selector:
12     matchLabels:
13       app: employee-portal-frontend
14   template:
15     metadata:
16       labels:
17         app: employee-portal-frontend
18         tier: frontend
19   spec:
20     containers:
21       - name: frontend
22         image: 098347675427.dkr.ecr.eu-central-1.amazonaws.com/employee-portal-frontend:latest
23         imagePullPolicy: Always
24         ports:
25           - containerPort: 80
26           name: http
27         resources:
28           requests:
29             memory: "64Mi"
30             cpu: "100m"
31           limits:
32             memory: "128Mi"
33             cpu: "200m"
34         livenessProbe:
35           httpGet:
36             path: /health
37             port: 80
38           initialDelaySeconds: 10
39           periodSeconds: 10
40           timeoutSeconds: 3
41           failureThreshold: 3
42         readinessProbe:
43           httpGet:
44             path: /health
45             port: 80
46           initialDelaySeconds: 5
47           periodSeconds: 5
48           timeoutSeconds: 3
49           failureThreshold: 3
50 ---
51 apiVersion: v1
52 kind: Service
53 metadata:
54   name: employee-portal-frontend
55 namespace: employee-services
56 labels:
57   app: employee-portal-frontend
58   tier: frontend
59 spec:
60   type: LoadBalancer
61   selector:
62     app: employee-portal-frontend
63   ports:
64     - port: 80
65     targetPort: 80
66     protocol: TCP
67     name: http
68   sessionAffinity: None
```

Kubernetes manifests for deploying the Employee Portal

These manifests define how the portal components are deployed, scaled, and exposed inside the Kubernetes cluster.

4.4.1 Deployment Verification

After deployment I verified that pods and services were running correctly. The main commands are shown in the deployment instructions.

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure\employee-portal\frontend> kubectl get all -n employee-services
```

NAME	READY	STATUS	RESTARTS	AGE
pod/employee-portal-5474c9984c-9btcc	1/1	Running	0	3h20m
pod/employee-portal-5474c9984c-tszbm	1/1	Running	0	2d5h
pod/employee-portal-frontend-d59fbbffd-8czdl	1/1	Running	0	56m
pod/employee-portal-frontend-d59fbbffd-wffmg	1/1	Running	0	56m
pod/postgres-0	1/1	Running	0	3h20m

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/employee-portal	ClusterIP	172.20.166.188	<none>	5000/TCP	4d3h
service/employee-portal-frontend	LoadBalancer	172.20.141.141	ac71e2e553be340b9aeec43b5e65f2ea-396148643.eu-central-1.elb.amazonaws.com	80:30698/TCP	4d3h
service/postgres	ClusterIP	None	<none>	5432/TCP	4d3h

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/employee-portal	2/2	2	2	4d3h
deployment.apps/employee-portal-frontend	2/2	2	2	4d3h

NAME	READY	AGE
statefulset.apps/postgres	1/1	4d3h

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure\employee-portal\frontend>
```

Running Kubernetes resources for the Employee Portal

This output confirms that all portal components are successfully deployed and running in the cluster.

4.5 Database Persistence

The portal requires a relational database for employee records and access requests. Because Amazon RDS was blocked by AWS Academy policies, I deployed PostgreSQL inside the cluster as a StatefulSet with an EBS backed persistent volume claim.

```
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: postgres
5    namespace: employee-services
6  spec:
7    serviceName: postgres
8    replicas: 1
9    selector:
10     matchLabels:
11       app: postgres
12    template:
13     metadata:
14       labels:
15         app: postgres
16     spec:
17       containers:
18       - name: postgres
19         image: postgres:15
20         ports:
21         - containerPort: 5432
22         env:
23         - name: POSTGRES_DB
24           value: "employees"
25         - name: POSTGRES_USER
26           value: "admin"
27         - name: POSTGRES_PASSWORD
28           value: "SecurePass123!" # TODO: Move to K8s Secret
29         volumeMounts:
30         - name: postgres-storage
31           mountPath: /var/lib/postgresql/data
32           subPath: postgres
33     volumeClaimTemplates:
34     - metadata:
35       name: postgres-storage
36     spec:
37       accessModes: [ "ReadWriteOnce" ]
38       storageClassName: gp2
39       resources:
40         requests:
41           storage: 10Gi
42 ---
43 apiVersion: v1
44 kind: Service
45 metadata:
46   name: postgres
47   namespace: employee-services
48 spec:
49   selector:
50     app: postgres
51   ports:
52   - port: 5432
53     targetPort: 5432
54   clusterIP: None
```

PostgreSQL StatefulSet with persistent storage

PostgreSQL is deployed as a StatefulSet with an EBS-backed PVC to ensure database persistence across pod restarts.

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure\employee-portal\frontend> kubectl get pvc -n employee-services
NAME                                STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
postgres-storage-postgres-0        Bound    pvc-f6496ac7-4dfb-4a8c-b308-08f29a28fdc6   10Gi       RWO            gp2            <unset>                 4d3h
```

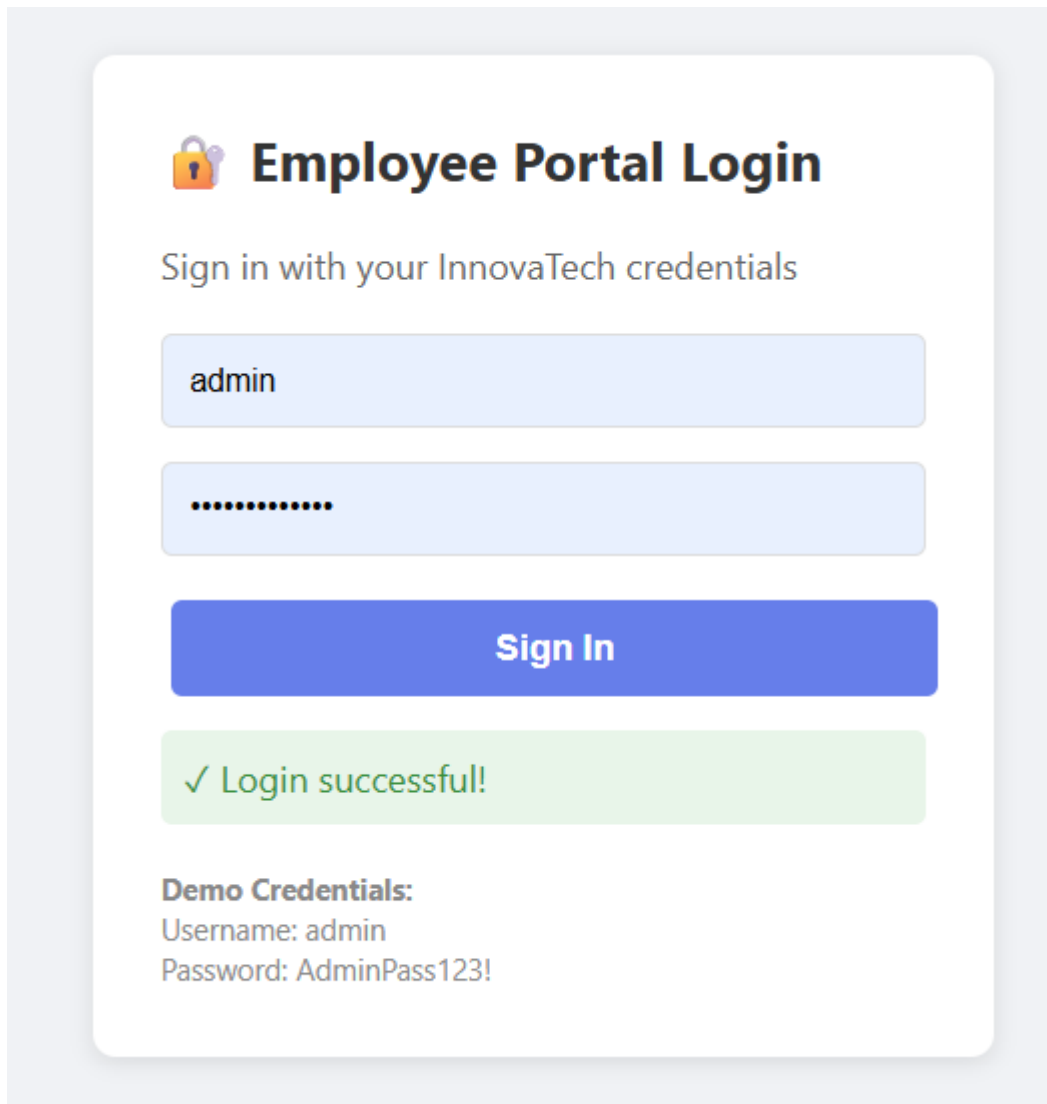
Persistent Volume Claim bound to PostgreSQL

The bound PVC confirms that database data is stored persistently outside the pod lifecycle.

Why StatefulSet Instead of Deployment:

1. **Stable Network Identity:** postgres-0.postgres.employee-services.svc.cluster.local
2. **Persistent Storage:** PVC survives pod restarts
3. **Ordered Deployment:** Guarantees single database instance at a time
4. **Data Preservation:** Deleting pod doesn't delete data

4.6 Portal Accessibility



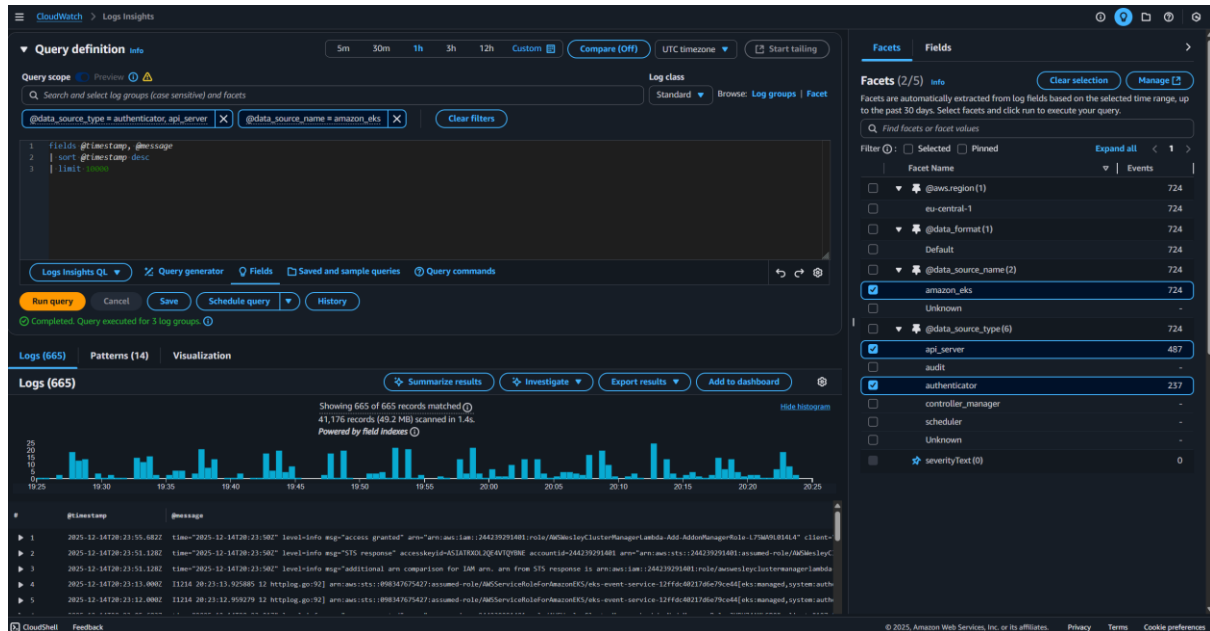
The screenshot displays the 'Employee Portal Login' interface. At the top, there is a lock icon followed by the title 'Employee Portal Login'. Below the title, a message reads 'Sign in with your InnovaTech credentials'. There are two input fields: the first contains the username 'admin', and the second contains a masked password represented by ten dots. A blue 'Sign In' button is positioned below the password field. A green success message '✓ Login successful!' is displayed in a light green box. At the bottom, a section titled 'Demo Credentials:' provides the test login information: 'Username: admin' and 'Password: AdminPass123!'.

Employee Portal accessible after Kubernetes deployment

This screenshot confirms that the frontend service is reachable and the portal is operational.

4.7 Logging

Phase 3 requires log parsing and aggregation for the portal and Kubernetes services. I used CloudWatch Logs to query onboarding and error events, and I documented example queries for verification.



Application logs queried using CloudWatch Logs Insights

CloudWatch Logs Insights is used to verify application behavior and troubleshoot runtime issues.

4.8 CI/CD Deployment Process

The repository includes a GitHub Actions workflow to automate deployment steps. In my implementation notes I also documented the manual rollout restart approach using `kubectl` for controlled updates.

```
1 ▶ Run aws eks update-kubeconfig --name cs3-employee-platform --region eu-central-1
14 Added new context arn:aws:eks:eu-central-1:098347675427:cluster/cs3-employee-platform to /home/runner/.kube/config
Kubernetes control plane is running at https://67E7D5A816EE05EC3DA19569757EAC7F.ap7.eu-central-1.eks.amazonaws.com
16 CoreDNS is running at https://67E7D5A816EE05EC3DA19569757EAC7F.ap7.eu-central-1.eks.amazonaws.com/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
17
18 To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
19 NAME STATUS ROLES AGE VERSION
20 ip-10-0-41-9.eu-central-1.compute.internal Ready <none> 3h1m v1.32.9-eks-ecaa3a6
21 ip-10-0-41-99.eu-central-1.compute.internal Ready <none> 39h v1.32.9-eks-ecaa3a6

▼ Deploy AWS Load Balancer Controller

1 ▶ Run echo "Installing Helm..."
40 Installing Helm...
41 % Total % Received % Xferd Average Speed Time Time Time Current
42 Dload Upload Total Spent Left Speed
43
44 0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0
45 100 11929 100 11929 0 0 288k 0 --:--:-- --:--:-- --:--:-- 291k
46 Helm v3.19.3 is available. Changing from version v3.19.2.
47 Downloading https://get.helm.sh/helm-v3.19.3-linux-amd64.tar.gz
48 Verifying checksum... Done.
49 Preparing to install helm into /usr/local/bin
50 helm installed into /usr/local/bin/helm
51 Cleaning up old ALB controller if exists...
52 deployment.apps "aws-load-balancer-controller" deleted from kube-system namespace
53 Installing ALB Controller CRDs...
54 customresourcedefinition.apiextensions.k8s.io/albtargetcontrolconfigs.elbv2.k8s.aws unchanged
55 customresourcedefinition.apiextensions.k8s.io/ingressclassparams.elbv2.k8s.aws unchanged
56 customresourcedefinition.apiextensions.k8s.io/targetgroupbindings.elbv2.k8s.aws unchanged
57 Waiting for CRDs to be ready...
58 Installing AWS Load Balancer Controller via Helm...
59 "eks" has been added to your repositories
60 Hang tight while we grab the latest from your chart repositories...
61 ...Successfully got an update from the "eks" chart repository
62 Update Complete. 🎉Happy Helming!🎉
63 Release "aws-load-balancer-controller" has been upgraded. Happy Helming!
64 NAME: aws-load-balancer-controller
65 LAST DEPLOYED: Fri Dec 12 10:10:54 2025
66 NAMESPACE: kube-system
67 STATUS: deployed
68 REVISION: 12
69 TEST SUITE: None
70 NOTES:
71 AWS Load Balancer controller installed!
72 AWS Load Balancer Controller deployed successfully!

▼ Deploy Employee Portal

1 ▶ Run echo "Deploying Employee Portal..."
19 Deploying Employee Portal...
20 namespace/employee-services unchanged
21 serviceaccount/employee-portal-sa unchanged
22 deployment.apps/employee-portal unchanged
23 service/employee-portal unchanged
24 deployment.apps/employee-portal-frontend unchanged
25 service/employee-portal-frontend unchanged
26 Waiting for deployments to be ready...
27 deployment.apps/employee-portal condition met
28 Employee Portal deployed successfully!

▼ Post Configure AWS Credentials
```

Successful CI/CD pipeline run deploying the portal

The CI/CD pipeline automates Kubernetes rollouts to deploy updates safely.

5. PHASE 4: VALIDATION, MONITORING & OPERATION

After completing the implementation of the employee portal and its supporting infrastructure, validation tests were performed to confirm that the system functions correctly under operational conditions and that key platform guarantees such as availability, security, and observability are met.

5.1 Functional Validation

The core onboarding and offboarding workflows were validated earlier in this case study using real executions through the portal. These tests demonstrated:

- Successful user onboarding across all integrated systems
- Controlled and auditable user offboarding
- Correct propagation of role-based access and system state changes

Because these workflows were already validated and documented with screenshots in Phase 2, they are not repeated here. Instead, Phase 4 focuses on **operational behavior, resilience, and monitoring**.

5.1.1 Onboarding Validation (Reference to Phase 2)

The onboarding workflow was executed end-to-end to validate correct system behavior. This includes identity creation, data persistence, directory integration, and desktop provisioning.

The following components were verified during testing:

- Employee record creation in PostgreSQL
- User creation and group assignment in Amazon Cognito
- User creation in Active Directory
- Amazon WorkSpace provisioning
- Role-based computer placement in Active Directory for GPO application



5.1.2 Offboarding validation (reference)

Offboarding was validated in Phase 2 by terminating a user through the portal. The process disables the Cognito account, disables the Active Directory user, terminates the Amazon WorkSpace, and removes the employee from the portal database. Screenshots in Phase 3 confirm correct execution across all systems.

5.2 Kubernetes Failover Testing

To validate the resilience of the Kubernetes deployment, a controlled failover test was performed on the backend application running in the EKS cluster.

The current state of running pods was inspected using:

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure> kubectl get pods -n employee-services
NAME                                READY   STATUS    RESTARTS   AGE
employee-portal-5474c9984c-6v8q2    1/1    Running   0          86m
employee-portal-5474c9984c-qjwzd     1/1    Running   0          88m
employee-portal-frontend-d59fbbffd-479cl 1/1    Running   0          86m
employee-portal-frontend-d59fbbffd-pb64t 1/1    Running   0          88m
postgres-0                          1/1    Running   0          86m
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure>
```

An active backend pod was manually deleted:

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure> kubectl delete pod employee-portal-5474c9984c-qjwzd -n employee-services
pod "employee-portal-5474c9984c-qjwzd" deleted
```

Kubernetes automatically detected the missing pod and scheduled a replacement:

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure> kubectl get pods -n employee-services
NAME                                READY   STATUS    RESTARTS   AGE
employee-portal-5474c9984c-6v8q2    1/1    Running   0          91m
employee-portal-5474c9984c-chm4k     1/1    Running   0          23s
employee-portal-5474c9984c-qjwzd     1/1    Terminating 0          93m
employee-portal-frontend-d59fbbffd-479cl 1/1    Running   0          91m
employee-portal-frontend-d59fbbffd-pb64t 1/1    Running   0          93m
postgres-0                          1/1    Running   0          91m
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure> kubectl get pods -n employee-services
NAME                                READY   STATUS    RESTARTS   AGE
employee-portal-5474c9984c-6v8q2    1/1    Running   0          91m
employee-portal-5474c9984c-chm4k     1/1    Running   0          36s
employee-portal-frontend-d59fbbffd-479cl 1/1    Running   0          91m
employee-portal-frontend-d59fbbffd-pb64t 1/1    Running   0          93m
postgres-0                          1/1    Running   0          91m
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure>
```

Test Result

- The deleted pod transitioned to the Terminating state
- A new pod was automatically created by the Deployment controller
- The application remained available during the replacement
- No manual recovery actions were required

This confirms that the Kubernetes deployment provides **self-healing behavior at the pod level**.

5.3 Kubernetes Network Security Validation

5.3.1 Active NetworkPolicies in production

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure> kubectl get networkpolicy -n employee-services
```

NAME	POD-SELECTOR	AGE
default-deny-all	<none>	4d8h
employee-portal-policy	app=employee-portal	4d8h

Active Kubernetes NetworkPolicies in the employee-services namespace

This screenshot shows that namespace-level NetworkPolicies are enforced, including a default deny policy and an application-specific policy for the employee portal.

```
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure> kubectl describe networkpolicy employee-portal-policy -n employee-services
```

Name: employee-portal-policy
Namespace: employee-services
Created on: 2025-12-10 17:43:50 +0100 CET
Labels: <none>
Annotations: <none>
Spec:
PodSelector: app=employee-portal
Allowing ingress traffic:
To Port: 5000/TCP
From:
PodSelector: <none>
Allowing egress traffic:
To Port: 53/UDP
To:
NamespaceSelector: kubernetes.io/metadata.name=kube-system

To Port: 5432/TCP
To:
PodSelector: app=postgres

To Port: 443/TCP
To:
IPBlock:
CIDR: 0.0.0.0/0
Except: 169.254.169.254/32

To Port: 389/TCP
To Port: 389/UDP
To Port: 636/TCP
To Port: 88/TCP
To Port: 88/UDP
To:
IPBlock:
CIDR: 10.0.0.0/16
Except:
Policy Types: Ingress, Egress
PS C:\Users\sonny\case_studies\cs3-ma-nca-infrastructure>

Employee portal NetworkPolicy defining allowed ingress and egress traffic

The employee portal NetworkPolicy restricts ingress to the application port and limits egress traffic to required services only, including PostgreSQL, DNS, Active Directory, and outbound HTTPS. This enforces least-privilege network communication inside the cluster.

5.4 Kubernetes Runtime Monitoring

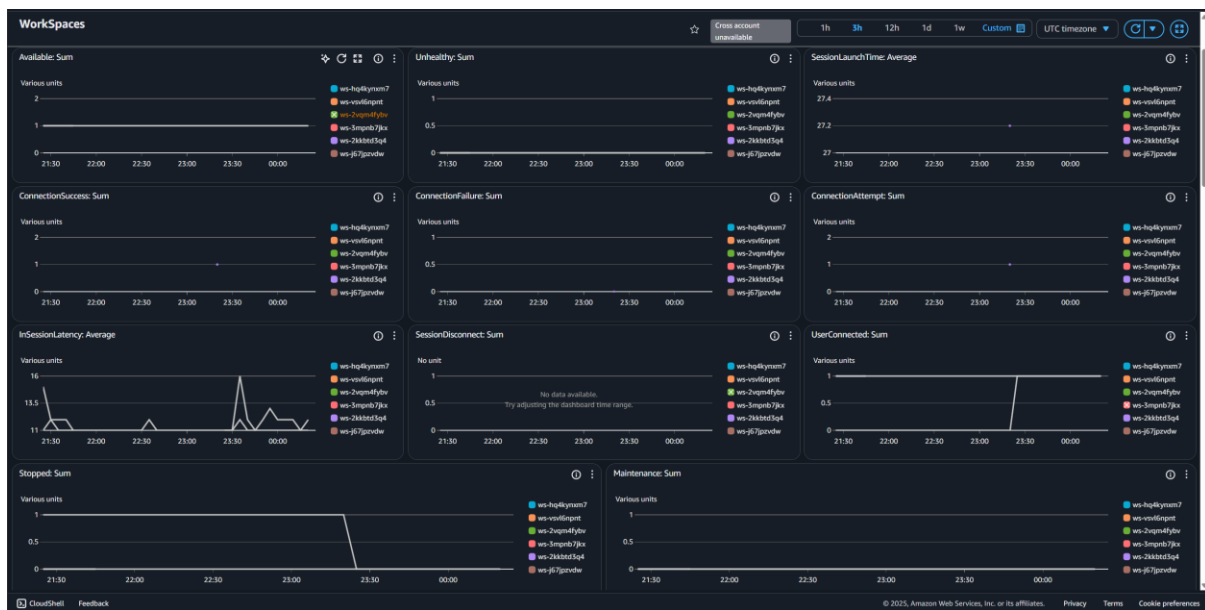
5.4.1 EKS cluster monitoring (CloudWatch)



CloudWatch dashboard showing EKS cluster health and resource utilization

This dashboard provides operational visibility into the EKS cluster hosting the employee portal. Resource utilization and node health metrics confirm stable cluster behavior during application usage.

5.4.2 Amazon WorkSpaces Monitoring



CloudWatch dashboard monitoring Amazon WorkSpaces availability and session metrics

WorkSpaces monitoring confirms successful provisioning, user connectivity, and termination behavior. Metrics demonstrate that virtual desktops are available when required and correctly transition state during offboarding.

5.5 CI/CD Deployment Verification

```
✓ Deploy Employee Portal

1  ▶ Run echo "Deploying Employee Portal..."
19 Deploying Employee Portal...
20 namespace/employee-services unchanged
21 serviceaccount/employee-portal-sa unchanged
22 deployment.apps/employee-portal unchanged
23 service/employee-portal unchanged
24 deployment.apps/employee-portal-frontend unchanged
25 service/employee-portal-frontend unchanged
26 Waiting for deployments to be ready...
27 deployment.apps/employee-portal condition met
28 Employee Portal deployed successfully!

✓ Post Configure AWS Credentials
```

Successful deployment rollout of the employee portal in EKS

The deployment pipeline applies updated Kubernetes manifests and triggers a rollout restart of the employee portal pods. This confirms that application updates are successfully deployed to the cluster.

5.6 Cost Management & Optimization

This section provides a concise overview of the operational costs of the Case Study 3 environment and the measures taken to keep the solution within a realistic budget.

Component	Description	Estimated Monthly Cost
Amazon EKS Control Plane	Managed Kubernetes control plane	~\$73
EKS Worker Nodes	2 × t3.medium EC2 instances	~\$60
Amazon EBS Volumes	Persistent storage for PostgreSQL	~\$8
Amazon WorkSpaces	Windows WorkSpaces (AutoStop enabled)	Variable
CloudWatch Logs & Metrics	Monitoring and log ingestion	~\$5
Estimated Total		~\$146 + WorkSpaces usage

WorkSpaces costs depend on the number of active users and session duration. AutoStop mode is enabled to minimize idle costs.

Cost Optimization Measures

Several cost control strategies were applied:

- **AutoStop WorkSpaces** to reduce idle compute costs
- **Right-sized EKS nodes** to match workload demand
- **Single PostgreSQL StatefulSet** to avoid unnecessary replication overhead
- **Use of managed services** to reduce operational and maintenance effort

These measures ensure the platform remains affordable while still demonstrating enterprise-grade architecture patterns.

5.7 Summary of Phase 4 validation

The validation activities performed in this phase demonstrate that the deployed solution operates as intended after deployment. Identity lifecycle management, workspace provisioning, security controls, and platform stability were all verified using real operational actions rather than simulated tests.

Phase 4 confirms that the deployed solution is operationally sound. The system demonstrates:

- Functional correctness
- Automated recovery from pod failures
- Centralized monitoring and visibility
- Controlled operational costs

This validates that the platform is suitable as a scalable and maintainable foundation for an enterprise self-service employee portal.

6. Evaluation & Reflection

This section evaluates the final solution and reflects on the design choices, challenges, and learning outcomes of Case Study 3. The goal of this evaluation is to assess whether the implemented solution meets the technical and functional requirements defined at the start of the project and to reflect on the learning progression compared to previous case studies.

6.1 Evaluation of the final Solution

The final solution delivers a fully automated employee lifecycle platform that integrates identity management, directory services, virtual desktop provisioning, and Kubernetes-based application deployment. The system successfully supports both onboarding and offboarding workflows through a centralized portal, eliminating the need for manual administrative actions across separate systems.

From a functional perspective, the solution meets the core requirements of the assignment. Employee onboarding results in the automatic creation of identity accounts, database records, Active Directory users, and Amazon WorkSpaces. Offboarding reliably revokes access by disabling identity accounts, directory users, and terminating WorkSpaces. These processes were validated through real operational testing as documented in the earlier phases.

From a security perspective, the solution applies multiple layers of access control. Role-based access is enforced at the application level, network isolation is implemented using Kubernetes NetworkPolicies, and identity boundaries are maintained through a combination of Amazon Cognito and Active Directory. Monitoring and observability using CloudWatch provide operational visibility into both the Kubernetes cluster and the WorkSpaces environment, supporting ongoing management and troubleshooting.

Overall, the final solution is considered stable, secure, and suitable for operational use within the defined scope of the case study.

6.2 Reflection on design decisions

Several key design decisions significantly influenced the final outcome of this project.

The choice to deploy the employee portal on Amazon EKS allowed the application to benefit from container orchestration, scalability, and clear separation of responsibilities between services. Compared to earlier case studies, this introduced additional complexity but also resulted in a more realistic and production-oriented platform.

For identity management, a hybrid approach combining Amazon Cognito and Active Directory was selected. Cognito provides cloud-native authentication and role-based

access for the portal, while Active Directory remains responsible for domain-based access and Group Policy enforcement. This separation reflects real-world enterprise environments where cloud and on-premises identity systems often coexist.

An important design decision involved role-based software provisioning. An initial approach using AWS Systems Manager was considered, but during implementation it proved unsuitable for the WorkSpaces and domain-joined environment used in this case study. Based on feedback from the supervising lecturer, the design was revised to use computer-based Group Policy Objects applied through Active Directory organizational units. This approach aligned better with the WorkSpaces lifecycle and resulted in a more reliable and maintainable solution.

6.3 Comparison with previous Case Studies

Compared to Case Study 1 and Case Study 2, this project represents a clear increase in scope and complexity. Case Study 1 focused primarily on infrastructure provisioning, while Case Study 2 introduced event-driven automation and security workflows. Case Study 3 builds on these foundations by combining automation, identity management, Kubernetes deployment, and operational monitoring into a single integrated platform.

A key improvement in Case Study 3 is the emphasis on end-to-end lifecycle management rather than isolated automation tasks. The solution demonstrates a more holistic approach, where identity, infrastructure, and application layers are tightly integrated and managed through a central portal. This reflects a more mature understanding of cloud platform design and operational responsibility.

6.4 Challenges, limitations, and improvements

One of the main challenges during this project was handling asynchronous behavior in Amazon WorkSpaces provisioning. Because WorkSpaces computer objects are not immediately available in Active Directory, additional logic was required to synchronize computer accounts into the correct organizational units after provisioning. While this was successfully implemented, it introduces a dependency on timing and availability that would require further refinement in a production environment.

Another limitation is the operational overhead introduced by Kubernetes. While EKS provides scalability and flexibility, it also requires additional monitoring, maintenance, and configuration compared to simpler deployment models. For smaller environments, a lighter-weight solution might be more appropriate.

If this solution were extended further, potential improvements could include enhanced automation around error handling, more granular audit logging, and tighter integration with CI/CD pipelines for container image management.

6.5 Teacher Feedback & Iterative Improvement

During multiple feedback sessions with my teacher (Checkpoints 4, 5, and 6), I presented the evolving architecture and functionality of the employee onboarding and offboarding platform. The feedback focused on role-based access, automation flow, and endpoint configuration. Based on this feedback, I refined the solution by replacing the initial SSM-based tooling approach with Group Policy Objects for WorkSpaces, which resulted in a more reliable and scalable implementation. These feedback moments confirmed that the overall design and technical choices aligned with the project requirements.

Checkpoint 4 28-11-2025

He, Sonny W.J.S. 11 days ago

In this feedback session, I showed my diagram and the key technologies that meet the requirements that I have chosen to use for the case study, also a bit of the of the documentation, because I finished Phase 1. Erik found the diagram good and the chosen technologies good. To continue Phase 2 properly, Erik asked some questions about user creation and the roles that I should consider. I took those into consideration and thought about the solutions.

Feedback session during Phase 2 architecture review

Checkpoint 5 08-12-2025

He, Sonny W.J.S. 3 days ago

In this feedback session, I showed mr. Erik my working Employee portal. I showed the system, the flow of how it works. When a user gets to the employee portal, you first have to login. The authentication happens via AWS Cognito. The username and password is stored there. Also, i use AWS Cognito for RBAC. In Cognito, you can add what kind of role the user has. So if you login with Admin, you'll get the JWT token for Admin and login as an Admin. The Admin has the extra features to terminate users and creating users.

The creating user function works as follows:

The admin has to fill in the employee details. The important part of this form is that you can also select the role in the dropdown box to prevent typos. This role will be tagged to the newly created user. When the admin finishes filling in the details and presses the create user button, it will then create the data base record, AWS Cognito user, a new user in the AD and also the right group depending on what role the admin chose, and a WorkSpace will be provisioned for the created user. This happens via the backend sending the information the the APIs and LDAP port.

Erik was impressed with the architecture and the system and told me about the benefits it had because of the way the architecture was thought out. Because with this kind of architecture, I can also easily add a function in the portal where an employee can request for example a WorkSpace with different tools to test. The tools a WorkSpace gets depends on the Role of the user after all. But as Erik said, the architecture already has the prerequisites to add the feature where I can allow to users to request for a WorkSpace with different tools. Erik also told me to make sure there are different tools installed depending on the Role. I plan to do that with AWS SSM. Erik advised me to also look into GPO. Overall it was a good feedback session.

Demonstration of onboarding and RBAC flow



He, Sonny W.J.S. 3 days ago

In this feedback session, I showed that I finished the role based tool installation. I initially tried doing it with SSM, that didn't work because personal AWS WorkSpaces are not the right instance types, so SSM can't target it even with the right conditions set. In the Fleet Manager of SSM, the WorkSpaces didn't show up, that's how I pretty much confirmed that it was likely not possible. I then followed Erik's advice to use GPO. I had to use Group Policy Manager and create a GPO for each OU(Admins, Developers, Employee). The GPO would have a package wherein I make sure it installs the right tool for each role. For Admins, I set the GPO to install PuTTY. For Developers node.js and for Employees just 7zip. These msi installers had to be in a shared folder, so I also had to create a shared folder on the Admin machine. I made sure every User can at least read it and that the Domain Controllers have the Read and Execute rights. I then also had to make sure the SBM port 445 inbound rule was set in my security group to allow the other non-admin machines to be able to access the and see the contents of the shared folder. After setting it up, I moved the msi installers of Node.js, PuTTY and 7ZIP in there to make sure the GPO can do its thing properly.

I also showed how I can as an Admin in the portal, terminate the user, which disabled the user in AWS Cognito and also the user in the Active Directory. I also showed that newly created users have to change password on login which is how I set it up in Cognito.

Erik was overall impressed and this talk went well.



Write a summary of what you discussed with your teacher...

Post feedback

Final feedback confirming GPO-based role tooling

6.6 Conclusion

Case Study 3 demonstrates the design and implementation of a realistic cloud-based employee lifecycle platform that integrates identity management, directory services, virtual desktops, and Kubernetes deployment. The solution fulfills the defined requirements and shows clear progression in technical skill and architectural thinking compared to earlier case studies.

Through iterative development, feedback-driven design decisions, and thorough validation, the final solution achieves a balance between automation, security, and operational control. This case study reflects a strong learning outcome and provides a solid foundation for future work in cloud-native platform engineering.