SWINBURNE UNIVERSITY OF TECHNOLOGY

SYDNEY CAMPUS

MASTER OF INFORMATION TECHNOLOGY (PROFESSIONAL COMPUTING)

COS80029-Technology Application Project

Project Report

XMOPS Accelerate

Submitted to

Dr. Ayesha Binte Ashfaq

Submitted by

Santosh Pokhrel : 104053011

Amir Maharjan : 104088013

Byambadorj Burentogtokh: 103133909

Semester: Second

Date: 17-04-2023

April, 2024

Table of Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
CSS	Cascading Style Sheets
EC2	Amazon Elastic Compute Cloud
HTML	Hypertext Markup Language
IAM	AWS Identity and Access Management
IDE	Integrated Development Environment
JS	JavaScript
JSON	Javascript Object Notation
R&D	Research and Development
RDS	Amazon Relational Database Service
S3	Amazon Simple Storage Service
SSH	Secure Shell
VPC	Value Proposition Canvas

Chapter 1 Introduction

1.1 Introduction

In the dynamic and ever-evolving landscape of cloud computing and infrastructure management, efficiency and automation stand as the cornerstone of successful operations. Recognizing this imperative, the XMOPS team has embarked on an ambitious endeavor to revolutionize the way redundant tasks and deployments are managed through the introduction of XMOPS Accelerate. This cutting-edge, centralized control plane is meticulously designed to streamline, simplify, and accelerate a wide array of operations critical to the team's success, ranging from WordPress deployments to intricate IAM roles and policy updates, as well as comprehensive infrastructure provisioning.

At the heart of XMOPS Accelerate lies a dual focus on innovation and user-centric design, aiming to address and fulfill several critical objectives. First and foremost, the platform endeavors to automate the deployment of various infrastructure components. This includes the seamless deployment of WordPress applications in configurations as varied as monolithic, distributed, managed, or even highly available architectures, thus ensuring efficiency and scalability. Furthermore, a paramount aspect of XMOPS Accelerate is its robust authentication mechanism, powered by Amazon Cognito. This feature not only facilitates secure login and access control for Xmops team members but also establishes a foundation of trust and reliability through a secure authentication framework.

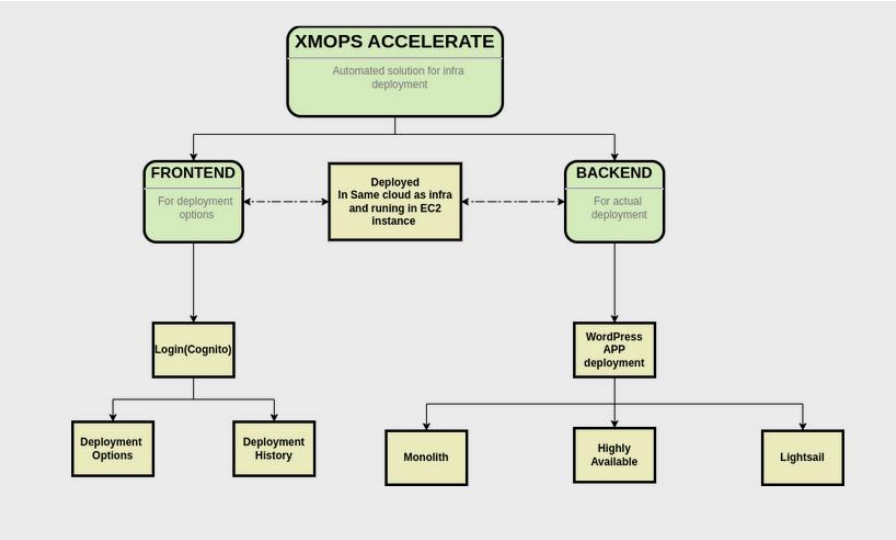

Figure 1 Workflow Diagram

1.2 Project Overview

XMOPS Accelerate represents a transformative initiative designed to redefine the landscape of infrastructure deployment and management within the XMOPS team's operational framework. By harnessing the power of

automation and centralized control, this project sets out to significantly streamline the deployment of various infrastructure components, with a keen focus on WordPress applications, IAM roles, policy updates, and comprehensive infrastructure provisioning. XMOPS Accelerate stands as a testament to the team's commitment to efficiency, security, scalability, and innovation, aiming to deliver a robust platform that not only meets but exceeds the operational requirements and expectations of its users.

1.3 Purpose

The purpose of XMOPS Accelerate is multi-fold:

- To automate and simplify the deployment process for various infrastructure components.
- To ensure secure access and management through robust authentication mechanisms.
- To offer a scalable and flexible solution that can adapt to future expansion and diverse deployment needs.

1.4 Scope

The software aims to enhance efficiency, security, and control over deployment processes. This document will cover the functionalities, user interactions, and technical specifications without delving into the implementation details. The XMOPS accelerate can be used in a wide range of industries and corporation where the use of cloud is essential specially in news agencies, for blogging, etc.

1.5 Objectives

The primary objectives are as follows:

1. **Automate Infrastructure Deployment**: To significantly reduce manual effort and human error by automating the deployment of infrastructure components.
2. **Enhance Security and Access Control**: To implement robust security measures ensuring that access to the XMOPS Accelerate platform is secure and controlled. This involves integrating Amazon Cognito for authentication, enabling multi-factor authentication, and implementing IP whitelisting to restrict access at the network level.
3. **Improve Operational Efficiency**: To enhance the overall efficiency of the Xmops team by streamlining the deployment process. This includes developing a user-friendly front-end interface that simplifies navigation and management of deployment options, thereby saving time and reducing the cognitive load on team members.
4. **Ensure Scalability and Flexibility**: To build a scalable system architecture that can accommodate future growth and expansion. XMOPS Accelerate should be flexible enough to adapt to evolving deployment needs and scenarios, ensuring the platform remains relevant and effective as the team's requirements change.
5. **Provide a Centralized Control Plane**: To offer a centralized platform for managing all aspects of infrastructure deployment. This centralization simplifies the process, making it easier for the Xmops team to oversee and control deployments across different environments and infrastructure types.

1.6 Architecture

The system architecture of XMOPS Accelerate is built on a robust, scalable, and secure framework that integrates seamlessly with AWS services, leveraging Terraform for infrastructure as code (IaC) to automate provisioning. The architecture encompasses two main components: the Front-End and the Back-End.

- **Front-End**: Crafted with a focus on user experience, the front-end interface of XMOPS Accelerate offers intuitive navigation and management capabilities.
- **Back-End**: The back-end serves as the backbone of XMOPS Accelerate, where Terraform scripts automate the provisioning of infrastructure. It supports a wide range of deployment scenarios, from WordPress applications to complex high-availability setups, and integrates with AWS services for comprehensive infrastructure management. We have three options when it comes to deployment, Monolithic architecture, Highly Available architecture and Lightsail architecture

All the three different architectures of XMOPS Accelerate are modular, comprising front- end, back-end, and integration with AWS services. It utilizes Terraform for infrastructure automation and Amazon Cognito for authentication, ensuring a scalable and secure environment.

1.6.1.1 Highly Available Architecture

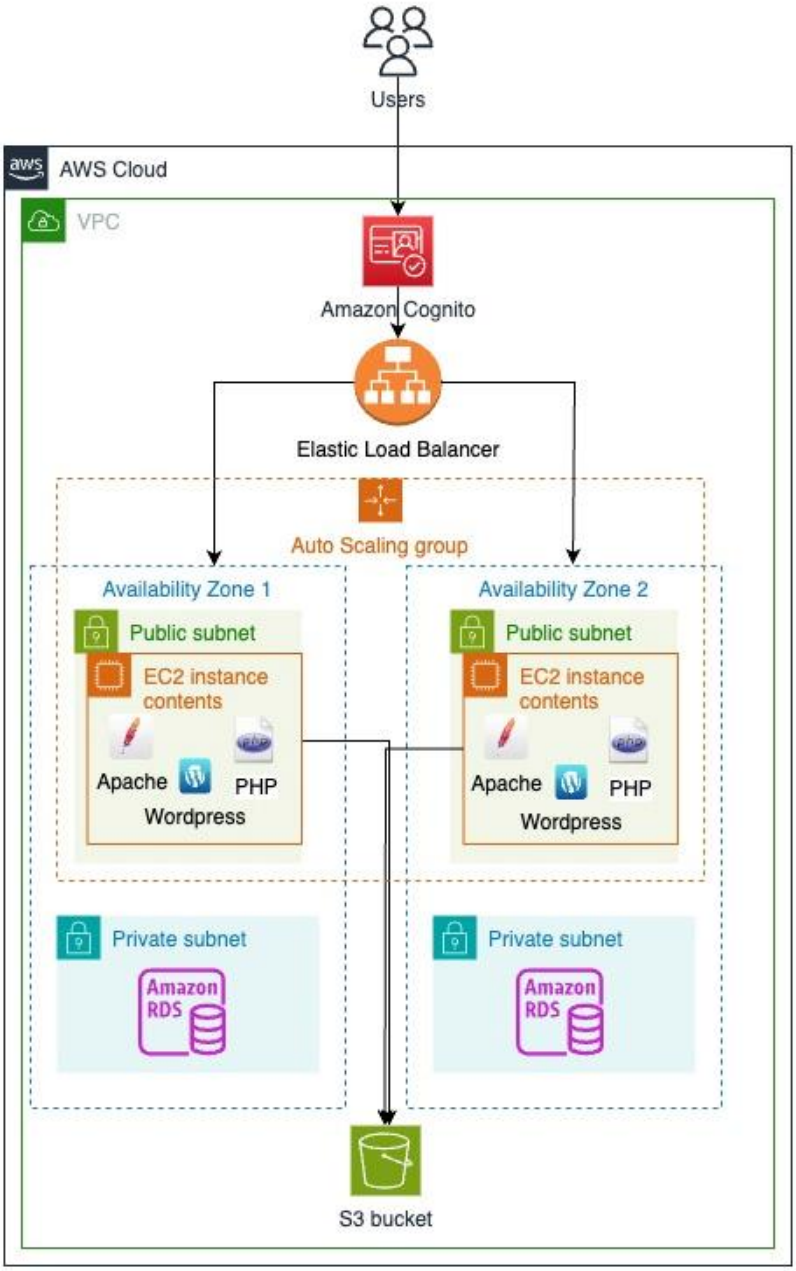

Figure 2 Highly Available Architecture

Implementing High Availability in deployments involves offering solutions for building durable and stable systems. In our structure, we employ an elastic load balancer to distribute incoming traffic evenly, and we intend to duplicate the entire system across an additional availability zone to serve as a backup, thereby minimizing downtime.

1.6.1.2 Lightsail Architecture

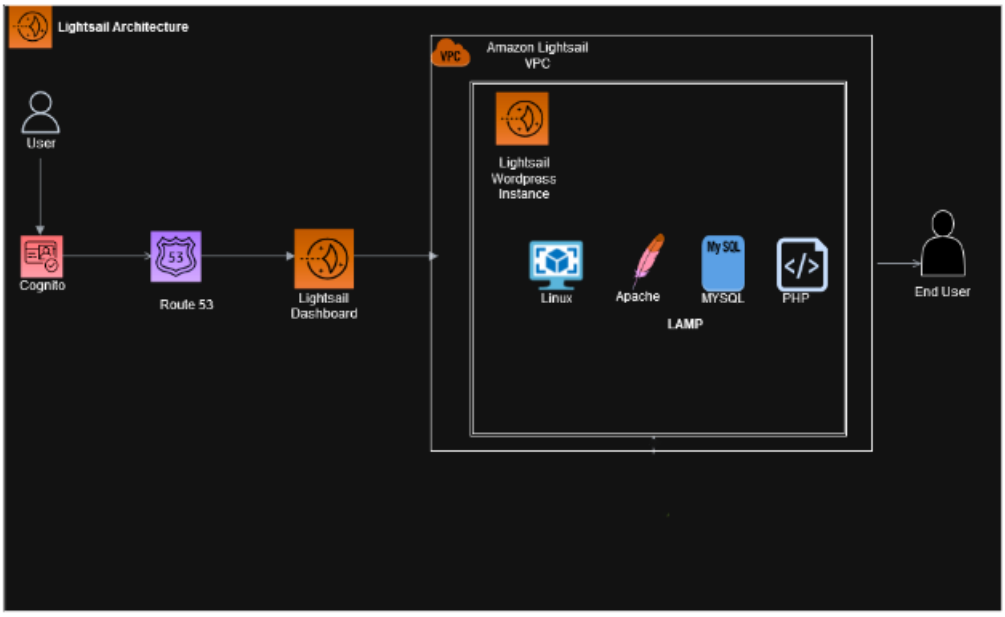

Figure 3 Lightsail Architecture

Lightsail is a simplified cloud computing platform provided by Amazon Web Services (AWS). It makes it easy to set up and deploy applications in the cloud. Supporting deployment on Lightsail instances means providing options specifically tailored for this platform. So, in lightsail architecture we do not need to install the wordpress components manually, and do not need to inject the installation script in our code

1.6.1.3 Monolithic Architecture

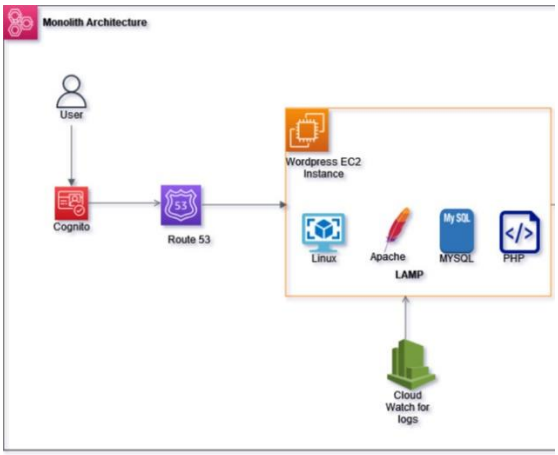

Figure 4 Monolithic Architecture

A monolithic application is built as a single, cohesive unit where all the components are interconnected and work together. It's a more traditional way of designing software. In our monolith architecture we only have one instance to host all the required components of the system.

1.7 Functional Requirements

The functional requirements of XMOPS Accelerate are as follows:

1. **Automated Infrastructure Deployment**
   - The system must allow users to deploy infrastructure components automatically using Terraform scripts.
   - Support deployment for WordPress applications, microservices, monolithic applications, Lightsail instances, and high-availability setups.
2. **Authentication and Authorization**
   - Integrate Amazon Cognito for user authentication and authorization.
   - Implement secure login features with multi-factor authentication.
3. **User Interface**
   - Offer a user-friendly dashboard that provides an overview of ongoing and completed deployments.
   - Enable users to input deployment parameters through a user-friendly form.
   - Display a comprehensive history of previous deployments, including details about the completion and type of deployments.
4. **Deployment Management**
   - Allow users to select deployment options and configurations for different scenarios (e.g., Lightsail, monolith, high-availability setups).
   - Provide functionality to start, monitor, and terminate deployments.
5. **Integration with AWS Services**
   - Seamlessly integrate with AWS services such as EC2, RDS, VPC for infrastructure provisioning.

1.8 Non functional Requirements

Non-functional requirements describe how the system works, focusing on reliability, efficiency, and other quality attributes. The non functional requirements of the XMOPS Accelerate are as follows:

1. **Scalability**
   - The system must be scalable to handle an increase in workloads, including a higher number of deployments and users, without degradation of performance.
   - It should support scalability in both infrastructure provisioning and user management.
2. **Performance**
   - The system should ensure rapid processing of deployments, aiming for minimal latency in initiating and executing deployment tasks.
   - User interface elements should load quickly, providing a responsive experience for the user.

3. **Security**
    - Employ robust security mechanisms to protect against unauthorized access and data breaches.
    - Implement IP whitelisting and secure communication channels (e.g., SSL/TLS) for all interactions with the system.
4. **Reliability**
    - Ensure high availability of the XMOPS Accelerate system, with minimal downtime.
    - Deployments should be executed reliably, with mechanisms in place to recover from failures and ensure data integrity.
5. **Usability**
    - The user interface should be intuitive, allowing users to easily navigate and perform tasks without extensive training.
    - Provide comprehensive documentation and support materials to assist users in understanding and utilizing the system effectively.
6. **Maintainability**
    - The system should be designed for easy maintenance and updates, allowing for quick adaptation to new requirements or changes in the technology landscape.
    - Employ coding best practices and standards to facilitate code readability and modularity.

Chapter 2 Design Concept

2.1 Methodology

The development of XMOPS Accelerate happened on a thorough understanding and meticulous analysis of the project's requirements, alongside a well-structured design and implementation strategy. This methodology chapter outlines the comprehensive approach taken from requirement analysis and design to the deployment, monitoring, and continuous improvement of the system. he methodology adheres to Agile principles and encompasses the following steps

1. **Requirement Analysis and Design**

   In the project's first phase, an exhaustive analysis of requirements was conducted to identify the essential components and functionalities needed for XMOPS Accelerate. This analysis was followed by the design of a system integrating various AWS services and technologies to effectively meet these identified requirements.

2. **Environment Setup**
   - **Setting up S3 bucket**: The setup of an Amazon S3 bucket to store deployment history in JSON format was identified as a crucial step. This enabled efficient tracking and auditing of deployment activities.
   - **API Configuration** : An API was configured to fetch AWS regions using boto3, facilitating the selection of appropriate regions for all the deployments.
   - **Terraform Configuration**: Terraform files were defined and configured to manage SSH keys, instance sizes (ranging from nano to extra large), and other necessary deployment parameters for various architecture types.
   - **Architecture Setup**: Specific configurations were identified as necessary for deploying monolithic architectures, including database type, PHP version, and the choice between Apache or Nginx servers. For highly available setups, detailed configurations for EC2 instances and RDS databases were necessitated, covering aspects like AWS Region, instance type, key pair management, security group settings, and database engine types and versions.
3. **Agile Development Process**
   - Sprints were organized to focus on specific sets of features or components. This approach was facilitated to ensure a clear focus and efficient tracking of progress.
   - To ensure alignment, address challenges, and refine strategies, regular stand-up meetings, sprint planning sessions, and retrospectives were conducted.

4. **Testing and Quality Assurance**
   - All routes and endpoints were subjected to unit testing to verify functionality and reliability.
   - The integrity and efficiency of the end-to-end workflow, including the authentication and authorization mechanisms and multi-factor authentication (MFA) on email, were ensured through integration testing.

- Validation against invalid inputs was conducted for all input fields to enhance system robustness and user experience.

5. **Documentation**
   Comprehensive user manuals and system documentation detailing the system's functionalities, deployment options, and alerting workflows were created.
6. **Deployment and Monitoring**

   - The implementation of continuous monitoring of S3 data, and deployment history ensured the system's health and operational efficiency. Automated scaling and recovery mechanisms were specially designed for high-availability components.

2.2 Software Requirements

The minimum software requirements are listed below:

- Terraform greater than version 1.0
- Python version 3.9
- Aws CLI must be installed and configured
- AWS account

2.3 Hardware Requirements

The software has no hardware requirements, yet The bare minimum hardware requirements to run the XMOPS.

1. Personal computer with at least 4GB RAM
2. Internet Connection

Chapter 3 Configuration

This chapter emphasizes on how to configure the AWS services and associated resources using Terraform. The provided configuration allows the developers to set up an authentication and authorization mechanism .

3.1 Terraform Variables

The configuration uses Terraform variables to customize the behavior of the resources. We will have to Review and adjust these variables according to the requirements.

3.1.1 Variables to be configured

In the XMOPS Accelerate project, the configuration of infrastructure components through Terraform is both dynamic and customizable. Below is an overview of the key variables that need to be reviewed and adjusted according to the specific requirements of the deployment.

- **instance_name**: Defines the name of the Lightsail or EC2 instance, serving as an identifier for the provisioned resources.
- **key_pair_name**: Specifies the base name for the Lightsail or EC2 Key Pair, to which a unique suffix might be appended for uniqueness.
- **ssh_key**: Holds the public SSH key, enabling secure shell access to the instance.
- **bundle_id**: Identifies the Lightsail bundle, determining the hardware and network capacity of the instance.
- **blueprint**: Specifies the blueprint for the Lightsail instance, such as WordPress, defining the software configuration.
- **aws_region**: Determines the AWS region where resources will be provisioned, impacting latency and availability.
- **vpc_name**, **vpc_cidr_block**, and related VPC configuration variables: Define the virtual network within which resources will be deployed, including its address space and subnets.
- **instance_type**, **ami_id**, and EC2-specific variables: Determine the hardware, software, and scale of EC2 instances for different deployment scenarios.
- **db_engine**, **engine_version**, and RDS-specific variables: Configure the database engine type, version, and scale for RDS instances.

3.1.2Steps to configure

Configuring XMOPS Accelerate's infrastructure with Terraform involves several steps that leverage the defined variables and resources within the variables.tf and main.tf files. Here's a structured approach to setting up the system:

1. **Review Requirements**: Start by thoroughly reviewing the deployment requirements to understand the specific configurations needed, such as instance sizes, regions, and the type of infrastructure (e.g., Lightsail, EC2, RDS).
2. **Adjust Variables**: Open the variables.tf file and modify the variables according to the project's needs.
3. **Define Resources**: In the main.tf file, use the adjusted variables to define the resources required for the deployment. This includes specifying the VPC setup, Internet Gateway, subnets, route tables, and the instances themselves (Lightsail, EC2, and RDS).
4. **Terraform Initialization**: Run terraform init to initialize the Terraform project,
5. **Plan Deployment**: Execute terraform plan to preview the actions .
6. **Apply Configuration**: Finally, run terraform apply to deploy the infrastructure.
7. **Verify Deployment**: Once the deployment is complete, verify that all resources have been provisioned as expected. This can involve logging into the AWS Console, checking the state of instances, and ensuring network configurations are correctly applied.
8. Cleaning Up: When you no longer need the configured resources, use the following command to destroy them and clean up:terraform destroy.

Chapter 4 Installation

This chapter guides the user through the installation process of the required tools and resources to set up xmops accelerate.

4.1 Deployment

With the code and configuration customized, follow these steps to deploy the xmops accelerate:

- Open a terminal window and navigate to the backend directory. Run the following command to activate the backend server:
    - From the root directory: cd backend
    - Then activate the server by running the command: python3 app.py, this command will start the backend server and all the routes will be available for the user
- In the next step we will have to activate the frontend server, to do this, open new terminal and switched to the frontend folder with the command: cd frontend.
- After this run the command: python3 -m http.server 8000, to activate the frontend server at 8000 port
- Finally open the given address on your favourite browser and the system will be ready for use.

4.2 Clean up

All the infrastructure will be initialised automatically but o destroy the infrastructure the user has to manually the command. To destroy the infrastructure,

Change directory to: cd terraform

And then Run: terraform destroy

This command will remove all the resources created by Terraform.

Chapter 5 Code Structure

Python Script : App.py

This is the main file which contains all the routes and endpoints for the whole system. Here is the description of each route:

The `app.py` script serves as the backbone of a deployment platform, orchestrating various operations through its Flask-based web server. It facilitates user interactions with the AWS environment, leveraging services like Amazon Cognito for authentication, EC2 for computing resources, and S3 for storage. Here's a brief description of each route and its functionality within the script:

- `/`
  Route Description: Serves as the landing page of the application.
  Functionality: Returns a welcoming message to indicate the service is operational.

- `/signup` (POST)
  Functionality: Registers a new user by creating an account in Amazon Cognito with the provided email and password.

- `/verify` (POST)
  Functionality: Confirms a user's email verification code to complete the signup process in Cognito.

- `/login` (POST)
  Functionality: Authenticates a user against Amazon Cognito credentials, returning session tokens upon successful login.

- `/menu`
  Functionality: Provides a list of available deployment options and related actions to the user.

- Deployment Routes
  These routes are prefixed with specific paths to categorize the deployment types (e.g., `/highly` for high-availability setups).

- `/highly/deploy` (POST)
  Functionality: Initiates the deployment of a high-availability infrastructure using Terraform scripts.

- `/highly/destroy` (POST)
  Functionality: Triggers the destruction of deployed infrastructure, ensuring resources are cleaned up.

- `/highly/validate_form` (POST)

Functionality: Validates the input form for deploying infrastructure, ensuring all required fields are correctly filled.

- `/highly/existing_key_pairs` (POST)
  Functionality: Retrieves a list of existing EC2 key pairs in a specified AWS region.

- `/highly/create_key_pair` (POST)
  Functionality: Generates a new EC2 key pair in the specified region and saves the private key file.

- `/highly/get_regions` (GET)
  Functionality: Lists all available AWS regions to inform user selection for resource provisioning.

- `/highly/amis` (POST)
  Functionality: Fetches available Amazon Machine Images (AMIs) based on the specified operating system type and region.

- `/highly/instance_types` (POST)
  Functionality: Retrieves available EC2 instance types within a specified region, aiding in resource selection.

- `/highly/db_engine_types` (POST) and `/highly/db_engine_versions` (POST)
  Functionality: Lists available database engines and their versions for RDS instance provisioning.

- `/highly/deployment_info` (GET)
  Functionality: Retrieves information about past deployments from an S3 bucket, providing a history of activities.

- `/deploy` (POST)
  Functionality: Generic deployment route that triggers Lightsail instance deployment with Terraform, using provided SSH keys and other parameters.

- S3-related Functions
  Functions related to creating buckets, saving deployment histories, and fetching such histories from S3 are included but not exposed as routes. They support the underlying functionality of deployment and history tracking.

- `/fetch-deployment-history` (GET)
  Functionality: Fetches the deployment history stored in an S3 bucket, allowing users to review past deployment activities.

```python
from flask import Flask, jsonify, request, redirect
from flask_cors import CORS
import subprocess
import json
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
import os
import boto3
from botocore.exceptions import ClientError
from botocore.exceptions import NoCredentialsError, ClientError
import logging
import datetime



from dotenv import load_dotenv

load_dotenv()

load_dotenv()

highbucket_name = 'amirxmopbucket'
region_name = 'ap-southeast-2'
create_bucket_config = {
    'LocationConstraint': region_name
}

# Define the prefix
high_prefix = '/highly'
mono_prefix = '/mono'

logging.basicConfig(level=logging.INFO)

app = Flask(__name__)
CORS(app)  # Enable CORS for all routes
CORS(app, resources={r"/*": {"origins": "http://127.0.0.1:8000"}})
CORS(app, resources={r"/*": {"origins": "*"}})
CORS(app, resources={r"/*": {"origins": "*"}}, supports_credentials=True)

# getting the userpool details from env file
AWS_REGION = os.getenv('AWS_REGION')
COGNITO_USER_POOL_ID = os.getenv('COGNITO_USER_POOL_ID')
COGNITO_APP_CLIENT_ID = os.getenv('COGNITO_APP_CLIENT_ID')

cognito_client = boto3.client('cognito-idp', region_name=AWS_REGION)

@app.route('/')
def index():
    return "Welcome to the Deployment Platform!"


@app.route('/signup', methods=['POST'])
def signup():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')


    try:
        response = cognito_client.sign_up(
            ClientId=os.getenv('COGNITO_APP_CLIENT_ID'),
```

```
            Username=email,
            Password=password,
            UserAttributes=[{'Name': 'email', 'Value': email}],
        )
        return jsonify({'message': 'Signup successful. Please check your
email to verify your account.'}), 200
    except ClientError as e:
        error_code = e.response['Error']['Code']
        if error_code == 'UsernameExistsException':
            return jsonify({'error': 'User already exists.'}), 409  # 409
Conflict
        else:
            return jsonify({'error': 'Failed to sign up. Please try
again.'}), 500


@app.route('/verify', methods=['POST'])
def verify():
    data = request.get_json()
    email = data.get('email')
    code = data.get('code')

    try:
        response = cognito_client.confirm_sign_up(
            ClientId=os.getenv('COGNITO_APP_CLIENT_ID'),
            Username=email,
            ConfirmationCode=code,
            ForceAliasCreation=False
        )
        return jsonify({'message': 'Account verified successfully'}), 200
    except cognito_client.exceptions.UserNotFoundException:
        return jsonify({'error': 'User does not exist.'}), 404
    except cognito_client.exceptions.CodeMismatchException:
        return jsonify({'error': 'Invalid verification code.'}), 400
    except cognito_client.exceptions.NotAuthorizedException:
        return jsonify({'error': 'User is already confirmed.'}), 400
    except Exception as e:
        # Generic error handling
        return jsonify({'error': str(e)}), 500

@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')

    try:
        auth_result = cognito_client.initiate_auth(
            ClientId=COGNITO_APP_CLIENT_ID,
            AuthFlow='USER_PASSWORD_AUTH',
            AuthParameters={'USERNAME': email, 'PASSWORD': password}
        )
        return jsonify({'message': 'Login successful', 'tokens':
auth_result}), 200
    except cognito_client.exceptions.NotAuthorizedException:
        return jsonify({'error': 'Incorrect username or password.'}), 401
    except cognito_client.exceptions.UserNotFoundException:
        return jsonify({'error': 'User does not exist.'}), 404
    except cognito_client.exceptions.UserNotConfirmedException:
        return jsonify({'error': 'User is not confirmed. Please check your
email for the confirmation link.'}), 401
    except Exception as e:
```

```
        # Log the exception details for debugging purposes
        print(e)
        return jsonify({'error': 'Login failed due to an unexpected error.
Please try again later.'}), 500



@app.route('/menu')
def menu():
    # Placeholder for actual logic
    return jsonify([
        {"name": "Deploy Monolith", "url": "/deploy-monolith"},
        {"name": "Deploy Highly Available", "url": "/deploy-highly-
available"},
        {"name": "Deploy Lightsail", "url": "/deploy-lightsail"},
        {"name": "Deployment History", "url": "/deployment-history"},
    ]), 200


@app.route(f'{high_prefix}/deploy', methods=['POST'])
def deploy_infrastructure():
    try:

        high_terraform_dir = "../../terraform/highly"

        # Initialize Terraform
        init_output = subprocess.run(["terraform", "init"],
cwd=high_terraform_dir, capture_output=True, text=True)
        print(init_output.stdout)
        if init_output.returncode != 0:
            print(init_output.stderr)
            raise Exception('Error initializing Terraform')

        # Plan Terraform
        plan_output = subprocess.run(["terraform", "plan"],
cwd=high_terraform_dir, capture_output=True, text=True)
        print(plan_output.stdout)
        if plan_output.returncode != 0:
            print(plan_output.stderr)
            raise Exception('Error planning with Terraform')

        return jsonify({'message': 'High deployed successfully!'})

    except Exception as e:
        # Log the error and save the 'failed' status if an exception is
caught

        return jsonify({'message': 'Error deploying High instance',
'error': str(e)}), 500

@app.route(f'{high_prefix}/destroy', methods=['POST'])
def destroy_infrastructure():
    try:
        # Check if the destroy.sh script exists
        script_path = '../../terraform/highly/destroy.sh'
        print("Absolute path of script:", os.path.abspath(script_path))  #
Debugging line
        if not os.path.exists(script_path):
            error_message = 'Error: destroy.sh script not found'
            return jsonify({'error': error_message}), 500

        # Execute the destroy.sh script
```

```
        subprocess.run([script_path], check=True)

        return jsonify({'message': 'Infrastructure destruction triggered
successfully by py'}), 200
    except subprocess.CalledProcessError as e:
        error_message = f'Error destroying infrastructure by py:
{e.stderr.decode()}'
        return jsonify({'error': error_message}), 500
    except Exception as e:
        error_message = f'Unknown error: {str(e)}'
        return jsonify({'error': error_message}), 500


@app.route(f'{high_prefix}/validate_form', methods=['POST'])
def validate_form():
    try:
        data = request.get_json()  # Get JSON data from request

        # Perform form validations
        errors = {}
        for key, value in data.items():
            # Check for required fields
            if not value.strip():
                errors[key] = 'This field is required'
            # Check for numeric fields
            if key.endswith('min_instances') or
key.endswith('max_instances'):
                if not value.isdigit():
                    errors[key] = 'Please enter a numeric value'

        if errors:
            return jsonify({'error': 'Form validation failed', 'errors':
errors}), 400
        else:

            # Check if the user chose to create a new key pair
            if 'new_key_pair_name' in data:

                create_key_pair(data)

                data['key_pair_name'] = data['new_key_pair_name']
                data.pop('new_key_pair_name')

            else:

                data['key_pair_name'] = data['existing_key_pair_name']
                data.pop('existing_key_pair_name')

            # Read existing terraform.auto.tfvars content
            with open('terraform.auto.tfvars', 'r') as f:
                existing_content = f.readlines()

            # Update validated input fields
            for key, value in data.items():
                # Exclude double quotations for numeric fields
                if key.endswith('min_instances') or
key.endswith('max_instances'):
                    new_line = f"{key} = {value}\n"
                else:
                    new_line = f"{key} = \"{value}\"\n"
```

```
                # Check if the variable already exists in
terraform.auto.tfvars
                for i, line in enumerate(existing_content):
                    if key in line:
                        existing_content[i] = new_line
                        break
                else:
                    # If the variable doesn't exist, append it to the
content
                    existing_content.append(new_line)

            # Write updated content back to terraform.auto.tfvars
            with open('terraform.auto.tfvars', 'w') as f:
                f.writelines(existing_content)

                # Trigger infrastructure deployment
                # response = deploy_infrastructure()

                # if response[1] == 200:  # Check the status code
                return jsonify({'message': 'Form validation successful and
infrastructure deployment triggered'}), 200
            # else:
            #   return jsonify({'error': 'Failed to trigger infrastructure
deployment after form validation'}), 500

    except Exception as e:
        error_message = f'Error validating form data: {str(e)}'
        return jsonify({'error': error_message}), 500

@app.route(f'{high_prefix}/existing_key_pairs', methods=['POST'])
def get_key_pairs():
    region = request.form.get('region')

    try:
        ec2 = boto3.client('ec2', region_name=region)

        # Retrieve the list of existing key pairs
        response = ec2.describe_key_pairs()
        key_pairs = [key_pair['KeyName'] for key_pair in
response['KeyPairs']]

        return jsonify({'existing_key_pairs': key_pairs})
    except Exception as e:
        return jsonify({'error': str(e)}), 500


@app.route(f'{high_prefix}/create_key_pair', methods=['POST'])
def create_key_pair():
    try:

        region = request.form.get('region')
        new_key_pair_name = request.form.get('new_key_pair_name')

        ec2 = boto3.client('ec2', region_name=region)

        # Generate a new key pair
        response = ec2.create_key_pair(KeyName=new_key_pair_name)

        # Save the private key to a file
        with open(f"{new_key_pair_name}.pem", 'w') as key_file:
            key_file.write(response['KeyMaterial'])
```

```
        return jsonify({'message': 'Key pair created successfully'}), 200
    except Exception as e:
        return jsonify({'error': str(e)}), 500


@app.route(f'{high_prefix}/get_regions', methods=['GET'])
def get_all_regions():
    ec2_client = boto3.client('ec2')

    try:
        response = ec2_client.describe_regions()

        # Extract region names from the response
        all_regions = [region['RegionName'] for region in
response['Regions']]
        return all_regions
    except Exception as e:
        print(f"Error retrieving regions: {e}")
        return []

@app.route(f'{high_prefix}/amis', methods=['POST'])
def get_amis_by_os():
    region = request.form.get('region')
    os_type = request.form.get('os_type')

    ec2_client = boto3.client('ec2', region_name=region)

    if os_type == 'linux':
        name_filter = 'amzn2-ami-hvm-*'
    elif os_type == 'windows':
        name_filter = 'Windows_Server-*'
    elif os_type == 'ubuntu':
        name_filter = 'ubuntu/images/*'
    else:
        return jsonify({'error': 'Invalid OS type'}), 400

    filters = [
        {'Name': 'name', 'Values': [name_filter]},
        {'Name': 'architecture', 'Values': ['x86_64']},
        {'Name': 'root-device-type', 'Values': ['ebs']},
        {'Name': 'virtualization-type', 'Values': ['hvm']},
        {'Name': 'state', 'Values': ['available']},

    ]

    # Retrieve AMIs based on the filters
    try:
        response = ec2_client.describe_images(Filters=filters)
    except Exception as e:
        return jsonify({'error': str(e)}), 500

    # Extract relevant information from the response
    ami_list = []
    for image in response['Images']:
        ami_list.append({
            'ImageId': image['ImageId'],
            'Name': image['Name'],
            'Description': image['Description'],
        })

    return jsonify(ami_list)
```

```
@app.route(f'{high_prefix}/instance_types', methods=['POST'])
def get_instance_types():
    try:
        region = request.form.get('region')

        # Initialize AWS EC2 client
        ec2_client = boto3.client('ec2', region_name=region)

        # Get all instance types available in the region
        response = ec2_client.describe_instance_type_offerings(
            LocationType='region',
            Filters=[
                {'Name': 'location', 'Values': [region]}
            ]
        )

        instance_types = [instance['InstanceType'] for instance in
response['InstanceTypeOfferings']]

        return jsonify(instance_types)

    except Exception as e:
        error_message = f"Error: {e}"
        return jsonify({"error": error_message})

@app.route(f'{high_prefix}/db_engine_types', methods=['POST'])
def list_db_engine_types():
    region = request.form.get('region')

    rds_client = boto3.client('rds',
                              region_name=region)
    response = rds_client.describe_db_engine_versions()

    engine_types = set()  # Using a set to store unique engine types

    for engine in response['DBEngineVersions']:
        engine_types.add(engine['Engine'])

    return jsonify(list(engine_types))


@app.route(f'{high_prefix}/db_engine_versions', methods=['POST'])
def list_db_engine_versions():
    region = request.form.get('region')
    engine = request.form.get('engine')

    rds_client = boto3.client('rds',
                              region_name=region)

    response = rds_client.describe_db_engine_versions(Engine=engine)

    mysql_versions = [version['EngineVersion'] for version in
response['DBEngineVersions']]

    return jsonify(mysql_versions)


@app.route(f'{high_prefix}/deployment_info', methods=['GET'])
def deployment_info():
    try:
```

```
        s3_client = boto3.client('s3', region_name=region_name)

        # List objects in the specified bucket
        response = s3_client.list_objects_v2(Bucket=highbucket_name)

        # Extract object keys from the response
        object_keys = []
        if 'Contents' in response:
            object_keys = [obj['Key'] for obj in response['Contents']]

        # Retrieve and extract content of each object
        object_contents = []
        for key in object_keys:
            obj_response = s3_client.get_object(Bucket=highbucket_name,
Key=key)
            obj_content = obj_response['Body'].read().decode('utf-8')
            object_contents.append(json.loads(obj_content))

        return jsonify({'objects': object_contents}), 200
    except Exception as e:
        error_message = f'Error listing S3 objects: {str(e)}'
        return jsonify({'error': error_message}), 500

if __name__ == '__main__':
    app.run(debug=True, port=5000, use_reloader=False)
```

Variables.tf:

```
variable "region" {
  description = "AWS region for deploying resources"
  type        = string
  default     = "ap-southeast-2"
}

variable "project_name" {
  description = "Name of the project"
  type        = string
  default     = "Xmopsteam2"
}

variable "vpc_name" {
  default     = "HighVPC"
  description = "Name of the Virtual Private Cloud (VPC)"
}

variable "vpc_cidr_block" {
  default     = "10.0.0.0/16"
  description = "CIDR block for the VPC"
}

variable "internet_gateway_name" {
  default     = "MainInternetGateway"
  description = "Name of the Internet Gateway"
}

variable "instance_name" {
  description = "Name of the EC2 or Lightsail instance"
  type        = string
}

variable "ami_id" {
  description = "ID of the AMI to use for the EC2 instance"
  type        = string
}

variable "instance_type" {
  description = "Type of EC2 instance to launch"
  type        = string
}

variable "min_instances" {
  description = "Minimum number of EC2 instances to launch"
  type        = number
}

variable "max_instances" {
  description = "Maximum number of EC2 instances to launch"
  type        = number
}

variable "desired_instances" {
  description = "Desired number of EC2 instances to launch"
  type        = number
}

variable "key_pair_name" {
  description = "Name of the key pair to associate with the EC2 or Lightsail instance"
  type        = string
  default     = "XMOPSTeamTwo"
}

variable "ssh_key" {
  description = "The public SSH key to access the instance"
  type        = string
}

variable "bundle_id" {
  description = "ID of the Lightsail bundle (applicable for Lightsail instances)"
  type        = string
  default     = "nano_3_2"
}

variable "blueprint" {
  description = "ID of the blueprint (e.g., wordpress, applicable for Lightsail instances)"
  type        = string
  default     = "wordpress"
}

variable "storage_size" {
  description = "Size of the root volume for the EC2 instance (in GiB)"
  type        = number
  default     = 8
}

variable "storage_type" {
  description = "Type of storage for the root volume (e.g., gp2, gp3)"
  type        = string
  default     = "gp3"
}

variable "db_engine" {
  description = "Engine type for RDS (e.g., MYSQL or PostgreSQL)"
  type        = string
  default     = "mysql"
}

variable "engine_version" {
  description = "Version of the database engine"
  type        = string
  default     = "8.0"
}

variable "instance_class" {
  description = "Instance type for the RDS instance"
  type        = string
  default     = "db.t2.micro"
}

variable "environment" {
  description = "Environment for RDS (Production or Development)"
  type        = string
  default     = "Development"
}

variable "identifier" {
  description = "Identifier for the RDS instance"
  type        = string
  default     = "amirrdsinstance1"
}

variable "db_username" {
  description = "Username for the RDS database"
  type        = string
  default     = "admin"
}

variable "db_password" {
  description = "Password for the RDS database"
  type        = string
  default     = "password"
}
```

Figure 5 Vaiables.tf

General Configuration

- **variable "region"**: Specifies the AWS region where the resources will be deployed, defaulting to "ap-southeast-2" to ensure resources are provisioned in the desired geographical location.

Project Identification

- **variable "project_name"**: Defines the name of the project, "Xmopsteam2", serving as a identifier for organizing and managing resources related to this specific project within AWS.

Virtual Private Cloud (VPC) Configuration

- **variable "vpc_name"**: Names the Virtual Private Cloud, "HighVPC", providing a virtual network dedicated to the project for deploying resources securely.
- **variable "vpc_cidr_block"**: Sets the CIDR block "10.0.0.0/16" for the VPC, determining the IP address range available for resources within the VPC.
- **variable "internet_gateway_name"**: Identifies the Internet Gateway, "MainInternetGateway", which connects the VPC to the internet, enabling communication between resources in the VPC and the outside world.

Instance Configuration

- **variable "instance_name"**: Establishes a generic name for an EC2 or Lightsail instance, offering a way to uniquely identify and manage instances.
- **variable "ami_id"**: Specifies the Amazon Machine Image (AMI) ID to use for EC2 instances, dictating the OS and software configuration.
- **variable "instance_type"**: Defines the type of EC2 instance to launch, impacting the computing, memory, and storage capabilities.
- **variable "min_instances", variable "max_instances", variable "desired_instances"**: Control the scale of deployment by setting the minimum, maximum, and desired counts of EC2 instances to ensure scalability and availability.

Key Pair and Access

- **variable "key_pair_name"**: Names the SSH key pair for secure access to the instance, with a default value indicating a project-specific naming convention.
- **variable "ssh_key"**: Holds the public SSH key to enable secure shell access to the instance, essential for administrative tasks.

Lightsail Configuration

- **variable "bundle_id"**: Identifies the Lightsail bundle, determining the instance's hardware specification, like CPU and memory.

- **variable "blueprint"**: Specifies the software setup for Lightsail instances, defaulting to a WordPress configuration for rapid web deployment.

Storage Configuration

- **variable "storage_size"**: Determines the size of the root storage volume for EC2 instances, impacting data storage capacity.
- **variable "storage_type"**: Selects the storage type for the root volume, with "gp3" offering improved performance and cost-efficiency compared to previous generations.

Database Configuration

- **variable "db_engine"**, **variable "engine_version"**: Define the RDS database engine and its version, crucial for compatibility and feature availability.
- **variable "instance_class"**: Specifies the compute and memory capacity of the RDS instance, affecting database performance.
- **variable "environment"**: Indicates the deployment environment (e.g., Development or Production), aiding in environment-specific configurations.
- **variable "identifier"**: Provides a unique identifier for the RDS instance, simplifying database management and reference.
- **variable "db_username"**, **variable "db_password"**: Set credentials for accessing the RDS database, ensuring secure and authenticated database interactions.

Chapter 6 Troubleshooting

Troubleshooting issues that arise during the deployment and operation of the XMOPS Accelerate platform involves a systematic approach to identify and resolve common problems related to the Flask application (app.py), the environment configuration, and the Terraform infrastructure provisioning process. This chapter outlines effective strategies for addressing common issues encountered across these components.

6.1 Flask Application (app.py) Troubleshooting

1. Authentication Issues:

- **Symptom**: Users cannot sign up, verify their account, or log in.
- **Resolution**: Ensure that the Amazon Cognito User Pool IDs and App Client IDs are correctly set in the environment variables. Verify that the Cognito service is correctly configured in the AWS Console.

2. CORS Errors:

- **Symptom**: Cross-Origin Resource Sharing (CORS) errors in the browser console when trying to interact with the API from a different origin.
- **Resolution**: Review the CORS configuration in app.py. Ensure the origins are correctly specified and that CORS(app) is properly configured to allow requests from the intended client origins.

6.2 Terraform Configuration Troubleshooting

1. Terraform Initialization and Plan Errors:

- **Symptom**: Errors during terraform init or terraform plan executions, such as provider initialization failures or configuration syntax errors.
- **Resolution**: Ensure all required Terraform providers are specified with the correct versions in the Terraform configuration files. Check for syntax errors in the .tf files and validate the configuration with terraform validate.

2. AWS Resource Provisioning Failures:

- **Symptom**: Resources fail to provision, with errors indicating insufficient permissions or unavailable resources.
- **Resolution**: Verify that the AWS credentials used have the necessary permissions for resource creation. Check the AWS region specified in the Terraform variables and ensure it supports the resources you're trying to provision.

6.3 Common Infrastructure Deployment Issues

1. Connectivity Issues with Deployed Instances:

- **Symptom**: Unable to connect to EC2 instances or access deployed applications.
- **Resolution**: Check the security group rules associated with the EC2 instances to ensure they allow inbound traffic on the required ports. For Lightsail instances, verify the instance's public IP and ensure no network ACLs or firewalls are blocking access.

2. Database Connection Errors

- **Symptom**: Applications fail to connect to the RDS database instance.
- **Resolution**: Confirm that the database endpoint and credentials in the application configuration match those of the deployed RDS instance. Ensure the RDS instance's security group allows inbound connections from the application's EC2 instances.

6.4 Debugging and Logs

1. Viewing Application Logs:

- Utilize the logging module configured in app.py to review application logs for errors or unexpected behavior. Adjust the logging level as needed to capture more detailed information during troubleshooting.

2. Terraform State Issues:

- **Symptom**: Terraform state conflicts or errors about state lock.
- **Resolution**: Use terraform state list to review the current state and terraform state rm to remove any stale resources. Resolve any state lock issues by manually releasing the lock if you're certain no other operations are in progress.

3. S3 Bucket Access Errors:

- **Symptom**: Errors related to listing, reading, or writing to the S3 bucket.
- **Resolution**: Verify that the S3 bucket name is correctly specified in the application and Terraform configurations. Check the IAM role or user permissions to ensure they include S3 read/write access.

6.5 File Not Found

- **Symptom**: "File Not Found" errors when the application tries to access local files or when users request specific resources that should exist.
- **Resolution**: Verify the file paths specified in your application are correct and relative to the correct base directory. Ensure all required files are included in the deployment package and not excluded by .gitignore or similar configurations. For web resources, check that static files are correctly served by Flask and accessible via the correct URLs.

6.6 HTTP Request Error

- **Symptom**: Errors during HTTP requests between the frontend and the backend, manifesting as failed fetch or XMLHttpRequests in the browser console.
- **Resolution**: Check the endpoint URLs for typos or incorrect paths. Ensure the backend service is running and accessible from the client making the request. For CORS issues, verify the CORS settings in your Flask application to ensure the request origin is allowed.

6.7 Import Error in Python

- **Symptom**: "ImportError" messages when starting the Flask application, indicating missing Python modules or incorrect imports.
- **Resolution**: Ensure all required modules are listed in your requirements.txt and have been installed with pip install -r requirements.txt. If using virtual environments, verify that the environment is activated. Check for typos in module names and consistency in case sensitivity on filesystems that are case-sensitive.

6.8 Stylesheet or CSS Errors

- **Symptom**: Web pages not rendering as expected, missing styles, or browser console errors related to CSS files not loading.
- **Resolution**: Confirm the <link> tags in your HTML templates point to the correct path for your CSS files. Ensure static file serving is correctly configured in Flask. Use the browser's network debugging tools to check if CSS files are requested from the correct URL and examine the server response.

6.9 JavaScript and DOM-related Errors

Symptom: JavaScript errors in the browser console, such as "Uncaught ReferenceError" or issues manipulating the DOM that prevent web pages from functioning correctly.

Resolution:

- **Script Loading Order**: Ensure that scripts are loaded in the correct order, especially if they have dependencies. Scripts that manipulate the DOM should be loaded after the DOM is fully loaded or executed in response to the DOMContentLoaded event.
- **Path to JS Files**: Verify the paths to JavaScript files are correct and that the server properly serves these files. Like CSS, use the network debugging tool to inspect the requests and responses for JS files.
- **JavaScript Syntax**: Review the JavaScript code for syntax errors or typos, especially in variable names or function calls. Modern IDEs and linters can help identify these issues before runtime.

- **External Libraries**: If using external libraries or frameworks, ensure they are correctly included and initialized in your project. For CDN resources, check the URLs and availability of those resources.

Chapter 7 FAQ (Frequently Asked Questions)

Q: What is XMOPS Accelerate, and why was it created?

A: XMOPS Accelerate is a centralized control plane developed to streamline and accelerate redundant tasks such as WordPress deployments, IAM roles and policy updates, and broader infrastructure provisioning for the Xmops team. It aims to enhance efficiency, security, and user experience in managing cloud infrastructure, reducing the time and effort required for these tasks.

Q: How does XMOPS Accelerate improve infrastructure deployment?

A: It automates the deployment of infrastructure components, including WordPress applications and various architectural setups like monolithic, distributed, and highly available systems. Automation ensures consistency, reliability, and speed in deployments.

Q: What authentication mechanism is used in XMOPS Accelerate?

A: XMOPS Accelerate utilizes Amazon Cognito for user authentication and authorization, incorporating secure login features and multi-factor authentication to ensure that only authorized team members can access and manage deployment options.

Q: How is security enhanced in XMOPS Accelerate?

A: Besides secure authentication via Amazon Cognito, XMOPS Accelerate employs IP whitelisting to limit access at the network level, ensuring that only designated IPs can access certain functionalities, thereby enhancing security.

Q: What features does the front-end interface of XMOPS Accelerate include?

A: The front-end interface offers a user-friendly dashboard for an overview of ongoing and completed deployments, quick access to deployment options and settings, and a comprehensive history of previous deployments to track progress and actions over time.

Q: Can you describe the types of deployments XMOPS Accelerate supports?

A: XMOPS Accelerate supports a variety of deployment scenarios, including:

- **Lightsail deployments**: For straightforward, managed instances with options for WordPress.
- **Monolith deployments**: Traditional single-instance deployments, suitable for simpler applications.
- **Highly Available deployments**: For critical applications requiring high availability across multiple instances and zones.

Q: What input fields are required for different deployment types?

A: Depending on the deployment type, required inputs can include AWS Region, instance type, AMI, key pair name, security group settings, storage configurations, database engine types, and version among others. These inputs enable customized and precise infrastructure provisioning according to project needs.

Q: What should I do if I encounter issues or errors with XMOPS Accelerate?

A: Start by consulting the troubleshooting chapter provided in the documentation, which addresses common problems and their solutions. If the issue persists, check the application logs and AWS CloudWatch for error messages that can provide more insight into the problem.

Q: What are the expected outcomes of using XMOPS Accelerate?

A: Teams can expect significant time and effort savings on infrastructure deployment, enhanced security measures, scalable and flexible architecture to support various deployment scenarios, and reliable, automated deployments.

Q: Will XMOPS Accelerate accommodate future expansion and updates?

A: Yes, the architecture of XMOPS Accelerate is designed to be scalable and flexible, making it well-suited to accommodate future expansions and the addition of new features or deployment options.

Chapter 8 Contact Information

For any inquiries, feedback, or if you require further assistance with XMOPS Accelerate, including its setup, deployment strategies, or general operation, please do not hesitate to reach out through the following channels:

- Email: Our primary communication channel for detailed inquiries and support requests. Contact us at swinabs@gmail.com for personalized assistance or with any questions you might have about XMOPS Accelerate.
- GitHub: For those interested in diving into the code, contributing improvements, or reporting issues, our GitHub repository is the perfect place to collaborate and contribute to the project's development. Visit our repository https://github.com/san-pok/XMPOS-ACCELERATE to explore the codebase, submit issues, or propose pull requests.
- Slack: Join our vibrant Slack community to connect with other XMOPS Accelerate users, share your experiences, and seek help from our network of experts. It's a great space for real-time discussions, troubleshooting help, and networking with peers. Access our Slack channel by joining through https://app.slack.com/client/T06FJLPT25D/C06HH7H9XB3

Our team is dedicated to supporting your journey with XMOPS Accelerate, from initial setup and deployment to optimizing your infrastructure management workflows. Whether you're encountering technical challenges, seeking advice on best practices, or looking to contribute to the project, we're here to assist. Reach out at any time, and we'll ensure you have the resources and support needed to achieve success with XMOPS Accelerate.

References

[1] Amazon Web Services, Inc. (n.d.) *AWS Documentation*. Available at: https://docs.aws.amazon.com (Accessed: 10-03-2024]).

[2] HashiCorp. (n.d.) *Terraform by HashiCorp*. Available at: https://www.terraform.io/docs (Accessed: 15-02-2024).

[3] Pallets Projects. (n.d.) *Flask Documentation*. Available at: https://flask.palletsprojects.com/en/2.0.x/ (Accessed:26-02-2024).

[4] Mozilla Foundation. (n.d.) *JavaScript | MDN*. MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript (Accessed:20-03-2024).

[5] Tailwind Labs. (n.d.) *Tailwind CSS Documentation*. Available at: https://tailwindcss.com/docs (Accessed: 25-03-2024).

[6] GitHub, Inc. (n.d.) *GitHub Documentation*. Available at: https://docs.github.com/en (Accessed: 20-03-2024).

[7] Microsoft. (n.d.) *Office Help & Training*. Available at: https://support.microsoft.com/en-us/office (01-04-2024).

[8] OpenAI. (n.d.) *ChatGPT: Optimizing Language Models for Dialogue*. Available at: https://openai.com/research/chatgpt (Accessed: 05-03-2024).

[9] Hosting a website. (n.d.) *Guide to host a wordpress website in amazon lightsail*. Available at:https://www.smashingmagazine.com/2023/02/host-wordpress-site-amazon-lightsail/ (Accessed:25-02-2024).