Bridgewater State University

Virtual Commons – Bridgewater State University

Honors Thesis

Undergraduate Honors Program

How do server-based architectures compare to serverless architectures in terms of development, deployment process, scalability, and cost-effectiveness?

Natalio F. Gomes, Undergraduate Researcher

Dr. John Santore, Thesis Advisor

Dr. Paul Kim, Computer Science Committee Member

Dr. Seikyung Jung, Computer Science Committee Member

Abstract

Applications are tested in developers' machines before deployment, since this is the standard practice that ensures the software is functioning as it should before deploying it to a specific environment. The deployment process consists of setting up the environmental requirements, infrastructure configuration on cloud or on-premises servers. Consequently, developers often run into the infamous "It works on my machine"[1] problem. This issue is defined by the fact that applications function correctly in development environments but fail during deployment to production.

Furthermore, scalability concerns and cost optimization are another factor that developers and architects must take into consideration. As user demand increases, the application must be able to scale based on the traffic without performance degradation to avoid unnecessary expense.

This Thesis will investigate the research question: "*How do server-based architectures compare to serverless architectures in terms of development, deployment process, scalability, and cost-effectiveness?"* This research aims to provide guided details on the planning and analysis, design, development, testing, and deployment of a resume analyzer web application using two distinct AWS architectural approaches:

1. **Server-Based Architecture**: Utilizing AWS EC2 (virtual servers), RDS (relational database), Route 53 (DNS management), and VPC (network isolation)
2. **Serverless Architecture**: AWS S3 (for static files), CloudFront (content delivery), AWS Cognito User Pools (Authentication), API Gateway (API management), Lambda (serverless compute), and DynamoDB (NoSQL database)

Table of Contents

Introduction

We often take for granted the websites that we use daily. When one is exposed to the complexity behind planning, designing, coding, testing and deploying a web application, it becomes clear that what we see as working fast or well isn't magic, but incredible engineering. For your machine (e.g.: computers, smart phones, iPads, etc..) to communicate with the website that you are currently accessing, there are a multitude of processes that permit your machine to send and receive data. Moreover, at the other end of your request there could be multiple servers which will be responsible for answering it. A *server* can refer to two related but different things, which can be confusing. In this thesis, server will mean server software—a program running on a machine that listens to client requests and responds using standard network protocols. This software must run on some underlying hardware, often called a physical server or server machine. A physical server, also "known as a bare-metal server or hardware server, is a computing machine that performs services such as running an operating system to provide networking resources to users or other computers."[2] Over the years, companies have deployed many of these physical servers around the world to host their server software and manage their computing resources. It is crucial to point out the complexity behind maintaining physical servers on premises, as it requires significant upfront investment, manual network configuration, ongoing security, etc. All in all, it is incredibly challenging to maintain physical servers.

In 2006, "[Amazon Web Services (AWS)] introduced Elastic Compute Cloud (EC2), an offering that allowed users to rent virtual computers to run their applications." [3] With this innovation, AWS sought to solve the physical server's maintenance challenge by providing a service that removes the problem of self-managing servers and worrying about security, scalability, manual hard drive configuration and troubleshooting.

As technologies rise to prominence, new problems follow. Cloud web servers provide the ability to manage unpredictable levels of traffic, cost optimization, and security are less of a worry compared to the traditional method. Nonetheless, there are still concerns regarding security, application infrastructure, cost optimization and more. Moreover, in “2014, Amazon introduced AWS Lambda, the first serverless platform” [4] which originated from the concept of serverless computing. Serverless computing, or simply ‘serverless’, “is a cloud computing model that offloads all the back-end infrastructure management tasks”. [5] The model which sought to address new concerns with the renting web servers.

I have recently encountered the critical decision of choosing between server-based architecture and serverless-based approach. This problem originates from the need to optimize costs, improving security and scalability. This research investigates and reaches a solid conclusion based on the information gathered by planning, analyzing, designing, implementing, testing and deploying a web application ultimately performs the same task. The application fetch jobs for the user based on their input and provide feedback on their resumes utilizing two different architectural approaches: server-based architecture and serverless-based architecture. This research thesis will compare aspects of architectural design, development, deployment complexity, operational costs, scalability patterns and maintenance requirements and performance analysis. This research is important because by implementing and comparing these 2 different architectures for the same application, this thesis will not only explore cloud architecture concepts but also document the development of each application, challenges encountered and the trade-offs between server-based and serverless architectures.

Related Works

In September 2023 a paper was published that analyzed a private enterprise located in United Kingdom named Deloitte Industry Insight. According to the authors of the paper “In 2019, [Deloitte] introduced a framework for comparing the total cost of ownership (TCO) for both serverless and server-based applications” [6, p. 2] This framework considered infrastructure, development, and maintenance. Since serverless-based architecture does not focus on handling the servers and the backend logic, the conclusion Deloitte arrived in the comparison is that while infrastructure costs can sometimes look similar, or perhaps even higher for serverless, overall TCO typically favors serverless due to reduced development effort and operations. In the light of exploring the duration of running tasks, the study discovered that a “serverless strategy may be least suited for long-running computational tasks” [6, p. 21], the reason for such conclusion is that according to AWS documentation, “Lambda is designed for short-lived compute tasks.” Furthermore, the paper explicitly stated that the architecture must cautiously design the infrastructure and deliberately select the absolute required technologies. Moreover, the cost-optimization must be taken into an account, and the use of serverless functions must be appropriately placed in order to fit the objective of the application.

In May of 2025 Research Gate published a research paper that “present[ed] a comparative implementation of server-based and serverless application models using Amazon Web Services (AWS), aiming to balance scalability, cost efficiency, and security.” [7] The paper showed that the authors Sarrah ElGazzar, Wagdy A. Aziz, and John N. Soliman implement both server and serverless-based workload using these technologies: MongoDB for database management, Express.JS(JavaScript framework) for backend business logic, React(JavaScript framework) for building dynamic and interactive user interface, and Node.js which is a runtime

environment that allows JavaScript to run on server-side commonly called the MERN stack. For their server-based architecture they used Elastic Computer Cloud (EC2) /Relational Databases Services (RDS); and for the serverless-based one they used Lambda/API Gateway/DynamoDB/S3. Furthermore, they empirically compare configuration complexity, scalability, performance, and cost. In terms of Infrastructure Management, they observed that deployment with EC2s required manual network and server configuration, whereas they didn't have to worry about these issues using the serverless architecture services such as AWS Lambda and API Gateway. In terms of scalability, they found limitations using server-based architecture, while the serverless handled the peak load much more efficiently. When considering cost, AWS charges the user for the EC2 instances that are active regardless of the activity, while Serverless-based architecture only incurred costs while it was actively running. In the matter of performance, EC2 performed well with low latency, however this changed as soon as there were traffic spikes, whereas the serverless functions performed well under traffic spikes though they showed higher latency when first started. When analyzing the deployment complexity, the authors documented the use of SSH access, server updates configuration, library installations and PM2 configuration. The serverless model on the other hand provided simplified deployment pipelines.

Mohammad Amman who is a senior Computer Science student at LUT University researched the difference between Serverless and Traditional Computing by building the same Python application on EC2 (traditional) and Lambda (serverless) and concluded that Lambda performed significantly better regarding short scaling, deployment duration, lower energy use, and cost for bursty workloads. "A [server-based] or traditional client-server model is significantly cheaper than serverless for sustained workloads, but serverless is more cost-

effective for variable workloads." [8] Moreover the thesis condenses these findings into a decision table located in page 37 summarizing the performance, cost, scalability, and productivity analysis. The conclusion that can be drawn from the study is that the "right choice" depends on the functionality of the application. Serverless is favored for variable traffic and EC2 preferred for compute-heavy demand.

The purpose of this paper

My thesis compares the software development lifecycle "which is a structured and iterative methodology used by development teams to build, deliver and maintain high-quality and cost-effective software systems." [9] process (planning and analysis, design, implementation, testing, deployment and maintenance) on a resume-analyzer web application using server and serverless-based architectures. The server-based infrastructure uses EC2 to deploy the application, using RDS for database management, Route 53, for Domain Name System configuration, and a Virtual Private Cloud where the database and the application reside. On the other hand, the serverless-based architecture uses CloudFront for content delivery, Cognito User pools for user authentication, API Gateway for handling users request, S3 for static content, Lambda for short workloads, and DynamoDB to store relevant user information. Unlike Deloitte's framework summary, this thesis will not focus on the enterprise development level, and it will document the hands-on, junior developer experience. In contrast to ElGazzar's MERN implementations, The Cognito User Pools will be responsible for handling the user authentication, and CloudFront will be responsible for global content delivery and S3 for object hosting. In short, my study documents the software development lifecycle for a web application that retrieves jobs for the user based on their input and provides constructive feedback, using both server-based and serverless-based architectures.

Tooling Section

This section will be exclusively dedicated to explaining the technological jargon mentioned throughout this paper.

Nginx short for engine x, according to the official website “HTTP web server, reverse proxy, content cache, load balancer, TCP/UDP proxy server, and mail proxy server.” [10]. However, in this application, its sole purpose is to be an HTTP Server.

Linux: Popular Server Web Server Operating System [11, Sec. What is Linux?]

WSGI – is the Web Server Gateway Interface. “It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.” [12]

AWS IAM: AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. With IAM, you can manage permissions that control which AWS resources users can access.[13]

IAM Role: An IAM *role* is an IAM identity that you can create in your account that has specific permissions.[14]

IAM policies define permissions for identities or resources in AWS. [15]

CloudWatch: Amazon CloudWatch monitors Amazon Web Services (AWS) resources and the applications you run on AWS in real time and offers many tools to give you system-wide observability of your application performance, operational health, and resource utilization.[16]

A **framework** is a collection of reusable software components that make it more efficient to develop new applications. [17]

Django is a Python web framework that encourages rapid development and clean, pragmatic design.[18]

Hypertext transfer protocol (HTTP) is a protocol or set of communication rules for client-server communication. When you visit a website, your browser sends an HTTP request to the web server, which responds with an HTTP response.[19]

Hypertext transfer protocol secure (HTTPS) is a more secure version or an extension of HTTP. In HTTPS, the browser and server establish a secure, encrypted connection before transferring data.[20]

Transport Layer Security, or **TLS,** is a widely adopted security protocol designed to facilitate privacy and data security for communications over the Internet. A primary use case of TLS is encrypting the communication between web applications and servers, such as web browsers loading a website. [21]

The **Domain Name System (DNS)** is the phonebook of the Internet.[22] When users access the website resume-analyzer.net or google.com. Web browsers interact through Internet Protocol (IP) addresses. DNS translates domain names to IP addresses so browsers can load Internet resources.

Amazon Route 53 is AWS' Domain Name System (DNS) web service.[23]

Virtual Private Cloud (VPC) is a virtual network that closely resembles a traditional network that you'd operate in your own data center. [23, Sec. Virtual Private Clouds] Imagine this as a big house for your applications.

Internet Gateway is the entry point that allows traffic from the public internet into a Virtual Private Cloud (VPC) [24, Sec. Internet gateways]

Subnet is a range of IP addresses in your VPC. [25, Sec. Subnets] Imagine this as a room inside the VPC where you can specify if they are private or public

A network access control list (NACL) allows or denies specific inbound or outbound traffic at the subnet level. [26]

A **route table** serves as the traffic controller for your virtual private cloud (VPC). Each route table contains a set of rules, called *routes*, that determine where network traffic from your subnet or gateway is directed. [27]

In AWS VPCs, **AWS Security Groups** are the rules of a security group control the inbound traffic that's allowed to reach the resources that are associated with the security group. The rules also control the outbound traffic that's allowed to leave them. [28]

An **elastic network interface (ENI)** is a logical networking component in a VPC that represents a virtual network card.[29]

An **Amazon Elastic Computer Cloud (EC2)** instance is a virtual server in the AWS cloud environment. [30]

A EC2 **security group** acts as a virtual firewall for EC2 instances to control incoming and outgoing traffic. Inbound rules control the incoming traffic to your instance, and outbound rules control the outgoing traffic from your instance. [31]

Amazon Relational Database Service (Amazon RDS) is a web service that makes it easier to set up, operate, and scale a relational database in the AWS Cloud. [32]

SSH (Secure Shell or Secure Socket Shell) is a network protocol that gives users -- particularly systems administrators -- a secure way to access a computer over an unsecured network. - https://www.techtarget.com/searchsecurity/definition/Secure-Shell [33]

Version control tracks every code change, providing a complete history and a single source of truth so teams can experiment safely, roll back[redeploying a prior version of the code] if needed, and collaborate effectively.[34]

Git is a distributed version control system, which means that a local clone of the project is a complete version control repository. These fully functional local repositories make it easy to work offline or remotely. Developers commit their work locally, and then sync their copy of the repository with the copy on the server. [35]

Every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. [35, p. Snapshots, Not Differences]

A content delivery network (CDN) is a network of interconnected servers that speeds up webpage loading for data-heavy applications.[36]

Amazon CloudFront is a **CDN** that “delivers your content through a worldwide network of data centers called edge locations. When a user requests content that you're serving with CloudFront, the request is routed to the edge location that provides the lowest latency, so that content is delivered with the best possible performance.” [37]

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. [38]

An **Amazon Cognito user pool** is a user directory for web and mobile app authentication and authorization.[39] Instead of developers coding their own authentication system, AWS Cognito User pools handle the authentication.

AWS Lambda is a computer service that runs code without the need to manage servers. Your code runs, scaling up and down automatically, with pay-per-use pricing. [40]

Application Programming Interface (API): is a set of rules or protocols that enables software applications to communicate with each other to exchange data, features and functionality.[41] In other words, is a programmatic way to GET, POST, DELETE, PATCH, and UPDATE any type of data. An example of this would be when a user submits a website review. The website API makes a POST request to the server with the user information.

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing APIs at any scale. [42]

Business logic is the custom rules or algorithms [set of steps to solve a problem] that handle the exchange of information between a database and user interface. [43]

A Django **model** is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.[44]

Object-relational mapping (ORM) is a way to align programming code
with database structures.[45]

Gunicorn 'Green Unicorn' is a Python WSGI HTTP Server for UNIX.[46]

A **shell script** is a file that contains one or more [terminal] commands. Shell scripts provide an easy way to carry out tedious commands, large or complicated sequences of commands, and routine tasks.[47]

Let's Encrypt is a Certificate Authority that provides free TLS certificates, making it easy for websites to enable HTTPS encryption and create a more secure Internet for everyone.[48]

Certbot is a free, open-source software tool for automatically using Let’s Encrypt certificates on manually-administrated websites to enable HTTPS.[49]

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easily readable and writable for humans and robots that are generated and parsed by computers. [50]

Amazon Simple Notification Service (Amazon SNS) is a service that provides message delivery from publishers (producers) to subscribers (consumers). Publishers communicate asynchronously with subscribers by sending messages to a *topic*, which is a logical access point and communication channel.[51]

Docker is an open-source platform that enables developers to build, deploy, run, update and manage containers.[52]

Containers are standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.[53]

Cross-origin resource sharing (CORS) is a browser security feature that restricts cross-origin [a request made from a web page's origin to a resource on a different origin] HTTP requests that are initiated from scripts running in the browser.[54]

Server-Based Architecture

Phase 1 - Planning and analysis:

This is the most important part of the software development lifecycle because it lays out the high-level architecture of the application, and predicts future errors and prevents the waste of development time. Given the project idea, the next step is to consider the technologies available to build this system. The result afterwards is the design of the application.

Phase 2 - Design:

This section consists of designing the architecture of the Django framework for the resume analyzer application. Django's framework infrastructure allows you to divide the application into multiple "apps". These apps can handle the different functionality of the application. Each app consists of:

- **urls.py** is responsible for redirecting the user to the correct business logic.
- **views.py** is responsible for verifying the authentication and the validity of the data being requested.
- **models.py** is an entity that interacts with the database by using built-in ORM

In the case of the Resume Analyzer, there are 4 apps highlighted with the red border in Figure 1 below. The apps below were architected by me to handle different responsibilities based on the user request. At the project level, the root URL configuration forwards all of the incoming traffic to the UserInterface app except urls that starts with "/admin/". The purpose of this routing is to delegate the administrative traffic to be redirected to Django's build-in admin site.

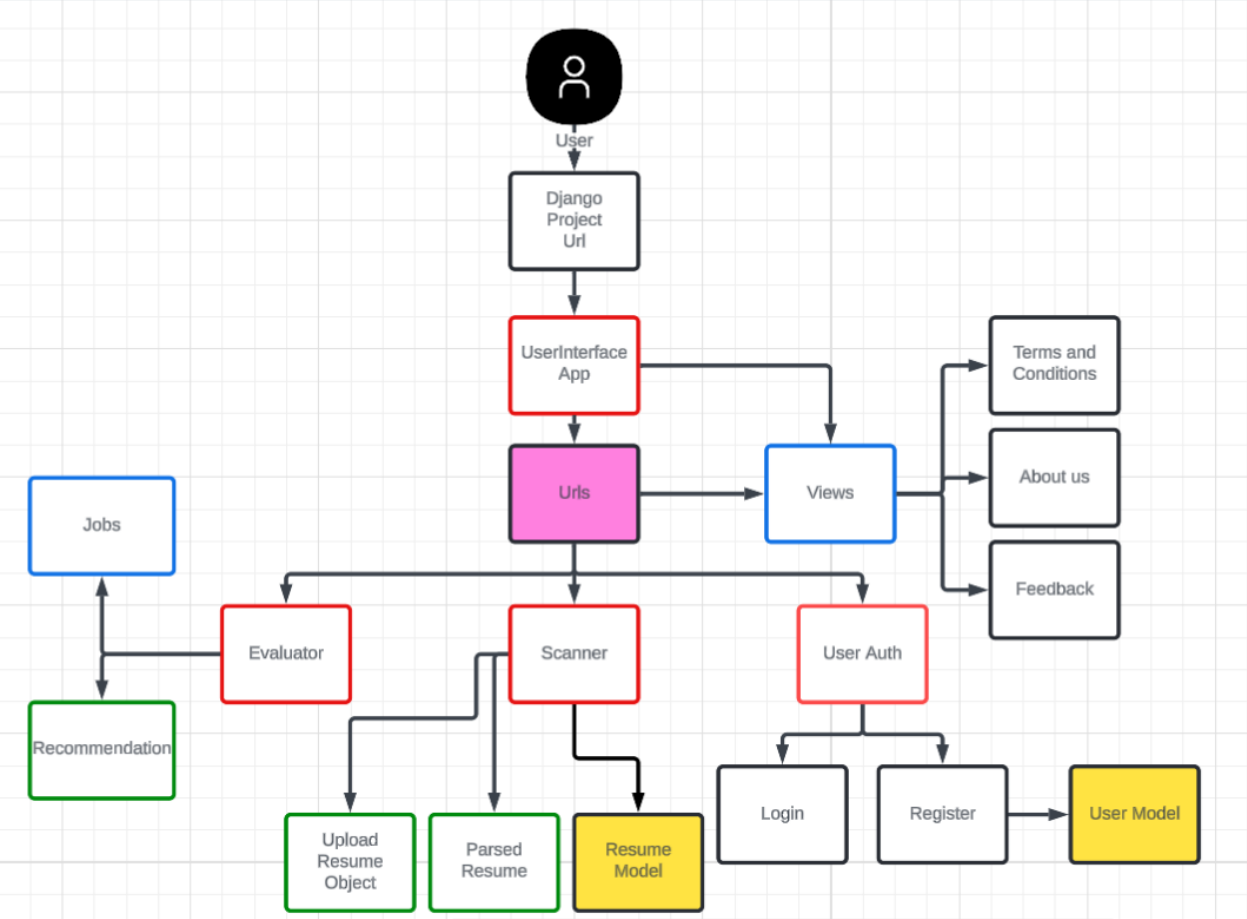

Figure 1: Server-Based architecture

As can be seen from the figure 1 above, the apps which have a red border are UserInterface, UserAuth, Scanner and Evaluator. Each of these holds a different responsibility to ensure that the business logic is separated in manner that is comprehensible to the developer. Their functionality is further explained below:

- **UserInterface:** Presentation app that serves public HTML pages such as Home, Terms and Services, About, and Feedback. Moreover, it handles form submissions and routes the user to the correct business logic based on the request.

- **UserAuth:** Control center for authentication and account management. Creates new user accounts using Django's built-in authentication libraries. Validates user credentials during login and manages user sessions.
- **Scanner**: Responsible for resume file processing: handles resume upload, file format validation, and text extraction. Integrate with Claude AI to analyze the extracted text and parse it into a comprehensible JSON file.
- **Evaluator**: Pulls job postings from the third-party API named RapidAPI based on the user's selected career field, experience level, and preferred job location. Uses Claude AI to compare the structured resume data against the jobs returned by RapidAPI. Produces an evaluation by identifying missing skills, education, certifications, and experience gaps. Generates recommendations to help the user update their resume and address those gaps.

Phase 3 - Development:

All code for Resume Analyzer was uploaded to a GitHub repository over SSH, using git version control to manage changes, isolate features, and enforce review before integration. SSH keys were provisioned for secure push/pull access. This ensured that only authorized personnel could modify the code. Moreover, a dedicated Python virtual environment, which is an isolated workspace was created to install the necessary dependencies for this application. The purpose of this environment is to guarantee the isolation of dependencies. Dependencies are libraries, frameworks, or other software components that a project relies on.[55] By default the virtual environments are deactivated. When activated, all of the libraries installed are only part of the Resume Analyzer project environment, rather than all of the other projects on the machine. Each logical component (UserInterface, Scanner, Evaluator, UserAuth) was implemented as a Django app and registered with the project's central configuration (INSTALLED_APPS). Registering an

app in this list allows Django to load its database models, URL routes, and views, and to include it in admin and migration workflows.

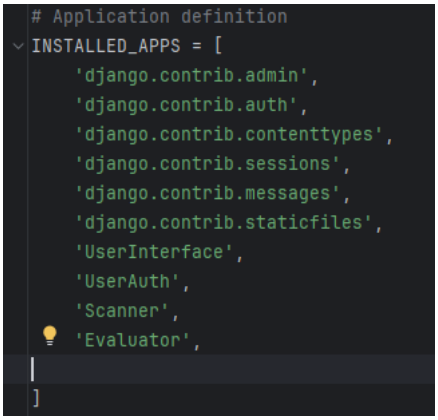

Figure 2: Django installed applications

There are 4 database tables:

- **UserProfile**: Uses Django's built-in User model to store additional profile data, including tracking how many resumes each user has uploaded.
- **Resume**: Central store for uploaded resumes, including upload timestamp, extracted text, jobs returned by the third-party API, and recommended skills generated during evaluation.
- **Feedback**: Enables users to submit website feedback by selecting a category, writing a message, and providing a rating.
- **Contact**: Saves user contact details along with their inquiries or support requests so administrators can follow up.

These models allow the system to associate analysis results and job matches with a specific authenticated user, rather than storing them anonymously. After each change in the model in the local development, migrations were generated and applied. Migrations are simply versioned database schema changes, which means database evolution is tracked as part of the codebase instead of being done manually and silently on a live database.

Phase 4 - Testing:

The goal of this phase is to verify that each module in the system behaves correctly in isolation and that core user-facing routes function as expected before deployment. The Resume Analyzer application tested the model/data integrity and the request/response behavior.

First, to test the model/data integration, a synthetic user account is created, a record is stored for that user, and then assertions are made to confirm that the model is linked to the correct user account, the content is stored accurately and the sensitive values such as the password are hashed and verifiable, not stored in plain text for security purposes. For an example see figure 3 below of the automated tests.

```python
class TestFeedbackModel(TestCase):
    def setUp(self):
        self.test_user_one = User.objects.create_user(username="user_one", first_name = "Jacob", last_name="Lii",
                                                      email="jacob_lii@email.com", password="mypassword")

        Feedback.objects.create(
            user=self.test_user_one,
            feedback="This is a great project.",
            rating=5
        )

        self.get_user_one = Feedback.objects.get(pk=1)

    def test_correct_user(self):
        self.assertEqual(self.get_user_one.user_id, second: 1)

    def test_feedback_field(self):
        self.assertEqual(self.get_user_one.feedback, second: "This is a great project.")

    def test_rating_field(self):
        self.assertEqual(self.get_user_one.rating, second: 5)

    def test_user_creation(self):
        self.assertEqual(self.get_user_one.user.username, second: "user_one")
        self.assertEqual(self.get_user_one.user.first_name, second: "Jacob")
        self.assertEqual(self.get_user_one.user.last_name, second: "Lii")
        self.assertEqual(self.get_user_one.user.email, second: "jacob_lii@email.com")
        self.assertTrue(check_password( password: "mypassword", self.get_user_one.user.password) )
```

Figure 3: Unit Tests for the Feedback feature

Second, the test class exercises the request/response cycle for several public-facing and authentication-related views: the homepage, about, feedback submission, login, and registration page. For each route, two properties are checked:

- **HTTP status**: The response should return 200 OK. This confirms that the route is registered, that the corresponding view executes without error, and that template rendering succeeds.
- **Template resolution**: Using assertTemplateUsed, the test confirms that each view returns the correct HTML template (index.html, about-us.html, feedback.html, login.html,

register.html). That proves that URL routing → view → template pipeline is wired correctly.

```python
class TestViewFunctions(TestCase):
    def setUp(self):
        self.client = Client()
        self.index_view = self.client.get(reverse('home'))
        self.about_view = self.client.get((reverse('about')))
        self.feedback_page_view = self.client.get(reverse('feedback_page'))
        self.login_page_view = self.client.get(reverse('login'))
        self.register_page_view = self.client.get(reverse('register'))


    def test_views_status(self):
        """By accessing a specific name view, what is the response status"""
        self.assertEqual(self.index_view.status_code, second: 200)
        self.assertEqual(self.about_view.status_code, second: 200)
        self.assertEqual(self.feedback_page_view.status_code, second: 200)
        self.assertEqual(self.login_page_view.status_code, second: 200)
        self.assertEqual(self.register_page_view.status_code, second: 200)


    def test_views_templates(self):
        """assertTemplateUsed checks the view function template"""
        self.assertTemplateUsed(self.index_view, template_name: 'index.html')
        self.assertTemplateUsed(self.about_view, template_name: "about-us.html")
        self.assertTemplateUsed(self.feedback_page_view, template_name: "feedback.html")
        self.assertTemplateUsed(self.login_page_view, template_name: "login.html")
        self.assertTemplateUsed(self.register_page_view, template_name: "register.html")
```

Figure 4: Unit Testing for Redirection

Phase 5 - Deployment:

The process of successful deployment requires knowledge of the deployment environment. By studying for AWS Solutions Architect Associate, I learned about many services which AWS offers. There are a variety of ways to combine these services to deploy a system. For

this research, the following Figure 5 showcases the architecture of the AWS cloud for the server-based deployment:

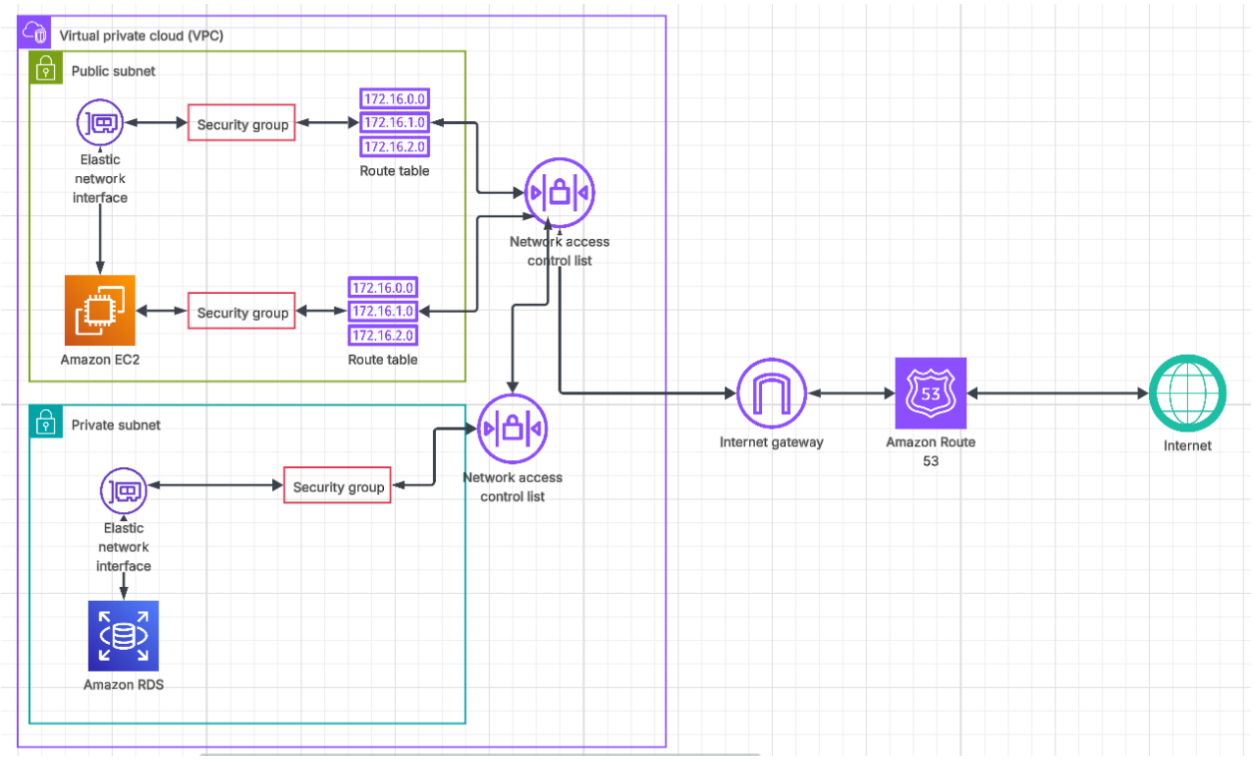

Figure 5: Resume Analyzer Cloud Architecture

The public domain name(resume-analyzer.net) is managed in **Route 53**. DNS A-records are mapped to the EC2 instance's public IP. TLS certificates are provisioned via Encrypt/Certbot that is installed at the Nginx layer so that end users access the site over HTTPS. This protects the user data in transit with Transport Layer Security (TLS). Route 53 maps the domain resume-analyzer.net to 54.123.45.67.

The request will access the server with the **public IP address 54.123.45.67** by going through the following steps:

1. **Internet Gateway (IGW)** - Entry point to the Virtual Private Cloud (VPC)
2. **VPC boundary** - Enters the AWS Virtual Private Cloud
3. **Network Access Control Lists (NACL):** Stateless subnet-level firewalls that do not track traffic state, requiring explicit inbound and outbound rules for both request and response traffic.
4. **Public Subnet** - Where the **EC2** instance (web server) lives and by default it comes with a private IP address (Ex: 10.0.1.0/24)
5. **Route Table** - Evaluates routing rules (traffic is destined for local subnet)
6. **Security Group (Inbound traffic)** – Stateful instance-level firewall that remember the traffic, if traffic is allowed in, it is allowed out.
7. **ENI (Elastic Network Interface)** - Virtual network card attached to EC2

    Example of the Internet Protocol:

    - Public IP: 54.123.45.67
    - Private IP: 10.0.1.10

8. **EC2 Instance** - Where the Django application is hosted

A Linux-based EC2 instance runs the Django application using Gunicorn as the WSGI server and Nginx as a reverse proxy. Nginx terminates HTTP/HTTPS requests from the public internet, applies security headers, and forwards only valid traffic to Gunicorn on localhost. Gunicorn then executes the Django code.

9. **Django application** – Located in the EC2 initiates database connection to RDS endpoint (e.g., db. resume-analyzer. internal:5432)
10. **Security Group (outbound traffic)** - ALLOWS outbound traffic to Database security Group on port 5432
11. **Route Table** - Evaluates destination: 10.0.3.10 is within VPC (10.0.0.0/16), routes **locally**
12. **NACL (outbound from public subnet)** - Allows traffic to flow out of the Subnet
13. **NACL (inbound to private subnet)** - ALLOWS traffic from the public subnet
14. **Private Subnet** - Where RDS lives (10.0.3.0/24)
15. **Security Group (DB-SG)** – ALLOWS traffic flow on port 5432. ONLY from EC2 Security Group
16. **ENI (RDS)** - Elastic Network interface with private IP: 10.0.3.10
17. **RDS PostgreSQL** - Database fetches the data

The persistent application data lives on Amazon RDS (relational database service). In order for the EC2 instance to talk to the database, there must be IAM role that allows the web server to access the database credentials stored in the SSM parameter Store. The IAM role must contain a policy that allows the EC2 instance to access the database. This prevents arbitrary unintended users from connecting directly to the database. After the traffic bypasses all of the layers of security, the RDS returns the data, and the data traces all the way to the user in reverse order by going through the same components.

Phase 6 - Maintenance:

This process consisted of applying Operating System and dependency updates on the EC2 instance on an ongoing basis, verifying that the service is reachable and responding correctly (via Nginx, Gunicorn logs and HTTP health checks), and analyzing the logs located in CloudWatch logs.

Continuous Integration/Contiguous Delivery (CI/CD) Pipeline Method on Server-Based architecture:

A CI/CD pipeline is an automated workflow that is responsible for the software delivery process. CI/CO is defined as a "series of established steps that developers must follow in order to deliver a new version of software." [56] The technology chosen for the CI/CD pipeline for this project was GitHub Actions, which is a"platform that allows you to automate your build, test, and deployment pipeline."[57]. GitHub Actions expects workflow files to be located in the directory: ".github/workflows/filename.yml" The workflow file shown in figure 6 was configured to run the unit tests and, if they pass, it will deploy the program to production. For a seamless pipeline integration, both of the EC2 instances must have *git* installed and connected to the individual's GitHub account and the EC2 instance security group must allow the connection to the GitHub repository, and once the connection is made, the EC2 web server will use SSH to connect to the GitHub repository to pull the changes. This simply means, it will replace the current application version with a specified one. Afterwards, using a shell script file, the web server it will automatically activate the virtual environment and install all the dependencies specified on the requirements.txt file, run database migrations "are controlled sets of changes developed to modify the structure of the objects within a relational database." and finally it restarts the Gunicorn-backed systemd service.[58]

```yaml
name: Deploy to EC2

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production

    steps:
    - name: Simple SSH Test
      uses: appleboy/ssh-action@v1.0.0
      with:
        host: ${{ secrets.EC2_HOST }}
        username: ${{ secrets.EC2_USERNAME }}
        key: ${{ secrets.EC2_SSH_KEY }}
        script: |
          echo "🚀 Starting deployment..."
          echo "Current date: $(date)"
          echo "Current user: $(whoami)"
          echo "Current directory: $(pwd)"

          # Activate virtual environment (adjust path as needed)
          source prodVirt/bin/activate

          # Navigate to your project directory (adjust path as needed)
          cd /home/ec2-user/ATP-PROJECT

          # Pull latest code
          git pull origin main

          # Install/update dependencies
          pip install -r requirements.txt

          # Run Django commands
          python3 manage.py makemigrations
          python3 manage.py collectstatic --noinput
          python3 manage.py migrate

          cd ..
          ./reload_services.sh

          echo "✅ Deployment completed successfully!"
```

Figure 6: GitHub Actions workflow file

Serverless-Based Architecture

Phase 1 - Planning and analysis:

Similar to the server-based architecture, this phase consisted of thorough design multiple times. This process consisted of understanding how the AWS services involved worked and how to use them to accomplish the desired goal.

Phase 2 - Design:

The purpose of this phase is to design the architecture of the user workflow of the website. Unlike a Server-based architecture, there are no web frameworks involved during the development or deployment process. For this reason, the functionalities such as user authentication, resume parsing, job featuring, and recommendation consisted of the configuration of an event-driven architecture (EDA) which "is a software design model built around the publication, capture, processing and storage of events." [59] This design uses the application events to trigger and communicate between different components. In the resume analyzer serverless-architecture there are a multitude of AWS Services that handle different functionalities shown in Figure 7 below.

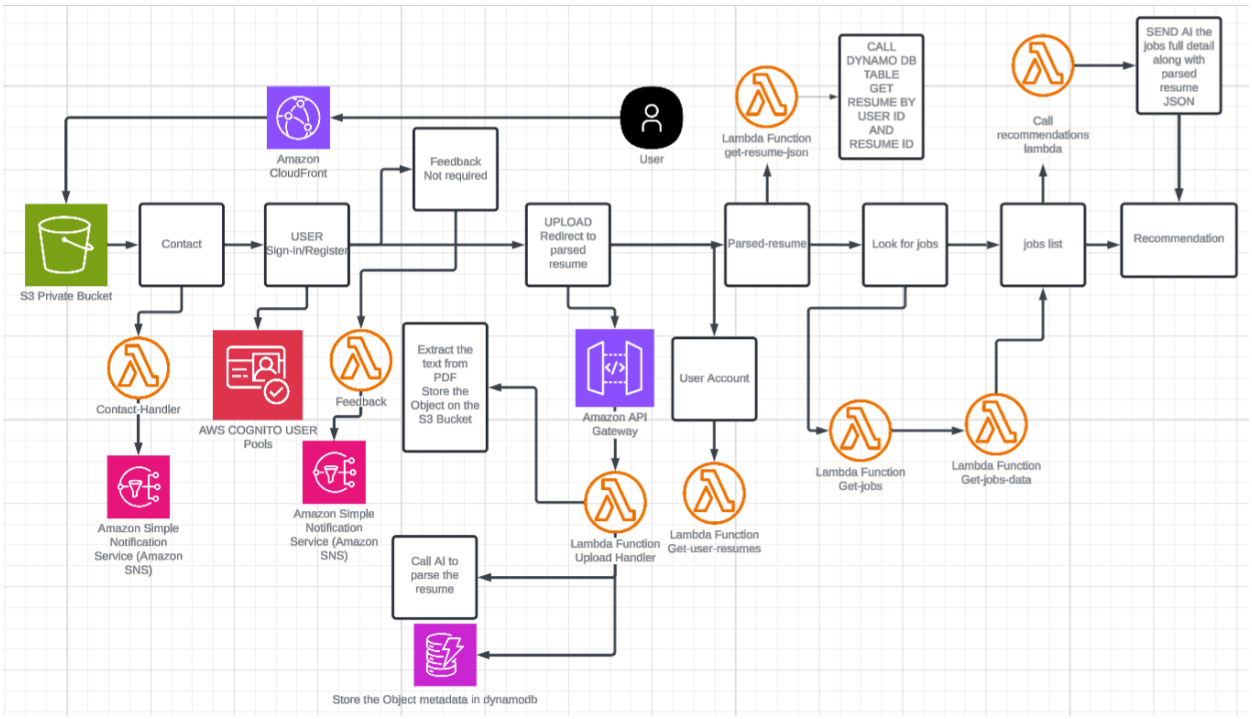

Figure 7: Serverless Application Architecture

To comprehend the cloud architecture of the serverless-based application, let's assume that the User, using the public internet accesses the web application via the URL: https://d390ayroow0bsc.cloudfront.net/. The workflow is as follows:

1. **CloudFront** – Content Delivery Network responsible for serving the users the static content stored in a private S3 Bucket via origin access control which "secures S3 origins by permitting access to the designated distributions only"[60]
2. **S3 (website bucket)**– Contains the public static websites files which are Html, Cascading Style Sheets (CSS) and JavaScript. Each of these files performs a different function. While CSS is responsible for the User Interface design, HTML is responsible for the

website markup and page structure and JavaScript is the backbone of the functionality, handling forms validation, calling the APIs, and updating Document Object Model.

3. **S3 (Data Bucket)** – Stores user files: original PDFs, extracted text, and parsed data in JSON format.
4. **Cognito User Pools** – Handles user sign-up/sign-in, issues temporary tokens.
5. **API Gateway (REST)** – Validates tokens via a Cognito authorizer and redirects user requests to the correct lambda function.
6. **AWS Lambda** – Serverless functions that compute each business capability. In the Case of resume-analyzer there are 8 lambda functions:
    - *upload-handler:* Accepts a user's resume upload, validates file type/size, extracts the text from the resume, stores it in the S3 bucket, calls Claude AI for resume parsing and creates a DynamoDB "resume" record with relevant user information such as the resume number and path in the S3 Bucket.
    - *get-user-resumes*: fetches all the user resume data on the S3 bucket
    - *get-resume-json*: Fetches individual parsed resume data
    - *get-Jobs*: Calls a Third-party API to get jobs based on the user location preference, career field and experience level
    - *get-job-data*: Gets the Jobs data located in the S3 Bucket
    - *recommendations-handler*: Calls Claude AI to perform recommendations based on the user resume and jobs fetched.
    - *feedback-Handler:* Calls the Simple Notification Service and sends the form with the user feedback.

- *Contact-Handler:* Calls the Simple Notification Service and sends the form with the user inquiry.

7. **DynamoDB** – AWS's native NoSQL database. Stores user profiles, resume metadata, evaluation results, job matches.
8. **SSM Parameter Store**– Contains the sensitive credentials of the 3rd party API keys.
9. **CloudWatch (Logs/Metrics/Alarms)** – Central logging for Lambdas and API Gateway access logs.

Phase 3 – Development

The development phase of this application was far different than server-based architecture. To illustrate, this architecture does not need GitHub for version control; instead, I used AWS Command Line Interface and AWS Console to deploy the code files to a versioned S3 Bucket, which keeps track of file versions uploaded. Moreover, the development of functionality such as job search, resume parsing, and recommendation was coded in the Python programming language in a separate directory, as can be seen in Figure 8.

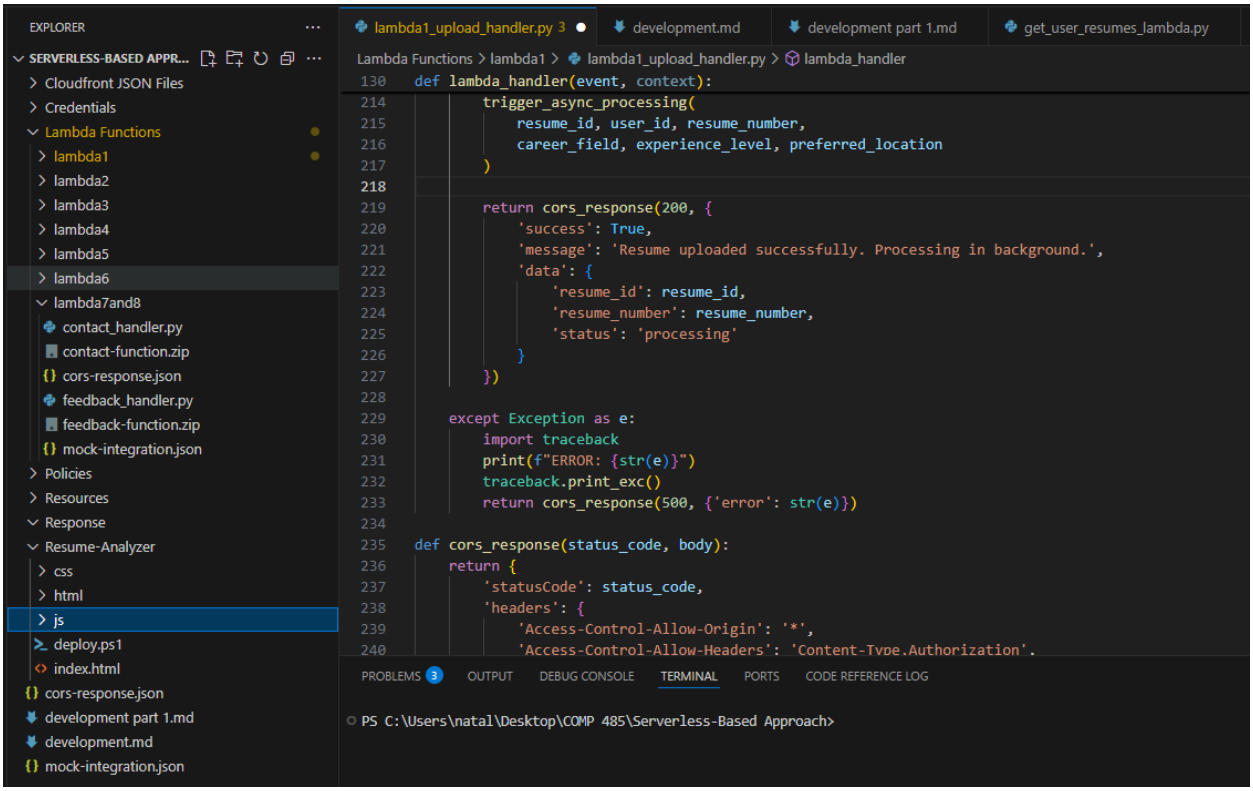

Figure 8: Lambda Function coded in python programming language

Since the application relies on external libraries to perform certain tasks a lambda layer must be created for the application to access these libraries. A Lambda layer “is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.” [61] Lambda functions run on Amazon Linux 2 according to the official AWS documentation [62. Sec. Supported runtimes] and therefore to ensure the libraries run successfully, the external Python libraries were installed with Docker commands locally on a dedicated directory in order to run as is, regardless of the environment. Afterwards the directory is compressed into a zip file to be deployed to AWS. The local authentication setup consisted of creating a Cognito identity provider, followed by an app client, which is “a configuration within

a user pool that interacts with one mobile or web application that authenticates with Amazon Cognito.” [63]

Phase 4 - Testing:

The testing phase consisted of two steps: Unit testing of the Lambda Function and simple website workflow testing. Lambda functions were tested on the developer machine to ensure the functionalities worked closely to a real cloud environment using the unit tests library shown in figure 9.

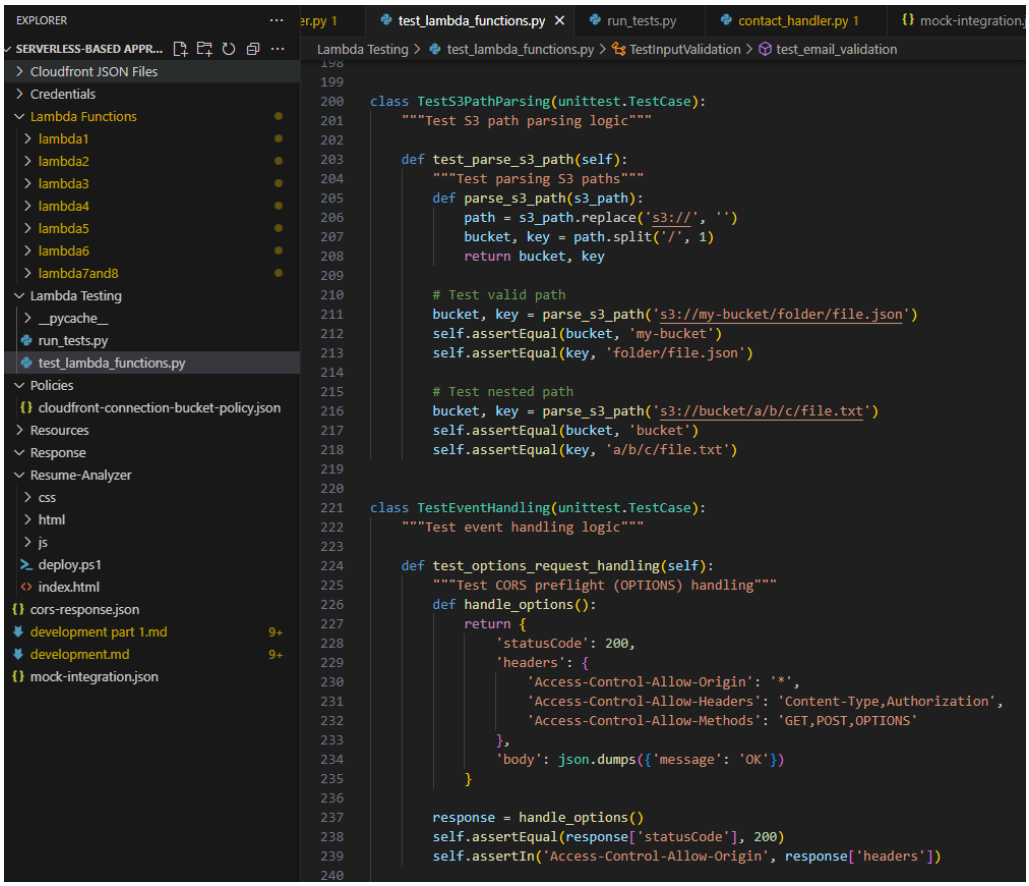

Figure 9: Lambda Function Unit Testing

Phase 5 - Deployment:

This phase differs significantly from the server-based deployment, since this consisted of uploading static files to the S3 bucket and integrating between the S3 bucket and CloudFront to serve the static files within the bucket to the users. This process involves allowing CloudFront to access the S3 bucket using Origin Access Control. Next, a CloudFront Distribution was created to serve the user the correct files. All S3 Buckets contain a Bucket Policy which "secures access to objects in bucket, so that only users with the appropriate permissions can access them." [64] to ensure the CloudFront distribution gets the files, the S3 Bucket policy was updated to allow the CloudFront distribution to access the content. Furthermore, the distribution is deployed, and now users can access the content via https://d390ayroow0bsc.cloudfront.net/ URL.

To handle the user file upload, API gateway endpoints were necessary to handle the backend APIs. The endpoint receives the user upload information, and it will pass the data to the correct lambda function for processing. This process was quite arduous due to the difficult configuration process. To enumerate the procedure, first, a REST API was created, followed by the creation of Cognito Authorizer, which validates the token from Cognito. "The API call succeeds only if the required token is supplied and the supplied token is valid." [65] Second, an API resource must be created. "A collection of HTTP resources and methods that are integrated with backend HTTP endpoints, Lambda functions, or other AWS services." [66] These are the URL paths (e.g. /upload, /resumes), and each of these paths must contain a method. "Each API resource can expose one or more API methods that have unique HTTP verbs supported by API Gateway." This is illustrated in Table 1 below. Third, to allow CloudFront to call the API Gateway from the browser, Cross-Origin Resource Sharing (CORS) must be configured. Fourth, for the API Gateway to call the lambda functions, each function must explicitly allow the API

Gateway to invoke them. After these changes, the API Gateway must be deployed to a stage, which is “a named reference to a deployment.” [67]

Method	Path	Lambda Function	Authentication	Purpose
POST	/get-jobs	get-jobs	Required	Fetch jobs for the user and stores it in the S3 Bucket
GET, OPTIONS	/get-jobs-data	get-jobs-data	Required	Gets the jobs from the S3 Bucket
GET, OPTIONS	/get-recommendations	recommendations-handler	Required	Makes recommendations
GET, OPTIONS	/get-resume-json	get-resume-json	Required	Get individual resume from the s3 bucket
GET, OPTIONS	/get-user-resumes	get-user-resumes	Required	Fetch resumes in the S3 Bucket
POST	/submit-contact	Contact-handler	OPTIONAL	Submit contact
POST	/submit-feedback	Feedback-handler	Required	Submit feedback
POST	/upload	upload-handler	Required	Handles he user resume upload

Table 1: API Resources and Lambda Functions they are associated with.

Phase 6 - Maintenance:

The maintenance consisted of monitoring the user request response via a browser's inspection tool. The purpose of this is to verify that the website is responding accurately based on the request and analyzing logs via CloudWatch logs. After each change to the static files, the CloudFront cache was invalidated in order to serve the latest version of the files.

Continuous Integration/Contiguous Delivery (CI/CD) Pipeline Method on Serverless-Based architecture:

In contrast to the CI/CD Pipeline workflow of the server-based architecture, where GitHub Actions was responsible for handling the automatic deployment, after all the necessary changes were made to the static files in this Serverless-based architecture, they were uploaded using a simple PowerShell configuration file illustrated in figure 10 below. The implementation of new features consisted of pushing new changes with AWS CLI commands.

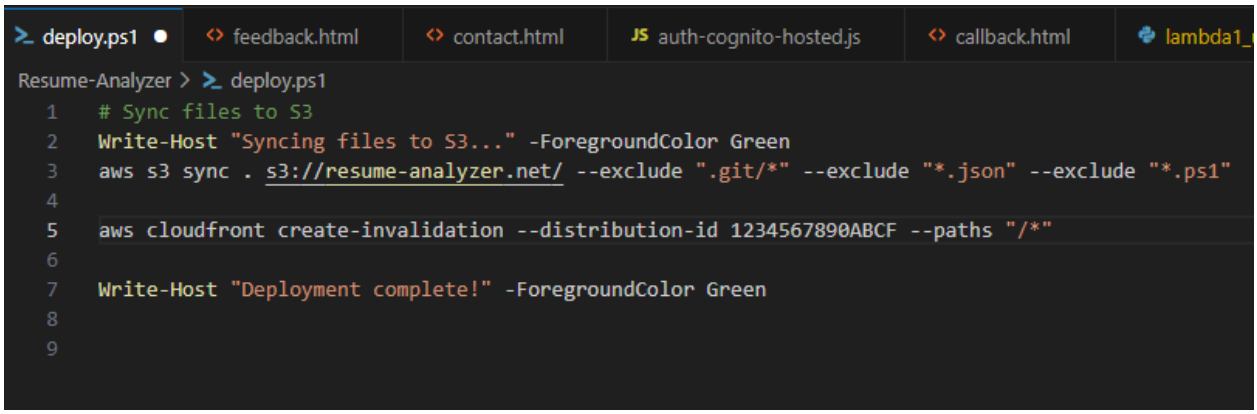

Figure 10: PowerShell Script to upload Code

Conclusion

After using the software development life cycle process to build both server-based and serverless-based architecture, I concluded that both web applications work similarly from a regular user point-of-view. However, the software is configured, coded, tested and deployed significantly different.

In terms of Development Complexity, the server-based architecture used Django framework, which facilitated the implementation of additional features whereas the serverless-based architecture required far more complexity from a developer point of view, due to extensive configuration when adding new resources and the testing to ensure the components are communicating as expected. This being stated, if developer time continues to be the greatest expense for software business, server-based architecture is the optimal solution. The conclusion drawn from this differentiates from the study done by Deloitte in 2019 which concluded that the total cost of ownership typically favors serverless due to reduced development effort and operations.

With regard to deployment process, server-based deployment is initially harder to configure, due to extensive configuration between GitHub Actions and AWS Resources to ensure proper environment configuration, however, this is done only once. On the other hand, the serverless-based deployment consisted of using AWS CLI and Console to deploy new changes.

The CI/CD pipeline implementation showed that GitHub Actions properly configured, seamlessly integrated with EC2 deployments, whereas serverless deployments relied on a simple PowerShell script for static file updates, however deployment of changes and new resources of Lambda function, API Gateway, DynamoDB, Cognito User Pools and CloudFront remained

manual. When considering Cost-Effectiveness, I noted a drastic difference between the two architectures. The server-based implementation consistently costs approximately $95 per month, maintaining operation 24/7 and availability regardless of actual usage. This included EC2 instance charges, RDS database costs, and data transfer fees. Conversely, the serverless architecture monthly bills as of November 7th, 2025, cost incurred a total of $0.0002. This comparison is further explored in the tables below.

Month	Server-based Architecture Cost	Serverless Architecture Cost
May of 2025	$82.20	-
Jun 2025	$101.25	-
Jul 2025	$105.94	-
August 2025	$93.94	-
September 2025	$86.19	-
October 2025	$99.26	-
November 2025	$57.09	$0.432

Table 2: Timeline of Cost difference of server-based and server-based architecture

AWS SERVICE	Total Cost	Percentage	Purpose
Relational Database Service (RDS)	$211.01	37.1%	PostgreSQL database for user data, resumes, jobs
EC2-Other	$164.31	28.9%	Data transfer, EBS volumes, snapshots
VPC	$97.38	17.1%	Network infrastructure, NAT Gateway
EC2-Instances	$32.31	5.7%	Virtual server running Django application
Domain Registration	$32.00	5.6%	resume-analyzer.net domain (annual)
AWS Certificate Manager	$15.00	2.6%	SSL/TLS certificates
Route 53	$5.56	1.0%	DNS management

AWS Secrets Manager	$4.86	0.9%	Storing API keys securely
AWS WAF	$4.79	0.8%	Web Application Firewall
Tax	$1.55	0.3%	AWS service tax
TOTAL	$568.77	100%	

Table 3: Server-Based Architecture Costs

AWS SERVICE	Total Cost	Percentage	Purpose
API Gateway	$0.001925	96.1%	REST API endpoints
DynamoDB	$0.000058	2.9%	NoSQL database for metadata
CloudFront	$0.000001	0.1%	Content delivery network
Lambda	$0.00	0%	Serverless compute (within free tier as of this writing)
S3	$0.43	0.1%	Static website hosting
Cognito	$0.00	0%	User authentication (within free tier)
CloudWatch	$5.56	0%	Logging and monitoring (within free tier)
TOTAL	$0.432	100%	-

Table 4: Serverless-Based Architecture Costs

Maintenance requirements differed between architectures. The server-based deployment demanded ongoing attention due to Operating System security patches, dependency updates, and monitoring Nginx/Gunicorn logs through CLI. These tasks, while manageable, required consistent administrative overhead. The serverless architecture eliminated server maintenance entirely. Maintenance focused instead on monitoring Lambda execution logs, and invalidating

CloudFront cache after updates—maintenance tasks were less frequent and less complicated than traditional server administration.

For the Resume Analyzer application, in terms of variable traffic and cost-sensitivity, the serverless architecture demonstrates clear superiority. The 99% cost reduction, automatic scaling, and elimination of server maintenance. The optimal architecture ultimately depends on specific application requirements, traffic patterns, performance needs, required database, etc.

Future Work

So far, this thesis explored a small-scale project with low traffic, however when considering a large-scale project with millions of users world-wide and the need to deploy new features regularly, a CI/CD pipeline which ensures the services are working accurately in a specific environment before pushing the code to the production environment. Furthermore, the User Interface and security must also be thoroughly tested. The future of resume-analyzer consisted of implementing a robust infrastructure to handle the CI/CD pipeline to deliver fast, safe and consistent changes.

References

[1] J. A. Pardo, “Sometimes, the ‘it works on my machine’ statement is a red flag for inadequate testing practices,” *Medium*, Dec. 04, 2023. Available: https://medium.com/@josetecangas/but-it-works-on-my-machine-cc8cca80660c. [Accessed: Nov. 01, 2025]

[2] D. Gohel, “What is a Physical Server? A Comprehensive Guide for 2025,” *Cantech*, Oct. 30, 2025. Available: https://www.cantech.in/blog/what-is-physical-server/

[3] S. Susnjara and I. Smalley, “Cloud Computing,” *IBM Think*, 2025. Available: https://www.ibm.com/think/topics/cloud-computing

[4] S. Susnjara and I. Smalley, “Serverless,” *IBM Think*, 2025. Available: https://www.ibm.com/think/topics/serverless

[5] S. Susnjara, “Public cloud use cases,” *IBM Think*, Jul. 22, 2025. Available: https://www.ibm.com/think/topics/public-cloud-use-cases. [Accessed: Nov. 01, 2025]

[6] G. Arora, W. Choi, N. Jethi, B. K. Sahoo, and Deloitte, “Determining the total cost of ownership: comparing serverless and server-based technologies,” Sep. 2023. Available: https://d1.awsstatic.com/psc-digital/2023/gc-300/deloitte-tco-mod/determining-the-total-cost-of-ownership.pdf. [Accessed: Nov. 01, 2025]

[7] S. ElGazzar, W. A. Aziz, and J. N. Soliman, “Implementing AWS Server-based and Serverless Application Model,” May 2025. Available: https://www.researchgate.net/publication/392075001_Implementing_AWS_Server-based_and_Serverless_Application_Model. [Accessed: Nov. 01, 2025]

[8] M. Amman, “Serverless vs. Traditional Computing: A Multi-Dimensional Comparative Analysis,” Bachelor’s thesis, Lappeenranta–Lahti University of Technology LUT, 2025.

Available:

https://lutpub.lut.fi/bitstream/handle/10024/170251/Thesis_Mohammad_Amman_Final.pdf.

[Accessed: Nov. 01, 2025]

[9] G. Jackson, M. Kosinski, and J. Holdsworth, "SDLC," *IBM Think*, Nov. 17, 2025. Available: https://www.ibm.com/think/topics/sdlc. [Accessed: Nov. 28, 2025]

[10] "nginx." Available: https://nginx.org/en/

[11] "What is Linux? - Linux.com," *Linux.com*, Aug. 26, 2022. Available: https://www.linux.com/what-is-linux/

[12] "What is WSGI? — WSGI.org." Available: https://wsgi.readthedocs.io/en/latest/what.html

[13] "What is IAM? - AWS Identity and Access Management." Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

[14] "IAM roles - AWS Identity and Access Management." Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html

[15] "Define custom IAM permissions with customer managed policies - AWS Identity and Access Management." Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_create.html

[16] "What is Amazon CloudWatch? - Amazon CloudWatch." Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html

[17] "What is a Framework? - Framework in Programming and Engineering Explained - AWS," Amazon Web Services, Inc. Available: https://aws.amazon.com/what-is/framework/

[18] "Django," Django Project. Available: https://www.djangoproject.com/

[19] “HTTP vs HTTPS - Difference Between Transfer Protocols - AWS,” Amazon Web Services, Inc. Available: https://aws.amazon.com/compare/the-difference-between-https-and-http/

[20] Cloudflare, “What is Transport Layer Security (TLS)?,” Cloudflare. Available: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/. [Accessed: Nov. 01, 2025]

[21] Cloudflare, “What is DNS? | How DNS works,” Cloudflare. Available: https://www.cloudflare.com/learning/dns/what-is-dns/. [Accessed: Nov. 01, 2025]

[22] “What is Amazon Route 53? - Amazon Route 53.” Available: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html

[23] “What is Amazon VPC? - Amazon Virtual Private Cloud.” Available: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

[24] “What is a VPC?,” wiz.io, Jun. 05, 2025. Available: https://www.wiz.io/academy/virtual-private-cloud-vpc

[25] “Control subnet traffic with network access control lists - Amazon Virtual Private Cloud.” Available: https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html

[26] “Configure route tables - Amazon Virtual Private Cloud.” Available: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Route_Tables.html

[27] “Security group rules - Amazon Virtual Private Cloud.” Available: https://docs.aws.amazon.com/vpc/latest/userguide/security-group-rules.html

[28] “Elastic network interfaces - Amazon Elastic Compute Cloud.” Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-eni.html

[29] “Amazon EC2 instances - Amazon Elastic Compute Cloud.” Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Instances.html

[30] “Amazon EC2 security groups for your EC2 instances - Amazon Elastic Compute Cloud.” Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-security-groups.html

[31] “What is Amazon Relational Database Service (Amazon RDS)? - Amazon Relational Database Service.” Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html

[32] A. S. Gillis, P. Loshin, and M. Cobb, “What is SSH (Secure Shell) and How Does It Work?,” *SearchSecurity*, Jul. 29, 2024. Available: https://www.techtarget.com/searchsecurity/definition/Secure-Shell

[33] GitLab, “What is version control?,” about.gitlab.com. Available: https://about.gitlab.com/topics/version-control/

[34] Mijacobs, “What is Git? - Azure DevOps,” Microsoft Learn. Available: https://learn.microsoft.com/en-us/devops/develop/git/what-is-git

[35] “Git - What is Git?” Available: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

[36] “What is Amazon CloudFront? - Amazon CloudFront.” Available: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html

[37] “What is Amazon S3? - Amazon Simple Storage Service.” Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

[38] “Amazon Cognito user pools - Amazon Cognito.” Available: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-pools.html

[39] “What is AWS Lambda? - AWS Lambda.” Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[40] M. Goodwin, “API,” *IBM Think*, Oct. 31, 2025. Available: https://www.ibm.com/think/topics/api. [Accessed: Nov. 02, 2025]

[41] “What is Amazon API Gateway? - Amazon API Gateway.” Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

[42] I. Team, “Business logic: definition, benefits, and example,” *Investopedia*, Oct. 02, 2019. Available: https://www.investopedia.com/terms/b/businesslogic.asp

[43] “Models | Django documentation,” Django Project. Available: https://docs.djangoproject.com/en/5.2/topics/db/models/

[44] R. Awati, “object-relational mapping (ORM),” *TheServerSide.com*, Mar. 28, 2023. Available: https://www.theserverside.com/definition/object-relational-mapping-ORM

[45] Sakai-Nako, “Understanding dependencies in programming,” *DEV Community*, Apr. 14, 2024. Available: https://dev.to/sakai-nako/understanding-dependencies-in-programming-4201

[46] “Gunicorn - Python WSGI HTTP Server for UNIX.” Available: https://gunicorn.org/#quickstart

[47] “What is a CI/CD pipeline?” Red Hat. Available: https://www.redhat.com/en/topics/devops/what-cicd-pipeline

[48] “Understanding GitHub Actions - GitHub Docs,” GitHub Docs. Available: https://docs.github.com/en/actions/get-started/understand-github-actions

[49] “Database Migrations: What are the Types of DB Migrations?” Prisma’s Data Guide. Available: https://www.prisma.io/dataguide/types/relational/what-are-database-migrations#what-are-database-migrations

[50] “AIX,” *IBM Documentation*. Available: https://www.ibm.com/docs/en/aix/7.2.0?topic=concepts-creating-running-shell-script

[51] “Let’s Encrypt,” Nov. 24, 2025. Available: https://letsencrypt.org/

[52] “About Certbot.” Available: https://certbot.eff.org/pages/about

[53] “Migrations | Django documentation,” Django Project. Available: https://docs.djangoproject.com/en/5.2/topics/migrations/

[54] IBM, “Event-driven architecture,” *IBM Think*, Oct. 03, 2025. Available: https://www.ibm.com/think/topics/event-driven-architecture. [Accessed: Nov. 05, 2025]

[55] S. Susnjara and I. Smalley, “What is Docker?,” *IBM Think*, Jul. 22, 2025. Available: https://www.ibm.com/think/topics/docker. [Accessed: Nov. 07, 2025]

[56] I. Garg, “Study on JSON, its Uses and Applications in Engineering Organizations,” *ResearchGate*, Mar. 2024. Available: https://www.researchgate.net/publication/379001324_Study_on_JSON_its_Uses_and_Applications_in_Engineering_Organizations. [Accessed: Nov. 07, 2025]

[57] “What is Amazon SNS? - Amazon Simple Notification Service.” Available: https://docs.aws.amazon.com/sns/latest/dg/welcome.html

[58] “Amazon CloudFront introduces Origin Access Control (OAC) | Amazon Web Services,” Amazon Web Services, Aug. 30, 2022. Available: https://aws.amazon.com/blogs/networking-and-content-delivery/amazon-cloudfront-introduces-origin-access-control-oac/

[59] “Managing Lambda dependencies with layers - AWS Lambda.” Available: https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html

[60] “Application-specific settings with app clients - Amazon Cognito.” Available: https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-settings-client-apps.html

[61] “Adding layers to functions - AWS Lambda.” Available: https://docs.aws.amazon.com/lambda/latest/dg/adding-layers.html

[62] “Lambda runtimes - AWS Lambda.” Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

[63] “Examples of Amazon S3 bucket policies - Amazon Simple Storage Service.” Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/example-bucket-policies.html

[64] “Control access to REST APIs using Amazon Cognito user pools as an authorizer - Amazon API Gateway.” Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-integrate-with-cognito.html

[65] “Amazon API Gateway concepts - Amazon API Gateway.” Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-basic-concept.html

[66] “CORS for REST APIs in API Gateway - Amazon API Gateway.” Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-cors.html

[67] “Set up a stage for a REST API in API Gateway - Amazon API Gateway.” Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-stages.html