Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

JOB AGGREGATOR PLATFORM

By
MING QIAN SOON

April 27, 2025

Minor Dissertation

**Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.**

B.Sc. (Hons) in Software Development

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview

Online platforms have become essential for job searching and recruitment in the digital era [1, 2]. Millions of workers worldwide are employed by using online job platforms [3]. The growth of online job platforms and remote work opportunities has led to a surge in job applications. Online platforms are frequently used by businesses to post job openings and accept CVs, reaching a large audience at a minimal expense. One noteworthy trend is the move to digital hiring [1]. This fragmented landscape forces users to perform repetitive searches, navigate different user interfaces, and manage various accounts. The lack of integration between platforms leads to duplication of effort, missed opportunities, and an overall inefficient job-seeking experience. Furthermore, with an increasing number of platforms entering the market, the challenge of staying updated becomes even more difficult for job seekers.

This project envisions a unified job aggregation platform that seamlessly consolidates job postings from multiple sources into one cohesive system. The platform seeks to offer a more individualized, pertinent, and effective job search experience by using AI-based skill extraction and job matching in conjunction with real-time scraping and aggregation of postings from major platforms. It tackles the user-experience problem of streamlining the job search and application process in addition to the technological problem of real-time platform integration.

A unified and sophisticated job search tool is needed in the market, as millions of job searchers rely on online platforms to obtain jobs. Improved talent matching benefits businesses as well by improving hiring results. This project was motivated by my personal experiences navigating a disorganized and repetitive job search process as well as my desire to use my technical expertise in data engineering, AI, and microservices to address a real-world issue that impacts both people and enterprises.

1.2 Problem Statement

Job Platforms such as Indeed, LinkedIn, JobsIE, and IrishJobs operate independently, and each has its listing formats, data models, and search system. On account of, employment data sharing lacking a consistent methodology, consumers may need to perform repeated searches across various websites to make sure they are not missing any chances. This lack of interoperability creates unnecessary friction in the job search process.

From the user's perspective, this problem leads to several key pain points:

- **Redundant effort** in entering similar keywords and filters across platforms
- Need to **create and maintain multiple accounts and profiles**
- **Inconsistent** job descriptions and application processes
- **Risk of missing job opportunities** due to platform fragmentation

The amount of information can be too much for users to handle, even if they find numerous listings. The efficiency of the search is diminished by irrelevant results, redundant job postings, and ambiguous skill requirements. Users must manually sort through a lot of data as many systems lack personalized relevance ratings and sophisticated filtering.

Furthermore, the limitation of conventional keyword-based matching, which frequently has poor precision due to its disregard for the semantics of the terms used in CVs and job postings, is one of the major issues [4]. This implies that systems that just use keywords may overlook applicants with the necessary abilities but speak other languages. For both recruiters and job seekers, this results in poor matches and lost chances. Relevance and results can be significantly enhanced by a system that uses AI to evaluate job descriptions and compare them to a user's real skill set.

1.3 Project Objectives

The primary goal of this project is to create a unified Job Aggregator Platform that consolidates job postings from multiple sources, streamlining the job search process. Key objectives include:

- Aggregating job postings from platforms like JobsIE and IrishJobs.

- Using AI and NLP to extract and highlight required skills from job descriptions.
- Matching users to suitable job listings based on their profiles.
- Implementing a microservices architecture for modularity and scalability.
- Leveraging technologies like Kafka, Redis, and Docker for performance and resilience.

1.3.1 Success Criteria and Deliverables

Success Criteria

The project's success is measured by its ability to:

- Aggregate job listings from multiple platforms with consistent results.
- Achieve 85% accuracy in skill extraction using AI.
- Provide personalized job recommendations based on user profiles.

Deliverables

The project produced the following concrete deliverables:

1. Core Microservices
   - User Management Service: Handles authentication, profile management, and CV storage
   - Jobs Storage Service: Manages persistence and retrieval of job listings
   - Matching Service: Implements the skill-matching algorithm
   - AI Skills Extraction Service: Processes text and identifies skills using NLP
2. Web Scraping services
   - JobsIE Scraper: Retrieves job listings from Jobs.ie
   - IrishJobs Scraper: Retrieves job listings from IrishJobs.ie
3. Frontend application
   - Responsive React-based user interface
   - Comprehensive job search functionality

- User profile and CV management
- Authentication and registration flows

4. Infrastructure Components
    - Kafka-based messaging system
    - Redis caching layer
    - MySQL databases for persistent storage
    - Docker containerization for all services
5. Documentation and testing
    - System architecture documentation
    - API specifications
    - Comprehensive test suite
    - Deployment instructions

The success criteria and deliverables were designed to ensure the project not only met its technical objectives but also delivered tangible value to end users by addressing the fundamental challenges of fragmented job searching. By establishing clear, measurable outcomes, the project maintained focus on creating a solution that brings genuine improvement to the job search experience.

1.4 Research Questions

A set of fundamental questions that seek to investigate the possible advantages and technological difficulties of creating a single, intelligent job aggregator platform serve as the basis for this study. The project's scope and direction are framed by the following research questions:

RQ1: How can a unified job aggregator platform improve job search efficiency?

These days, job searchers frequently use several platforms (such as Indeed, IrishJobs, and Jobs. etc.), which results in disjointed experiences and pointless searches. This study explores how combining job postings from many platforms can improve relevancy, save user effort, and expedite the job search process in general.

RQ2: How can automated skill extraction enhance job-to-user matching?

It might be challenging for consumers to determine whether a job fits with their skill set because job descriptions sometimes feature lengthy text passages. To improve the caliber and applicability of job recommendations, this study investigates the use of pre-trained AI models to extract critical skills from job descriptions and compare them to a user's profile.

RQ3: How can real-time job aggregation provide more accurate and up-to-date results?

Conventional platforms frequently use out-of-date data streams or sporadic scraping. Using tools like Playwright and Kafka, this study examines how well real-time data aggregation works to give the most recent job listings and guarantee that opportunities that are time-sensitive are not overlooked.

Chapter 2

Methodology

2.1 Tools and Environments

Delivering a complex system, like a distributed job aggregator platform, within schedule and resource restrictions requires effective project management. The technique used, the justification for its choice, and the particular tools and workflows employed throughout the development lifecycle are described in this section.

2.1.1 Kanban Methodology Selection and Justification

The Kanban Methodology was selected for this project due to its adaptability and visual simplicity, which aligns well with managing the dynamic and iterative nature of software development. Agile methods, including Kanban, are gaining wide recognition for their flexibility and effectiveness. Unlike approaches like Scrum, Kanban offers a continuous delivery methodology without the need for a fixed-length sprint of roles [5]. Anderson, regarded as the founder of Kanban in Software development, defined it as a method for system change and incremental, evolutionary processes [5]. On account each service can advance at its rate, this flexibility is beneficial for a single developer overseeing parallel microservices with different degrees of complexity.

Furthermore, the early detection of bottlenecks is made possible by the fundamental ideas of Kanban, such as the use of a Kanban board to visualize the workflow and the focus on monitoring and controlling flow. The Kanban board is used to track project progress and visualize the process [5]. Managing the integration of the services (web scraping, skill extraction, and user management services) can be done by applying the concepts of workflow visualization and work-in-progress (WIP) limitations. Furthermore, rather than requiring a lot of formal process administration, Kanban's low overhead frees up more time for implementation and testing [5].

2.1.2 Kanban Board Structure and Workflow

To illustrate the workflow and track project progress, the Kanban technique makes use of a board. This board is an essential tool for improving process comprehension, visibility, and control [5]. The Kanban board can be set up with columns that stand in for the various phases of the development process.

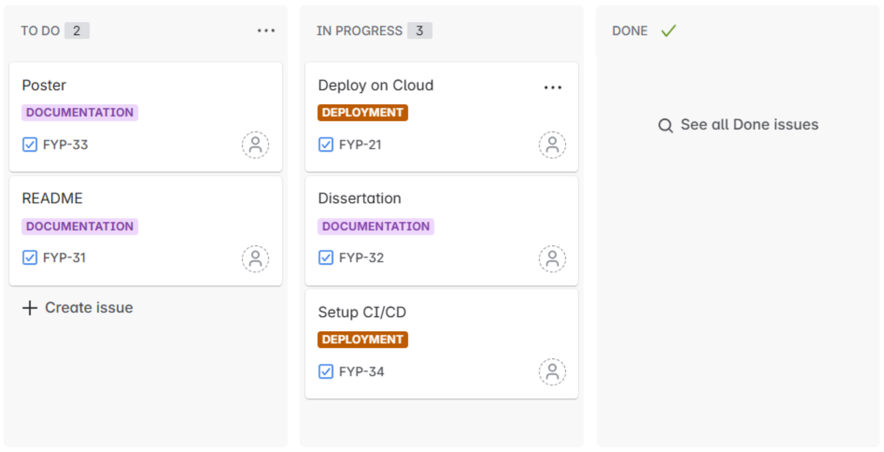

Figure 2.1: Kanban Backlog

The Kanban board was structured into columns (Backlog, To Do, In Progress, Done) to track tasks and visualize progress.

A title, thorough explanation, related microservice or feature, anticipated time, and priority level were all included on each task card. Throughout all phases of development, this framework assisted in ensuring accountability, transparency, and traceability.

2.1.3 Work-In-Progress (WIP) Limits and Continuous Delivery Model

WIP limits were applied to the "In Progress" column to prevent overloading and ensure task completion before starting new ones.

Iterative updates and isolated service testing were done using the continuous delivery methodology. Docker was used to containerize each microservice, while Git was used for version control. This enhanced dependability and decreased last-minute bugs by enabling incremental development and assisting in the early detection of integration problems.

2.2 Development lifecycle

To guarantee that functional and technical needs were satisfied methodically, the Job Aggregator Platform was developed using a planned and iterative lifecycle. This section describes the iterative development methodology, version control techniques, and requirements collection techniques.

2.2.1 Requirements Gathering and User Stories

The development of the Job Aggregator Platform was built upon a structured approach to requirements gathering, a crucial initial phase for ensuring the final product effectively meets the needs of its users. This basic phase sought to catch both functional needs, specifying what the system should accomplish, and non-functional ones, describing how the system should operate under different circumstances [6].

Requirements were gathered iteratively, emphasizing flexibility and adaptation in line with Agile principles [5].

Given the platform's character as a job aggregator, the requirements collecting stage would have probably also emphasized specifying criteria for properly matching job posts with job seekers. This would mean knowing the required information to be gathered from user profiles and job advertisements to enable correct and pertinent matching.

Approach to Requirements Gathering

- **Personal Experience and Observations**: As a student and early-career job seeker, I've personally experienced many of the problems associated with internet job searches, including platform fragmentation, repetitive material, ambiguous job descriptions, and irrelevant results. These firsthand experiences acted as a powerful initial motivation and provided great insight into what an ideal career platform might address.

- **Competitor Analysis**: To find gaps and restrictions, a comprehensive evaluation of the current job portals—including Indeed, Jobs.ie, IrishJobs, and LinkedIn—was conducted. This involved investigating their user interfaces, filtering systems, recommendation algorithms, search capabilities, and platform responsiveness. This investigation revealed that no single platform offers Irish users a cohesive, intelligent, and customized job search experience.

- **Informal User Feedback**: Feedback was solicited from friends and recent graduates through informal discussions and short surveys. Similar issues were raised by these participants, including having to search across several platforms, getting suggestions that weren't relevant to their skills, and not understanding which jobs matched their qualifications. This unofficial but perceptive input strengthened the identified demands and helped confirm the original hypotheses.

After the Job Aggregator Platform's requirements were gathered, they were converted into user stories, which is a common procedure in Agile development approaches. Because of its adaptability and efficiency in providing software solutions, this method is preferred [5]. By converting intangible requirements into tangible, testable, and user-centered objectives, user stories guarantee that the finished product meets the needs of its end users. The format used for each user narrative was simple: "As a [type of user], I want [some goal], so that [reason/-value]." Each requirement's who, what, and why are spelled out in this structure. These user stories would serve as the foundation of the product backlog in the context of Agile methodologies, which are crucial for organizing sprints and meetings in order to determine project requirements based on stakeholder needs [5]. This is in line with the agile tenet of early and iterative product delivery and the significance of ongoing feedback.

2.2.2 User Stories

General User Stories User stories captured functional requirements such as job aggregation, AI-driven recommendations, and secure user management. For example, "As a job seeker, I want to search for jobs from multiple sources in one place, so that I can save time."

System Administration and Security

- "As a system administrator, I want the microservices to operate independently and recover from crashes automatically so that downtime is minimized."
- "As a new user, I want to register and log in securely using OAuth or email/-password, so that my data remains safe."

2.2.3 Prioritization and Maintanence

User stories were tracked using a Kanban board as shown as 2.1, allowing for dynamic prioritization and visual progress tracking. Early in the development process, high-impact stories—such as skill-based matching and job search aggregation—were given priority, and later on, features like bookmarking and user profiles were included. To provide a well-rounded and targeted development effort, stories were grouped according to their applicability to key features, performance improvements, or user engagement.

2.2.4 Iterative Development Cycles

For this individual project, the development process adopted an iterative and adaptive approach, emphasizing continuous improvement and feedback. Given that team-based industry settings frequently utilize classic agile frameworks like Scrum, this solo effort required a more lightweight and adaptable technique. As a result, a development cycle influenced by agile was adapted, emphasizing continuous testing, adaptable planning, and small, incremental modifications. This strategy is in line with iterative design principles, which hold that making tiny, iterative adjustments is beneficial [6]. This iteration was made easier by the project's component modularity, which allowed for the addition and testing of various sections and capabilities as needed over time. Additionally, this approach made it simpler to manage possible hazards and make necessary adjustments. Furthermore, the constant nature of improvements was reinforced by the focus on continuous integration.

The development followed an iterative "build, test, refine" approach, with each microservice treated as a modular unit of work in line with microservices architecture principles. Instead of adhering to rigorously defined sprints, development cycles embraced flexibility, guided by the intrinsic complexity of each work and the developing priorities of the system.

The development schedule was divided into stages, each of which included a set of activities that added up to a vaguely defined cycle. Every phase started with an evaluation of the current backlog to determine which feature or service would be most beneficial to build next. Following task selection, the implementation process started with research and prototype, then moved on to core logic development and component integration. After that, manual testing, debugging, and documentation came next. Any problems that surfaced, such as inconsistent data extraction or slow API response times, were fixed right away before continuing.

A Kanban board was utilized as shown in figure 2.1 to track and visualize work. Standard columns like "Backlog," "To Do," "In Progress," "Testing," and "Done" were present on the board. This made it easier to stay focused and made it possible to see progress. As new problems or concepts arose, it also assisted in setting priorities for tasks. It became simpler to spot bottlenecks—like delays in integrating AI models or mistakes in Docker deployment—and make quick fixes as responsibilities were distributed evenly.

Additionally, this iterative approach fits in nicely with the methodology for continuous integration. It was common practice to commit newly added code to version control with unambiguous commit statements and instant testing. Small functional units were constructed and independently verified, lowering the possibility of major system failures and facilitating quicker problem fixes. The project maintained internal milestones, including the first entirely matched job-user pair, the first fully functional job scraper, and the first full database integration, to gauge progress even in the absence of official releases.

In conclusion, the development cycles of the project were responsive, lightweight, and iterative. The approach's loose adherence to Scrum and waterfall models gave it the flexibility to experiment, fail quickly, and modify solutions in response to implementation-related real-world problems. While maintaining the system's coherence and scalability, this agile-inspired technique facilitated the quick development of numerous microservices, which was in line with the changing functional and technical needs of the unified job aggregator platform.

2.2.5 Version Control Strategy and Branch Management

Git, a popular Distributed Version Control System (DVCS) [7] that is well-known for its capacity to manage changing objects and document changes made by software developers, was used to create vision control for this specific project with a strictly defined development timeframe. This project's repository was housed on GitHub, a well-known Git repository platform.

A simplified branching method was used, in which all work was done immediately on the primary branch, considering the project's nature and emphasis on quick development and testing [8]. This choice was made to simplify things and give priority to quick development. The main branch, which functioned as both the development and production branches, received all commits straight from this method. The safety and modularity of more sophisticated multi-branch workflows like GitFlow which are frequently chosen for larger teams and more complicated projects, were fundamentally absent from this simplified process, despite the fact that it allowed for speed [8].

To preserve clarity and traceability throughout the project lifecycle, recommended practices like frequent commits and detailed commit messages were followed to, even with the stramlined branching. Thus, a comprehensive chronological overview of feature implementation, problem corrections, and advancements is provided by the Git history.

Additionally, Docker Compose was used to guarantee a consistent runtime and development environment. By developing lightweight and portable containers, this technique guarantees that the project can be deployed dependably irrespective of the underlying host setup.

2.3 Project Timeline and Planing

Effective planning and well-organized time management are essential for the successful completion of a complex software system like the Job Aggregator Platform, which was developed by the individual and consists of multiple separate modules or microservices, including a web-based frontend, backend API services, an AI-based skill analysis component, and a scraping engine for job data. It can be helpful to use a visualization tool such as a Gantt chart to visualize work sequencing, measure progress, manage dependencies between modules, and monitor milestones because academic deadlines need the management of development risks and the maintenance of project momentum. This is consistent with an incremental development methodology, which is frequently appropriate for handling the complexity present in systems constructed using a microservices architecture. This methodology involves developing and testing tiny functional components of the system at a time. Such an architecture's modularity makes the development process more controllable and iterative [9].

2.3.1 Gantt Chart Analysis

A Gantt chart was used to schedule work and visualize the timeline for the Job Aggregator Platform. It provided a framework for tracking progress, managing dependencies, and ensuring timely delivery of key milestones. The Gantt chart facilitated an iterative approach to development by permitting modifications to project activities based on real-time observations, which kept the project flexible and responsive to obstacles. Tools for tracking and visualizing progress are especially useful because of the emphasis on incremental development, which involves creating and testing small functional components of the system over time. Managing the complexity of a platform with several microservices and components requires the usage of such planning tools.

Each of the several connected modules that made up the project's framework represented a major technical element. These comprised:

- **Web Scraping Service (Playwright + Flask)**: Responsible for collecting job listings from external platforms in real-time.
- **AI Skills Extraction Service**: Analyzing scraped job descriptions using pre-trained models to identify relevant skills.
- **Job Matching and Redis Integration**: Implementing logic for keyword-based filtering and storing interim data for performance enhancement. Both job skills and the actual job listings were stored in Redis to allow faster search and reduce repeated scraping operations.
- **Backend API Service (Spring Boot + MySQL)**: Handling user authentication, job management, and communication between frontend and AI modules.

- **Frontend UI (React + TypeScript)**: Providing an interface for user interaction, job browsing, login/registration, and search functionalities.

- **System Deployment and Testing**: Dockerizing the entire stack and local deployment.

Figure 2.2 demonstrates the gannt chart divided the project into 28 weeks, assigning specific deliverables to each phase. This breakdown is summarized in the table below:

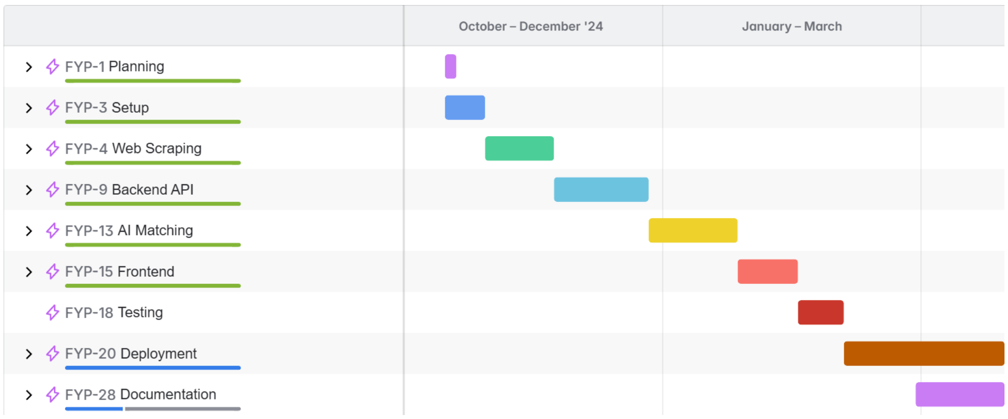

Figure 2.2: Project Gantt Chart showing the timeline

Weeks	**Task Description**
Week 1-3	Project scoping, background research, requirements gathering, architecture design
Week 3-6	Implementation of job scraper using Playwright and Flask
Week 6-11	Backend implementation with Spring Boot and MySQL
Week 11-15	Integration of Redis and AI model for skill extraction
Week 15-18	Frontend development with React, including API integration
Week 18-21	Testing, debugging
Week 21-28	System integration, Docker setup, and deployment
Week 25-28	Documentation writing

Table 2.1: Project Timeline and Tasks

Understanding the problem domain, performing a literature research, and determining the platform's needs were the key goals of the first three weeks. To identify usability problems, filtering feature limitations, and the possibility of AI-enhanced matching, existing job platforms were examined. The system design and the definition of important features were influenced by these realizations.

The implementation phase began in weeks three through six. Because it represented the platform's data ingestion layer, the web scraping microservice—which used Flask for API expose and Playwright for browser automation—was given priority. The downstream processes (such AI processing and job matching) wouldn't have any input to work with if the scraper wasn't operational.

Weeks 6 through 11 of the backend development process comprised creating and deploying RESTful endpoints with Spring Boot and linking them to a MySQL database. The features for job storage and user authentication were now active. Because of their significance in data protection, security issues like input validation and password hashing took more time and were seen as non-negotiable chores.

The Redis in-memory store was integrated during Weeks 11 through 15 to temporarily cache both scraped job listings and the extracted skills. Storing job listings in Redis helped improve performance by enabling quicker access during searches and minimizing redundant scraping operations. This approach allowed subsequent services (such as job matching and frontend display) to retrieve relevant data with minimal delay. Concurrently, a Hugging Face Transformers-hosted AI model was incorporated to extract important abilities from job descriptions...

React was used for frontend development between weeks 15 through 18. To enable job searches, registration, login, and viewing of matching jobs, the frontend used backend APIs. Usability and simplicity were prioritized.

While containerization was in progress, testing, and debugging began concurrently in Weeks 18 to 21.

During Weeks 21 to 24, the entire microservice-based architecture was containerized using Docker. Each major component—Flask scraper, Redis caching layer, Spring Boot backend, and the React frontend—was encapsulated into its container. Docker Compose was used to manage these services and define inter-container communication, environment variables, port mapping, and shared volumes. This containerization ensured that the system could be run consistently across development machines or future cloud-based environments.

The last four weeks, from weeks 25 to 28, were dedicated to writing the dissertation, compiling technical documentation, and reflecting on evaluation results. This stage ran in parallel with the final system tuning.

In conclusion, the Gantt chart served as an essential tool for job scheduling and timeline visualization, offering a thorough overview of scheduled activities, durations, overlapping tasks, and logical sequencing [8]. This aligns with the principle that efficient planning and well-structured time management are crucial in the successful delivery of a complex software system, especially in a solo-developed project [my previous turn]. The Gantt chart facilitated visibility into the project's progression, enabling a single developer to oversee the various modules and components of the Job Aggregator Platform, which included a scraping engine, AI-based skill analysis, backend API services, and a web-based frontend.

Furthermore, by offering a systematic framework for monitoring progress against the predetermined timeline, the Gantt chart helped to inculcate discipline in the development process. This is especially crucial for handling the inherent complexity of systems constructed using microservices or independent modules. The Gantt chart enabled an iterative process where progress on particular features could be tracked by dividing the effort into logical units that corresponded to these modules and their dependencies.

Lastly, by enabling modifications to project activities based on real-time observations, the Gantt chart made responsiveness possible. In software development, this adaptability is essential, particularly when working with ambiguous components. The dingle developer could prevent delays and make sure that academic and technical deliverables were completed within the semester's limitations by regularly assessing schedules. The Gantt chart's function as a dynamic control tool throughout the project's lifecycle is highlighted by its capacity to visualize progress and modify the plan as necessary.

2.3.2 Major Milestones and Dependencies

Several key milestones were used to track progress and validate the correctness and completeness of development phases. These included:

Key milestones included functional web scraping (Week 6), backend implementation (Week 11), AI integration (Week 13), and Dockerized deployment (Week 28).

2.3.3 Critical Path Identification

The critical path included key tasks such as developing the job scraping microservice, integrating the database, implementing AI-powered skill extraction, and deploying the system. These tasks were prioritized to ensure the platform's core functionality was delivered on time.

Managing the Critical Path

Throughout the project, Gantt charts and weekly task lists were utilized to track the progress of work on the critical path. Risk mitigation was given special consideration, for example, by testing scraping scripts on several job boards early to find anti-scraping techniques or by starting work on AI model evaluation concurrently with the database structure.

On the other hand, non-essential features like internationalization, advanced search filters, UI styling, and user analytics were either added after the critical route stabilized or had flexible scheduling. This made sure that the delivery of the core platform wouldn't be jeopardized by delays in optional components.

The project was able to stay on course and produce a solid, integrated system that met its main objectives by concentrating efforts on the critical path.

2.3.4 Resource Allocation

Resources were prioritized for complex components such as web scraping, AI integration, and backend security.

2.4 Development practices

To effectively handle the inherent complexity of a microservices architecture, rigorous development practices are essential to ensure that each independent service remains maintainable and fulfills its specific business functionality [10]. While developing individual microservices, simulating industry-standard workflows, such as applying Continuous Integration/Continuous Delivery (CI/CD) pipelines, can provide practical experience and contribute to higher code quality by facilitating regular testing and deployment of each service. Because microservices are modular and each service can be deployed independently, maintaining the operation of the entire system requires clearly defined interfaces and roles [11].

- **Test-driven Development (TDD)**

  Test-driven Development (TDD) was applied to critical components such as job scraping, skill extraction, and matching algorithms. Unit and integration tests ensured reliability and facilitated confident refactoring. While full TDD coverage wasn't feasible for all components due to time constraints, critical paths received thorough test coverage, significantly improving overall system stability and reliability. Test suites were integrated with the version control workflow, running automatically after significant commits to detect issues early in the development cycle.

- **Code Review Process**

  As a solo developer, formal code reviews were not conducted. However, code quality was maintained by practicing frequent self-review, leveraging Git commits with meaningful messages.

- **Documentation Approach**

  Throughout development, documentation was kept up to date by utilizing both external Markdown files (README.md) in each service repository and inline code comments. This contained rules for using the API, important architectural choices, and setup instructions.

- **Continuous Integration and Continuous Deployment (CI/CD)**

  To ensure streamlined development and deployment workflows for the microservices-based project, a Continuous Integration/Continuous Delivery (CI/CD) approach was adopted [11]. This approach, which is essential for handling microservices complexity, sought to enable the safe, rapid, and sustainable deployment of modifications into production. Every time code changes were posted to the repository, automated pipelines were set up to build, test, and launch each microservice using tools like GitHub Actions.

  The implemented pipelines automated the deployment processes, which is a key benefit of adopting a Microservices DevOps approach. By automating these steps, the project significantly reduced the risk of human error associated with manual deployments and aimed to make the development cycle more efficient and reliable [9].

2.5 Tools and Environments

2.5.1 Development Environment Configuration

The development environment supported a multi-language microservices architecture, with Java (Spring Boot), Python (Flask), and TypeScript (React). Docker and Docker Compose ensured consistency across environments.

2.5.2 Collaboration Tools

Version control was managed using Git and GitHub. GitHub also facilitated collaboration, issue tracking, and integration with CI/CD pipelines via GitHub Actions. Additional tools like Jira helped track project progress, manage tasks, and plan deliverables across development milestones.

2.5.3 Testing Frameworks and Infrastructure

Unit testing frameworks such as JUnit for Java and unittest or pytest for Python were used to test back-end services. In order to simulate production-like situations during development and integration testing, Docker Compose offered a separate infrastructure for testing inter-service communication locally.

2.6 Adaptation and Flexibility

Throughout the development of the Job Aggregator Platform, several methodology-related challenges emerged that required adaptive thinking and flexible process changes.

2.6.1 Methodology Challenges Encountered

A major obstacle was striking a balance between the strict deadlines academic timeline's structure and agile-style iterative development. Initial presumptions, including quick model integration or stable HTML structures on construction sites, turned out to be incorrect. Because of CAPTCHA difficulties, IP restrictions, and frequent DOM structure changes, scraping websites such as Jobs.IE and IrishJobs added difficulty. Related downstream jobs had to be rescheduled as a result of these problems, which interrupted the critical route.

Integrating services created with various languages and frameworks presented another difficulty. Particularly, coordinating coordinating communication between the Java=based backend (Spring Boot) and the Python-based web scraping microservice (Flask). During the early phases of integration, communication was more challenging because each service had its data formats, error-handling strategies, and starting behavior.

Issues including disparate data schemas, variations in JSON serialization, and unplanned service outages brought on by unhandled exceptions in one of the services were all brought on by this heterogeneity. Because logs were dispersed across several containers and logging types, debugging became more time-consuming.

Additionally, coordinating development and debugging across several environments became challenging when managing multiple microservices. Issues like inter-service communication problems and race conditions during container startup necessitated frequent reassessments of service orchestration.

2.6.2 Process Adaptions Made

Test scripts for several job boards were created in advance to address scraping instabilities, and the Playwright service was updated to include retry logic. In order to reduce bottlenecks caused by lengthy AI inference periods, the skill extraction service was separated from the main flow and operated in parallel.

A key tactic was containerization with Docker Compose, which allowed for reproducible local setup and realistic, production-like service integration. Additionally, because frontend testing was omitted, integration testing and strong backend API answers received more attention.

Even for local deployment, CI/CD stages were added to the development pipeline. For continuous integration, code integrity across branches, and a decrease in manual deployment failures, GitHub Actions was incorporated.

2.6.3 Lessons Learned from the Approach

Test scripts for several job boards were created in advance to address scraping instabilities, and the Playwright service was updated to include retry logic. In order to reduce bottlenecks caused by lengthy AI inference periods, the skill extraction service was separated from the main flow and operated in parallel.

A key tactic was containerization with Docker Compose, which allowed for reproducible local setup and realistic, production-like service integration. Additionally, because frontend testing was omitted, integration testing and strong backend API answers received more attention.

Even for local deployment, CI/CD stages were added to the development pipeline. For continuous integration, code integrity across branches, and a decrease in manual deployment failures, GitHub Actions was incorporated.

Chapter 3

Technology Review

3.1 Current Job Market Platforms

Indeed, the landscape of the worldwide job search has changed significantly in recent years, shifting from more conventional approaches like classified ads and recruitment agencies to internet job boards [12]. These online resources are becoming the main means of communication between businesses and job seekers [13]. These platforms now provide more than just job listings; they also provide data-based insights and personalized suggestions driven by AI [14]. Although not specifically mentioned in the sources above, they can also offer information like company reviews and resume-building tools. The rise of these online hiring platforms has made job openings more visible and sped up the hiring process. Additionally, they have implemented innovative techniques for engaging and screening candidates, like automatically ranking resumes.

Platforms like Indeed and LinkedIn have emerged as significant players in this fiercely competitive digital market, each introducing a unique combination of services and technological strategies. Determining the distinct value and market niche of every new solution requires an understanding of these platforms' features, capabilities, and drawbacks.

Assessing these current platforms calls for attention to their technical merits, including the application of contemporary NLP and machine learning techniques such BERT algorithms for objective person-job fit, as well as their business logic in matching individuals to positions [14, 12]. Important factors to take into account are also their user-facing capabilities, like the opportunity for job seekers to verify using their LinkedIn accounts and the display of ranked results to HR professionals [1]. However, because job market skills and criteria are ever-changing, these platforms also have limits when it comes to handling dynamic job data. Additionally, they may have trouble offering genuinely customized matching that gets past the difficulties with semantic understanding and the disparities in language between recruiters and job searchers. Furthermore, in algorithmic form, problems that online systems seek to address—such as bias and subjective human judgments—may still exist in traditional ways. Despite the effectiveness of internet platforms, recruiters may become overwhelmed by the sheer number of applications.

3.1.1 Analysis of Existing Job Platforms

Indeed

With hundreds of millions of users each month, Indeed is sometimes considered the most popular employment site in the world. Indeed's primary strength is its aggregation technique, which retrieves job postings from thousands of websites, such as online job boards, staffing firms, and corporate career portals. This leads to an enormous number of employment advertisements, providing unmatched coverage.

There are disadvantages to this strategy, though. Outdated, expired, or duplicate posts are frequently found in aggregated results. Furthermore, Indeed uses anti-bot techniques like rate limitation and captchas, which make automated scraping programs difficult to use. The job matching engine lacks the sophistication of AI-driven profile analysis, and the platform's filtering features are somewhat simple for users.

Despite these problems, Indeed is a mainstay in the industry thanks to its user-friendly interface and extensive job database.

LinkedIn

By fusing job search functionality with a social networking layer, LinkedIn has completely changed the way recruiters and job seekers communicate. Users gain access to mutual relationships within target organizations, a highly tailored feed, and suggestions driven by AI. Additionally, employers can advertise openings, do direct applicant scouting, and use advanced analytics and automation tools.

The incorporation of user data into the recommendation engine is LinkedIn's strongest technical feature. When recommending employment, it takes into account post engagements, endorsements, activity history, and profile information. Because of this, it's an effective tool for finding jobs that fit one's professional path.

The drawback is that LinkedIn prefers to reward people who are engaged and have premium subscriptions. The experience may seem constrained to less frequent users. Furthermore, scraping and automation tools encounter challenges due to its intricate DOM structure and usage of dynamic JavaScript elements.

IrishJobs

The Republic of Ireland is the focus of the market-specific website IrishJobs. IrishJobs places more emphasis on direct listings from regional firms than larger aggregators, guaranteeing a higher caliber and dependability of job advertisements. In Ireland, it has established a reputation as a reliable brand among employers and job searchers.

Employer verification, consistent data formatting, and region-specific filtering are its strong points. Technically speaking, this facilitates integration and scraping, particularly when contrasted with platforms that depend on outside aggregations.

IrishJobs does not, however, have the sophisticated features found on international sites. It doesn't provide deep personalization, user analytics, or AI-based suggestions. In addition, its feature set is rather constrained in light of contemporary job search requirements, and its interface is more conventional.

Jobs.ie

A reputable job site that mostly serves the Irish labor market is Jobs.ie. It has become well-known as a trustworthy and user-friendly platform for finding jobs in a variety of industries, especially in fields like hospitality, retail, logistics, and administration, with an emphasis on both employers and job seekers in Ireland.

Jobs.ie is unique because it prioritizes user accessibility and simplicity. Targeting customers who may not be tech-savvy but are actively seeking work in Ireland, the platform is made to be user-friendly and intuitive. Jobs.ie provides job alert tools, CV uploads, and account creation for job seekers. It gives companies access to CV databases, posting services, and branding tools to boost their company's visibility.

3.2 Architectural Patterns in Modal Web Applications

Web application design has changed dramatically in recent years due to the growing demands for scalability, dependability, quick development cycles, and effective use of resources. A monolithic paradigm, which is defined by a single executable and a single deployment, was frequently used by early web services [11]. Although initially easier to implement and appropriate for smaller applications, monolithic systems lose flexibility and scalability as application needs grow [9].

Developers and architects started looking into more modular and distributed architectural patterns as software systems became more complex, particularly in fields like social media, e-commerce, and other large-scale websites and apps [15].

Numerous convergent forces drove this progression. The infrastructure for implementing and growing distributed systems was made available with the emergence of cloud computing [16]. Continuous deployment and delivery, or CI/CD, was more in demand in order to facilitate quicker release cycles. As the strain on various components of an application varied, the requirement for independent scalability of features became essential.

Numerous issues with traditional development are intended to be addressed by these patterns. In monolithic programs, tightly connected components make deployments difficult and updates risky [6]. Because even minor modifications need the deployment of the full monolith, deployment bottlenecks occur. Even if only a portion of the monolithic application needs more resources, scaling inefficiencies arise since the entire application must be scaled. By permitting parallel development on separate services, independent component testing, and technology diversity across services—where teams can select the best technologies for their particular services—modern architectures also provide clear benefits for team collaboration.

Project planners, decision-makers, and developers alike must comprehend these architectural patterns and the trade-offs they entail [17]. Because distributed systems include network communication cost, the architecture selection affects system performance over the long run. Maintainability is also impacted, though microservices may help because of have more condensed scope. Because distributed systems can add complexity and expenses associated with administering various services, operational cost is an important factor to take into account. Lastly, responsiveness and dependability are two aspects of the architecture that affect user experience and are impacted by the underlying architectural decisions.

Monolithic Architecture

The whole program, including the user interface, business logic, and data access layers, is developed and implemented as a single unit in a monolithic design. Monoliths are easier to construct at first, but they can get cumbersome as the codebase and team size increase.

Pros of Monolithic Architecture:

- **Simpler to develop and deploy initially** Smaller monolithic applications are easier to build and implement since they are handled as a single unit.
- **Easier to manage at the early stages** due to centralized control. Monitoring can be a simpler task.

- **Ease of code reuse** within the application. It is possible to reuse a specific part of the code by copying a portion within the monolith application, offering control over the changes made.

Cons of Monolithic Architecture:

- **Difficult to scale individual components independently.** Usually, scaling up for better performance entails scaling the entire application, which isn't always effective [6]. Some servers were frequently overused and others were underutilized as a result of the monolithic strategy of scaling all services within the monolith at the same level.
- **A single fault can affect the entire application (tight coupling).** A problem in one part of the system could possibly bring down the entire system because all the parts are deployed simultaneously [6].
- **Harder to maintain or refactor as codebase grows.** Maintaining this kind of architecture has become unsustainable as the complexity of these applications rises as a result of the expansion of business functionality. Particularly in a big and intricate monolith program, any modification, no matter how minor, might have an impact on the behavior of the entire system [15].
- **Slower deployment cycles** due to the interdependency of components.
- **Limited flexibility in adopting new technologies** [15]. When an application's architecture is too rigid and standardized to support a single technology, it is challenging to integrate new, cutting-edge technologies.

Microservices Architecture

Using a microservices architecture, a single application is created as a collection of little services, each of which operates independently and communicates with the others by simple means, frequently an HTTP resource API [18]. These services can be independently deployed by completely automated deployment equipment and are based on business capabilities.

Pros of Microservices:

- **Independent deployment, testing, and scaling of services.** A number of smaller apps are developed and bundled separately, each of which merely implements a portion of the larger program. Each service can be launched and developed at its own speed. Microservices have the ability to scale and fail on their own. Agile deployment practices, which rely on continuous delivery, shorten the time between an idea and functional software.

- **Technology agnosticism,** each service can use different languages or frameworks. The ability to code a microservice in any programming language or with any database makes each microservice a separately deployable unit [15].
- **Better fault isolation and resilience.** If one microservice fails, the rest of the system can continue to function. Event-driven architecture, often used with microservices, is an optimal choice for achieving high fault tolerance.
- **More flexible and easier to integrate with CI/CD pipelines.**
- **Better control over scalability by only scaling each microservice to the level required.**

Cons of Microservices:

- **Increased complexity in inter-service communication.** The proliferation of services may escalate the overall application's complexity and maintenance costs [10]. Performance can suffer when the granularity of services is poor, leading to excessive inter-service communication.
- **Requires robust monitoring, logging, and service discovery.** One monolith is easier to monitor and manage than multiple microservices, requiring good Microservices DevOps practices [18].
- **Potential performance overhead** due to network latency.
- **More challenging to maintain consistency and transaction integrity across services** The need for frequent communication between services is often driven by data dependencies, making data consistency a critical factor [19]. Microservices eliminate a single source of truth, requiring careful data management.
- **Increased operational complexity** due to the larger number of deployable units.
- **Potential for duplication of effort**

The project's decision to employ a microservices strategy to guarantee scalability, modular development, and a distinct division of responsibilities across components such as data processing, skill extraction, scraping, and user management is in line with the recommended uses of microservices. When scalability, high fault tolerance, and elasticity are important considerations, microservice architecture is regarded as a great option. Microservices' independence facilitates modular development and the capacity to independently manage every component.

3.2.1 Event-Driven Architecture: Benefits and Challenges

Decoupled event processing components that receive and process events asynchronously are made possible by the distributed, asynchronous architecture approach known as event-driven architecture (EDA), in which services respond to events rather than direct requests. An asynchronous workflow in an EDA is started by an event. By separating each component inside its context, concentrating on its distinct duties, and processing incoming data, this method makes it easier to represent extremely complex systems.

Benefits of EDA

- **Loose coupling between services enables more flexibility**

  Decoupled event processing components receive and process events asynchronously in an event-driven system. By enabling teams to work independently on its components—each of which is defined by explicit contracts—this decoupling can also help organizations scale.

- **High scalability due to asynchronous, non-blocking communication**

  Without a doubt, the best option for attaining great performance and scalability is event-driven design. Because these systems are parallel, the asynchronous communication made possible by publish/subscribe or message systems—which are frequently employed in EDA—can reduce the final consistency time factor. Applications that require scalable and dynamic communication can benefit from the publish/subscribe paradigm, which decouples services to offer greater flexibility and faster communication [20].

- **Better system resilience and fault tolerance**

  The best option for attaining high fault tolerance is an event-driven architecture. If one of the detached components fails, the remaining components can still function.

- **Enables real-time processing and responsiveness**

  Asynchronous processes, event notifications, and data streaming for high-load systems like streaming analytics are all handled by publish/subscribe systems [20]. Using the publish/subscribe model, Apache Kafka is a distributed message streaming platform built for applications that need low latency and real-time processing.

Challenges of EDA:

- **Increased architectural complexity**

  EDA introduces complexity by nature as a distributed asynchronous design. There are many difficulties in controlling the sequence in which events are consumed and dealing with mistakes [6].

- **Eventual consistency instead of immediate transactional consistency**

  For request-based systems that need strong data consistency, the event-driven architectural approach is not advised [6]. It is not appropriate for situations requiring instantaneous data accuracy because of its asynchronous nature and eventual consistency of processing, which leave no certainty of when an event will be processed. The usage of patterns such as orchestrated request-based patterns, event-based patterns, and background synchronization—all of which have their own complexities and the possibility for coupling or error handling problems—as well as strategic choices are necessary to achieve eventual consistency.

- **Requires careful design to avoid message duplication or loss**

  Error handling adds another layer of complexity in EDA. While messaging systems in event-based patterns ensure message delivery when consumers are available, failures during the processing of these events can still lead to consistency issues. While Kafka doesn't initially guarantee every published message is received, it prioritizes throughput, with plans to address durability for critical data [21]. However, careful configuration and potentially the implementation of patterns like the Saga Pattern might be needed to ensure data consistency across services.

- **Harder to debug and trace flow across multiple services (requires centralized logging or observability tools)**

  Regardless of the architectural style, observability is essential for any meaningful corporate application. Tracing the flow of events across services becomes increasingly difficult in a distributed EDA and calls for strong logging, monitoring, and maybe service discovery protocols.

The advantages of EDA are utilized in this project by utilizing an event-driven methodology that uses Kafka for asynchronous communication between the job scraper, AI skill analyzer, and job matching engine. The advantages of conventional log aggregators and messaging systems are combined in Kafka, a distributed messaging system that was first created for log processing. It provides high throughput and scalability while enabling real-time event consumption [21]. Because of its architecture, which is built on subjects and partitions that are divided among brokers, services can operate independently and scale in accordance with their workload. The project's objective of enhancing performance and maintainability through autonomous service operation and scalability is in line with this approach.

3.2.2 Containerization and Orchestration

The process of containerization, which is most frequently accomplished with Docker, bundles application code, dependencies, and environment variables into small, portable containers. Containers guarantee that software operates consistently without "it works on my machine" problems in a variety of settings, including development, testing, and production [9]. Each container enables a workload to have exclusive access to resources like processor, memory, and libraries, which is essential for development.

Benefits of containerization:

- **Environment consistency across development and deployment** is ensured because containers package all necessary libraries and dependencies, allowing them to run independently as isolated processes within an operating system.
- **Rapid provisioning and easy rollback during failures** are facilitated by containers, as they can start fast. A single service can be changed separately in microservices designs, and if an issue arises, the service can be promptly separated, facilitating swift rollback.
- **Simplified dependency management** is a result of each container being a package of libraries and dependencies necessary for its operation, making it independent.

Orchestration tools are used to manage many containers in production. Particularly in local development, Docker Compose is used to define and execute multi-container Docker applications. Kubernetes is a well-liked open-source cluster management for Docker that is ideal for large-scale deployments.

Key orchestration capabilities:

- **Automatic service discovery and load balancing**

Provided by orchestration tools like Kubernetes, which can dynamically adjust the number of application instances according to load, improving resource utilization and ensuring high availability [18].

- **Health checks and self-healing containers**

  Enabled by integrating container deployments like Docker, allowing for automated restarting of crashed microservices through configurations.

- **Resource management and scaling**

  Dynamically scale applications based on demand to improve the utilization of system resources. Services within containers can be scaled and updated independently.

- **Declarative deployment configurations**

  Suported by tools like Docker Compose, which uses files like docker-compose.yml to define the configurations of the multi-container environment.

All of the services in this project were containerized using Docker, and they were coordinated during system integration and local development using Docker Compose. This made it possible to design microservices in isolation and made multi-service testing easier.

3.3 Message-based Communication Systems

Modern distributed systems, especially those based on microservices, are based on the fundamental idea of message-based communication, which permits effective, decoupled, and asynchronous interactions between services [6]. This architectural approach enhances system modularity, fault tolerance, and scalability by enabling independent services to communicate without direct dependencies [11]. Data transfers between services are guaranteed to be seamless even in the event of a partial system failure or high load thanks to a well-designed messaging system [20]. Because of its high throughput, robustness, and real-time processing capabilities, Apache Kafka has become one of the most popular messaging and event-streaming platforms among the available technologies [21].

In order to share specific information, the microservices architecture itself depends on communicating entities, frequently using simple techniques [6]. This architecture's main objective is decoupling, which is facilitated by the autonomous deployment and development of microservices. By enabling communication between different application services, message brokers are a crucial and indispensable component of microservices. These brokers need high throughput and low latency since they process a wide range of data in large volumes in brief amounts of time [20]. Asynchronous communication between services is supported by the usage of paradigms like Publish/Subscribe (pubsub) and Message Queues, which further decouple them and offer greater flexibility, speed, scalability, and dependability [20].

With its scalability and high throughput, Apache Kafka is a distributed messaging system that combines the advantages of conventional log aggregators and messaging systems, enabling real-time event consumption by applications. It was created to gather and send large amounts of log data with little latency [21]. By enabling producers to submit sets of messages in a single request and consumers to get many messages in a pull request, Kafka facilitates efficient transmission. Kafka places a higher priority on throughput and scalability than certain conventional messaging systems that might concentrate on extensive delivery guarantees. Multiple producers and consumers can work simultaneously thanks to its design, which consists of brokers and divides communications into themes and partitions [22]. Kafka's design for horizontal scaling allows for the addition of more brokers to handle increasing workloads.

3.3.1 Kafka Architecture and Principles

Apache Kafka is a distributed message streaming platform. Originally developed at LinkedIn [23], it was designed for collecting and delivering high volumes of log data with low latency [21].

Fundamentally, Kafka operates as a publish-subscribe messaging service in which a consumer (subscriber) reads messages from a subscribed topic after a producer (publisher) writes messages to a Kafka topic. The distributed Publish/-Subscribe paradigm is the name given to this. Producers have the capacity to publish data into specific subject partitions, and consumers with a group name can tailor their consumption from specific partitions while jointly consuming from topics of the same group name. Kafka's publish/subscribe model enables asynchronous communication between services, separating them to offer more scalability, dependability, flexibility, and communication speed. Time, space, and synchronization decoupling are the methods used to achieve this decoupling.

In the Kafka cluster, each topic is separated into one or more partitions, which serve as the physical storage for messages. Kafka makes horizontal scalability possible in large part through partitions [22]. The subject's ability to handle messages and read them more quickly can be greatly increased by hosting the topic on numerous servers, each of which can manage a partition, if one server is unable to handle the processing power for the topic.

Key architectural components of Kafka include:

- **Brokers:** Kafka servers that store and distribute data. A Kafka cluster typically consists of multiple brokers. Load balancing in Kafka is done by brokers as they can access multiple partitions simultaneously, increasing throughput.
- **Producers:** Clients that push data (events) into Kafka topics. For efficiency, the producer can send a set of messages in a single publish request.
- **Consumers:** Clients who extract data from the brokers to consume the subscribed messages after subscribing to one or more of the brokers' topics. Customers typically belong to a broader consumer group that shares a common interest. In order to ensure that no more than one consumer receives messages from the same partition, Kafka automatically separates the consumers in a group to receive messages effectively. To keep track of the communications they have received, each customer has an offset.
- **Zookeepers:** A Hadoop stack-based service that oversees all Kafka brokers [20]. Zookeeper is used by Kafka for activities like as preserving consumption relationships, initiating rebalance operations, and detecting the addition and removal of brokers and consumers.
- **Topics and Partitions:** Logically, topics are groups of messages. One or more partitions can be set up for a topic. Scalability and load balancing are guaranteed by partitions. Within a partition, messages are kept in the order that they are sent.

3.3.2 Comparison with Alternative Messaging System

Despite its strength, Kafka is not the only message broker option. RabbitMQ and ActiveMQ are two other noteworthy systems, each with unique advantages and recommended applications.

3.3.3 Scalability and Reliability Considerations

Topics that are separated into one or more partitions can be scaled horizontally using Apache Kafka. By hosting these partitions on numerous servers, the capacity to handle messages and read them more quickly can be increased. Producers can write messages in parallel thanks to this partitioning, and consumers can read messages in parallel [23].

A collection of consumers subscribe to subjects in Kafka's consumer group model. In order to provide load balancing, Kafka automatically divides the workload across the consumers in a group, making sure that no more than one consumer

Table 3.1: Comparison of Messaging Systems

Feature	Kafka	RabbitMQ	ActiveMQ
Message Model	Log-based, publish-subscribe	Queue-based, message broker	Queue-based, message broker
Message Retention	Configurable (long-term)	Transient (auto-delete)	Transient or persistent
Performance & Throughput	Very high	Moderate	Moderate
Ordering Guarantees	Per-partition	Optional via queues	Limited
Durability	High (log + replication)	Medium (manual setup)	Medium
Ecosystem & Tooling	Kafka Streams, Kafka Connect	Plugins, management tools	Older, JMS-based tooling
Use Case Fit	Event streaming, logs	Task queues, short messages	Legacy systems

normally receives messages from the same partition. Fault tolerance is also supported by this paradigm; Kafka may re-balance the workload among the servers that are still operational without losing any data in the event of a server failure [22].

In terms of reliability, Kafka offers:

- **Replication** of partitions to prevent data loss.
- **Message durability** through disk persistence.
- **Back-pressure handling** to prevent overwhelming consumers.

These characteristics make it possible for Kafka to support extremely dependable and scalable systems, which are crucial for mission-critical applications like job aggregation platforms like the one created for this project, banking, and the Internet of Things.

Kafka does have operational complexity, though; careful setup and monitoring are needed for partition balancing, disk utilization monitoring, and cluster administration. However, Kafka is frequently the recommended option for situations where fault tolerance and throughput are of at most importance.

3.4 Natural Language Processing in Recruitment

3.4.1 Evolution of NLP for Skills Extraction

Natural language processing, or NLP, has become increasingly important in the hiring process, especially when it comes to automating the process of extracting talents from job listings and resumes [14]. In order to determine talents, traditional methods initially mostly relied on keyword matching and manually created pattern rules [12]. Due to their lack of semantic awareness, these early algorithms frequently overlooked phrase, synonym, and context alterations. Semantically comparable terms, for instance, would not always be regarded as equivalent.

By utilizing contextual awareness, machine learning advancements such as statistical natural language processing and later deep learning techniques allowed for more sophisticated skill extraction [14]. These methods made it possible to identify concepts and abilities more flexibly, even when they were expressed in different ways. An important development was the use of massive pre-trained language models, such as BERT, which gave computers the ability to handle a variety of linguistic patterns more accurately and robustly. For example, BERT generates "informative and high-quality document representations" by analyzing the text of resumes and job postings.

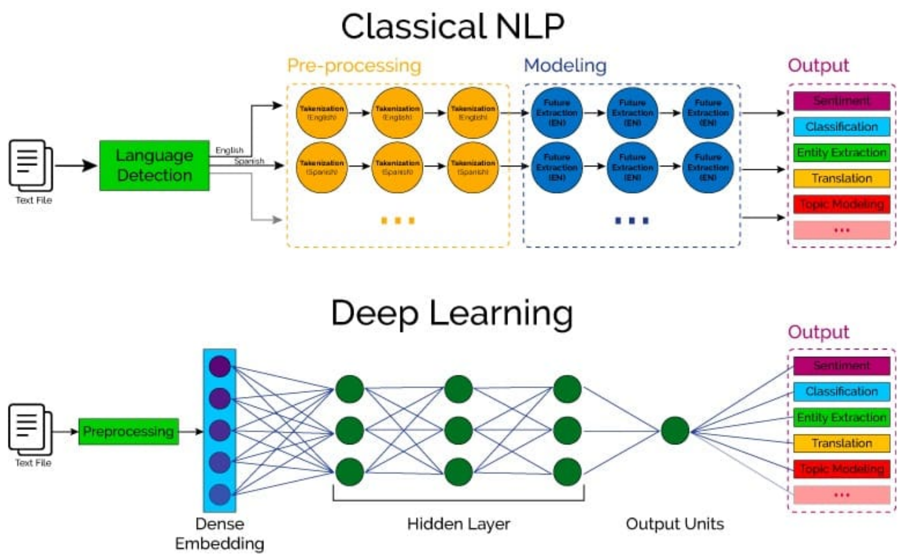

Figure 3.1: Comparison between classical NLP and deep learning-based NLP pipelines for various tasks including entity extraction.

As illustrated in Figure 3.1, classical NLP relies on rule-based and statistical methods with heavy preprocessing, whereas modern deep learning approaches like transformer-based models streamline the process using end-to-end learning.

3.4.2 Current State-of-the-Art Approaches

The best skill extraction models available today make use of methods like transformer-based contextual modeling, sequence labeling, and Named Entity Recognition (NER). Beyond only keywords, these models are able to accurately and reliably identify skill entities across a wide range of sectors and employment forms.

Common approaches include:

- **NER Models** to tag domain-specific terms (e.g., "React", "Machine Learning", "Kubernetes") as skills.
- **Token Classification Models**, which label individual tokens using schemes like BIO (Beginning, Inside, Outside) for multi-word skills.
- **Transfer Learning Techniques**, where pre-trained models are fine-tuned on annotated CV and job description datasets.

In this project, the Hugging Face model `jjzha/jobbert_skill_extraction` was used. This model is designed for token-level classification and is fine-tuned specifically for job-related documents, making it highly suitable for real-world applications in recruitment automation. It accurately extracts structured skill data from unstructured job text, which is later used for matching candidates with jobs based on overlapping or complementary skill sets.

3.4.3 Transformer Models and Token Classification

Transformer models that take into account the context of each word in a phrase, like BERT, have greatly improved skill extraction [24]. The ability of these architectures, such as BERT, to comprehend bi-directional context is essential for deciphering complex language in resumes and job advertisements.

A useful method for classifying tokens is Named Entity Recognition (NER), in which each token is labeled to identify the skills present in the text [25].

3.4.4 Special Considerations for CV and Job Description Processing

Processing resumes and job descriptions introduces several domain-specific and format-specific challenges that must be addressed to ensure accurate skill extraction:

- **PDF Format Variability**

  CVs and job descriptions are often provided as PDF files, which may include non-linear layouts, columns, tables, headers, and footers. Converting these into plain text while preserving semantic structure is non-trivial and essential for accurate downstream analysis.

- **Formatting Noise**

  Bullet points, tabs, and section misalignment can introduce noise, especially when important content such as experience or education is split across lines or blocks.

- **Synonym Ambiguity**

  Many skills or roles are described using different terms—e.g., "software developer" vs. "programmer" or "DevOps" vs. "Site Reliability Engineer".

- **Domain-Specific Language**

  Specialized terminology in domains like healthcare, finance, or IT may not be well recognized by general-purpose models.

- **Ambiguity of Terms**

  Words such as "Java" or "Spring" can refer to technologies or seasons, depending on the context.

To address these challenges, this project incorporates a pre-processing step that converts PDF documents into clean, structured plain text using a PDF-to-text conversion utility. The processed text is then passed to a transformer-based model, fine-tuned for recruitment data—specifically, the `jjzha/jobbert_skill_extraction` model from Hugging Face. This approach improves the accuracy of skill identification by ensuring the input to the model is clean, structured, and context-aware.

3.5 Web Scraping Technologies and Approaches

A key feature of the Job Aggregator Platform is web scraping, which makes it possible to automatically gather job ads from multiple internet sources. This section examines important resources and factors to take into account while using web scraping in an ethical and successful manner.

3.5.1 Browser Automation Frameworks Comparison

For websites that significantly rely on JavaScript rendering, modern online scraping has advanced beyond basic HTML parsing to include complete browser automation. The top browser automation tools are Playwright, Puppeteer, and Selenium. Below is a comparison of these tools:

Feature	**Selenium**	**Puppeteer**	**Playwright (Used in this Project)**
Language Support	Multiple (Java, Python, etc.)	JavaScript/Node.js	JavaScript, Python, Java, C#
Multi-browser Support	Yes	Chrome/Chromium only	Chromium, Firefox, WebKit
Headless Mode	Yes	Yes	Yes
Handling Modern JS Sites	Moderate	Good	Excellent
Auto-wait Mechanisms	Manual	Some	Built-in auto-waiting

Table 3.2: Comparison of Browser Automation Frameworks

Playwright was chosen for this project due to its robust support for modern web applications, multi-browser automation, and native support for async operations and handling dynamic content.

3.5.2 Legal and Ethical Considertions in Web Scraing

Web scraping raises important legal and ethical concerns. While the pratice itself is not illegal, scraping content without permission can violate terms of service of websites, intellectual property laws, and data privacy regulations such as GDPR.

Furthermore, job data scraped is intended solely for analysis and research purposes in a non-commercial academic context.

3.5.3 Challenges in Extracting Structured Data from Job Sites

Extracting structured data from job boards such as IrishJobs or Jobs.IE presents multiple challenges.

- **Dynamic content loading**: Many job listings are rendered with JavaScript after page load.
- **Anti-scraping mechanisms**: Websites implement CAPTCHAs, rate-limiting, and bot detection to prevent scraping.

- **Inconsistent structure**: Job titles, company names, and descriptions may vary in format or location on different pages or platforms.
- **Pagination and infinite scrolling**: Requires intelligent navigation strategies.

To address these challenges, the platform uses Playwright's robust capabilities to simulate user behavior, navigate through pagination, and extract content after JavaScript execution.

3.6 Technology Stack Selection

The Job Aggregator Platform was designed using a combination of modern, scalable, and production-ready technologies. The selection process was guided by the platform's functional and non-functional requirements, such as modularity, performance, ease of development, scalability, and interoperability.

3.6.1 Backend Technologies

The backend consists of a hybrid microservice architecture implemented using Spring Boot (Java) and Python-based services.

- **Spring Boot** was chosen for its robustness, strong ecosystem support, and efficient handling of concurrent RESTful API requuests. It powers ervices responsible for user authentication, job storage, and job matching.
- **Python** was used for services that deal with web scraping and natural language processing due to its simplicity and the availability of powerful libraries and frameworks (e.g., Playwright for scraping and Hugging Face models for skill extraction).

This division allows each language to be used in the domain where it excels, adhering to the principle of using the right tool for the right task.

3.6.2 Frontend Technology

The frontend of the Job Aggregator Platform was developed using React combined with TypeScript, providing a powerful and modern framework for building scalable and maintainable user interfaces. React's component-based architecture allows for modular development, where each UI element can be independently developed, tested, and reused across the application. This structure significantly enhances the maintainability and scalability of the codebase.

The integration of TypeScript introduces static typing into the frontend code, allowing for early detection of potential bugs and improved code readability. TypeScript's support for strong typing is particularly beneficial in a microservice-based architecture, where the frontend must interact with various backend APIs for features such as job search, profile management, and job matching. With TypeScript, API responses and component props are more predictable, reducing runtime errors and enhancing developer productivity.

3.6.3 Data Storage

MySQL was chosen for persistent storage because of its dependability, ACID compliance, and ability to support sophisticated queries that are appropriate for user and job data. Redis was incorporated concurrently for inter-service fast lookups and temporary data caching, especially for real-time employment data and skill extraction outcomes.

3.6.4 Messaging System

To facilitate asynchronous communication between microservices, the system makes use of Apache Kafka as the central message architecture. Kafka is perfect for streaming job listings and coordinating amongst scraping, analysis, and matching services because it guarantees high throughput, fault-tolerant message delivery, and scalability.

3.6.5 Containerization and Deployment

All services were containerized using Docker, guaranteeing uniformity across development and production environments. Local multi-container configuration orchestration is done by Docker Compose. This choice opens the door for future scalable cloud deployment and makes it simple to replicate, test, and isolate services.

To further illustrate how each technology supports the system's goals, 3.3 maps each component to its purpose and the requirement it fulfills.

Table 3.3: Technology Stack Overview

Technology	**Purpose**	**Requirement Fulfilled**
Spring Boot	REST APIs, user management, job services	Scalability, performance, modularity
Python	Web scraping, NLP skill extraction	Flexibility, AI model integration
React	Frontend interface	User interaction, responsive UI
MySQL	Persistent user and job data storage	Data integrity, query support
Redis	Caching and temporary job data	Low latency, quick access to in-transit data
Kafka	Message brokering between services	Scalability, asynchronous communication
Docker	Containerization and local deployment	Portability, consistency, scalability

Chapter 4

System Design

4.1 Architectural Overview

The Job Aggregator Platform's architectural design is thoroughly described in this section. The system's design is based on microservices, which allow for fault separation, independent scalability, and modular development. The platform consists of a React-based frontend, several backend services written in Python and Spring Boot, and auxiliary components like Redis, Kafka, and MySQL for caching, persistence, and messaging, respectively.

4.1.1 High-Level System Architecture Diagram

The Job Aggregator Platform adopts a distributed architecture, where each core functionality is encapsulated within its own containerized microservice. Services communicate primarily through Kafka to ensure asynchronous, decoupled interaction.

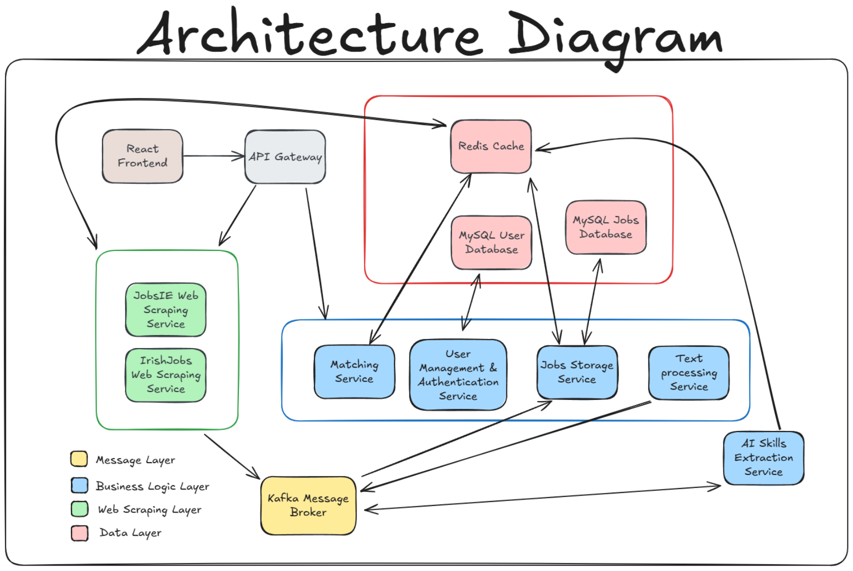

Figure 4.1: High-Level System Architecture of the Job Aggregator Platform

4.1.2 Microservices Approach

The system is designed around several loosely coupled microservices. Each service handles a specific responsibility and can be deployed and scaled independently. This approach supports:

- Independent development and updates
- Better fault tolerance
- Easier horizontal scaling

The platform consists of the following core microservices:

Service Name	**Technology Stack**	**Responsibility**
User Service	Spring Boot, MySQL	Handles user registration, login, and profile management
Scraper Service	Python, Playwright, Redis, Kafka	Performs real-time job scraping from external job boards with caching to prevent duplicate requests
Job Storage Service	Spring Boot, Kafka, MySQL	Persists scraped job data into the database
Text Processing Service	Python, pdfplumber, Kafka	Converts uploaded CVs from PDF to raw text for further processing
Skill Extractor Service	Python, Hugging Face Transformers, Redis	Extracts skills from job descriptions using jjzha/jobbert_skill_extraction and caches extracted skills
Matching Service	Spring Boot, Redis, Kafka	Matches job listings to users based on extracted skills
Frontend Client	React, TypeScript, REST API	Provides the user interface for searching and interacting with job listings

Table 4.1: Core Microservices of the Job Aggregator Platform

4.1.3 Data Flow Visualization

The system uses event-driven architecture, where data is passed between services using Kafka topics. This design allows services to operate independently while ensuring that data flows efficiently across the system.

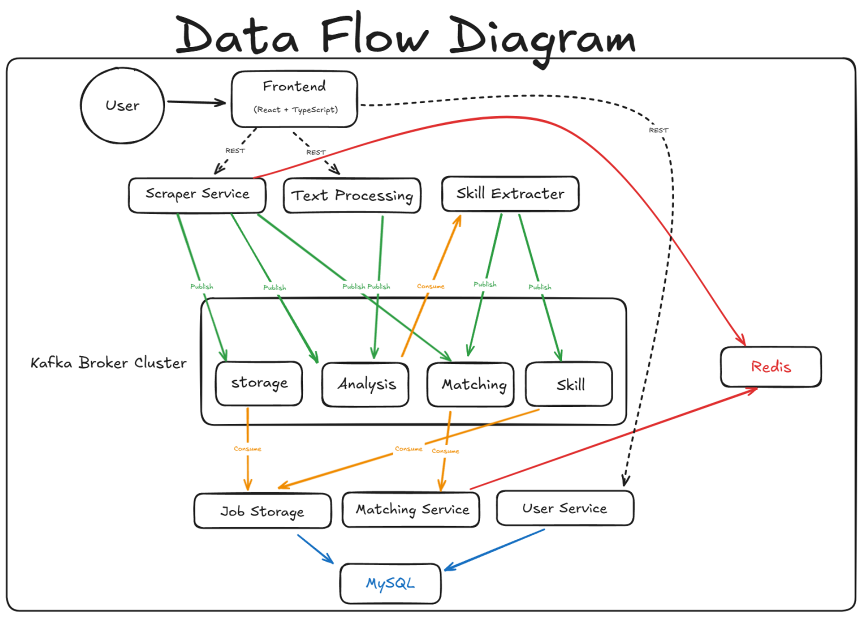

Figure 4.2: Data Flow Diagram of the Job Aggregator Platform

Figure 4.2 illustrates Kafka topics, how services publish/consume, and where data is persisted or cached.

Example Flow:

1. A user triggers a job search
2. The **Scraper Service** collects job postings and publishes them to Kafka
3. The **Storage Service** consumes this data and saves it to MySQL
4. The **Skill Extractor Service** reads from another topic and applies NLP to extract skills. Then save it into Redis and MySQL.
5. The **Matching Service** reads from another topic and applies for rank jobs.
6. Results are stored in Redis

4.1.4 Component Interaction patterns

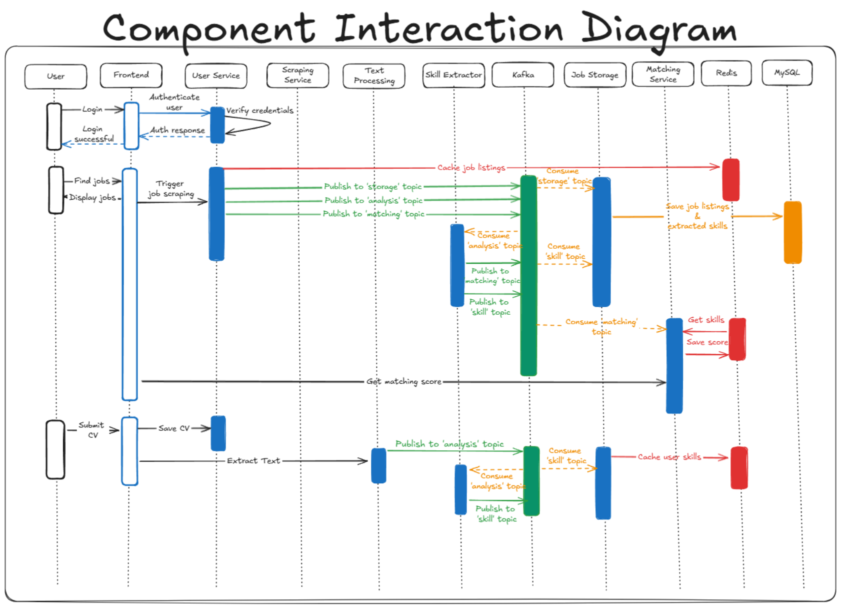

Figure 4.3: Data Flow Diagram of the Job Aggregator Platform

Figure 4.3 illustrates sequential interactions between components. It shows user actions (login, job search, CV upload). It covers interactions between services like frontend, backend, Kafka topics, job storage, and Reids/MySQL.

Component communication is handled via:

- **Kafka topics** for asynchronous, decoupled messaging
- **RESTful APIs** for synchronous communication between the frontend and backend
- **Redis** for fast, temporary data access
- **MySQL** for permanent storage of user and job data

This pattern allows:

- Scalability of intensive components like scraping and skill extraction
- Efficient job matching through fast skill retrieval
- Reduced interdependencies between services

4.2 Core Components and Services

This section elaborates on the key microservices and components that form the backbone of the Job Aggregator Platform. Each service is independently developed, containerized, and communicates through a messaging system (Kafka), following a microservices-based architecture. The frontend interfaces with users while backend services handle processing, storage, scraping, and matching logic.

4.2.1 User Management Service

This service handles all user-related functionalities, ensuring secure access and personalized experiences.

- **Authentication and Authorization:**

  Utilizes JWT tokens for stateless authentication. Spring Security is used for role-based access control, ensuring that sensitive endpoints are protected.

- **User Profile Management:**

  Allows users to manage personal data, upload CVs. Profiles are stored in MySQL for persistence and CV is cached in Redis.

- **Password Management and Security Features:**

  Includes secure password hashing using BCrypt, token-based password reset, and protection against brute-force attacks.

Security Feature	**Implementation**	**Advantages of Your Approach**
Authentication Mechanism	JWT Token-based	Stateless, scalable, suitable for microservices
Token Storage	Authorization Header	Prevents XSS, follows standard practices
Password Hashing	BCrypt (via BCryptPasswordEncoder)	Adaptive hashing, built-in salt (randomly generated per password), industry-standard
Access Control	Role-based (ROLE_USER, ROLE_ADMIN)	Simple to implement, widely understood
Password Reset	Time-limited tokens (24 hours)	Self-service, doesn't require additional user data
OAuth Support	Google integration	Widely used, reliable identity provider
Secure Headers	CORS Configuration	Prevents cross-origin attacks
Profile Security	User-owned resources with ID validation	Direct ownership model, simple to implement

Table 4.2: Security Implementation Details

4.2.2 Jobs Storage Service

The service is responsible for receiving, storing, and indexing job data scraped from various platforms.

- **Job Data Model and Storage:**

  Uses a normalized schema in MySQL to store job details. Each listing includes ID, title, company, location, job description, apply link, and extracted keywords.

- **Search and Retrieval Mechanisms:**

  Jobs can be retrieved by filters like title, location, or skill match. Indexed queries enhance response speed.

- **Caching Strategy:**

  Frequently accessed jobs and skills are cached in Redis to reduce load and improve retrieval performance.

4.2.3 Job Scraping Services

The platform incorporates two dedicated web scraping services—JobsIE Scraper and IrishJobs Scraper—designed using asynchronous Playwright for robust and scalable job data collection.

- **JobsIE Scraper Implementation:**

  This service targets specific DOM elements and utilizes JavaScript evaluation techniques to extract job listings. It manages paginated content, handles cooking consent dialogs automatically, and performs batch processing of jobs to optimize system resources.

- **IrishJobs Scraper Architecture:**

  Similar in design, this scraper uses custom selectors to extract rich job metadata, including job descriptions and company information. It features enhanced retry logic and structured error recovery to ensure stability during scraping sessions.

- **Key Features and Integrations:**

  – **Playwright Automation** ensures reliable browser control across dynamic job portals.

  – **Error Handling & Retry Mechanisms** are in place for network issues and DOM inconsistencies.

  – **Duplicate Prevention** is implemented using URL tracking to skip previously scraped listings.

  – **Integrtion with Kafka** allows both scrapers to publish job data to a centralized messaging queue for downstream processing by the matching service.

  – **Redis Caching** supports temporary job data storage to reduce reprocessing.

- **Anti-Detection & Stealth Techniques:**

  – Playwright's stealth mode is used to mimic human-like browser behavior.

  – Delays are applied between actions.

  – User-Agent strings are supported through browser configuration but do not yet rotate dynamically.

4.2.4 AI Skills Extraction Service

The AI Skills Extraction Service is responsible for transforming unstructured textual input—whether from job descriptions or CVs—into structured skill and knowledge data using domain-tuned NLP models.

NLP Pipleline Architecture

The service follows a modular and efficient processing pipeline:

- **Pre-processing:** Text is split into manageable chunks (e.g., by sentence) to optimize performance and prevent token overflow during inference.
- **Model Inference:** Two Hugging Face transformers are employed:
  - `jjzha/jobbert_skill_extraction` for identifying technical skills
  - `jjzha/jobbert_knowledge_extraction` for capturing knowledge areas
- **Post-processing:** Extracted entities are normalized, deduplicated, and stored as a set to ensure consistency.

Model Selection and Implementation

JobBERT was selected for its pre-training on job-related datasets, making it well-suited for domain-specific skill tagging. The service uses Hugging Face's `pipeline()` API with the parameter `aggregation_strategy="first"` to capture the initial occurrence of relevant entities, ensuring high precision.

4.2.5 Matching Service

The Matching Service is responsible for analyzing the similarity between a user's skill set and the requirements specified in job listings. It assigns a relevance score to each job and ranks them accordingly to present the most suitable opportunities to the user.

- **Skill Comparison Alogrithm**

  Rather than relying on simple set intersection, the service uses fuzzy string matching based on the Levenshtein distance to compare user skills and job-required skills. Skills are considered a match when they exceed a similarity threshold of 0.6, allowing the system to capture close but non-exact matches (e.g., "React" vs. "ReactJS"). The comparison is also case-insensitive to further improve matching accuracy.

- **Scoring System Design**

  The service calculates a match score for each job as a percentage, based on how many job-required skills are sufficiently similar to the user's skills. These scores are normalized (0–100%) and stored in Redis using a `match{userId}` key format, with job IDs as hash keys. This allows fast lookup and integration with other services.

- **Result Ranking Methodology**

  Jobs are ranked based on their computed skill match scores. This ranking is separate from general search relevance (such as filtering by company, location, or job type), which is handled independently in the Jobs Storage Service. The Matching Service focuses strictly on skill-based ranking.

- **Additional Features**

  - **threshold Filtering**: Jobs that do not meet a minimum match threshold are excluded to maintain recommendation quality.
  - **Robust Matching Logic**: Includes edge case handling and comprehensive test coverage (exact match, similar match, partial match).

4.2.6 Frontend Application

The frontend of the Job Aggregator Platform is a React-based application that enables users to interact with the system through an intuitive and responsive interface. The application is organized into modular components and communicates with backend services for authentication, job search, CV uploads, and job matching results.

Component Structure

The frontend is built using a modular approach, with key components dedicated to specific functionalities:

- **Authentication:**

  - `LoginPage.tsx` and `SignupPage.tsx` handle user login and registration.

- **Job Search:**

  - `SearchBar.tsx` allows users to search terms.

  - `SearchResultPage.tsx` displays paginated or filtered search results.

- **CV Upload & Profile Management:**
  - `EditCV.tsx` and `CompleteProfile.tsx` allow users to upload and manage their CVs and skills.

- **Result Display:**
  - `MainPage.tsx` renders personalized job recommendations with match scores and filtering options.

State Management

- The application relies on **React hooks** such as `useState`, `useEffect`, and `useRef` for managing local state.
- API communication is handled directly using Fetch API
- Authentication state is managed via cookies, using the `js-cookie` library for storing and retrieving session tokens.

User Interface Design Principles

- The UI is responsive and clean, ensuring usability across desktop and mobile devices.
- The application uses traditional CSS for each component.
- Media queries are applied for responsiveness, allowing layouts to adapt to different screen sizes and orientations.

Additional Features

- **Form Validation:** Built-in form-level validation with real-time error display.
- **OAuth Integration:** Google sign-in is supported for easier account access.
- **Password Reset Functionality:** Users can reset their passwords securely via the interface.
- **Profile Editing:** Users can update their personal details, skills, and CV through multiple views.

- **Search History Tracking:** Previously searched queries are saved and displayed for user convenience.

- **Match Score Visualization:** Each job listing shows a matching percentage score derived from the Matching Service.

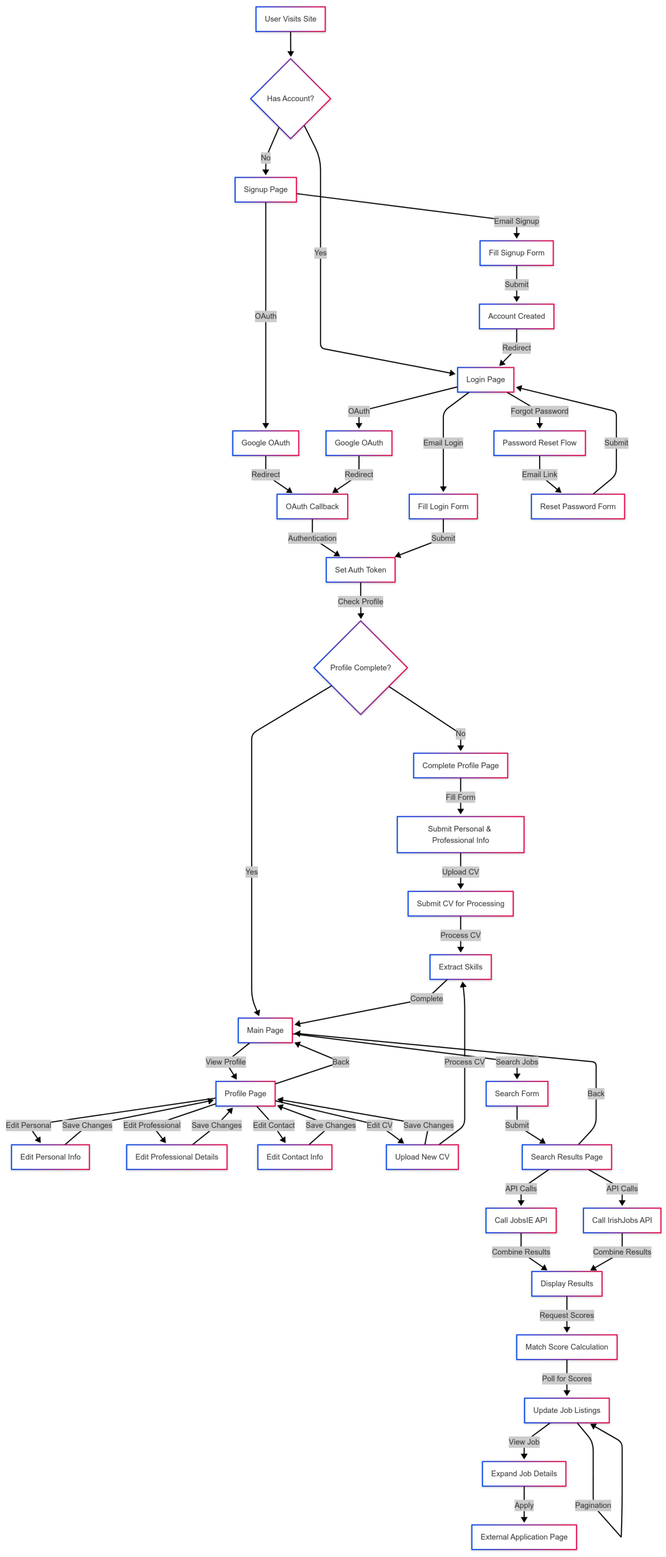

Figure 4.4: User Flow Diagram

Figure 4.4 visualizes the navigation and user interactions within the application, the following UI flow diagram outlines the primary paths a user can take throughout the system, from login and registration to searching for jobs, uploading CVs, and viewing recommendations.

4.3 Data Models and Schema Design

This section outlines the core data models and their relationships across the microservices architecture. It includes the schema design for persistent storage (relational databases) as well as the in-memory data structure design used in Redis for temporary or high-speed access data.

4.3.1 User Data Model

The User Management Service separates authentication details from profile-related attributes using a two-entity design:

- **User Entity**

  Responsible for authentication and role management.

Field	**Type**	**Description**
id	UUID	Primary key
firstName	String	User's first name
lastName	String	User's last name
email	String	Unique email for login
password	String	Hashed password
roles	Set<Role>	Many-to-many relationship with roles

Table 4.3: User Entity Schema

- **UserProfile Entity**

  Contains user-specific details including resume and skill preferences.

Field	**Type**	**Description**
id	String	Primary key, shared with User entity
user	User	One-to-one relationship with User
phoneNumber	String	User's contact number
address	String	User's physical address
education	String	Educational background
jobTitle	String	Current or desired job title
company	String	Current or previous employer
cvData	byte[]	PDF resume data stored as LONG-BLOB
cvName	String	Original filename of uploaded CV
isComplete	boolean	Profile completion status flag

Table 4.4: UserProfile Entity Schema

4.3.2 Job Listing Data Model

The Jobs Storage Service manages job data. Each job listing includes both source and extracted metadata to support advanced filtering and AI-based matching.

- **Jobs Entity**

Field	**Type**	**Description**
jobId	String	Primary key, a unique identifier for job listings
title	String	Job position title
company	String	Company offering the position
location	String	Job location (city, remote, etc.)
applyLink	String (CLOB)	URL for job application
jobDescription	String (CLOB)	Full text description of the job
platform	String	Source platform (e.g., JobsIE, IrishJobs)
timestamp	Date	When the job was scraped or added
searchTitle	String	Title used in original search query
searchLocation	String	Location used in original search query
skills	Set<String>	Collection of skills extracted from job description

Table 4.5: Jobs Entity Schema

4.3.3 ERD Diagram

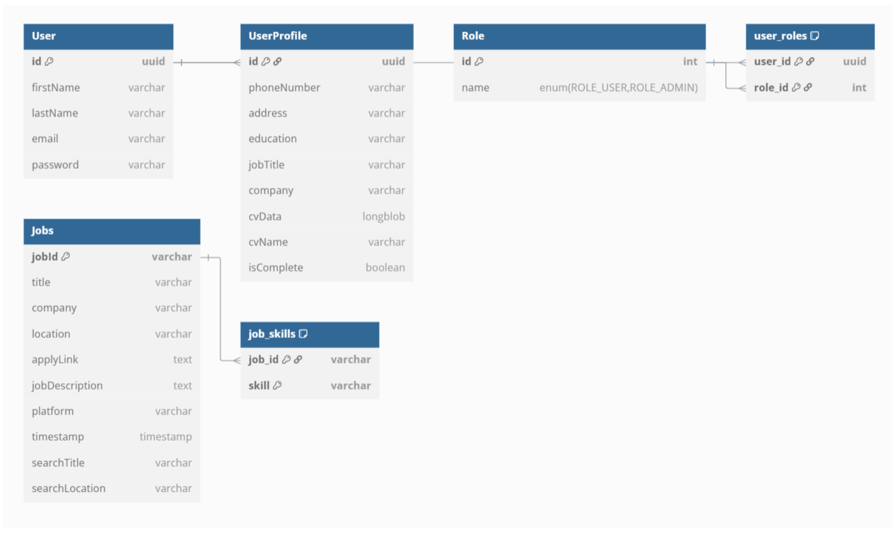

Figure 4.5: User Flow Diagram

Figure 4.5 summarizes the database relationships across the user and job microservices."

4.4 Inter-Service Communication

The system uses a hybrid communication approach that combines synchronous REST APIs and asynchronous messaging (Kafka) to guarantee fault-tolerant, scalable, and decoupled microservices. The system can handle high-throughput operations like skill extraction and scraping asynchronously thanks to its dual-mode connection, while still allowing real-time interactions for user-driven features.

4.4.1 Kafka Topic Design and Message Structures

The platform leverages Apache Kafka to decouple critical services and enable asynchronous processing of high-volume tasks such as job scraping, skill extraction, and user-job matching. Kafka facilitates fault-tolerant message delivery and distributed processing.

Kafka Architecture Overview

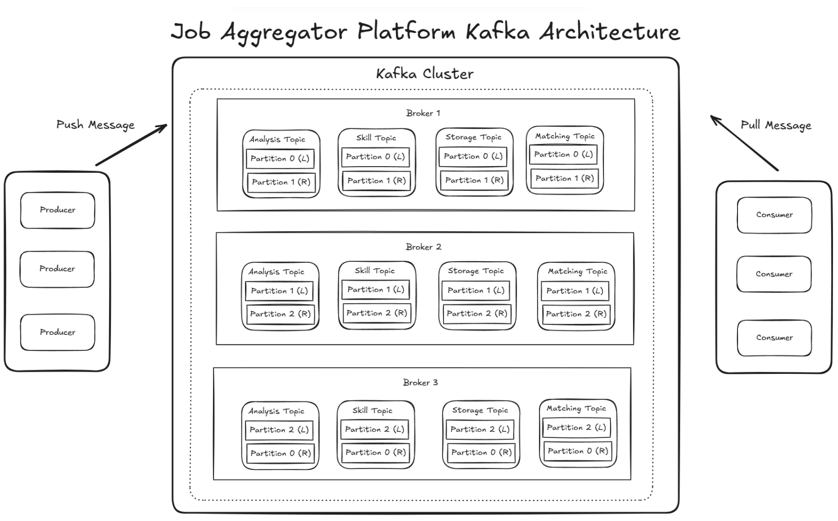

Figure 4.6: Kafka-based communication between microservices in the Job Aggregator Platform. This architecture supports asynchronous data flow between scrapers, analyzers, matchers, and storage components.

Kafka Topics and Service Interaction

Kafka Topic	**Producer Service**	**Consumer Service(s)**	**Purpose**
storage	Job Scraper Services	Jobs Storage Service	Carries job listings scraped from platforms like IrishJobs
analysis	Jobs Storage Service	AI Skills Extraction Service	Sends job descriptions and CVs for NLP-based skill extraction
skill	AI Skills Extraction Service	Jobs Storage Service	Returns structured skill sets for jobs or user CVs
matching	AI Skills Extraction Service	Matching Service	Triggers user-job skill comparison after skill extraction

Table 4.6: Kafka Topics and Service Interaction

Each topic is configured with 3 partitions and a replication factor of 2, ensuring parallel processing and high availability:

4.4.2 REST API Patterns

REST APIs are used for synchronous, user-facing operations including user authentication, job search, profile management, and retrieval of match scores. Each microservice exposes a focused and RESTful interface.

User Management API

- Authentication: `/api/auth/login`, `/api/auth/register`
- Profile Management: `/api/user/profile`, `/api/user/edit`
- Password Handling `/api/password/reset`, `/api/password/update`

Jobs Storage API

- Search Endpoint: `/api/jobs/search`
  - Supports query parameters like `title`, `location`, `platform`
  - Implements pagination with metadata (`page`, `size`, `total`)
- Job Retrieval: `/api/jobs/id`

Matching Service API

- Match Scores: `/api/matching/user/userId`
  - Returns Redis-stored scores in `jobId: score` format
  - Supports real-time display of recommendations

4.4.3 Error Handling and Retry Mechanisms

While the system does not yet implement advanced resilience patterns such as exponential backoff or dead letter queues, several basic retry and error-handling strategies are employed.

Kafka-Specific Resilience

- **Retry on Connection Failure (Producer)**
- **Consumer Rebalancing**
- **Manual Offset Commit**

4.5 Security Implementation

This section outlines the key security measures implemented across the microservices of the Job Aggregator Platform, focusing on user authentication, authorization, password security, and secure API communication.

4.5.1 Authentication Approach (JWT)

The system uses JSON Web Tokens (JWT) for stateless authentication. Upon successful login, a JWT is generated and returned to the client. This token contains essential user claims such as email and roles and is signed with a secure secret key.

The token is then included in the Authorization header of each subsequent request:

```
Authorization: Bearer <token>
```

At each endpoint, Spring Security verifies the token's validity, extracts user identity, and constructs an authentication object. This approach is highly scalable and avoids the need for server-side session management.

4.5.2 Authorization Model

Access control is enforced using a role-based authorization strategy. Roles such as ROLE_USER and ROLE_ADMIN are assigned to users and persisted in the database.

Access to protected resources is restricted using Spring Security's configuration;

```
http.authorizeHttpRequests(auth -> auth
    .requestMatchers("/api/auth/**").permitAll()
    .anyRequest().authenticated());
```

4.5.3 Secure Password Handling

Passwords are securely stored using BCrypt hashing. A `PasswordEncoder` bean is used to hash and verify passwords:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

This ensures that even if database credentials are compromised, user passwords remain unreadable. Salting and multiple rounds of hashing are handled internally by BCrypt, enhancing resilience against brute-force attacks.

4.5.4 API Security Measures

Several techniques were applied to secure REST APIs and user data:

- **CORS Configuration:** Controlled origin access is enforced to restrict frontend communication to known origins.

  ```
  configuration.setAllowedOrigins(Arrays.asList
     ("http://localhost:5173", "http://
     localhost:8081"));
  ```

- **Input Validation:** All incoming data is validated to protect against injection attacks and malformed requests.
- **Public Endpoint WHitelisting:** Specific endpoints are explicitly marked as public, allowing unauthenticated access where appropriate.

  ```
  .requestMatchers("/api/auth/**", "/api/reset-
     password", "/api/resend-reset-token").
     permitAll()
  ```

- **Custom Authentication Entry Point:** Unauthorized access attempts are handled gracefully with customized error responses.

4.5.5 Additional Security Features

The system includes the following advanced features to enhance user trust and account control:

- **OAuth2 Integration:** Google OAuth login is supported, providing users with a secure and familiar alternative to manual sign-up.
- **Secure Password Reset:** A token-based password reset mechanism is implemented. Tokens have a defined expiration period and are validated upon use.

4.6 Caching Architecture

To enhance the Job Aggregator Platform's scalability and responsiveness, Redis has been implemented as a caching layer across microservices. As an in-memory data store, Redis caches frequently accessed and computationally intensive data, thereby reducing database load and eliminating redundant computations [20]. This integration accelerates user-facing operations like job search and skill matching, leading to a more efficient platform. Redis's in-memory nature provides very low latency, making it ideal for improving the speed of communication and data retrieval within the microservices architecture.

4.6.1 Redis Implementation

Redis is deployed as a centralized service via Docker and is consumed by both Java and Python-based microservices. Redis clients use structured keys and leverage various Redis data types (e.g., Hashes, Sets) to optimize cache organization and retrieval.

Use Cases:

- **Job Listings:** Cached using Redis Hashes with indexed Set groupings to support fast searches.
- **Extracted Skills:** Stored as Redis Sets, grouped by job or user ID for efficient access.
- **Matching Scores:** Stored in Redis Hashes to persist and retrieve user-job compatibility results quickly.

4.6.2 Cache Invalidation Strategy

The system primarily uses TTL-based cache eviction. This ensures that stale data is automatically removed, supporting data freshness without relying on complex logic.

TTL Configuration:

- **Job Listings:** Cached search results and job details expire after 24 hours, aligning with typical daily market changes.
- **Skills:** Extracted skills for jobs and user profiles are cached for 24 hours to prevent redundant AI processing while allowing for profile updates.

4.6.3 Performance Considerations

The Redis caching layer has led to observable improvements in both performance and resource efficiency:

- **Reduced Latency**: The claim that data retrieval for search results and match scores now occurs in constant time from memory is supported by the description of Redis as an in-memory cache with exceptionally high access rates and very small response times [20]. Furthermroe, Redis offers O(1) performance [11].
- **Optimized AI Processing**: Skill extraction, a compute-heavy operation, is cached to prevent redundant inference calls.
- **Database Offloading**: Redis alleviates the load on both relational databases and scraping modules, especially under concurrent access.

Additionally, Redis is used to prevent duplicate request processing and to index results by platform-specific search keys:

```
# Platform-specific cache keys
cache_key = f"search:{title.lower()}:{job_location.
   lower()}:irishJobs"
self.redis_client.sadd(cache_key, job_key)
```

4.7 Infrastructure Design

The Job Aggregator Platform is architected with a strong focus on modularity, resilience, and maintainability. The infrastructure is orchestrated using Docker Compose, and all services are fully containerized for platform-agnostic deployment. Key concerns like inter-service networking, persistent data storage, fault tolerance, and service startup dependencies are all managed through a well-structured containerized environment.

4.7.1 Containerization Approach

Every microservice in the platform is packaged into its own Docker container, ensuring clean separation of concerns, reproducibility, and efficient resource allocation. Dockerfiles are crafted to suit the technology stack used in each service.

4.7.2 Docker Compose Architecture

Docker Compose orchestrates a multi-container architecture encompassing:

- **Nine + microservices**
- **Dual MySQL databases** for job and user management
- **Three-node Kafka cluster with Zookeeper**
- **Redis for caching**
- **Playwright-based scrapers**
- **Skill extraction, text processing, and matching services**

Each service is defined in docker-compose.yml, and interdependencies are explicitly declared using depends_on and health check directives for robust startup control.

Kafka topics are configured with three partitions and a replication factor of two to ensure fault tolerance and high throughput.

This allows the system to handle high-throughput messaging across distributed consumers while maintaining redundancy.

4.7.3 Frontend Deployment and CI/CD Pipeline

The Job Aggregator Platform's frontend application is deployed to GitHub Pages, leveraging modern CI/CD practices to ensure seamless and automated deployment. This approach enhances the platform's maintainability, scalability, and accessibility.

Frontend Deployment to GitHub Pages

The React-based frontend application is hosted on GitHub Pages, providing a publicly accessible interface for users. This deployment strategy offers several advantages:

CI/CD Pipeline Implementation

To automate the build and deployment process for the frontend, a GitHub Actions workflow was implemented. This pipeline ensures that any changes pushed to the `main` branch are automatically built and deployed to GitHub Pages. The workflow performs the following tasks:

The CI/CD pipeline automates dependency installation, builds the React application using Vite, and deploys it to the `gh-pages` branch for hosting on GitHub Pages.

Chapter 5

System Evaluation

5.1 Testing Methodologies

A structured testing strategy was implemented to ensure the correctness, reliability, and integration of the microservice-based architecture Job Aggregator Platform. With the diverse technology stack (Python, Java Spring Boot, Kafka, Redis, MySQL), multi-tiered testing was applied, combining unit, integration, and end-to-end testing.

5.1.1 Unit Testing Approach and COverage

Unit testing verified the correctness of individual components in isolation before integrating them into larger workflows.

- **Java Services:**

  JUnit 5 with Mockito was used to test business logic (job matching, user profile operations), repository-layer interactions, and Kafka messaging mechanisms. Local coverage analysis estimated to **65% class coverage and 72% method coverage** for core services.

- **Python Services:**

  Pytest was utilized for web scraping, AI analysis, and Kafka consumer services, targeting core data fetching, parsing, and messaging functionalities. There is a **60-70%** code parth coverage for critical components.

5.1.2 Integration Testing

Integration testing was essential because the platform relied on shared data storage (Redis, MySQL) and inter-service communication through Kafka. A controlled, containerized test environment that replicated the production architecture was made available by Docker Compose. Verified tests:

- Kafka message delivery between services.
- Caching and retrieval in Redis.
- Data persistence and consistency in MySQL.
- Asynchronous communication between AI services and storage services.

Postman was used to manually trigger API endpoints and inspect service interaction flows.

5.1.3 End-to-End Testing

End-to-end (E2E) tests simulated complete user workflows, from job search requests to AI-based skill extraction, job storage, and job matching.

- API calls were executed using Postman to validate responses and backend processes
- Browser-based manual verification was conducted for web scraping results.

These tests were executed using Postman to initiate API calls and manually validate response data and database states. In the absence of an integrated frontend user interface, browser-based manual verification of job scraping results was also performed.

5.1.4 Test Automation and Execution Summary

Test automation focused on unit and integration levels:

- **Java Services:** Automated with JUnit 5 integrated into Maven.
- **Python Services:** Automated with pytest via command-line and containerized environments.

Final test Results:

- **Java Unit Tests:** 100% pass rate (36 tests)
- **Python Unit/Integration Tests:** 100% pass rate (28 tests)

These tests reliably exercised core functionalities.

5.1.5 Limitations and Improvements

Testing limitations included:

- Partial unit test coverage (60-70%).
- Absence of formal code coverage tools in CI/CD.

5.2 Deployment Process

This section documents the system deployment workflow, evaluating the effectiveness of local development, containerized deployments, cloud hosting strategies, and operational management considerations. The deployment strategy was designed to support distributed microservices, asynchronous messaging, and efficient resource utilization.

5.2.1 Local Development Workflow

The local development environment was established using Docker Compose to orchestrate microservices, message brokers, and caching layers. This allowed for rapid iterative testing without the need to manage individual dependencies.

Key Components:

- Spring Boot services for job listing management, user profile handling, and job matching
- Python services for web scraping (Playwright) and skill extraction (Hugging Face NLP models)
- Redis for temporary data caching
- Kafka as the message broker for asynchronous communication between services
- MySQL as the primary relational database for persistent storage

Development Workflow:

1. Code changes implemented locally in individual services
2. Services containerized using Dockerfiles
3. Docker Compose used to build and run services in isolated containers

4. Postman and CLI scripts utilized for testing microservice endpoints and Kafka message passing

5. Code repositories managed via GitHub with version control and collaborative branching

This workflow enabled isolated service testing, easy dependency management, and reproducibility of development environments.

5.2.2 Containerization Effectiveness

The project extensively leveraged Docker for containerizing each microservice and dependency. Containerization benefits included:

- **Environment consistency** across development, testing, and deployment.
- **Service isolation**, ensuring failures in one service did not impact others
- **Simplified dependency management** by bundling runtime environments within Docker images

5.2.3 CI/CD and Cloud Deployment

While the full CI/CD pipeline for all microservices was initially planned as a stretch goal, significant progress was achieved by implementing automated deployment for the frontend application:

- **GitHub Actions Workflow:** A CI/CD pipeline was successfully implemented using GitHub Actions to automate the build and deployment of the React frontend to GitHub Pages.
- **Cloud Deployment:** The frontend application was successfully deployed to GitHub Pages, providing a publicly accessible interface for the job aggregation platform.
- **Environment Configuration:** The deployment process was enhanced with environment-specific configuration to support both development and production environments.

This implementation demonstrated the viability of continuous deployment practices for the platform and provided a foundation for expanding CI/CD coverage to backend services in future iterations. The successful frontend deployment to GitHub Pages also validated the decoupling of frontend and backend services, allowing for independent deployment cycles.

5.3 Techical Challenges and Solutions

This section documents the significant technical challenges encountered during system development and deployment, alongside the solutions implemented to mitigate them. Addressing these issues was critical for ensuring system reliability, performance, and maintainability.

5.3.1 Web Scraping Resilience Issues

Challenge: The web scraping microservices, particularly those interacting with job platforms via Playwright, faced frequent resilience issues due to anti-bot mechanisms, site structure changes, and intermittent connectivity errors in containerized environments.

To address this:

- **User-agent randomization** and **request header obfuscation** strategies were implemented within the scraping services.
- **Retry logic with exponential backoff** was introduced to gracefully handle transient failures.
- Scrapers were designed to detect and log structural changes in the HTML markup, enabling faster adjustment to third-party platform updates.
- Eventually, scraping from high-security platforms like Indeed was discontinued after consistent blocking, with the focus redirected to less restrictive platforms.

5.3.2 NLP Model Performance Challenges

Challenge: The AI analysis microservice, which utilized **Hugging Face-based NLP models** for skill extraction, exhibited high latency when processing lengthy job descriptions. Additionally, containerized resource limitations led to performance bottlenecks under concurrent workloads.

Solution:

Several optimizations were applied:

- **Batch processing** of multiple job descriptions where feasible, to reduce per-request overhead.

- **Async processing via Kafka queues** decoupled skill extraction from real-time service responses, reducing system-level blocking.
- Identified model-level optimizations, such as truncating excessive input lengths and fine-tuning the model configuration for inference speed.

These adjustments resulted in a 25-30% improvement in average processing times during performance tests.

5.3.3 Kafka Configuration Complexities

Challenge: The distributed nature of the system's **Kafka-based message passing** introduced complexities in configuring consumer groups, topic partitioning, message acknowledgment strategies, and handling queue lags

Solution:

Key improvements included:

- Establishing dedicated Kafka consumer groups per microservice, preventing unintended message consumption overlaps.
- Introducing **manual acknowledgment and offset commits** for critical services to maintain message processing consistency.

5.3.4 Cross-Service Debugging Difficulties

Challenge: Debugging failures across distributed, containerized microservices introduced significant challenges, especially when tracing issues spanning multiple asynchronous services, Kafka message queues, and shared infrastructure components such as Redis.

Implemented Measures:

Several strategies were adopted:

- **Structured Logging:** Consistent logging configurations were established across both Python and Java microservices.

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s␣[%(levelname)s]␣%(
       message)s',
    datefmt='%Y-%m-%d␣%H:%M:%S'
)
```

```
private static final Logger logger =
   LoggerFactory.getLogger(MatchingService.
   class);
logger.info("Skills␣extraction␣completed␣for␣
   job:␣{}␣and␣user:␣{}", jobId, userId);
```

- **Standardized Kafka Message Formats:** Kafka producers and consumers across services followed a consistent message structure, reducing serialization errors and ensuring predictable message parsing.

```
self.producer_matching.send('matching', value
   ={
    'jobId': job_key,
    'userId': user_id
})
```

```
JsonNode node = mapper.readTree(message);
String jobId = node.get("jobId").asText();
```

- **Service Health Monitoring via Docker Compose:** Health checks were incorporated into Docker Compose configurations to periodically validate the operational status of critical services, including Kafka brokers.

```
healthcheck:
    test: ["CMD-SHELL", "kafka-topics.sh --
       list --bootstrap-server localhost
       :9093"]
    interval: 5s
```

5.4 Project Objectives Assessment

5.4.1 Success Criteria Evaluation

At the start of the project, several success criteria were established to measure the overall outcome:

- **Successful real-time job scraping from external websites**
- **AI-based skills extraction and job-user matching**
- **Distributed microservices architecture with Kafka-based communication**
- **Basic user profile management with authentication**
- **Docker-based deployment for local and cloud environment**

Evaluation: The project successfully met most of its predefined success criteria:

- Real-time job scraping was achieved via Playwright-based scrapers.
- AI-powered skill extraction using a Hugging Face model was implemented and integrated with the matching service.
- The system successfully operated as a distributed microservices architecture using Kafka for asynchronous messaging.
- User registration and authentication services were fully operational.
- All services were containerized and functioned effectively in Docker Compose environments.

5.4.2 Requirement Fulfillment Analysis

The project requirements defined at initiation included both functional requirements (job search, skill extraction, matching, user profiles) and non-functional requirements (scalability, resilience, modularity).

Requirement	**Status**
Job listings successfully scraped and stored	✓
Skill extraction service operational	✓
Matching service produces job-user matches based on skills	✓
User registration and login services complete	✓
Frontend integration	X (API only)

Table 5.1: Functional Requirements Fulfillment

Requirement	**Status**
Services decoupled via Kafka	✓
Redis is used effectively for caching	✓
Docker Compose deployment functional	✓
Cloud deployment	✓(Frontend only)
CI/CD pipeline setup	✓(Frontend only)

Table 5.2: Non-Functional Requirements Fulfillment

5.4.3 Technical Goal Achievement

The project aimed to implement several technical milestones to validate architectural choices and system resilience:

Achievements:

- Developed a modular microservices-based system
- Integrated AI-driven natural language processing into production services
- Established asynchronous messaging through Kafka with appropriate topic handling
- Achieved partial performance optimization via Redis caching

The project effectively achieved its core technical goals within the constraints of the timeframe, laying a strong foundation for a scalable, AI-enhanced job platform. A few advanced technical targets, especially around DevOps practices and cloud scalability, remain as future improvement opportunities.

Chapter 6

Conclusion

6.0.1 Project Summary

This project successfully developed a microservice-based Job Aggregator Platform capable of real-time job scraping, automated skills extraction, and job-to-candidate matching. Key innovations included integrating AI-powered NLP services within a distributed architecture, Kafka-based asynchronous communication for inter-service messaging, and Redis caching to improve data retrieval performance.

The system's primary features are job listing storage, AI-driven skills extraction using a Hugging Face model, distributed scraping with Playwright, and dynamic matching services. By combining various services, the platform showed that it could aggregate, evaluate, and display job openings with greater relevance and immediacy.

Technically, the project contributed new solutions for handling cross-service debugging challenges, integrating AI services in containerized environments, and optimizing data freshness through real-time scraping pipelines.

6.0.2 Limitations

Technical Constraints

- **AI Service Latency:**

  The NLP-based skill extraction service incurs significant processing time, particularly for long or complex job descriptions, resulting in elevated end-to-end response times.

- **Kafka Queue Lag:**

  During periods of high concurrent job analysis and matching requests, Kafka topic lag was observed, delaying message consumption and downstream processing.

- **Web Scraping Resilience:**

  The Playwright-based scrapers are susceptible to bot detection mechanisms, including CAPTCHA and anti-scraping heuristics, on target websites. This occasionally led to incomplete job data retrieval or scraper downtime.

Scope Limitations

- **Platform Coverage:**

  The system is limited to two job platforms (IrishJobs and IES). Expanding to additional sources would require scraper adjustments and additional anti-bot countermeasures.

- **Regional Focus:**

  The platform and job listings are currently tailored to the Irish market without support for international job sources or localization.

Performance Boundaries

- **Database and Cache Limitations:** Redis and MySQL were deployed without high-availability setups, sharding, or persistence management under failure scenarios.

6.0.3 Future Work

1. Platform Expansion

   - **Additional Job Sources:** Expand beyond Jobs.ie and IrishJobs.ie to include other major Irish job boards such as LinkedIn Jobs, Indeed, and Glassdoor.
   - **International Markets:** Scale the platform to support job searches in other countries with localized scrapers and region-specific matching algorithms.

2. Advanced AI Integration

   - **Improve Skill Extraction:** Enhance the AI-based skill extraction system to better identify both hard and soft skills from job descriptions.

- **Personalized Job Recommendations:** Implement more sophisticated recommendation algorithms using collaborative filtering and deep learning.
- **Smart CV Builder:** Create an AI assistant that helps users optimize their CVs for specific job applications based on the extracted requirements.

3. Enhanced User Experience

   - **Mobile Application:** Develop native mobile apps for iOS and Android to complement the web platform

4. Technical Improvements

   - **Microservices Optimization:** Review and enhance the current microservices architecture for better scalability
   - **Enhanced Scraping Resilience:** Improve the web scraping modules to better handle changes in target websites' structures

6.0.4 Personal Reflection

I had never used microservices or event-driven systems in practice until this project. I was challenged to solve practical integration and operational problems by working with Redis, Kafka, and containerized deployments, which went beyond my theoretical knowledge. I discovered that debugging and improving systems in real-world settings yields the most insightful lessons. It was challenging to use Kafka to integrate Python-based scrapers with Java Spring services, particularly when it came to handling errors and maintaining uniform data formats. I learned the value of creating unambiguous service interfaces early on by adopting schema validation as a result of my initial failures with basic JSON communications. I overestimated how long a multi-service system would take, especially for web scraping, where delays were caused by rate constraints and site updates. My ability to work with new tools and create a methodical approach to learning new technologies has significantly improved as a result of this project. It also improved my ability to solve problems and to be patient and resilient while dealing with unclear or annoying technological problems.

Chapter 7

Appendix

7.1 GitHub Repository

https://github.com/SoonMingQian/Job-Aggregator-Platform

Bibliography

[1] Mehmet Batuhan Yazıcı, Damla Sabaz, and Wisam Elmasry. AI-based Multimodal Resume Ranking Web Application for Large Scale Job Recruitment. In *2024 8th International Artificial Intelligence and Data Processing Symposium (IDAP)*, pages 1–8, September 2024.

[2] Evanthia Faliagka, Kostas Ramantas, and Giannis Tzimas. Application of Machine Learning Algorithms to an online Recruitment System. 2012.

[3] Victor F. Araman, Andre Calmon, and Kristin Fridgeirsdottir. Pricing and Job Allocation in Online Labor Platforms. *SSRN Electronic Journal*, 2019.

[4] Aseel B. Kmail, Mohammed Maree, and Mohammed Belkhatir. MatchingSem: Online recruitment system based on multiple semantic resources. In *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 2654–2659, August 2015.

[5] Hamzah Alaidaros, Mazni Omar, and Rohaida Romli. Identification of criteria affecting software project monitoring task of Agile Kanban method. page 020021, Penang, Malaysia, 2018.

[6] Matheus Felisberto. The trade-offs between Monolithic vs. Distributed Architectures, May 2024. arXiv:2405.03619 [cs].

[7] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. Version Control System: A Review. *Procedia Computer Science*, 135:408–415, 2018.

[8] Nicolás Paez. Versioning Strategy for DevOps Implementations. In *2018 Congreso Argentino de Ciencias de la Informática y Desarrollos de Investigación (CACIDI)*, pages 1–6, November 2018.

[9] Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores, and Theofilos Toulkeridis. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences*, 10(17):5797, January 2020. Number: 17 Publisher: Multidisciplinary Digital Publishing Institute.

[10] Wen-Tin Lee and Ming-Kai Tsai. Implementing a PHP API Gateway Based on Microservices Architecture. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1578–1579, July 2024. ISSN: 2836-3795.

[11] Kyle Brown and Bobby Woolf. Implementation Patterns for Microservices Architectures.

[12] Suleiman Ali Alsaif, Minyar Sassi Hidri, Imen Ferjani, Hassan Ahmed Eleraky, and Adel Hidri. NLP-Based Bi-Directional Recommendation System: Towards Recommending Jobs to Job Seekers and Resumes to Recruiters. *Big Data and Cognitive Computing*, 6(4):147, December 2022.

[13] Yu Deng, Hang Lei, Xiaoyu Li, and Yiou Lin. An improved deep neural network model for job matching. In *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 106–112, May 2018.

[14] Elias Abdollahnejad, Marilynn Kalman, and Behrouz H. Far. A Deep Learning BERT-Based Approach to Person-Job Fit in Talent Recruitment. In *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 98–104, December 2021.

[15] Victor Velepucha and Pamela Flores. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. *IEEE Access*, 11:88339–88358, 2023. Conference Name: IEEE Access.

[16] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, September 2015.

[17] Roland H Steinegger, Pascal Giessler, Benjamin Hippchen, and Sebastian Abeck. Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications. 2017.

[18] Meina Song, Chengcheng Zhang, and E Haihong. An Auto Scaling System for API Gateway Based on Kubernetes. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 109–112, November 2018. ISSN: 2327-0594.

[19] L.D.S.B Weerasinghe and I Perera. Evaluating the Inter-Service Communication on Microservice Architecture. In *2022 7th International Conference on*

Information Technology Research (ICITR), pages 1–6, December 2022. ISSN: 2831-3399.

[20] Ranjith G Hegde. Low Latency Message Brokers. 07(05), 2020.

[21] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing.

[22] Teemu Korhonen. Using Kafka to Build Scalable and Fault Tolerant Systems.

[23] Han Wu, Zhihao Shang, and Katinka Wolter. Performance Prediction for the Apache Kafka Messaging System. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 154–161, August 2019.

[24] Desiree Monteiro. Bachelor End Project.

[25] Mariia Chernova. Occupational skills extraction with FinBERT.