**Yarmouk Private University**

**Faculty of Informatics and**

**Communication Engineering**



**Prepared by:**  
Ayham Kefo Nawrz Qal Amr Barakat

**Supervisor:**

Dr. Hiyam Khaddam  
Eng. Anas Mokhalati

Semester: 2024/2025-2

# Abstract

Nohoud is a distributed microservices-based platform designed to support all Syrians, including refugees, returnees, and residents, in developing their skills, improving employability, and building personalized development plans powered by AI.

The platform’s primary goal is to evaluate users’ current skill sets, match them with relevant job opportunities, and provide AI-generated personal development plans. It leverages a scalable architecture using NestJS microservices, MongoDB, and an AI pipeline orchestrated through n8n, connected to a stateless LLM (Gemini).

Users complete their profiles, which are then analyzed to produce job suggestions and tailored growth plans. The system emphasizes usability, modularity, and personalization, making it a practical solution for job seekers across the Syrian population—regardless of their background or displacement status. The results show that combining AI with scalable backend services can  
meaningfully assist in career development and employment integration.

# Acknowledgments

We extend our heartfelt gratitude to Dr. Hiyam Khaddam and Eng. Anas Mukhalati, our esteemed supervisors, for her invaluable guidance and mentorship, and to Dr. Moutasem Shafaumri, the esteemed Dean of our Faculty, for fostering a culture of innovation and scholarly pursuit. Their wise advice and unwavering support have been instrumental in the development of the NOHOUD project.

Additionally, we are deeply grateful to our families and friends for their continuous encouragement and emotional support. This achievement would not have been possible without their constant backing.

# Table of Contents

Contents

[Abstract 1](#_Toc201824688)

[Acknowledgments 2](#_Toc201824689)

[Table of Contents 3](#_Toc201824690)

[Table of Figures 6](#_Toc201824691)

[List of Abbreviation and Terminology 7](#_Toc201824692)

[Chapter 1: Introduction 8](#_Toc201824693)

[1.1 Motivation 9](#_Toc201824694)

[1.2 Problem Description: 9](#_Toc201824695)

[1.3 Project Objectives 10](#_Toc201824696)

[Chapter 2: Background 12](#_Toc201824697)

[2.2 Related Works 13](#_Toc201824698)

[2.3 The Proposed Solution 13](#_Toc201824699)

[Chapter 3: Project Management & Methodology 14](#_Toc201824700)

[3.1 Development Methodology: Agile Scrum 15](#_Toc201824701)

[3.1.1 Why Agile Scrum Fits This Project 15](#_Toc201824702)

[3.1.2. Roles and Responsibilities 15](#_Toc201824703)

[3.1.3. Sprint Planning and Execution 16](#_Toc201824704)

[3.1.4. Self-Organizing Team Structure 16](#_Toc201824705)

[3.1.5. Scrum Ceremonies 16](#_Toc201824706)

[3.1.6. Sustainable Engineering Practices 17](#_Toc201824707)

[3.1.7. Iteration Structure and Feature Lifecycle 17](#_Toc201824708)

[3.1.8. Code Publication & Versioning Strategy 18](#_Toc201824709)

[2.2. Remote Pair Programming for Complex Tasks 18](#_Toc201824710)

[3.2.1 Purpose and Benefits 18](#_Toc201824711)

[3.2.2 Setup and Approach 19](#_Toc201824712)

[2.3. Managing a Multi-Service Platform with Scrum 19](#_Toc201824713)

[3.4. Project Management Tools: Jira 20](#_Toc201824714)

[3.4.1. Epics, User Stories, and Tasks 20](#_Toc201824715)

[3.4.2. Sprint Board and Burndown Charts 20](#_Toc201824716)

[3.4.3. Visual Artifacts from Jira Workspace 21](#_Toc201824717)

[3.5. Version Control Strategy 22](#_Toc201824718)

[2.6. Communication and Collaboration 23](#_Toc201824719)

[3.6.1 Primary Communication Channels 23](#_Toc201824720)

[3.6.2 Meeting Cadence 23](#_Toc201824721)

[Chapter 4: Requirements Engineering 24](#_Toc201824722)

[4.1 Requirements Elicitation 25](#_Toc201824723)

[4.2 Functional Requirements 25](#_Toc201824724)

[4.2.1 User Management 25](#_Toc201824725)

[4.2.2 Job Application Flow 25](#_Toc201824726)

[4.2.3 Notification & Alerts 25](#_Toc201824727)

[4.2.4 AI Recommendations 25](#_Toc201824728)

[4.3 Non-Functional Requirements 25](#_Toc201824729)

[4.4 Use Case Diagram 25](#_Toc201824730)

[4.5 Use Case Specifications 25](#_Toc201824731)

[Chapter 5: System Design and Architecture 26](#_Toc201824732)

[5.1. Architectural Style: Microservices 27](#_Toc201824733)

[5.1.1. Rationale for Choosing Microservices 28](#_Toc201824734)

[5.1.2. Service Decomposition and Granularity 29](#_Toc201824735)

[5.2. High-Level System Architecture Diagram 30](#_Toc201824736)

[5.3. Core Design Principles 31](#_Toc201824737)

[5.3.1. SOLID Principles Application 31](#_Toc201824738)

[5.3.2. Design Patterns 32](#_Toc201824739)

[5.4. Inter-Service Communication 34](#_Toc201824740)

[5.4.1. Communication Patterns 34](#_Toc201824741)

[5.4.2. Asynchronous Communication with Apache Kafka 34](#_Toc201824742)

[5.4.3. Synchronous Communication with gRPC 35](#_Toc201824743)

[5.5. Data Management Strategy (Database-per-Service) 36](#_Toc201824744)

[5.6. AI Service Integration using n8n 37](#_Toc201824745)

[Chapter 6: Technology Stack and Rationale 38](#_Toc201824746)

[6.1. Programming Language & Runtime: Node.js 39](#_Toc201824747)

[6.2. Backend Framework: NestJS 40](#_Toc201824748)

[6.3. API Documentation: Swagger 40](#_Toc201824749)

[6.4. Database and ODM: MongoDB with Mongoose 41](#_Toc201824750)

[6.5. Messaging/Event-Driven Architecture: Apache Kafka 41](#_Toc201824751)

[6.5. RPC Communication: gRPC with Protocol Buffers 42](#_Toc201824752)

[6.6. Containerization & Orchestration: Docker Compose 42](#_Toc201824753)

[6.7. Kafka Monitoring Tool: Kafdrop 43](#_Toc201824754)

[Chapter 7: Implementation 44](#_Toc201824755)

[Chapter 8: Testing 46](#_Toc201824756)

[Chapter 9: Conclusion and Future Work 47](#_Toc201824757)

[9.1. Conclusion 48](#_Toc201824758)

[9.2. Future Enhancements 48](#_Toc201824759)

[9.3 Resources 48](#_Toc201824760)

[Appendices 49](#_Toc201824761)

# Table of Figures

[Figure 1:logo 8](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373580)

[Figure 2: Nodejs 17](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373581)

[Figure 3: Express js 17](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373582)

[Figure 4: Express architecture 17](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373583)

[Figure 5: MongoDB 17](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373584)

[Figure 6: Mongoose 18](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373585)

[Figure 7: scss 18](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373586)

[Figure 8: EJS template engine 18](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373587)

[Figure 9: Git 18](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373588)

[Figure 10: Github 18](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373589)

[Figure 11: Vs Code 19](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373590)

[Figure 12: Syria-Visa.sy 21](#_Toc172373591)

[Figure 13: UseCase Diagram 28](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373592)

[Figure 14: class diagram 43](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373593)

[Figure 15: entity relationship diagram 45](file:///C:\Users\ACER\Desktop\APPLIED-vamos.docx#_Toc172373594)

[Figure 16: Register Sequence diagram 46](#_Toc172373595)

[Figure 17: book appointment Sequence diagram 46](#_Toc172373596)

[Figure 18: Login Sequence diagram 47](#_Toc172373597)

[Figure 19: logout Sequence diagram 47](#_Toc172373598)

[Figure 20: Reset password Sequence diagram 48](#_Toc172373599)

[Figure 21: check wallet Sequence diagram 48](#_Toc172373600)

[Figure 22: add money Sequence diagram 49](#_Toc172373601)

[Figure 23: Add Ddferment Sequence diagram 50](#_Toc172373602)

[Figure 24: request state Sequence diagram 50](#_Toc172373603)

[Figure 25: Browse Requests Sequence diagram 51](#_Toc172373604)

[Figure 26: Add Request to work list Sequence diagram 51](#_Toc172373605)

[Figure 27: change Request State Sequence diagram 52](#_Toc172373606)

[Figure 28: Nodejs architecture 55](#_Toc172373607)

[Figure 29: email notification 58](#_Toc172373608)

# List of Abbreviation and Terminology

|  |  |
| --- | --- |
| **Abbreviation/Term** | **Definition** |
| **Nohoud** |  |
| **UI/UX** | User Interface/User Experience; principles and practices involved in designing the interface and experience of the platform. |
| **API** | Application Programming Interface; a set of protocols and tools for building software applications. |
| **BSON** | Binary JSON; a binary-encoded serialization of JSON-like documents. |

# Chapter 1: Introduction

## 1.1 Motivation

Syria’s civil war has caused one of the worst displacement crises, leaving millions of refugees and internally displaced persons (IDPs) in urgent need of support. By 2025, an estimated 16.7 million Syrians will require humanitarian aid, including 6.2 million refugees abroad and 7.2 million IDPs. A major challenge is unemployment, as many skilled Syrians struggle to find work due to broken labor market connections. AI and modern technologies, such as machine learning for job matching and microservices for scalable platforms, offer solutions. The Nohoud project leverages these innovations to provide personalized job recommendations and career development plans, helping Syrians rebuild their livelihoods.

## 1.2 Problem Description:

Despite the clear need, existing support systems for Syrian job-seekers are fragmented and incomplete. Most humanitarian job platforms or job boards are either basic registries of openings or manual counseling programs; they typically do not personalize matches or guide users on skills development.

Technically, most current platforms suffer from rigidity and poor scalability. Traditional systems are often monolithic (single-unit) applications that are hard to maintain or extend. Communication between services can be slow or unreliable, especially across regions with limited network infrastructure.

no existing system holistically solves the problem of guiding all Syrians — whether in camps, urban centers, or abroad — through a personalized pathway from skill assessment to employment. Nohoud is designed to fill these gaps. It identifies three core problems: (1) lack of personalized matching: existing platforms do not use AI or user profiling to adapt to individual needs; (2) lack of integrated skill development: refugees need guidance on training/certification, not just job listings; and (3) poor system design for scale: many solutions cannot easily add features or handle high loads. By targeting these issues, Nohoud aims to overcome the shortcomings of current practice and effectively support Syrian livelihoods.

## 1.3 Project Objectives

The Nohoud system has the following specific, actionable objectives:

AI-driven Job Matching using n8n workflow: Develop an intelligent recommendation engine that matches each user’s profile (skills, experience, preferences) to relevant job openings and opportunities. The matching will leverage including large language model to analyze curricula vitae and job descriptions for optimal alignment.

Personal Development Plans using n8n workflow: For every user, generate a customized career development plan outlining recommended training, courses, or certifications. These plans will be tailored using the user model and labor-market insights, enabling users to systematically build employable skills over time.

Microservices Architecture: Implement the platform as a set of distributed microservices (using NestJS) so that each major function (e.g. user management, job search, recommendation, content delivery) is an independent service. This will ensure the system is scalable, maintainable, and fault-tolerant.

Efficient Communication: Use gRPC (Protocol Buffers over HTTP/2) for inter-service communication to maximize performance. Studies show gRPC can significantly outperform traditional REST APIs (e.g. 7–10× faster throughput in benchmark tests), which will allow Nohoud to handle high volumes of requests across components.

Flexible Data Storage: Use MongoDB as the primary data store, taking advantage of its schemaless document model to handle diverse user profiles and job data. This will allow the system to evolve (e.g. adding new user attributes or content types) without disruptive schema migrations.

Together, these objectives ensure that Nohoud not only delivers intelligent matching and guidance, but does so on a robust technical foundation. By meeting these goals, Nohoud will empower Syrian refugees, returnees, and residents to develop marketable skills and find employment opportunities in an effective, scalable way.

# Chapter 2: Background

## 2.2 Related Works

## 2.3 The Proposed Solution

# Chapter 3: Project Management & Methodology

## 3.1 Development Methodology: Agile Scrum

We adopted **Agile Scrum** as the development methodology for this project due to its flexibility, iterative nature, and proven effectiveness in managing complex projects with evolving requirements. With a team of three members possessing diverse skills, ranging from backend development to AI services. Scrum enabled us to deliver functionality incrementally while continuously adapting to new insights.

### 3.1.1 Why Agile Scrum Fits This Project

* **Adaptability to Changing Requirements**: Our platform targets nuanced needs of migrants and refugees. As we progressed and understood our users better, our requirements evolved. Scrum allows backlog reprioritization between sprints without derailing the project plan.
* **Iterative and Incremental Delivery**: We used 1-week sprints. At the end of each sprint, we delivered a working increment—whether a functional endpoint, a microservice, or a UI

feature—allowing for constant feedback and continuous validation.

* **Hands-on Collaboration**: As a small team, we benefitted from close communication and fast decision-making. The daily standups (virtual or async) enabled full visibility of who was working on what, and reduced blockers.

### 3.1.2. Roles and Responsibilities

While a formal Scrum team has distinct roles (Scrum Master, Product Owner, Development Team), we streamlined responsibilities for our context:

* **Scrum Facilitator** (nawrz qal): Ensured Scrum events happened on time, facilitated

planning, and managed progress tracking.

* **Product Ownership** (shared): Prioritized backlog items based on user impact and project goals. Decisions were made collaboratively.
* **Development Team**: All three members contributed to development across all services, splitting work based on technical strength and availability.

### 3.1.3. Sprint Planning and Execution

 We held sprint planning at the start of each sprint to:

* Define the **Sprint Goal**.
* Select user stories from the Product Backlog.
* Break them down into tasks and estimate effort (using relative sizing).

 We executed in **short, focused iterations**, allowing us to continuously adjust course based on learnings or blockers.

 Example Sprint Goal: "Implement authentication service and integrate user role-based

routing for the job platform."

### 3.1.4. Self-Organizing Team Structure

Our team operated as a self-organizing unit:

* **Autonomy**: Each member owned their tasks. Work was pulled, not pushed.
* **Cross-fertilization**: Our team members had varied skills (backend, frontend, AI, DevOps), allowing diverse input across the board.
* **Requisite Variety**: When tasks demanded new approaches, team members quickly adapted and experimented (e.g., new workflows for handling async communication or CI scripts).
* **Learning to Learn**: Retrospectives were used to re-evaluate how we work, not just what we built.
* **Self-evaluation**: We reflected on velocity, quality, and team satisfaction every sprint and adjusted our flow.

### 3.1.5. Scrum Ceremonies

 **Daily Stand-ups**: Brief check-ins, async via chat or GitHub comments. Covered progress, blockers, and coordination.

 **Sprint Review**: Demoed completed features and gathered feedback (both internal and

supervisor input).

 **Sprint Retrospective**: Reviewed what went well, what didn’t, and what to improve. Actions were added to the next sprint’s plan.

### 3.1.6. Sustainable Engineering Practices

* We operated under the principle of **Sustainable Pace**. No team member worked nights or weekends. This led to higher-quality, more maintainable code, and consistent delivery.
* We enforced **Collective Code Ownership** through pull requests, shared knowledge of services, and agreed coding standards.
* No module or service was siloed—anyone could improve or refactor any part of the   
  system with proper testing and PR review.

### 3.1.7. Iteration Structure and Feature Lifecycle

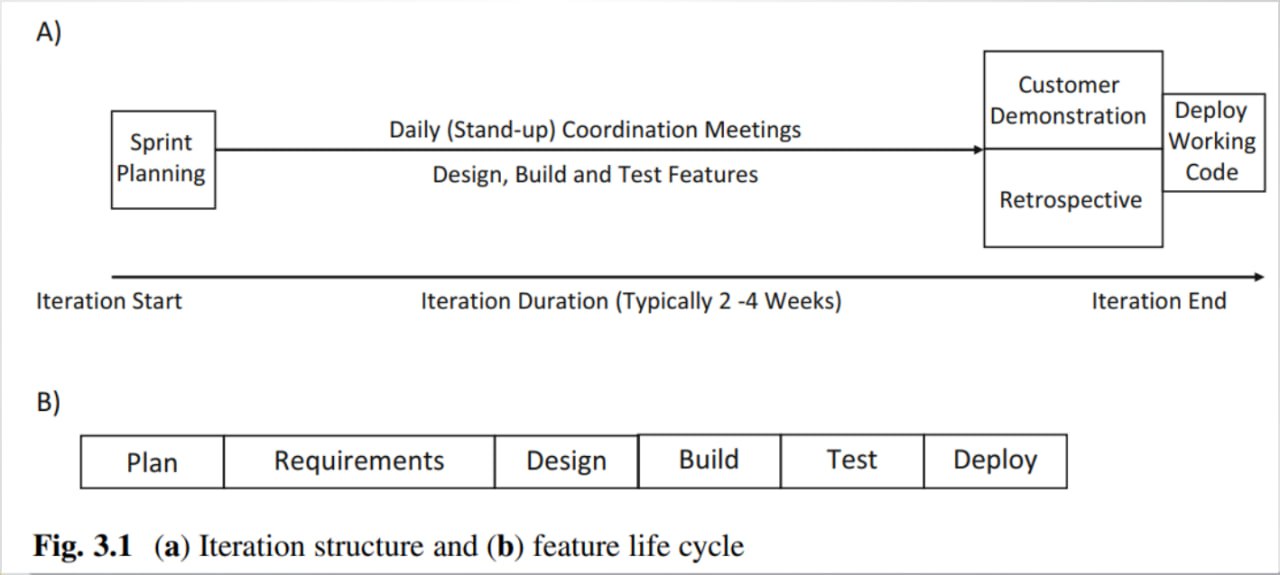
****

Figure 1

Iteration structure showing the lifecycle of features through Plan → Requirements → Design → Build → Test within each sprint.

This diagram reflects how we structured each 1-week sprint: we planned, broke down user stories, designed, built locally, and tested before merging.

### 3.1.8. Code Publication & Versioning Strategy

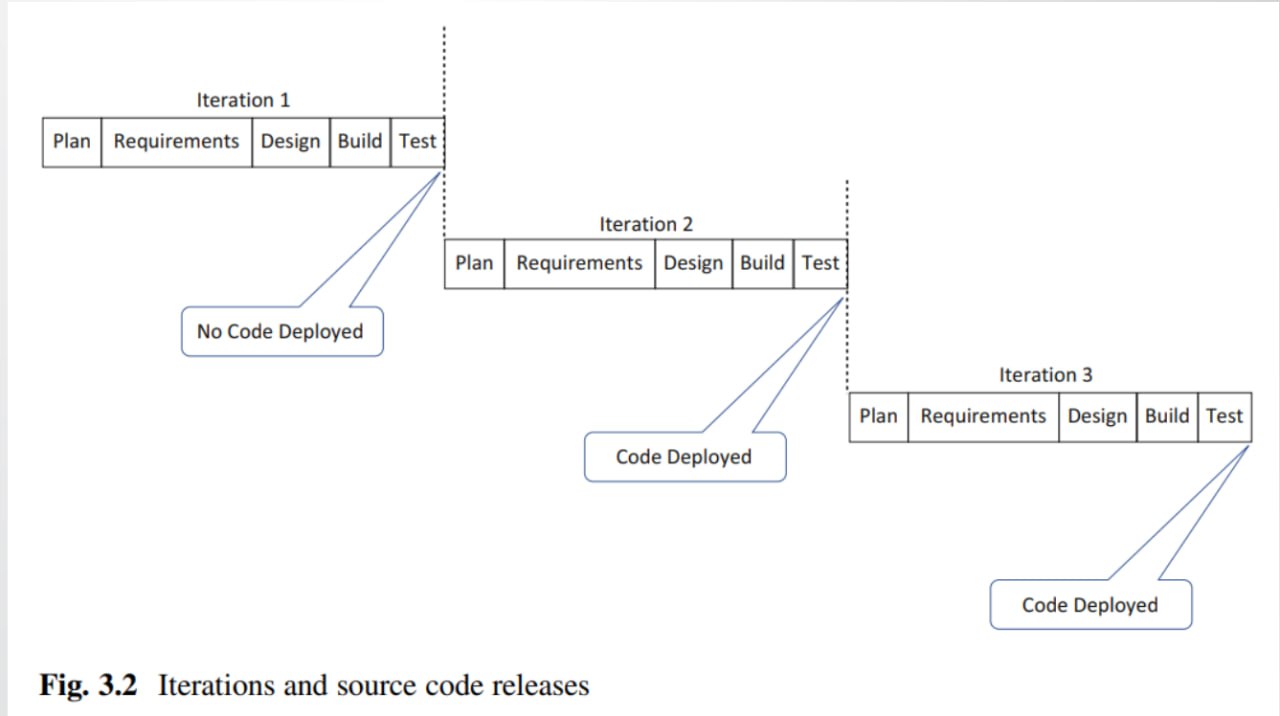
****

Figure 2

## 2.2. Remote Pair Programming for Complex Tasks

Although our team was small and partially co-located, we operated as a **virtual team** due to varying schedules and remote collaboration. To overcome the challenges of asynchronous development, we adopted **Remote Pair Programming** during critical implementation phases—especially for complex services like authentication, AI job matching logic, and cross-service integration.

### 3.2.1 Purpose and Benefits

* **Tactical Collaboration**: We used remote pair programming as a focused tactic to tackle complex logic, resolve integration bugs, or review sensitive code (e.g., security features, async flows, database interactions).
* **Real-time Code Quality Feedback**: One member acted as the **Driver** (writing the code), while the other was the **Navigator** (reviewing in real time, thinking about structure, edge cases, and design consistency).
* **Onboarding Support**: When a team member worked on an unfamiliar service, pair programming was used to onboard them quickly without long delays or documentation overhead.

### 3.2.2 Setup and Approach

* We used **Google Meet** for implementation sessions.
* Pairing was scheduled during sprint planning when high-risk or shared ownership tasks were identified.
* Sessions had **clear goals** and were time-boxed to avoid fatigue and maintain flow.
* We occasionally used the **ping-pong** method: one person wrote tests, the other wrote the implementation, and then roles were swapped.

## 2.3. Managing a Multi-Service Platform with Scrum

Our project consists of four core services, each with its own focus, but all contributing to a

unified platform. Scrum helped us manage this structure effectively:

* **Single Product Backlog**: Centralized and prioritized across all services. This ensured that we always focused on the most impactful items regardless of where they belonged in the system.
* **Service-Focused Sprint Goals**: Each sprint goal either cut across multiple services or focused on delivering complete functionality within one. This avoided context switching and maintained team focus.
* **Example**: One sprint focused on user authentication (Auth + Frontend), while another focused on job listing filtering and display logic (Job Service + Search Layer).

## 3.4. Project Management Tools: Jira

We used **Jira** as our main project management and tracking tool, configured to reflect Scrum best practices.

**Jira** is a **project management and issue tracking software** developed by **Atlassian**. It is widely used by teams to plan, track, and manage tasks, bugs, and agile (Scrum & Kanban)   
workflows.

**Key Features of Jira:**

* **Issue Tracking** – Log and track bugs, tasks, and improvements.
* **Agile Project Management** – Supports Scrum and Kanban boards for sprint planning.
* **Customizable Workflows** – Adapt processes to team needs.
* **Dashboards & Reports** – Visualize progress with burndown charts, velocity reports, etc.
* **Integration** – Works with Confluence, Bitbucket, Slack, and other tools.

### 3.4.1. Epics, User Stories, and Tasks

* **Epics**: Represented large features or services (e.g., AI Recommendation, Job Matching,

Authentication).

* **User Stories**: Written in user-centric format and linked to the product goals.
* As a job seeker, I want to receive personalized job listings based on my profile so that I can find relevant opportunities faster.
* **Tasks/Sub-tasks**: Broke user stories down into actionable development items with effort

estimates.

### 3.4.2. Sprint Board and Burndown Charts

* The Jira board had columns for To Do, In Progress, Code Review, and Done.
* **Burndown Charts** were used to track progress during each sprint. Irregularities in the burndown graph helped us identify underestimations or scope creep early.

### 3.4.3. Visual Artifacts from Jira Workspace

**Still empty**

## 3.5. Version Control Strategy

Effective version control is the backbone of collaborative software development. For this project, we utilized Git, a distributed version control system, and managed our codebase within a centralized **GitHub Organization**. This approach provided a structured environment for code management, collaboration, and quality assurance.

<https://github.com/Nuhoud>

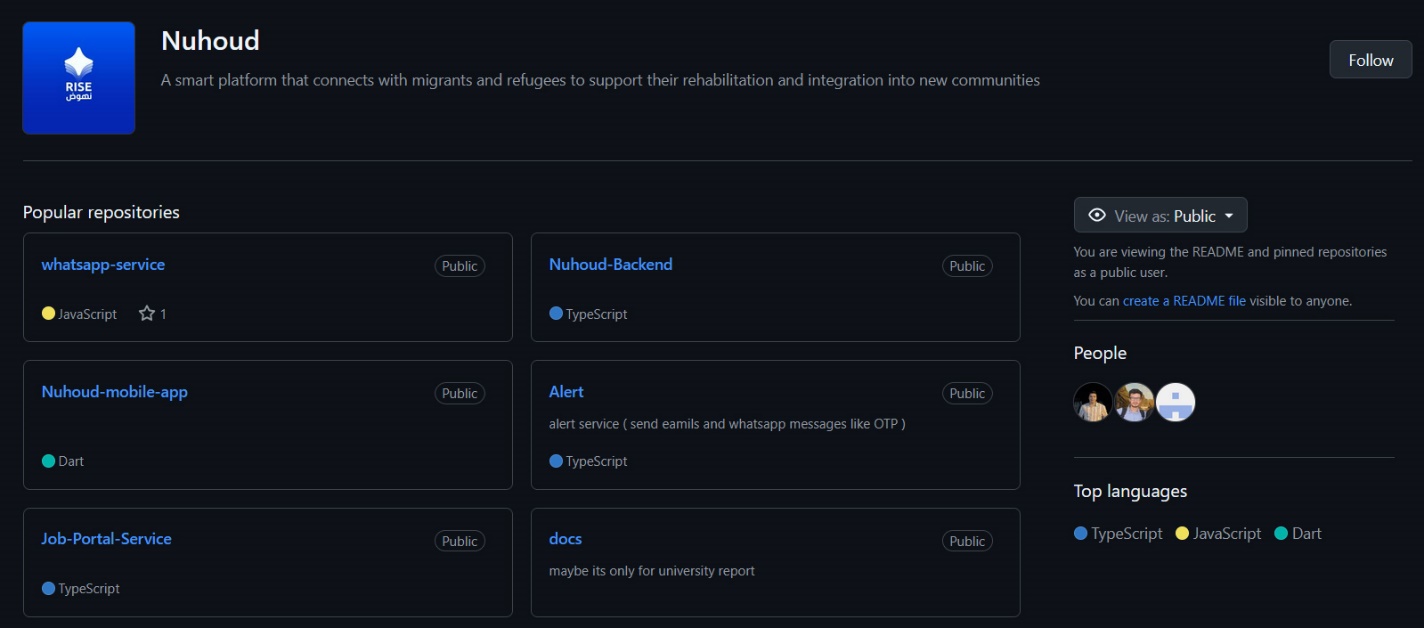


Figure 3

By centralizing our repositories in one organization, we achieved a transparent overview of all ongoing work. Any team member could see the status of different services, track feature development, and understand the overall architecture at a glance. This was particularly crucial for our multi-service platform, as it helped us manage dependencies and coordinate cross-service integrations.

## 2.6. Communication and Collaboration

Effective communication and collaboration were essential to the success of our self-organizing team, especially given our hybrid work environment. We established clear communication channels and structured interactions to ensure transparency, accountability, and fast feedback.

### 3.6.1 Primary Communication Channels

* **Google Meet**: Used for real-time meetings such as sprint planning, sprint reviews, and occasional ad-hoc discussions.
* **Telegram Group**: Our central communication hub. We organized conversations into topic-specific threads to maintain clarity and focus:
  + #visual-identity: UI/UX and branding discussions.
  + #discussion: General project decisions, blockers, and coordination.
  + #files-and-docs: Shared links to documentation, files, and assets.

### 3.6.2 Meeting Cadence

* **Weekly Sprint Meetings**:
  + Sprint Planning at the start of each sprint.
  + Sprint Review/Demo at the end of the sprint.
* **Daily Async Stand-ups**:
  + Team members posted brief updates (what was done, what’s next, blockers) in the Telegram #discussion thread or via GitHub issue comments.

# Chapter 4: Requirements Engineering

## 4.1 Requirements Elicitation

## 4.2 Functional Requirements

### 4.2.1 User Management

### 4.2.2 Job Application Flow

### 4.2.3 Notification & Alerts

### 4.2.4 AI Recommendations

## 4.3 Non-Functional Requirements

## 4.4 Use Case Diagram

Figure

## 4.5 Use Case Specifications

# Chapter 5: System Design and Architecture

## 5.1. Architectural Style: Microservices

**What is Microservices Architecture?**

Microservices architecture (often shortened to microservices) refers to an architectural style for developing applications. Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility. To serve a single user request, a microservices-based application can call on many internal microservices to compose its response.

**Monolithic vs. microservices architecture**

Traditional monolithic applications are built as a single, unified unit. All components are tightly coupled, sharing resources and data. This can lead to challenges in scaling, deploying, and maintaining the application, especially as it grows in complexity. In contrast, microservices architecture decomposes an application into a suite of small, independent services. Each microservice is self-contained, with its own code, data, and dependencies. This approach offers several potential advantages:

* **Improved scalability:** Individual microservices can be scaled independently based on their specific needs
* **Increased agility:** Microservices can be developed, deployed, and updated independently, enabling faster release cycles
* **Enhanced resilience:** If one microservice fails, it doesn't necessarily impact the entire application
* **Technology diversity:** The flexibility of microservices allows teams to use the most suitable technology for each service

### 5.1.1. Rationale for Choosing Microservices

The Nuhoud system uses a microservices architecture to maximize scalability and flexibility. Each microservice can be scaled independently – for example, if the Job Service experiences heavy load, only its instances need to be increased. This avoids over-provisioning and improves resource utilization. Microservices also enforce **separation of concerns**: distinct business   
domains (user management, job postings, notifications, AI) reside in separate services, making each codebase smaller and more maintainable. Changes or faults in one service (say, the AI   
Recommendation Service) do not directly affect others, improving fault isolation. Independent services can be deployed and updated on their own schedules, enabling faster iteration.   
In practice, we organize teams and development pipelines around these bounded domains; for example, one team owns user/profile features while another owns job postings. Overall,   
microservices offer the agility (parallel development, polyglot persistence), resilience, and fine-grained scalability that suit Nuhoud’s requirements.

* **Scalability:** Each service can be scaled on demand (e.g. adding instances of the Job Service) without scaling unrelated components.
* **Separation of Concerns:** Services are aligned to single business capabilities (following the Single Responsibility Principle) so they are cohesive and independently maintainable.
* **Independent Deployment:** Teams can deploy or upgrade services separately, reducing risk. The database-per-service pattern decouples data, so updates do not require cross-database schema changes.

These factors make microservices well-suited for Nuhoud’s event-driven, domain-oriented system. (As one guide notes, “Bounded Contexts (each BC correlates to a microservice)” when using DDD principles

### 5.1.2. Service Decomposition and Granularity

We decomposed Nuhoud according to Domain-Driven Design (DDD). Each microservice corresponds to a bounded context – a coherent subdomain with its own data and logicIn Nuhoud, the main services are:

* **Authentication & User Service:** Manages the ***User*** domain. It handles registration, login (email or phone), profiles (personal info, experience, education, skills) and tracks each user’s job applications and development history. All user-related data is stored here. When an OTP is needed, it generate it and invokes the Alerts Service over gRPC to send it.
* **Alerts & Notifications Service:** Manages the ***Notifications*** domain. It exposes a gRPC interface for sending one-time passwords (used by Auth) via WhatsApp or email.   
  It subscribes to Kafka events (e.g. job application or status events) to notify users of job matches, application updates
* **AI Recommendation Service:** Handles the ***Recommendation*** domain. Orchestrated by an n8n workflow, it analyzes user profiles (skills, experience) and job data to generate personalized job recommendations and Personal Development Plans. It relies on external AI model ( Gemini ).
* **Job Service:** Encapsulates the ***Job Posting*** domain. Employers post jobs here; users browse and apply. It tracks job status (Active, Closed, Draft, Expired) and application status. When a user submits an application, this service records it and consume a   
  job.application.submit Kafka message to proccess it. when an application’s status changes, it emits job.application.statusChange Kafka event.

Each service thus owns a single, well-defined domain. This decomposition follows best practices (DDD/Single-Responsibility) by grouping related functionality within one service. For example, all user authentication and profile logic stays in the User Service, and job logic stays in the Job Service. This keeps services highly cohesive and loosely coupled.

## 5.2. High-Level System Architecture Diagram

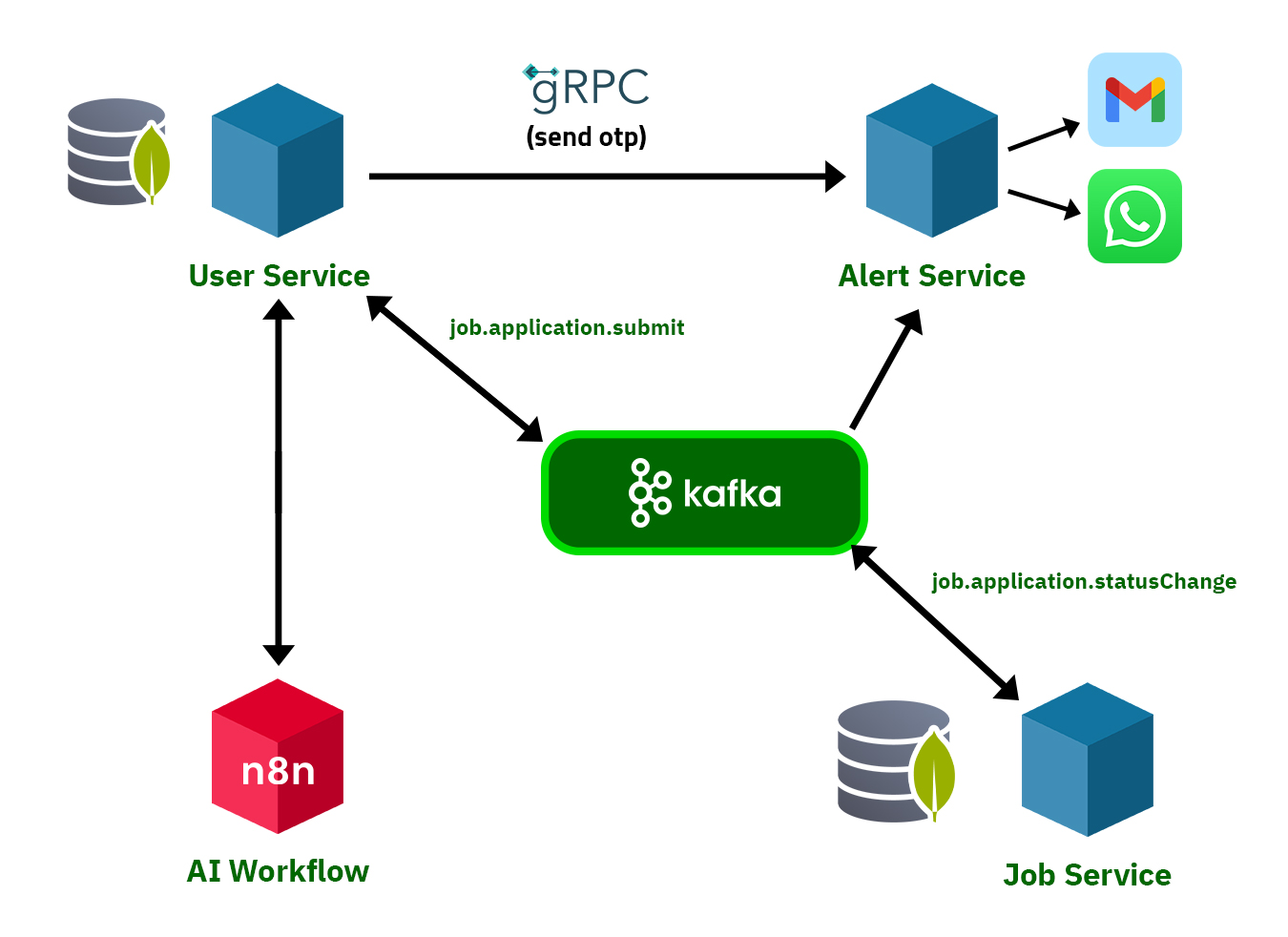
****

Figure 5

the diagram shows four main service blocks (User, Job, Alerts, AI) each connected to its own MongoDB, and lines between them indicating gRPC or Kafka links. Kafka occupies the center as the pub/sub backbone. External systems (WhatsApp, email, AI APIs) attach to the Alerts and AI services respectively. This topology illustrates how each service is a self-contained unit, with the grpc, Kafka, and http request enabling communication between them.

## 5.3. Core Design Principles

### 5.3.1. SOLID Principles Application

The Nuhoud system demonstrates consistent adherence to the SOLID principles across its microservices. The architecture benefits from the structure and conventions of the NestJS framework, with good separation of concerns and use of dependency injection. The application of these principles is outlined below with emphasis on the User and Job Services.

**Single Responsibility Principle (SRP):**

SRP is effectively applied throughout the system. In both the User and Job services, controllers (e.g., job-offers.controller.ts, application.controller.ts) handle request/response logic, while service classes (e.g., job-offers.service.ts, users.service.ts) encapsulate business logic and data access. DTOs are used for input validation and data transfer, and entities define the persistence models. This clear separation aligns well with SRP. However, in the User service, classes such as AuthService and UsersService are beginning to accumulate multiple responsibilities (e.g., authentication logic, role management, hashing), which may merit refactoring into more specialized components as complexity grows.

**Open/Closed Principle (OCP):**

OCP is evident in the consistent use of class extension and composition patterns. For example, UpdateJobOfferDto extends PartialType(CreateJobOfferDto), allowing behavior to be extended without modifying base structures. Decorators and guards in the authentication domain are also designed to be extended for new roles or permissions. In service logic, new functionality is generally introduced through new methods rather than altering existing ones, which preserves stability while enabling change.

**Liskov Substitution Principle (LSP):**

LSP is upheld through the use of well-structured DTOs and inheritance. Extended DTOs like UpdateJobOfferDto are used interchangeably in contexts where their base classes (CreateJobOfferDto) are expected. The system avoids violating substitutability by ensuring that extended types do not alter the expected behavior of their base types. No structural or behavioral violations were observed in the service or controller layers.

**Interface Segregation Principle (ISP):**

The codebase favors small, purpose-driven abstractions such as DTOs and Mongoose models. While explicit TypeScript interfaces are not widely used for services, each service class exposes narrowly scoped behavior aligned with a specific domain. There are no bloated or general-purpose interfaces forcing consumers to depend on unused methods. Future improvements could include formalizing interfaces for core services to further enhance testability and flexibility.

**Dependency Inversion Principle (DIP):**

DIP is strongly supported by the use of NestJS’s dependency injection mechanism. Services are injected into controllers and into one another via constructor injection, decoupling high-level modules from concrete implementations. For instance, the ApplicationService and JobOffersService inject Kafka clients and database models using NestJS providers. While the direct use of new this.model(...) is necessary for Mongoose models, all external services—such as messaging or notification services—could benefit from being abstracted behind interfaces to facilitate mocking and future replacement.

### 5.3.2. Design Patterns

This section outlines the design patterns currently implemented across the **Job Service** and **User Service**, evaluates how they are applied in the context of the NestJS framework

**1. Dependency Injection Pattern**

Dependency Injection (DI) is at the core of both services and is leveraged extensively through NestJS’s built-in DI system. Services, models, and external clients are injected using constructor injection with the @Injectable() decorator. Controllers consume these services without instantiating them manually, promoting clean separation of concerns and ease of testing.

**2. Decorator Pattern**

Both services rely heavily on the Decorator pattern. NestJS’s native decorators such as @Controller, @Post, @Body, @Injectable, and others are used throughout the application. Additionally, custom decorators like @Public() and @Roles() are used to encapsulate route-level metadata such as authentication and authorization requirements. DTO classes also use validation decorators from the class-validator library to enforce schema rules.

**3. Observer / Event Pattern**

The system implements the Observer pattern via Kafka. Events like job.application.submit and job.application.statusChange are emitted and consumed using @EventPattern, allowing asynchronous and decoupled communication between microservices. This pattern is particularly useful for maintaining the integrity of workflows like job applications and status updates without tight coupling.

**4. Module Pattern**

The application architecture is modular by design. Each feature domain (users, authentication, job offers, applications, etc.) is encapsulated in its own NestJS module. This promotes clean separation of responsibilities and makes the codebase scalable and easier to maintain.

**5. Singleton Pattern**

All services in the application are singletons by default due to NestJS’s service lifecycle. This ensures that shared dependencies such as database connections or utility classes are instantiated once and reused across the application lifecycle.

**6. Factory Pattern**

While not explicitly implemented with custom factory classes, the Factory pattern is present implicitly. NestJS and Mongoose rely on factories internally, for instance through SchemaFactory.createForClass() when defining schemas. The global exception handling pipeline also uses custom exceptionFactory functions to standardize error formats.

**7. Proxy Pattern**

The Proxy pattern is implemented indirectly through NestJS guards and interceptors. Guards such as AuthGuard and RolesGuard act as access control proxies that execute logic before reaching the actual route handler, enforcing authentication and authorization policies transparently.

## 5.4. Inter-Service Communication

### 5.4.1. Communication Patterns

Nuhoud employs a hybrid communication model. Public-facing APIs use traditional REST/HTTP, allowing clients (web or mobile) to interact with services in a simple, standardized way.

For example, logging in or searching jobs are done via REST endpoints. Internally, synchronous calls between services use gRPC. gRPC offers high-performance RPC over HTTP/2 with Protocol Buffers, which is more efficient than JSON/HTTP for inter-service calls. We use gRPC for operations where low latency is important (such as sending OTPs or real-time status checks). For asynchronous workflows and loose coupling, we use Apache Kafka. Services publish events (fire-and-forget) and other services subscribe to them. This event-driven approach ensures that services remain decoupled: a service emits an event and does not wait for a response.

### 5.4.2. Asynchronous Communication with Apache Kafka

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

Asynchronous, event-driven communication is handled by Apache Kafka. We define specific Kafka events for key domain activities. For example, when a user applies to a job, the Job Service publishes a job.application.submit event. When an application’s status changes (approved/rejected), it publishes job.application.statusChange. Other services subscribe to these events as needed.

Kafka provides a robust, scalable backbone for Nuhoud’s asynchronous communication, enabling reliable event delivery and independent service processing.

### 5.4.3. Synchronous Communication with gRPC

gRPC is a modern, high-performance, open-source framework for building APIs. It utilizes Remote Procedure Calls (RPC) for efficient communication between services, especially in microservices architectures. gRPC leverages HTTP/2 for transport and Protocol Buffers (protobuf) for serialization, resulting in faster and more lightweight communication compared to traditional REST APIs using JSON or XML.

Key Features and Benefits:

* High Performance:

gRPC utilizes HTTP/2 and protobuf, which leads to reduced latency and overhead compared to REST APIs.

* Language Agnostic:

gRPC supports various programming languages, allowing for polyglot microservice architectures.

* Contract-First API Development:

gRPC uses Protocol Buffers as an Interface Definition Language (IDL), ensuring a well-defined contract between services.

* Streaming Support:

gRPC offers support for client, server, and bidirectional streaming, making it suitable for real-time and data-intensive applications.

For synchronous inter-service calls that require low latency, we use gRPC. Notably, the Auth Service calls the Alerts Service’s gRPC API to send OTPs to users. This choice was made for performance reasons: gRPC runs over HTTP/2 with binary serialization (Protocol Buffers), which significantly reduces overhead. As one analysis notes, gRPC/HTTP2 can transmit messages “up to 10 times faster” than traditional HTTP1.1/REST.

## 5.5. Data Management Strategy (Database-per-Service)

We follow a database-per-service pattern using MongoDB for each service’s datastore. Each microservice has its own MongoDB instance or cluster and holds only its own data: the User Service stores users and profiles, the Job Service stores jobs and applications.

This encapsulation ensures loose coupling: a schema change in one service’s database has no impact on the others. All services use MongoDB (a document database) because our domain data (profiles, jobs, notifications) fits a flexible, schemaless model. Because services are isolated, we do not perform SQL-like joins across services. When data from another domain is needed, we use one of two approaches:

(1) the requesting service calls another service’s API synchronously, or (2) we replicate the necessary data asynchronously. To support efficient querying and avoid constant RPC calls, we do replicate selected data to read-optimized views.

. In summary, each service owns its MongoDB data and there are no live joins between databases.Controlled duplication of data (eventual-consistent views) is used only where needed for performance. This strategy maximizes service autonomy: each team can choose indexes and schemas suited to its needs, and the system remains scalable and fault-tolerant. Indeed, isolating data in this way helps ensure that the services are loosely coupled and supports independent scalability.

### 5.5.1. Data Modeling:

## 

Figure

## 5.6. Class Diagram

Figure

## 5.7. AI Service Integration using n8n

# Chapter 6: Technology Stack and Rationale

## 6.1. Programming Language & Runtime: Node.js

Node.js is the chosen runtime for the backend due to its asynchronous, event-driven architecture. It is a JavaScript runtime designed to build scalable network applications. Node.js employs a non-blocking, single-threaded event loop that can handle many concurrent connections efficiently, making it well-suited to high-throughput server tasks. Furthermore, Node.js benefits from JavaScript’s popularity , it has been the most widely used language in recent developer surveys , and from a vast open-source ecosystem. In fact, npm (Node’s package manager) is the largest package registry in the world, providing ready-made modules that significantly accelerate development and improve maintainability. Together, these properties (high concurrency, rich ecosystem, and a large developer community) make Node.js a performant and productive choice for our microservice-based system.

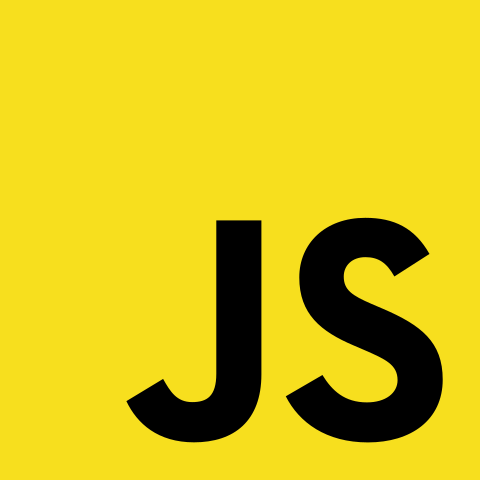


Figure 8

Figure 9

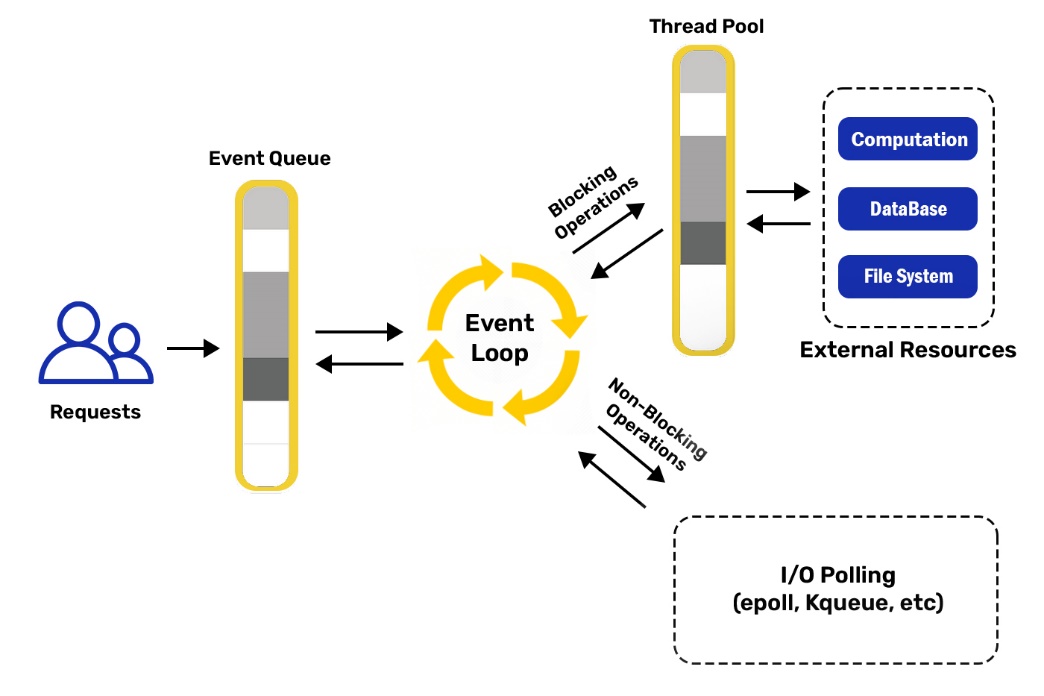


Figure 10

## 6.2. Backend Framework: NestJS

NestJS is a progressive Node.js framework built with TypeScript, chosen to structure the backend services. NestJS is described as a “progressive Node.js framework for building efficient, reliable and scalable server-side applications”. It introduces a modular architecture (inspired by Angular) that organizes code into self-contained modules and services, streamlining development and maintenance. Its built-in dependency injection and strong TypeScript type safety help catch errors at compile time and promote clean, testable code.

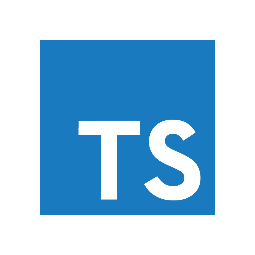


Figure 11

Figure 12

## 6.3. API Documentation: Swagger

the @nestjs/swagger module auto-generates a Swagger document from the code annotations. This approach provides an interactive, machine-readable API contract that is easy for developers and external clients to consume. Swagger’s advantages include automatic documentation generation (so that API docs stay in sync with code) and a built-in user interface for testing endpoints. As one source explains, “Swagger provides a unique and convenient platform to document, test, and write API structures”



Figure 13

## 6.4. Database and ODM: MongoDB with Mongoose

For data storage, MongoDB was selected as the primary database. MongoDB is a popular open-source NoSQL document database that stores data as JSON-like documents. This document oriented model provides a flexible schema, allowing the data model to evolve over time without rigid table definitions. MongoDB supports horizontal scaling (via sharding) and replication for high availability, enabling it to handle very large datasets and high traffic volumes. The JSON/BSON document format aligns naturally with JavaScript/Node.js, facilitating seamless data interchange between server and application code. To interface with MongoDB from Node, the Mongoose library (an Object Data Modeling – ODM – tool) was adopted. Mongoose enables developers to define schemas, models, and validation rules in Node.js, providing structure and consistency on top of MongoDB’s flexibility. According to MongoDB’s documentation, “Mongoose is an ODM (Object Data Modeling) library for MongoDB” that helps with data modeling, schema enforcement, model validation, and general data manipulation

Figure 14

Figure 15

## 6.5. Messaging/Event-Driven Architecture: Apache Kafka

[Apache Kafka](https://kafka.apache.org/) is a distributed streaming platform designed to handle large volumes of [real-time data](https://builtin.com/data-science/real-time-analytics). It’s an [open-source system](https://builtin.com/founders-entrepreneurship/open-source-future) used for stream processing, real-time [data pipelines](https://builtin.com/learn/tech-dictionary/data-pipeline) and data integration.

1. **Producer:** A producer generates a large amount of data and writes this into Kafka.
2. **Consumer:** Consumers act as end-users that read data from Kafka that comes from   
   producers.
3. **Topic:** Topic is a category or label on which records are stored and published. All Kafka records coming from   
   producers are organized into topics. Consumer applications read from topics.

Figure 16

1. **Brokers:** These are the Kafka servers that handle the data. Kafka brokers receives message from producers and stores them on its data
2. **Partition:** This is a unit of data storage. It’s a sequence of messages that is stored in a log and is   
   identified by a unique ID, known as the partition offset. Each partition is ordered and immutable, meaning that once a message has been written to a partition, it cannot be modified or deleted. A topic can have multiple partitions to handle a larger amount of data

## 6.5. RPC Communication: gRPC with Protocol Buffers

or inter-service RPC calls, gRPC with Protocol Buffers is employed. gRPC is a high-performance, open-source RPC framework developed by Google

Figure 17

gRPC was chosen for its efficiency and robust tooling: it provides high performance (HTTP/2 + Protobuf) and language-agnostic, type-safe service definitions, making it easier to build reliable, maintainable cross-service APIs

## 6.6. Containerization & Orchestration: Docker Compose

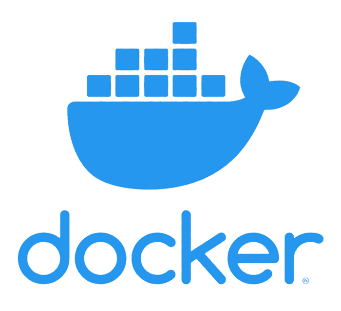
Docker is a platform that uses containerization to package and run   
applications. It allows developers to build, share, and run applications in isolated containers, ensuring consistency across different environments.

Figure 18

Docker Compose is used to containerize (Kafka and Kafdrop ) as a unified system. Docker Compose is a tool for defining and running multi-container Docker applications. In a single YAML file, and the entire stack can be started or scaled with a single command

## 6.7. Kafdrop Monitoring Tool: Kafdrop

While Kafka itself is a robust and high-throughput distributed log system, it lacks a native graphical interface for introspection. Kafdrop fills this gap by offering an intuitive interface to monitor and inspect Kafka activity during development and testing. Developers can browse topic metadata, view live messages within partitions, inspect headers and payloads, and validate that events are being published and consumed correctly.

Figure 19

# Chapter 7: Implementation

# Chapter 8: Testing

# Chapter 9: Conclusion and Future Work

## 9.1. Conclusion

## 9.2. Future Enhancements

## 9.3 Resources

# Appendices

Note:

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………

……………………………………………