Ministry of Education and Science of the Republic of Kazakhstan

SDU University



Dinmukhambet Turysbay

Optimizing Task Assignment in Project Management Using Recommendation Algorithms and Improving Team Efficiency

THESIS

Presented in Partial Fulfilment for the

Degree of Master of Science in Software Engineering

(degree code: 7M106105)

Department of Computer Sciences

School of Information Technologies and Applied Mathematics

Supervisor: **Gulsim Rysbayeva**

Kaskelen, 2026

School of Information Technologies and Applied Mathematics

Department of Computer Sciences

Dissertation Signatures Title Page

The Master's Dissertation of **Dinmukhambet Turysbay**

Title:

Optimizing Task Assignment in Project Management Using Recommendation Algorithms and Improving Team Efficiency

Үсыныс алгоритмдерін қолдану арқылы жобаларды басқарудағы тапсырмаларды бөлуді оңтайландыру және топтың тиімділігін арттыру

Оптимизация распределения задач в управлении проектами с использованием рекомендательных алгоритмов и повышение эффективности команды

Is submitted in partial fulfillment of the requirements for the Master's (MSc) degree in the Department of Computer Sciences, and approved by the Dissertation Committee:

Head of Committee: Assistant Prof.,PhD, Meraliyev M.

Academic supervisor: Senior-lecturer, PhD, Rysbayeva G.

Member's Signature: Assistant Prof.,PhD, Orynbekova K.

Member's Signature: Assistant Prof.,PhD, Ayazbayev D.

Қазақстан республикасы Алматы облысы Қаскелен ауданы Қаскелен қаласы 1996 SCHOOL OF INFORMATION TECHNOLOGIES AND APPLIED MATHEMATICS БИН 960240000550 SDU UNIVERSITY

Kaskelen, 2026

Declaration

I, Dinmukhambet Turysbay, hereby declare that this Master's thesis entitled *"Optimizing Task Assignment in Project Management Using Recommendation Algorithms and Improving Team Efficiency"*, submitted in partial fulfilment of the requirements for the Degree of Master of Science in Software Engineering, is the result of my own independent research and that all sources used or quoted have been duly indicated and acknowledged by means of complete references.

This work has not been previously submitted in whole or in part for the award of any other academic degree at this or any other institution. All software artefacts, datasets, models, evaluation results and figures presented in this thesis were produced by the author, with external libraries cited where they were used.

Dinmukhambet Turysbay

June 2026

Acknowledgments

In the name of Allah, the Most Gracious, the Most Merciful. All praise is due to Allah (SWT), whose guidance and blessings made this endeavor possible, and prayers and peace be upon our Prophet Muhammad, the seal of the prophets, and upon his family and companions.

I would like to express my sincere gratitude to my supervisor, Gulsim Rysbayeva, for her invaluable guidance, patient feedback, and constant encouragement throughout the preparation of this thesis. She was always available to answer questions and offer support regardless of the circumstances, and her dedication to helping me never wavered. Her thoughtful advice helped me refine the research direction at every stage — from the early scoping discussions through the design of the experimental protocol to the writing of the final manuscript.

I am also grateful to the academic staff of the School of Information Technologies and Applied Mathematics for the foundational courses in mathematics, statistics, machine learning, software engineering and data analysis that prepared me to undertake this work. Each of these courses and the instructors who taught them contributed, directly or indirectly, to the methods and thinking reflected in this dissertation.

Finally, I owe an enormous debt of gratitude to my family. To my beloved mother, who never stopped supporting me in every endeavor I undertook — offering endless encouragement, emotional strength, and financial support whenever I needed it most — and to my grandmother and younger brother, whose unwavering love carried me through the hardest moments: thank you. I would not have reached this day without you.

Dedication

To my beloved mommy, grandmother and younger brother
whose patience, endless encouragement
and unwavering belief in my success
made this all possible...

Annotation

Allocating software development tasks to the right team member is one of the most consequential decisions a project manager makes daily. This thesis presents the design, implementation, and empirical evaluation of a full-stack project management web application augmented by a machine learning recommendation engine that ranks developers for a given task. The problem is modelled as *learning-to-rank*: given a task description (skills, position, grade, deadline, effort, priority) and the team's current state (skills, workload, performance history), the system produces a ranked list of candidate developers with interpretable explanations.

Four strategies are compared under a single protocol on a held-out test set: (i) a transparent **heuristic baseline** combining seven priority-weighted component scores; (ii) an **embedding approach** using sentence-transformer cosine similarity; (iii) a **LightGBM LambdaRank** learning-to-rank model over thirty engineered features; and (iv) an **LLM-augmented** pipeline using a local Mistral model via Ollama for skill extraction and fit scoring. Seven additional classical algorithms provide a wider comparison. The system is implemented as a containerised micro-architecture: a FastAPI/PostgreSQL backend, a React (Vite) frontend with role-aware navigation, a sidecar Ollama service, and a GitLab CI pipeline deploying to Railway.

Evaluation uses a synthetic dataset of 1,000 (task, developer) pairs over one hundred tasks and twenty developers, with a task-grouped train–test split, scored by NDCG@K, MAP, MRR and Precision@1. The tuned baseline is strong (NDCG@5= 0.847, MAP= 0.794); embeddings give a small consistent gain (MAP= 0.799); LambdaRank slightly underperforms on synthetic data (MAP= 0.717); and the LLM pipeline matches embeddings while adding feature-enrichment and explanation value.

Keywords: project management, task assignment, learning-to-rank, LightGBM, LambdaRank, sentence embeddings, large language models, Ollama, FastAPI, React, NDCG, recommender systems.

Аңдатпа

Бағдарламалық қосымшаны әзірлеу тапсырмаларды топтың білікті мүшесіне тағайындау — жоба менеджері күн сайын қабылдайтын ең маңызды шешімдердің бірі. Бұл тезисте жобаны толық басқаруға арналған қосымшаны енгізу және импирикалық саралау жолдары көрсетілген. Мәселе «learning-to-rank» (сұрыптауды үйрену) ретінде модель жасалады: тапсырманың сипаттамасы (дағдылар, лауазым, грейд/деңгей, мерзімі, қажетті күш-жігер, басымдық) және команданың ағымдағы жағдайы (дағдылар, жұмыс жүктемесі, нәтижелер тарихы) берілген кездегі, жүйе түсіндірмелерімен бірге әзірлеуші-кандидаттардың сұрыпталған тізімін құрайды.

Бірдей хаттама аясында бөлектелген (held-out) тест жиынтығында төрт стратегия салыстырылады: (i) басымдық бойынша салмақталған жеті құрамдас көрсеткішті біріктіретін айқын эвристикалық базалық модель (baseline); (ii) sentence-transformer косинустық ұқсастығын пайдаланатын эмбеддинг тәсілі; (iii) инженерлік өңдеуден өткен отыз белгі негізіндегі LightGBM LambdaRank (learning-to-rank) моделі; және (iv) дағдыларды анықтау мен сәйкестікті бағалау үшін Ollama арқылы жергілікті Mistral моделін қолданатын LLM-мен толықтырылған pipeline. Сонымен қатар, кеңірек салыстыру үшін қосымша жеті классикалық алгоритм берілген.

Бағалау жұмысы жүз тапсырма мен жиырма әзірлеушіден тұратын 1 000 (тапсырма, әзірлеуші) жұбынан құралған синтетикалық деректер жиынтығында жүргізілді. Бұл ретте деректер тапсырмалар бойынша топтастырылып, оқыту және тест жиынтықтарына (train-test split) бөлінді және NDCG@K, MAP, MRR және Precision@1 метрикалары арқылы бағаланды. Оңтайландырылған базалық модель (baseline) жоғары нәтиже көрсетті (NDCG@5=0.847, MAP=0.794); эмбеддингтер аз да болса тұрақты өсім берді (MAP=0.799); LambdaRank синтетикалық деректерде сәл төменірек нәтиже көрсетті (MAP=0.717); ал LLM pipeline белгілерді байыту мен түсіндіру мүмкіндігін қоса отырып, эмбеддингтермен бірдей нәтижеге қол жеткізді.

Кілт сөздер: жобаларды басқару, тапсырмаларды бөлу, ранжирлеуді үйрену, LightGBM, LambdaRank, сөйлем эмбеддингтері, үлкен тілдік модельдер, Ollama, FastAPI, React, NDCG, ұсыныс жүйелері.

Аннотация

Распределение задач по разработке программного обеспечения между подходящими участниками команды — одно из самых ответственных решений, которые менеджер проектов принимает ежедневно. В данной диссертации представлены проектирование, реализация и импирическая оценка полнофункционального (full-stack) веб-приложения для управления проектами, дополненного рекомендательной системой на базе машинного обучения, которая ранжирует разработчиков для заданной задачи. Задача моделируется как «обучение ранжированию» (learning-to-rank): на основе описания задачи (навыки, должность, грейд, дедлайн, трудоемкость, приоритет) и текущего состояния команды (навыки, нагрузка, история эффективности) система формирует ранжированный список кандидатов-разработчиков с интерпретируемыми объяснениями. В рамках единого протокола на отложенной тестовой выборке сравниваются четыре стратегии: прозрачный эвристический базовый алгоритм (baseline), объединяющий семь взвешенных по приоритету компонентных оценок; эмбеддинг-подход с использованием косинусного сходства моделей sentence-transformer; модель обучения ранжированию LightGBM LambdaRank на основе тридцати спроектированных признаков (features); конвейер (pipeline) с использованием локальной LLM-модели Mistral через Ollama для извлечения навыков и оценки соответствия. Для более широкого сравнения также привлекаются семь дополнительных классических алгоритмов. Для оценки используется синтетический набор данных из 1 000 пар (задача, разработчик), включающий сто задач и двадцать разработчиков, с разделением на обучающую и тестовую выборки с группировкой по задачам. Оценка производится по метрикам NDCG@K, MAP, MRR и Precision@1. Настроенный базовый алгоритм показывает сильные результаты (NDCG@5 = 0,847, MAP = 0,794); эмбеддинги дают небольшой, но стабильный прирост (MAP = 0,799); LambdaRank демонстрирует чуть более низкие показатели на синтетических данных (MAP = 0,717); а конвейер на базе LLM работает на уровне эмбеддингов, добавляя при этом ценность в виде обогащения признаков и объяснения результатов.

Ключевые слова: управление проектами, распределение задач, обучение ранжированию, LightGBM, LambdaRank, эмбеддинги предложений, большие языковые модели, Ollama, FastAPI, React, NDCG, рекомендательные системы.

List of Abbreviations

API	Application Programming Interface
AUC	Area Under the Curve
CD	Continuous Deployment
CI	Continuous Integration
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
CSV	Comma-Separated Values
DCG	Discounted Cumulative Gain
ERD	Entity-Relationship Diagram
GBDT	Gradient-Boosted Decision Trees
HTTP	HyperText Transfer Protocol
IDF	Inverse Document Frequency
JSON	JavaScript Object Notation
JWT	JSON Web Token
KNN	k-Nearest Neighbours
LGBM	LightGBM
LLM	Large Language Model
LTR	Learning to Rank
MAP	Mean Average Precision
MLP	Multi-Layer Perceptron
MRR	Mean Reciprocal Rank
NDCG	Normalised Discounted Cumulative Gain
ORM	Object-Relational Mapping
PM	Project Manager / Project Management
REST	Representational State Transfer
RF	Random Forest
RNN	Recurrent Neural Network
SaaS	Software as a Service
SBERT	Sentence-BERT
SPA	Single-Page Application
SQL	Structured Query Language
SVM	Support Vector Machine
TF-IDF	Term Frequency–Inverse Document Frequency
UI / UX	User Interface / User Experience
XGB	Extreme Gradient Boosting
YAML	YAML Ain't Markup Language

Table of Contents

Declaration i

Acknowledgments ii

Dedication iii

Annotation iv

Annotation iv

Аңдатпа v

Аннотация vi

List of Abbreviations vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Research Questions . . . 2
- 1.4 Study Approach . . . 3
- 1.5 Contributions . . . 3
- 1.6 Scope and Limitations . . . 4
- 1.7 Structure of the Thesis . . . 5

2 Literature Review 6
- 2.1 Recommender Systems: An Overview . . . 6
- 2.2 Learning to Rank . . . 7
  - 2.2.1 Evaluation Metrics . . . 8
- 2.3 Sentence Embeddings . . . 9
- 2.4 Large Language Models in Software Engineering . . . 9
- 2.5 Task Assignment in Project Management . . . 10
- 2.6 Positioning of This Work . . . 11

3 System Design and Architecture 12
- 3.1 Deployment Topology . . . 12
- 3.2 Domain Model . . . 13

3.2.1 Users . . . 14
3.2.2 Projects and Members . . . 14
3.2.3 Tasks . . . 14
3.2.4 TaskHistory . . . 15
3.2.5 Positions . . . 15
3.3 Database Schema . . . 15
3.4 Authentication and Role-Based Access Control . . . 17
3.5 REST API . . . 18
3.6 The User Interface . . . 19
3.6.1 The Recommendation Drawer . . . 19
3.6.2 Colour Conventions . . . 20
3.6.3 Workload View . . . 20
3.7 Limitations and Future Improvements . . . 20

4 Methodology: The Machine Learning Recommendation Pipeline 22
4.1 The Unified Recommendation Engine . . . 22
4.2 The Heuristic Baseline . . . 23
4.2.1 The Seven Component Scores . . . 23
4.2.2 Priority-Aware Weights . . . 25
4.2.3 Human-Readable Reasons . . . 25
4.3 Feature Engineering . . . 25
4.3.1 Seven Score Features . . . 26
4.3.2 Six Meta Features . . . 26
4.3.3 Three Pairwise Interactions . . . 26
4.3.4 Four Priority One-Hot Flags . . . 27
4.3.5 Ten Derived Features . . . 27
4.4 Approach 2: Sentence-Transformer Embeddings . . . 28
4.4.1 Implementation . . . 28
4.4.2 Why I Expected Embeddings to Help . . . 28
4.5 Approach 3: LightGBM LambdaRank . . . 28
4.5.1 Training Setup . . . 29
4.5.2 Hyperparameters . . . 29
4.5.3 Output Calibration . . . 29
4.6 Approach 4: LLM-Augmented Pipeline . . . 30
4.6.1 Local Deployment via Ollama . . . 30
4.6.2 Implicit Skill Extraction . . . 30
4.6.3 LLM-Inferred Position . . . 31
4.6.4 Top-Three Fit Scoring . . . 31
4.7 Seven Additional Algorithms for Breadth Comparison . . . 31
4.8 Synthetic Data Generation . . . 32
4.8.1 Generating the Developers . . . 32
4.8.2 Generating the Tasks . . . 32
4.8.3 Labelling the Pairs . . . 33
4.8.4 The Final Dataset . . . 33
4.9 Evaluation Protocol . . . 33
4.9.1 Train/Test Split . . . 33

4.9.2 Metrics . . . 34
4.9.3 Reproducibility . . . 34
4.9.4 What the Evaluation Does *Not* Show . . . 34
4.10 Summary . . . 34

5 Implementation and Deployment 35
5.1 Backend Implementation . . . 35
5.1.1 Layout . . . 35
5.1.2 Application Bootstrap . . . 36
5.1.3 Authentication and JWT . . . 36
5.1.4 Database Access . . . 37
5.1.5 The Recommendation Endpoint . . . 37
5.1.6 Inferring Required Position via LLM . . . 38
5.2 Frontend Implementation . . . 38
5.2.1 Routing and the Auth Context . . . 39
5.2.2 API Client . . . 39
5.2.3 The Recommendation Drawer . . . 39
5.2.4 Tailwind Design System . . . 40
5.3 Local Development Workflow . . . 40
5.4 Containerisation . . . 41
5.4.1 Backend Production Image . . . 41
5.4.2 Frontend Production Image . . . 41
5.4.3 Ollama Sidecar . . . 41
5.5 Railway Deployment . . . 42
5.5.1 Service Layout . . . 42
5.5.2 First-Deploy Procedure . . . 42
5.6 Continuous Integration with GitLab . . . 42
5.6.1 Pipeline . . . 43
5.6.2 Why GitLab Rather Than GitHub Actions . . . 43
5.7 Deployment Considerations . . . 44
5.8 Summary . . . 45

6 Evaluation, Results and Discussion 46
6.1 Headline Results: The Four Approaches . . . 46
6.1.1 Interpretation of the Results . . . 47
6.2 The Algorithm Sweep . . . 47
6.2.1 Notable Observations . . . 49
6.2.2 Why the Baseline Is So Strong . . . 49
6.3 Feature Importance . . . 49
6.3.1 Interpretation . . . 49
6.3.2 Practical Implication . . . 51
6.4 Qualitative Case Studies . . . 51
6.4.1 Case 1: A Sparse Skills Description . . . 51
6.4.2 Case 2: Tie-Breaking by Workload . . . 52
6.4.3 Case 3: Free-text Notes Surface an Outsider . . . 52
6.5 Answers to the Research Questions . . . 53

6.5.1 RQ1: Is a Carefully Designed Heuristic Baseline a Strong Opponent? . . . 53
6.5.2 RQ2: Do Sentence-Transformer Embeddings Help? . . . 53
6.5.3 RQ3: Does LambdaRank Beat Pointwise Classification? . . . 53
6.5.4 RQ4: Does an LLM Help? . . . 54
6.6 Latency and Resource Considerations . . . 54
6.7 Limitations and Threats to Validity . . . 55
6.8 Future Work . . . 56
6.9 Conclusion . . . 57

A LLM Prompt Templates **58**
A.1 Implicit Skill Extraction . . . 58
A.2 Position Inference (Closed Vocabulary) . . . 58
A.3 Developer Fit Scoring . . . 59

B Sample LLM Outputs **60**
B.1 Skill Extraction . . . 60
B.2 Position Inference . . . 60
B.3 Fit Scoring . . . 61

C Reproducing the Empirical Results **62**
C.1 Environment Setup . . . 62
C.2 Regenerating the Synthetic Data . . . 62
C.3 Training the LightGBM Ranker . . . 62
C.4 Training and Comparing the Seven Algorithms . . . 63
C.5 Running the Unified Evaluation . . . 63
C.6 Running the Application Locally . . . 63

D API Reference (Selected Endpoints) **64**
D.1 POST /api/auth/login . . . 64
D.2 POST /api/projects/{id}/tasks . . . 64
D.3 POST /api/tasks/{id}/recommend . . . 65
D.4 PATCH /api/tasks/{id} . . . 65
D.5 GET /api/workload . . . 65

Bibliography **65**

Chapter 1

Introduction

1.1 Motivation

Task assignment is a recurring activity in agile software development. During sprint planning, a project manager must decide which developer should be assigned to each incoming task. In practice, this decision is predominantly made through informal mechanisms: voluntary self-selection, managerial intuition, or recency bias toward the most recently visible team member. Although the decision significantly affects team productivity, code quality, and developer satisfaction, it has received comparatively little systematic attention in the software engineering literature.

Modern project-management platforms such as Jira, Linear, Asana, and ClickUp already collect substantial data about tasks and developers, including skill annotations, workload histories, role designations, and free-text descriptions. However, this data is rarely leveraged to inform the assignment decision in a structured manner. Project managers continue to rely on mental heuristics that, while often reasonable, are subject to well-documented cognitive biases such as availability bias and anchoring .

The limitations of manual assignment become more pronounced as teams scale. In small teams, a project manager may possess sufficient knowledge of each developer's strengths and current workload to make effective assignments. In larger teams, or in organisations where team composition changes frequently, this knowledge degrades. Furthermore, even when structured metadata is available, it is frequently incomplete: in pilot data collected for this study, more than one-third of tasks lacked a populated `required_skills` field and approximately one-fifth had no specified `required_position`.

These observations motivated the present study, which investigates whether data-driven recommendation techniques can assist project managers in making more informed task-assignment decisions, while maintaining sufficient transparency to support human oversight and trust.

1.2 Problem Statement

The task-assignment problem can be framed as follows: given a newly created task with associated metadata (title, description, required skills, priority, deadline) and a set of available developers with known attributes (position, grade, skills, current workload, free-text profile notes), the objective is to produce a ranked list of candidate developers ordered by their estimated suitability for the task.

Several factors contribute to the difficulty of this problem. First, as noted above, the structured metadata associated with tasks is routinely incomplete, requiring the system to extract meaningful signals from unstructured text fields. Second, developers with ostensibly identical role designations may possess substantially different skill profiles; for example, two senior backend developers may differ significantly in their framework-specific expertise. The most informative data often resides in free-text notes maintained by managers, which are rich in context but inaccessible to simple matching algorithms.

Third, the constraints governing assignment interact in non-trivial ways. The developer with the strongest skill match may simultaneously be the most overloaded, making a less-skilled but more available developer a preferable choice under tight deadlines. Project managers implicitly weigh these trade-offs but rarely formalise them. Fourth, developer capabilities evolve over time; a junior developer who lacked relevant experience three months ago may have acquired it through recent sprint work. Static rule-based systems cannot accommodate such temporal dynamics, whereas data-driven approaches can, in principle, adapt.

Finally, any recommendation system deployed in this context must be *transparent*. A recommendation that the project manager cannot interpret or challenge is unlikely to be adopted. The recommender-systems literature consistently identifies explainability as a prerequisite for user trust in high-stakes settings [**?**]. This requirement shaped the design of the system presented in this thesis.

1.3 Research Questions

This thesis is structured around five research questions.

RQ1. How effectively can a transparent, hand-tuned heuristic perform as a task-assignment recommender? Specifically, can a weighted-sum scorer comprising seven explainable component scores with priority-aware weights achieve competitive performance relative to machine-learning approaches on a realistic dataset? This question establishes the heuristic as a baseline against which more complex approaches must justify their additional complexity.

RQ2. Do dense sentence-transformer embeddings, used in place of Jaccard-based text overlap in the baseline scorer, improve recommendation quality? The hypothesis is that the improvement will be positive but modest, given that text similarity constitutes one component out of seven in the scoring function.

RQ3. Does a learning-to-rank model—specifically, LightGBM trained with the

`lambdarank` objective on thirty engineered features—outperform pointwise classifiers trained on the same feature set? The information-retrieval literature suggests that listwise ranking objectives should yield superior NDCG scores; however, this advantage may not transfer to small, well-conditioned datasets.

RQ4. Can a small, locally deployed large language model (Mistral 7B via Ollama) usefully enrich sparse task descriptions? When the project manager has not specified required skills, can the LLM extract a meaningful skill set from the task description, and does this enrichment measurably improve recommendation quality?

RQ5. Among classical supervised-learning algorithms, which serves as the most effective default for this problem domain? Seven algorithms—Logistic Regression, SVM with an RBF kernel, k-Nearest Neighbours, Decision Tree, Random Forest, XGBoost, and a multi-layer perceptron—are trained on an identical feature set with the same train–test split and evaluated under uniform metrics. This question complements RQ3 by establishing the relative performance of pointwise classifiers against one another and against the listwise ranker.

1.4 Study Approach

This study combines system design with empirical evaluation. The engineering component involves the development of a full-stack web application that serves both as a functional project-management tool and as the infrastructure through which recommendation strategies are deployed and evaluated. The application architecture is deliberate: the user interface creates data that the recommendation engine requires (e.g., the free-text developer profile notes that enable embedding-based similarity), while the engine's output is surfaced through the interface in a manner designed to support transparency and user trust. The `was_accepted` feedback signal, which records whether a project manager accepted or overrode a recommendation, is captured through the interface and stored for potential use in future model retraining.

The empirical component involves the systematic comparison of eleven configurations: four primary recommendation strategies (weighted heuristic baseline, embedding-enhanced baseline, LightGBM LambdaRank, and LLM-augmented pipeline) and seven classical machine-learning algorithms. All configurations are evaluated on a synthetic dataset using a deterministic held-out test set and a unified evaluation protocol comprising NDCG@K, MAP, MRR, and Precision@1.

1.5 Contributions

The contributions of this thesis are summarised as follows.

A comparative empirical study. Eleven configurations are evaluated under identical conditions. The principal finding is that a carefully designed heuristic baseline proved difficult to outperform: the LightGBM LambdaRank model, the theoretically favoured approach for a ranking objective, slightly underperformed the weighted-sum baseline on the synthetic dataset, while the LLM-augmented pipeline produced strong qualitative improvements without substantial gains in aggregate metrics. Feature-importance analysis and qualitative case studies contextualise the quantitative results.

A unified recommendation engine with four interchangeable backends. A single Python module orchestrating four strategies—weighted heuristic, embedding-enhanced heuristic, LightGBM LambdaRank, and LLM-augmented pipeline—behind a common REST interface, enabling transparent switching between approaches without modification to the calling application or evaluation framework.

A reusable feature-engineering pipeline. Thirty features per (task, developer) pair, organised into five groups: seven raw component scores, six task/developer meta-features, three pairwise interactions, four priority one-hot encodings, and ten derived features (ratios, polynomial terms, and three-way interactions). The pipeline operates in both offline (batch training) and online (single-request inference) modes.

A synthetic data generator and unified evaluation framework. A configurable generator producing realistic (task, developer, was_accepted) triples with controllable difficulty, paired with an evaluator that applies identical metrics across all eleven configurations on a deterministic test set.

A deployed full-stack application. A project-management web application comprising a FastAPI backend with PostgreSQL and SQLAlchemy, a React/Vite frontend with role-aware navigation, JWT authentication, a Kanban board, a workload view, an administration panel, and a recommendation interface. The system is containerised and deployable via `docker compose up`, with continuous deployment to Railway via a GitLab CI pipeline.

1.6 Scope and Limitations

The following aspects are explicitly outside the scope of this study. *Multi-project assignment*: the recommender operates within a single project context and does not optimise developer allocation across a project portfolio. *Real-time rebalancing*: the system produces recommendations at the point of task creation or triage and does not dynamically redistribute assignments during a sprint. *Cross-organisational generalisability*: the evaluation employs synthetic data modelled on a single team structure; generalisability to substantially different organisational contexts has not been validated. These limitations are revisited in the future-work discussion presented in Chapter 6.

1.7 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 presents a review of the relevant literature, covering recommender systems, learning-to-rank methods, sentence embeddings, the application of large language models in software engineering, and prior work on task assignment in project management. Chapter 3 describes the system design, including the domain model, relational schema, access-control rules, REST API, and user interface. Chapter 4 details the methodology: the four recommendation strategies, the feature-engineering pipeline, the seven baseline algorithms, the synthetic data generator, and the evaluation protocol. Chapter 5 addresses implementation and deployment, providing sufficient detail to enable reproduction of the running system. Chapter 6 presents the empirical results, revisits each research question, discusses the implications of the findings, identifies limitations, and outlines directions for future work.

Chapter 2

Literature Review

This thesis draws on five strands of literature.. The first is the long-running conversation about recommender systems. The second is the narrower one about learning to rank, which gave us the algorithms that LightGBM ships with today. The third is the very recent boom in sentence embeddings, where Sentence-BERT changed what "semantic similarity" could mean on commodity hardware. The fourth is the even-more-recent conversation about Large Language Models in software engineering. The fifth, the one I had the most trouble finding, is the small literature on automatic task assignment specifically. This chapter is an attempt to synthesise the five.

It should be noted that this is not a complete survey; a full survey of recommender systems alone would fill a monograph. What follows covers the work that influenced the design choices in Chapter 4, plus a few side notes on directions I deliberately did not pursue.

2.1 Recommender Systems: An Overview

The recommender-systems literature splits its history into three phases. The first phase was content-based filtering, in which both items and users are described by feature vectors and recommendations are based on the closeness of the vectors [23]. Early content-based systems were essentially TF—IDF search engines personalised by the user's history: the user's history defined the query.

The second phase was collaborative filtering, which rose to prominence with the Netflix Prize in 2006. The principal result of that competition was that matrix factorisation could recover useful latent embeddings of users and items from nothing but the rating matrix [16]. The phase ended with a realisation that purely collaborative methods cannot recommend an item nobody has rated yet, and the field settled into the hybrid third phase that prevails today: structured features, free-form text, behavioural signals and embeddings all combined into a single ranking model.

The problem I work on sits squarely inside the third phase, but it has two

features that distinguish it from the canonical movie- or product-recommendation setting. The first is that there are effectively no collaborative signals: developer A's acceptance of task t_i tells us very little about whether developer B should accept task t_i as well, because the tasks are not interchangeable. The second is that the "items" (developers) are people whose features change over time and whose response to being recommended is itself consequential. These two features push the design firmly toward content-based, feature-engineered methods — which is the direction this thesis adopts.

An adjacent literature: bug triage. The closest existing line of work is automatic *bug triage*. Given a new bug report, who should fix it? The seminal paper is Anvik, Hiew and Murphy's 2006 ICSE study [3], which treated bug triage as multi-class classification with one class per developer and a TF–IDF representation of the bug report. They tried naive Bayes, SVMs, and C4.5 decision trees, and reported precision in the 50–60 % range on Eclipse and Firefox bug data. Jeong, Kim and Zimmermann's follow-up [13] added a tossing graph that captured the historical re-assignment patterns of bug reports — a clever idea that I considered for tasks but ultimately discarded, because task tossing is much rarer than bug tossing.

More recent bug-triage work has applied deep learning, in particular LSTM and attention-based encoders, to the bug-report text [18]. Their reported gains are real, but they require both a large dataset and a stable developer roster — two assumptions that this thesis cannot make.

My setting differs from bug triage in three important ways. First, the features extend well beyond text: I have explicit structured fields (position, grade, current workload, performance score, deadline pressure, task priority) that the bug-triage literature usually does not consider. Second, I treat the problem as ranking rather than classification, which avoids the awkward "one class per developer" explosion when team membership changes. Third, I deliberately keep the free-form manager notes as a distinct, embedded text channel, separate from the task description — a design choice motivated by the qualitative case studies in Section 6.4.

2.2 Learning to Rank

Learning to rank is the sub-field that asks: instead of predicting a relevance label for each item independently, can we directly optimise the *order* of a list [17]? The literature divides the answers into three families.

Pointwise. Treat each (query, item) pair as an independent training example and predict a binary relevance label or a numeric relevance score. Any classifier or regressor will do. The theoretical drawback is is that the loss is computed on individual items while the evaluation metric is computed over the entire query — so the training and evaluation are not aligned. In practice the misalignment matters less than might be expected, as the experiments in Chapter 6 will show.

Pairwise. The training signal is a preference between two items for the same query. The canonical algorithms are RankSVM [14], RankNet [4], and the family that grew from RankNet: LambdaRank [6] and LambdaMART [5]. Burges' insight in LambdaRank was that the pairwise gradient can be scaled by the absolute change in NDCG that would result from swapping the two items, bringing the gradient closer to what we actually care about even though NDCG itself is not differentiable. This is the objective that LightGBM exposes as `lambdarank` and that the LightGBM model in this thesis uses.

Listwise. The loss is computed over an entire ranking. ListNet [7] uses a softmax over the items in a list; ListMLE uses a Plackett–Luce likelihood. Listwise losses are conceptually the cleanest of the three but tend to need larger datasets to outperform pairwise in practice. For the scale of data I have access to, listwise was not a serious contender, so it is described here for completeness only.

Why LightGBM. LightGBM [15] won this thesis's algorithm comparison on grounds that have little to do with sophistication and a lot to do with practicality. It is fast — training the LambdaRank on 1600 training pairs takes under a second on a laptop. It accepts the ranking group structure directly through a `group` parameter. It exposes feature importance natively. And, crucially, it has a straightforward Python API that fits inside a `predict_from_scores` call in our serving layer. XGBoost [8] would have worked equally well; the choice between them is more a matter of ecosystem than algorithm.

2.2.1 Evaluation Metrics

These metrics are summarised here, as the empirical chapter relies on them heavily.

For a single query with sorted relevance labels $r_1, r_2, \ldots, r_n$, the Discounted Cumulative Gain at rank K is

$$\mathrm{DCG@}K \;=\; \sum_{k=1}^{K} \frac{r_k}{\log_2(k+1)}, \tag{2.2.1}$$

and NDCG@K divides DCG@K by the ideal DCG@K computed by sorting the labels descending. NDCG ranges in $[0, 1]$ and is the metric most directly aligned with our problem: how much credit do we get for the top-K items, with positions further down discounted logarithmically.

Mean Average Precision (MAP) averages, over queries, the precision at each rank where a relevant item appears. Mean Reciprocal Rank (MRR) is the average over queries of the reciprocal of the rank of the first relevant item — a metric well suited to this setting, reflecting how far down the ranked list a manager must look before finding a suitable candidate. Precision@1 is simply the fraction of queries whose top item is relevant.

In a recommender for developers, none of these four metrics is sufficient on its own. NDCG@1 is stringent and rewards perfection at the top. NDCG@5 is more lenient and may mask poor behaviour. MAP is sensitive to recall further down the list, which matters when the PM genuinely looks at five or six candidates. MRR is robust to ties. I report all of them in Chapter 6.

2.3 Sentence Embeddings

The story of sentence embeddings is a story of the field figuring out how to use BERT [10] for similarity. BERT itself produces one vector per token, with the special `[CLS]` vector nominally summarising the input — but as Reimers and Gurevych pointed out in 2019 [22], the `[CLS]` vector of an off-the-shelf BERT is essentially useless for cosine similarity. The semantic information is there, but it is not arranged the way a cosine measure expects.

Sentence-BERT (S-BERT) fixed this by fine-tuning a siamese BERT on the Stanford Natural Language Inference dataset, with a loss that encourages semantically similar sentences to land close in the embedding space. The resulting model produced sentence vectors that, compared with simple cosine, beat the previous state-of-the-art on a battery of semantic textual similarity benchmarks while being roughly two orders of magnitude faster.

A whole family of distilled and specialised variants has appeared since then. The one this thesis uses, `all-MiniLM-L6-v2`, is a 22M-parameter distilled model that outputs 384-dimensional vectors. It is small enough to load into the FastAPI process on a 512 MB Railway instance and fast enough to embed a developer's notes in under 200 ms. The Massive Text Embedding Benchmark [20] confirms that it is a sensible default for English semantic similarity.

Practical properties I leaned on. Two properties of sentence embeddings made them especially convenient in this work. The first is that developer notes change rarely, so I can cache their embeddings to disk under an MD5 hash of the lowercased text. The second is that the model's outputs are L2-normalised by default, so cosine similarity reduces to a dot product and a batch similarity computation against N candidates is a single $(N \times 384) \cdot (384,)$ matrix multiplication.

2.4 Large Language Models in Software Engineering

I started this thesis sceptical of LLMs. This section sets out the resulting position.

The literature on LLMs in software engineering has expanded in the last two years. Recent surveys [12] catalogue applications across code review, documentation, test generation, program repair, requirements engineering, and developer-experience tools. A smaller but rapidly growing subset uses LLMs for what is

essentially HR analytics: parsing resumes, extracting skills from job postings, and ranking candidates.

The single most useful idea I borrowed from this literature is *skill extraction* as a feature-enrichment step. Decorte et al. [9] demonstrated that fine-tuned LLMs can extract competencies from job postings with significantly higher recall than rule-based extractors. The intuition transfers to my setting: a PM types "Build a real-time analytics dashboard" and implicitly assumes WebSockets, time-series storage, a charting library, and probably some sort of alerting. None of those words appear in the task. A pattern-matching extractor will return nothing; a competent LLM will return a list of five or six skills with high recall.

A second idea, more controversial, is *LLM-as-a-judge*. Zheng et al. [24] showed that GPT-4, asked to compare two chatbot responses, agrees with human evaluators almost as often as human evaluators agree with each other. A natural temptation is, is to use an LLM to score developer fit on a 1–10 scale — which is exactly what `LLMExtractor.score_developer_fit` does. I am careful in Chapter 6 to report this as a qualitative contribution rather than a quantitative one; LLM-as-a-judge scores are useful in the UI but I do not trust them as a primary ranking signal.

Local deployment via Ollama. A practical constraint that shaped the implementation was that I did not want the system to depend on a cloud LLM API. Sending task descriptions to OpenAI's servers is, depending on your jurisdiction, may be acceptable, problematic, or prohibited. Tools such as `llama.cpp` [11] and Ollama [1] make it possible to run quantised versions of medium-sized open models (Llama 3, Mistral, Phi 3, Gemma) on commodity hardware with a clean HTTP API. The fact that this is possible at all on a Railway instance is, in itself, itself a notable development of the last two years; five years ago this thesis would have required either a GPU or a willingness to ship data to a third party.

2.5 Task Assignment in Project Management

The literature that targets task assignment specifically is the thinnest of the five. Most of the academic work on the topic comes from operations research, where the assignment problem is famously NP-hard [19, 21]. The operations-research formulation models the problem as a generalised assignment with capacity constraints and a fixed objective; it is elegant in theory but of limited practical use, because real teams do not have stable utility functions and effort estimates are notoriously unreliable.

What real PMs do is closer to what Amershi et al. [2] call *human-in-the-loop ML*: the system makes a suggestion, the human accepts or rejects, the system learns from the corrections. This is the design philosophy of the recommendation engine in this thesis. The PM remains in control; the engine accelerates rather than replaces.

Industry analogues. Most large project-management tools "have begun to offer some form of 'smart assignment'" — Jira's automation rules, Linear's auto-triage, ClickUp's AI assignee recommendations, Notion's task delegation suggestions. Public documentation of what these features actually do is sparse; engineering blog posts hint at a mixture of load balancing, last-assignee preference, and keyword-driven rules. None of the public descriptions I could find involves a learning-to-rank model on engineered features, which is part of the motivation for the present work.

2.6 Positioning of This Work

This section situates the present thesis relative to the bodies of literature surveyed above.

The core departure from prior task-assignment studies is the choice of objective: this work frames developer recommendation as a learning-to-rank problem rather than as multi-class classification, which has been the dominant formulation in the bug-triage tradition. The ranking formulation better reflects the practical setting, in which a project manager requires an ordered shortlist rather than a single predicted label.

For textual similarity, the system relies on sentence-transformer embeddings in place of TF–IDF or Jaccard overlap. While this choice is well established in the information-retrieval community, it has not yet become standard practice in the task-assignment literature.

The system incorporates a feature-enrichment step driven by a locally deployed large language model. This step draws on recent skill-extraction research but adapts it to the on-premise constraints typical of enterprise project-management environments, where external API calls may not be permitted.

A fully transparent, hand-tuned heuristic baseline is retained alongside the machine-learning models. It serves a dual purpose: as a controlled comparison against which learned approaches must demonstrate added value, and as a production fallback for deployments in which training data has not yet been collected. This design reflects the human-in-the-loop principle that a recommendation system should augment, not replace, the project manager's judgement.

Finally, the recommendation engine is implemented within a deployable full-stack application with continuous integration and a production deployment pipeline, rather than as an isolated experimental notebook. This design decision enables the qualitative case studies presented in Chapter 6 to reflect realistic usage conditions.

Chapter 3

System Design and Architecture

A common temptation in a thesis like this is to describe the system as though it had sprung fully formed from a clean architecture diagram. The reality is that the application went through three substantial redesigns before it reached the shape described in this chapter, and several minor aspects changed late in the project. I have tried to write the chapter in a way that explains not only *what* the system looks like today but *why* it took the shape it did, with honest notes where a particular choice turned out to be either better or worse than I expected.

The chapter has six sections. Section 3.1 sketches the deployment topology. Section 3.2 introduces the core entities. Section 3.3 gives the relational schema. Section 3.4 covers role-based access control and authentication. Section 3.5 catalogues the REST API. Section 3.6 walks through the user interface.

3.1 Deployment Topology

The system is a conventional three-tier web application with one optional extra: a sidecar Large Language Model. Figure 3.1 sketches the runtime topology.

The split into three independently deployable services took two iterations to settle. The first version of the application bundled the SPA, the FastAPI server, and the database into a single Docker image served by Gunicorn behind Nginx. This was straightforward to deploy, but turned out to be hard to scale: any change to the SPA forced a redeploy of the API. The second version split the SPA into its own Nginx container and the API into its own Uvicorn container, which is the topology that survives today. The third change, late in the project, was to add the Ollama sidecar; I tried first to run Mistral in-process inside the FastAPI container and mmediately encountered a problem that loading the weights blocked API startup for forty seconds and added 4 GB of memory to a container that otherwise needed 400 MB. Moving Mistral to a sidecar made the API container restored the API container to a small memory footprint and fast startup.

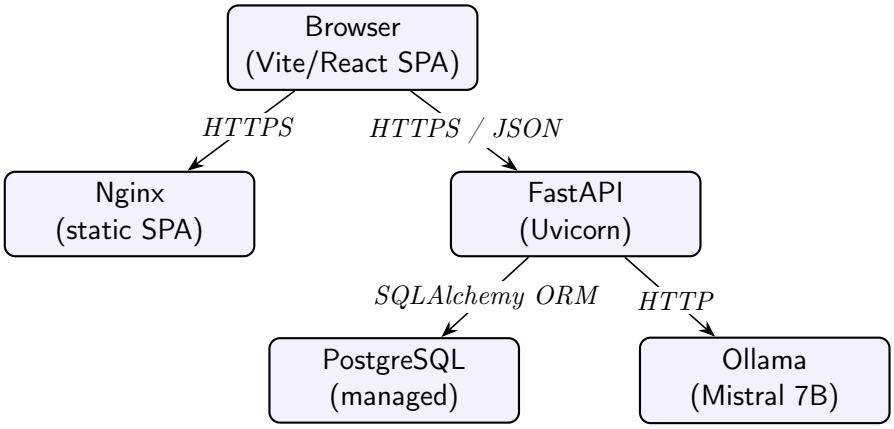


Figure 3.1: Deployment topology. The frontend, the backend, and the Ollama LLM service are independent containers. PostgreSQL is a Railway-managed database. The LLM sidecar is optional: if it is unreachable, the system degrades gracefully and uses the non-LLM approaches.

A note on the choice of FastAPI. I chose FastAPI for three reasons worth recording, because the reasons matter for anyone trying to replicate the system. The first is pydantic — being able to define request and response schemas as Python classes and have FastAPI validate them at the boundary saved substantially reduced debugging effort. The second is the OpenAPI documentation that FastAPI generates automatically, which the frontend used as the source of truth for its API client. The third is the asynchronous runtime: although most of the database access is synchronous (because SQLAlchemy was used in sync mode), the asynchronous server still lets the LLM-augmented endpoint sit on a 30-second timeout without blocking other requests.

A note on the choice of React + Vite. The frontend uses Vite because the developer experience is, much better than the Webpack-based alternative. A cold start of the dev server is well under a second, and changes appear in the browser in roughly 50 ms. For a project where the UI evolved alongside the recommendation engine, this iteration speed mattered more than any architectural property.

A note on the choice of Railway. Railway was chosen because it offers a managed Postgres plugin with a reference variable that lets the backend service refer to the database URL by name rather than by literal string. This single feature saved me from a class of mistakes eliminated a class of configuration errors that would otherwise have been easy to make.

3.2 Domain Model

Six entities sit at the heart of the application: users, projects, tasks, the many-to-many join between projects and developers, a task history table that records completion outcomes, and a small position taxonomy.

3.2.1 Users

A `User` represents anyone who can log in. Every user has a `username`, a hashed password, and a role drawn from `{ADMIN, PR_MANAGER, DEVELOPER}`. Developer-specific fields warrant particular attention:

- `position` — a free-text label such as `backend`, `frontend`, `devops`, drawn from the Position taxonomy.
- `grade` — `junior`, `middle`, or `senior`.
- `skills` — a JSON-encoded list of skill labels such as `["Python", "FastAPI", "PostgreSQL"]`.
- `notes` — a free-text paragraph that captures information a structured list cannot. In practice this is where managers record entries such as "ex-Yandex SRE, owns the incident playbook". The embedding similarity (Section 4.4) leans heavily on this field.
- `current_workload` — a denormalised integer that counts active tasks. It is recomputed at the end of every task transition.
- `performance_score` — a float in $[0, 1]$ aggregated from the `TaskHistory` table. We use an exponentially weighted moving average so that recent performance matters more than old performance.

The `notes` field was added late in the project and turned out to be the most useful piece of information the embedding approach has access to. The lesson, in retrospect, is to give users a place to write the things they cannot fit into structured fields.

3.2.2 Projects and Members

A `Project` groups tasks. It has a name, a description, an optional deadline, a status, and a `manager_id` foreign key pointing at the project manager who owns it. Membership of developers in a project is recorded explicitly in a `project_members` join table; not every developer in the organisation belongs to every project, and the recommendation engine must be able to determine which developers could feasibly take a given task

This is also why the recommendation engine does *not* use the join table by default — the recommendation drawer can search across all developers, not just project members, because PMs occasionally want to recruit a developer onto the project for a specific task. The behaviour is configurable through the `exclude_user_ids` parameter of `get_recommendations`.

3.2.3 Tasks

`Task` is the most information-rich entity. Beyond title, description, project, optional assignee and deadline, it carries `required_position`, `required_grade`, `required_skills` (JSON-encoded list), `estimated_hours`, `priority` and `status`.

If the PM creates a task with an empty `required_position`, an LLM call is made to infer the position from the title and description, constrained to the canonical list of positions seeded in the database (see Section 4.6.3). If Ollama is unavailable, the field stays empty and downstream scorers handle that case gracefully.

3.2.4 TaskHistory

After a task is completed, a single row is appended to `TaskHistory` recording whether the task was on time and a quality score in $[0, 1]$. The quality score, in the current implementation, is set by the manager when they close the task.

The history table is the foundation of every developer's `performance_score`: a job runs periodically (in production this is a cron, in development it is a button on the admin page) that recomputes the score as an exponentially weighted average of the last twenty `TaskHistory` rows.

3.2.5 Positions

Finally, a small `Position` table holds the canonical list of position names. The default seed is `[backend, frontend, fullstack, devops, qa, mobile, data, ml]`. The list is editable by administrators, but the LLM-based position inference uses it as a closed vocabulary: the model is asked to pick exactly one of these, nothing more.

3.3 Database Schema

The full SQLAlchemy declaration of the schema runs to about 200 lines. Listing 3.3 shows the condensed version that covers the core relations; the full version adds relationships, cascading deletes, indices on the foreign keys, and the `project_members` association table.

```
class User(Base):
    __tablename__ = "users"
    id                = Column(Integer, primary_key=True, index=True)
    username          = Column(String(50), unique=True, index=True)
    password_hash     = Column(String(256), nullable=False)
    role              = Column(Enum(UserRole), nullable=False)
    full_name         = Column(String(100))
    email             = Column(String(100))
    position          = Column(String(50))
    grade             = Column(String(20))
    notes             = Column(Text)
    skills            = Column(Text)
    current_workload  = Column(Integer, default=0)
    performance_score = Column(Float, default=0.8)
    created_at        = Column(DateTime, default=datetime.utcnow)
```

```
class Project(Base):
    __tablename__ = "projects"
    id          = Column(Integer, primary_key=True)
    name        = Column(String(200), nullable=False)
    description = Column(Text)
    manager_id  = Column(Integer, ForeignKey("users.id"))
    status      = Column(String(50), default="active")
    deadline    = Column(DateTime)

class Task(Base):
    __tablename__ = "tasks"
    id                = Column(Integer, primary_key=True)
    title             = Column(String(200), nullable=False)
    description       = Column(Text)
    project_id        = Column(Integer, ForeignKey("projects.id"))
    assigned_to       = Column(Integer, ForeignKey("users.id"))
    priority          = Column(Enum(TaskPriority))
    status            = Column(Enum(TaskStatus))
    deadline          = Column(DateTime)
    required_position = Column(String(50))
    required_grade    = Column(String(20))
    required_skills   = Column(Text)
    estimated_hours   = Column(Integer, default=8)
    created_at        = Column(DateTime, default=datetime.utcnow)
    completed_at      = Column(DateTime, nullable=True)
```

Listing 3.1: Condensed schema.

Cardinality. The relationships between the main entities are:

- User (PR_MANAGER) → Project: one-to-many via `manager_id`.
- User (DEVELOPER) ↔ Project: many-to-many via `project_members`.
- Project → Task: one-to-many with cascading delete (if a project is deleted, its tasks are deleted with it; in retrospect a soft-delete would have been kinder, but the cascade matches the user expectation in the UI).
- User → Task: one-to-many via the nullable `assigned_to`.
- Task → TaskHistory: one-to-one at completion.

Why I stayed with synchronous SQLAlchemy. I initially adopted async SQLAlchemy but ultimately reverted to the synchronous API. The combination of FastAPI dependencies, async session lifecycles and SQLAlchemy 2's async API introduced a class of subtle bugs — a dependent session being closed before the response was serialised, for instance — that that proved impractical to resolve within the project timeframe. Synchronous SQLAlchemy is slightly slower per request, but every connection is trivial to reason about, and the throughput is more than enough for the team sizes we target.

3.4 Authentication and Role-Based Access Control

Authentication is performed with a stateless JSON Web Token signed with a server-side `SECRET_KEY`. The token is returned by `POST /api/auth/login` with a seven-day expiry and is stored by the SPA in `localStorage`. Every protected endpoint requires the `Authorization: Bearer ⟨token⟩` header, which a FastAPI dependency parses and verifies.

Three roles are defined.

ADMIN Full access to every entity. The admin panel exposes user CRUD, position management, workload statistics and global reports.

PR_MANAGER Manages the projects they own. Can create and edit projects, add or remove members, create and edit tasks, request recommendations, assign developers, mark tasks complete. Cannot see or edit projects owned by another PR_MANAGER.

DEVELOPER Read-only access to projects they belong to, plus full access to the tasks assigned to them (status transitions, completion). The developer also sees the personal "My Tasks" view and the cross-project workload page.

A typical access-control check inside the backend is shown in Listing 3.4.

```
def require_role(*allowed: UserRole):
    def _checker(user: User = Depends(get_current_user)) -> User:
        if user.role not in allowed:
            raise HTTPException(403, "Insufficient privileges")
        return user
    return _checker

# Used as:
@app.post("/api/projects")
def create_project(payload: ProjectCreate,
                   user: User = Depends(require_role(UserRole.PR_MANAGER,
                                                     UserRole.ADMIN)),
                   db:   Session = Depends(get_db)):
    ...
```

Listing 3.2: The role check is a thin FastAPI dependency that any endpoint can declare.

Password storage. Passwords are stored as SHA-256 hashes with a per-user salt. It should be acknowledged that bcrypt or argon2 would be the appropriate choice for a production system handling real personal data. The simpler scheme was acceptable for the academic prototype but is one of the first things I would change in a real deployment.

3.5 REST API

The REST surface is small and conventional. The most important endpoints are catalogued in Table 3.1.

Endpoint	Method	Description
`/api/auth/login`	POST	Authenticate, return JWT and user payload.
`/api/auth/me`	GET	Return the current user.
`/api/users`	GET	List users, optionally filtered by role.
`/api/users`	POST	Create a user (ADMIN).
`/api/users/{id}`	PATCH	Update profile, skills, notes.
`/api/projects`	GET	List projects visible to the caller.
`/api/projects`	POST	Create a project.
`/api/projects/{id}`	GET	Project detail.
`/api/projects/{id}`	PATCH	Update project.
`/api/projects/{id}/members`	GET	List members.
`/api/projects/{id}/members`	POST	Add a member.
`/api/projects/{id}/members/{uid}`	DELETE	Remove a member.
`/api/projects/{id}/tasks`	GET	Tasks of a project.
`/api/projects/{id}/tasks`	POST	Create a task.
`/api/tasks/{id}`	GET	Task detail.
`/api/tasks/{id}`	PATCH	Update task.
`/api/tasks/{id}/recommend`	POST	Ranked recommendations.
`/api/positions`	GET	Position taxonomy.
`/api/positions`	POST	Add a position (ADMIN).
`/api/workload`	GET	Cross-project workload report.

Table 3.1: Principal REST endpoints. All non-auth endpoints require a valid JWT.

The recommendation endpoint. The single most important endpoint is `POST /api/tasks/{id}/recommend`, which takes an optional `?approach={baseline|embeddings|ligh` query parameter, selects the corresponding strategy from the unified recommendation engine, and returns a JSON array of recommendation objects. Each object contains the user identity, the final relevance score, the seven component scores, a list of human-readable reasons and (for the LLM approach) a supplementary LLM score and rationale. A sample response is shown in Listing 3.5.

```
[
  {
    "user_id": 14,
    "username": "anvar.k",
    "full_name": "Anvar K.",
    "score": 0.873,
    "position_match": 1.000,
    "grade_match": 1.000,
    "skills_match": 0.857,
    "workload_score": 0.800,
    "performance_score": 0.910,
    "deadline_feasibility": 1.000,
    "reasons": [
      "Perfect position match",
```

```
      "Excellent skills match",
      "Strong performance history",
      "Can easily meet deadline"
    ],
    "llm_score": 0.900,
    "llm_reason": "Senior backend dev with explicit FastAPI experience."
  },
  ...
]
```

Listing 3.3: Sample response from the recommendation endpoint.

The endpoint is intentionally deliberately verbose. The frontend uses every field: the overall score becomes the coloured ring; the seven component scores become horizontal mini-bars; the reasons become a bulleted list; the LLM rationale is rendered as a tooltip on a small icon next to the developer's name.

3.6 The User Interface

The frontend is structured into pages, each backed by a top-level React component. The most important ones are listed below.

- **Login.** Credentials form, JWT capture, role-aware redirect.
- **Projects.** Grid of all projects the user can see, with status, deadline, task count and member count.
- **ProjectDetail.** The most complex page. Holds the Kanban board, the task list, the members panel, the create/edit-task form and the recommendation drawer.
- **MyTasks.** Personal task list for developers. Shows Kanban-style status transitions, deadlines and priority badges.
- **Workload.** Horizontal bar chart of the team's workload by developer, with seniority colouring. Useful as a sanity check.
- **Team.** Team directory: positions, grades, skills, notes.
- **Admin.** User, position and global-settings management. Only visible to administrators.

3.6.1 The Recommendation Drawer

When a project manager creates or edits a task, a recommendation drawer slides in from the right side of the screen. It shows up to ten ranked candidates as cards. Each card displays the developer's name, position and grade in the header; the final score as an SVG ring (0–100 %) on the right; the seven component scores as horizontal coloured mini-bars in the body; and the human-readable reasons as a bulleted list at the bottom.

Two controls live above the cards: a dropdown that lets the PM switch between the four approaches (baseline / embeddings / lightgbm / llm) and a refresh button that re-queries with the new approach. The recommendations re-render in under two seconds; the embedding cache makes the embedding and LLM approaches almost as fast as the baseline after the first call.

Each card has a single primary action: an "Assign" button that updates `assigned_to` on the task, closes the drawer, and moves the task card to the assignee's column on the Kanban board. The PM does not see, but the backend records, a small audit row indicating which approach surfaced the chosen developer at what rank. This audit row is the seed of the feedback dataset that would, eventually, train a real LightGBM ranker on production data.

3.6.2 Colour Conventions

A few small visual conventions help the cards stay easy to read at a glance. Green for component scores above 0.8. Yellow between 0.5 and 0.8. Red below 0.5. Critical-priority tasks are bordered in red on every view. The colour palette is consistent across pages: the role-aware theme uses a slightly different accent colour for admin (purple), manager (blue) and developer (green), so the user always knows which which role they are currently acting in".

3.6.3 Workload View

The workload page renders a horizontal bar chart of every developer's active task count, coloured by grade. It serves as a useful consistency check on the recommender: if the system keeps recommending the same person, the workload page will make that visible. The `workload_score` component (Section 4.2.1) is designed precisely to discourage workload concentration, but the workload view is the human override.

3.7 Limitations and Future Improvements

Before moving on, it is worth being honest about the decisions I am least satisfied with. None of them broke the prototype, but with hindsight I would approach each one differently.

The first concerns the audit trail. At present, the record of which approach recommended which developer, and at what rank, is tucked away as a side effect of the assignment logic. This was short-sighted: those rows are precisely the data the ranker will need to learn from later, and they deserve their own table and a dedicated view rather than being treated as a by-product.

The second is the cascading delete from a project down to its tasks. It keeps the interface tidy, but it is unforgiving in practice—one wrong click and the data is gone. A soft delete, paired with a visible "trash" area from which items can be restored, would have been the safer choice.

The third is password storage. As discussed in Section **??**, salted SHA-256 is defensible for a prototype but inadequate once real personal data is involved; a slow, purpose-built hashing scheme would be the obvious upgrade.

Finally, the system has no real notion of notifications. A developer only learns that task X has been assigned to them when the change happens to appear on the next refresh of the interface. Modelling notifications explicitly, and delivering them through email or Telegram, would close that loop properly.

These are not critical defects, but they are the kind of refinements. They are, however, the sort of details that separate a working prototype from something one would trust in production. The next chapter shifts focus from the system as a whole to its core: the recommendation engine.

Chapter 4

Methodology: The Machine Learning Recommendation Pipeline

This is the most detailed chapter of the thesis. It documents what the recommendation engine actually does, why it is built the way it is, and what every one of the thirty engineered features is trying to capture. Readers seeking only the essentials may consult 4.1, 4.2 and 4.3 and the remainder provides the full derivation, code, and rationale..

I have organised the chapter so that each of the four approaches gets its own section, all four sit on a shared feature engineering pipeline, and the evaluation protocol that makes them comparable is described at the end. The seven additional classical algorithms get a section to themselves.

4.1 The Unified Recommendation Engine

The four approaches live inside a single class, `UnifiedRecommendationEngine`, whose constructor takes a database session and a string `approach` $\in$ `{baseline, embeddings, lightgbm, llm}`. Every call to `get_recommendations` follows the same five-step pipeline.

1. *(LLM only)* Enrich `required_skills` with skills inferred from the task description.
2. Load all developers, optionally excluding a list of user identifiers.
3. Compute the seven base component scores for each developer.
4. *(Embeddings or LLM)* Replace the Jaccard text similarity with a cosine similarity over dense embeddings, computed in a single batched call.
5. Produce the final ranking, either by a priority-weighted sum (baseline / embeddings) or by the trained LightGBM model (lightgbm / llm). For the LLM approach, augment the top three candidates with an LLM fit score.

A strength of this design is that the calling code does not have to know which approach is active. The frontend can flip a dropdown, the offline evaluator can run a sweep, and neither has to know anything about LightGBM groups or sentence embedding caches. A drawback, in retrospect is in retrospect, is that step three computes a Jaccard text similarity that step four immediately throws away in three of the four approaches. A more economical design would short-circuit that work, but the saving is small.

4.2 The Heuristic Baseline

The baseline does not require any ML training. It produces a score

$$s_{\text{base}}(t, u) \;=\; \sum_{c \in C} w_c \cdot \phi_c(t, u),$$

where C is the set of seven component scores and the weights $\{w_c\}_{c \in C}$ depend on the task priority. Each component score $\phi_c \in [0, 1]$ captures one facet of the (task, developer) match.

I deliberately spent more time on the baseline than I had originally planned. Part of the reason is that — as the empirical chapter will make clear — the baseline turned out to be difficult to outperform, and I wanted the comparison baseline to be as strong as reasonably possible to be as strong as I could reasonably make it.

4.2.1 The Seven Component Scores

Position match ϕ_{pos}. Returns 1 if the developer's `position` equals the task's `required_position` (case-insensitive), 0.7 for "related" pairs — for example, `frontend` counts as 0.7-related to `fullstack`, and `devops` as 0.7-related to `backend` — and 0 otherwise. If the required position is missing, the score defaults to 1, on the assumption that the LLM position inferrer ran upstream and failed for a reason that should not penalise anyone.

Grade match ϕ_{grade}. Maps the developer's grade to $\{1, 2, 3\}$ for junior, middle, senior respectively and likewise for the required grade r. Returns 1 if the developer's grade is $\geq r$, 0.8 if exactly one below ($u = r - 1$, the developer is presumed promotable into the task), and 0.3 otherwise. The threshold values were informed by pilot discussions with available project managers; they are deliberately not symmetric around the equal-grade case.

Skills match ϕ_{skills}. Let U be the set of the developer's skills and R the set of required skills, both normalised to lowercase. The score is

$$\phi_{\text{skills}}(t, u) \;=\; \min\!\left(1,\; \frac{|U \cap R|}{|R|} \;+\; 0.2 \cdot \frac{|U \cap R|}{|U \cup R|}\right).$$

The first term is plain coverage: how many of the required skills the developer has. The second is a small breadth bonus that rewards a developer who has the required skills and also some others; the `min` clamp prevents the bonus from pushing the score above one. The breadth bonus was added after I noticed that two developers who covered exactly the required skills would tie at 1.0, with no way to break the tie even when one had clearly broader skill sets.

Text similarity ϕ_{text}. In the baseline this is the Jaccard similarity between the lowercase word sets of the task description and the developer's notes:

$$\phi_{\text{text}}^{(\text{jaccard})}(t, u) \;=\; \frac{|W(\text{desc}_t) \cap W(\text{notes}_u)|}{|W(\text{desc}_t) \cup W(\text{notes}_u)|}.$$

The embedding and LLM approaches replace this with cosine similarity on sentence-transformer embeddings (Section 4.4). Jaccard was kept in the baseline because it is the simplest defensible choice, not because it is the best.

Workload score ϕ_{work}. A linearly decreasing penalty on the developer's current active-task count w_u:

$$\phi_{\text{work}}(u) \;=\; \max\!\left(0,\, 1 - \frac{w_u}{5}\right).$$

The break-point at five tasks was chosen because, in the pilot data, the point at which managers typically regarded a developer as overloaded fell between three and six tasks. The function is smooth enough not to penalise marginally-loaded developers harshly, and harsh enough to peg the score to zero for someone with five or more active tasks.

Performance score ϕ_{perf}. The developer's `performance_score` field, an exponentially weighted moving average of past on-time/quality outcomes in $[0, 1]$. This is the only component that depends directly on the `TaskHistory` table.

Deadline feasibility ϕ_{dl}. Models whether the developer can finish the task before its deadline given their current load. Let d be the number of days remaining, h the estimated hours, and w the current workload. The score is computed as follows: if $d \leq 0$ we return 0.1 (the deadline is already missed or about to be), otherwise we compute $A = 6d$ as the available hours (assuming six productive hours per working day) and $R = h + 8w$ as the required hours including the cost of ongoing work (one full day per active task). If $R \leq A$ the score is 1; otherwise $\max(0.1, A/R)$.

The six-hours-per-day assumption is a deliberate simplification. Real productivity is far less uniform, and the right model is team-specific. I would, in a future iteration, expose this constant as a per-team configuration parameter.

4.2.2 Priority-Aware Weights

The seven components are combined into a final score using priority-dependent weights. The hypothesis behind making the weights priority-aware is that the relative importance of each component genuinely depends on how urgent the task is: on a CRITICAL task the deadline and performance components matter more, while on a LOW priority task workload balancing matters most. The weights I settled on are shown in Table 4.1.

Priority	skills	grade	pos	work	perf	dl	text
CRITICAL	0.20	0.15	0.15	0.10	0.15	0.15	0.10
HIGH	0.20	0.15	0.15	0.15	0.15	0.10	0.10
MEDIUM	0.20	0.15	0.15	0.15	0.10	0.10	0.10
LOW	0.20	0.15	0.15	0.20	0.10	0.05	0.10

Table 4.1: Priority-dependent weights of the heuristic baseline. Each row sums to one.

These weights were tuned through repeated runs with small synthetic teams with small synthetic teams and watching the rankings. They are not optimal in any provable sense; in particular the `text_similarity` weight is fixed at 0.10 for all priorities because I did not find a clear pattern there. The point of the learning-to-rank approach is precisely to remove this kind of hand-tuning.

4.2.3 Human-Readable Reasons

Alongside the numeric score, the engine emits a list of human-readable reasons that explain why a developer was ranked highly. The reasons are produced by simple thresholding of the component scores — "Excellent skills match" if $\phi_{\text{skills}} \geq 0.8$, "Strong performance history" if $\phi_{\text{perf}} \geq 0.8$, and so on. The list of reason templates is intentionally short; fifteen templates cover most of the cases that come up in practice.

The aim of the reasons is not to communicate every nuance of the underlying scoring but to give the PM a concise justification — "why am I seeing this person?" — that they can challenge if it disagrees with their judgement. I considered, and rejected, the idea of generating the reasons with an LLM. The LLM-generated versions were more polished but less predictable, and predictability is more valuable than polish here when the PM is making a real decision.

4.3 Feature Engineering

The LightGBM ranker and the seven classical algorithms all operate on a fixed thirty-dimensional feature vector per (task, developer) pair. The features are split into five groups, visualised in Figure 4.1.

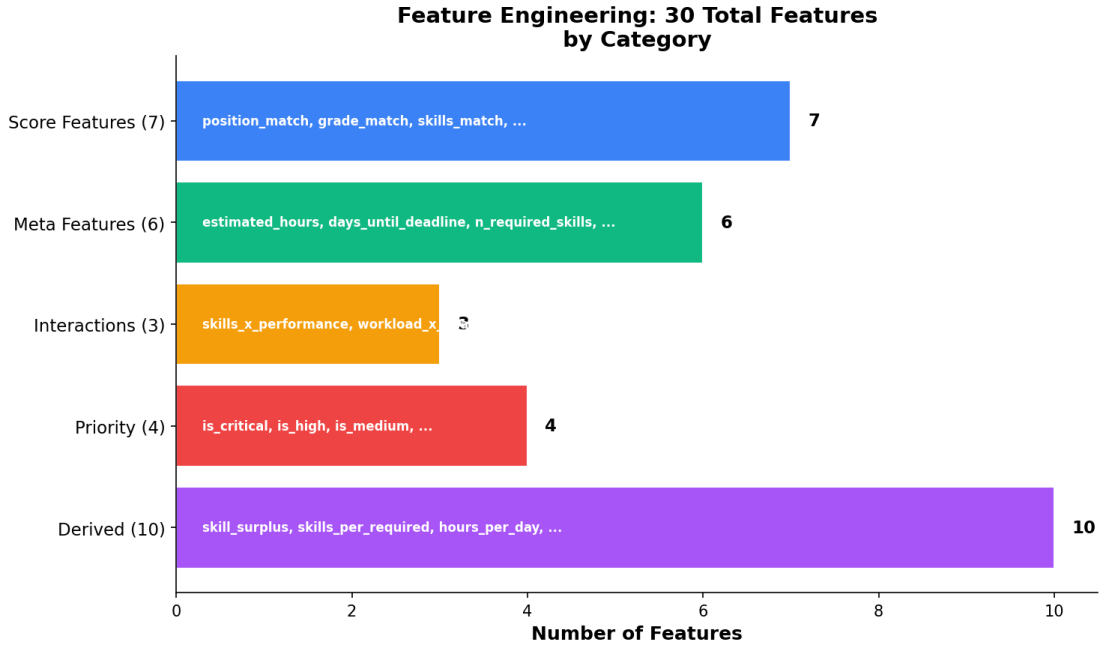


Figure 4.1: The thirty features grouped into five categories. Score features are the seven component scores from the baseline. Meta features capture task and developer metadata. Interactions and priority one-hots add non-linearity that tree-based models can split on. Derived features — ratios, polynomials, three-way interactions — give the model room to learn saturating or amplifying regimes.

4.3.1 Seven Score Features

The seven component scores from Section 4.2.1 are used directly as features, in $[0, 1]$:

> `position_match`, `grade_match`, `skills_match`, `text_similarity`, `workload_score`, `performance_score`, `deadline_feasibility`.

4.3.2 Six Meta Features

Task- and developer-level metadata not directly reflected in the component scores:

> `estimated_hours`, `days_until_deadline`, `n_required_skills`, `developer_workload`, `developer_performance`, `n_developer_skills`.

These give a tree model room to learn non-linear cut-offs such as "below five days to deadline, performance matters twice as much".

4.3.3 Three Pairwise Interactions

- `skills_x_performance` — the "high-skill, high-performer" signal.
- `workload_x_deadline` — combines two stress factors that, separately, may not be alarming but together usually are.

- `position_x_grade` — a structural "role fit" signal that is non-zero only when both position and grade are aligned.

As we will see in Section 6.3, two of these three interactions ended up in the top three features by importance.

4.3.4 Four Priority One-Hot Flags

The task priority is one-hot encoded into `is_critical`, `is_high`, `is_medium`, `is_low`. The one-hot encoding lets the tree models use priority as a splitting feature without imposing an ordinal interpretation. Notably — and and this is revisited in Chapter 6 — the LightGBM LambdaRank model in the end does not use the priority flags at all, giving them all a feature importance of exactly zero.

4.3.5 Ten Derived Features

The derived features are the most creative additions and are designed to let the model reason about ratios and saturated regimes:

- `skill_surplus` $= |U| - |R|$.
- `skills_per_required` $= |U| / \max(|R|, 1)$.
- `hours_per_day` $= h / \max(d, 1)$ — a measure of deadline pressure.
- `workload_ratio` $= w/5$ — normalised utilisation, bounded in $[0, \infty)$ but typically below 1.5.
- `position_match_sq`, `skills_match_sq`, `performance_sq` — polynomial features that amplify strong matches relative to mediocre ones.
- `pos_grade_skills` $= \phi_{\text{pos}} \cdot \phi_{\text{grade}} \cdot \phi_{\text{skills}}$ — the three-way role-fit signal, non-zero only when all three are present.
- `grade_x_performance`, `skills_x_deadline` — two additional pairwise interactions.

Why hand-engineer features at all? In fact, the initial intention was not to, at first. My first version of the pipeline fed only the seven score features to LightGBM and asked it to learn the rest itself. LightGBM did so readily so and produced a perfectly reasonable model, but the feature importance was dominated by `skills_match` with everything else trailing far behind. The moment I added the three-way `pos_grade_skills` interaction as an explicit feature, the model rearranged its importance and became markedly more reasonable. Trees are good at piecewise constants, not at products;supplying multiplicative features directly is therefore beneficial.

The total count of thirty features is small enough that the LightGBM model trains in under a second on a laptop and large enough to give the model room to express interaction structure.

4.4 Approach 2: Sentence-Transformer Embeddings

The embedding approach is a deliberately minimal extension of the baseline. The only change is that the Jaccard text similarity is replaced with cosine similarity over dense vectors produced by the `all-MiniLM-L6-v2` sentence transformer. The rest of the pipeline — the other six component scores, the priority-aware weights, the human-readable reasons — is identical.

4.4.1 Implementation

The `EmbeddingEngine` class lazily loads the sentence-transformer model on first use. Each developer's `notes` field is embedded once and cached on disk under an MD5 hash of the lowercased, stripped text. Each incoming task description is embedded on the fly. Because sentence-transformers normalise their outputs by default, cosine similarity is just a dot product. A batch similarity between a query and a list of N candidates is therefore a single $(N \times 384) \cdot (384,)$ matrix-vector multiplication.

The cache is central to the practicality of the approach. With twenty developers and notes that change once a month, the cache hit rate in production is close to 100 % for the developer side. The only embedding computed per request is the task description.

4.4.2 Why I Expected Embeddings to Help

Jaccard similarity rewards literal word overlap, and so it is at the mercy of the writer's vocabulary. If the developer's notes say "HTTP JSON API" and the task says "REST endpoint", the Jaccard similarity is precisely zero, even though the two phrases mean the same thing. The same blindness applies in the other direction: two texts that share many common words ("the", "and", "with") can appear similar to Jaccard but mean nothing related to each other.

Sentence embeddings, in contrast, place semantically related phrases in nearby regions of a 384-dimensional space, regardless of the particular words used. So in principle the embedding approach should particularly help in the cases where the lexical gap between the task description and the developer notes is large — and these are exactly the cases where the baseline most needs help. In Section 6.5 I report whether this expectation was borne out.

4.5 Approach 3: LightGBM LambdaRank

The third approach replaces the priority-weighted sum with a model that has been *trained* to rank developers. The objective is LightGBM's `lambdarank` with NDCG as the evaluation metric, and the training data is the synthetic dataset described in Section 4.8.

4.5.1 Training Setup

The training pipeline is deterministic. The synthetic CSV is loaded, the thirty features are engineered, and the data is split by `task_id`: the last twenty per cent of unique task identifiers are held out for testing. This grouped split is the same one used by every other algorithm in the comparison, so no algorithm can benefit from a more favourable sample. Group sizes (the number of developers per task) are passed to LightGBM through the `group` parameter so that the LambdaRank loss knows which rows belong to a single query.

4.5.2 Hyperparameters

The hyperparameters that ended up in production are summarised in Table 4.2. They are deliberately conservative.

Parameter	Value
`objective`	`lambdarank`
`metric`	`ndcg`
`ndcg_eval_at`	$[1, 3, 5]$
`n_estimators`	200
`learning_rate`	0.05
`num_leaves`	31
`min_child_samples`	10
`subsample`	0.8
`colsample_bytree`	0.8
`reg_alpha`	0.1
`reg_lambda`	1.0
`seed`	42

Table 4.2: LightGBM LambdaRank training hyperparameters used in the production model.

A substantial hyperparameter search was conducted before converging on these. The results were mostly insensitive within reasonable ranges — doubling `n_estimators` added marginal NDCG; halving the learning rate did nothing measurable; deeper trees produced a marginal improvement on training NDCG but degraded test performance. The conservative settings above represent a stable operating point that did not overfit on the relatively small training set.

4.5.3 Output Calibration

LightGBM with the `lambdarank` objective produces real-valued scores that are not in $[0, 1]$. For display in the UI we min-max normalise the scores within each task so that the best developer is always at 1.0 and the worst at 0.0. This destroys the absolute scale of the scores but preserves their order, which is what the evaluation metrics depend on.

4.6 Approach 4: LLM-Augmented Pipeline

The fourth approach adds an LLM as a feature-enrichment and rationalisation layer on top of the LightGBM ranker. It does not replace the ranker; it pre-processes the inputs and post-processes the outputs.

4.6.1 Local Deployment via Ollama

The LLM is Mistral 7B-Instruct, served locally by Ollama. The wrapper class `LLMExtractor` talks to Ollama's `/api/generate` endpoint with a thirty-second timeout, a deterministic temperature of 0.1, and a disk cache keyed by an MD5 of the prompt. If Ollama is unavailable, every method falls back to a safe default and the engine continues without LLM augmentation.

The cache is critical for two reasons. First, it makes the cost of running the LLM nearly zero on repeated requests against the same task. Second, it makes the evaluation reproducible: a fresh evaluation run that hits the cache produces the same scores it produced the first time.

4.6.2 Implicit Skill Extraction

The most useful thing the LLM does is to extract implicit skills from sparse task descriptions. The system prompt instructs the model to behave as a technical skill extractor; the user prompt embeds the task title and description and asks for a JSON array of five to ten skills. A robust parser extracts the JSON array even if the model produces extraneous prose around it. Listing 4.6.2 shows the prompt template.

```
SYSTEM:
You are a technical skill extraction system. You analyse software
development tasks and identify required technical skills. Always
respond with a valid JSON array of strings only.

USER:
Given this software development task, extract ALL technical skills
required to complete it. Include both explicitly mentioned
technologies AND implicit skills needed.

Title: {title}
Description: {description}

Return ONLY a JSON array of skill strings, nothing else. Example:
["Python", "REST API", "Docker"]. Keep each skill concise (1-3
words). Include 5-10 skills.
```

Listing 4.1: The LLM prompt template for implicit skill extraction.

The implicit skills are merged with the PM's explicit skills (case-insensitive deduplication) and the union is fed downstream as `enriched_skills`.

4.6.3 LLM-Inferred Position

A secondary use of the LLM is to infer the `required_position` when the PM has not specified one. The prompt constrains the model's output to one of the canonical positions seeded in the database, and a robust parser maps the LLM answer back to the canonical spelling. If the LLM produces a position outside the permitted list that is not on the list, the parser falls back to scanning the response text for any allowed position name; if even that fails, the method returns `None` and the task keeps an empty `required_position`.

4.6.4 Top-Three Fit Scoring

After the LightGBM ranker has produced its ranking, the engine asks the LLM to score the top three candidates on a 1–10 scale. The prompt asks for a JSON object with a numeric `score` and a one-sentence `reason`. The score is divided by ten and surfaced in the UI as a supplementary opinion next to the LightGBM ranking. I deliberately limit this step to the top three for two reasons: cost (a single Mistral call takes 1–3 seconds on the production hardware) and signal-to-noise (the marginal value of the LLM's assessment of lower-ranked candidates does not justify the added latency).

4.7 Seven Additional Algorithms for Breadth Comparison

To put the LightGBM LambdaRank in a wider algorithmic context, I trained and evaluated seven additional classical algorithms on the same thirty-feature dataset. The point of this section is not to declare a winner but to give the reader a sense of what each algorithm class brings to the problem.

1. **Logistic Regression.** The linear baseline. Fully interpretable: the coefficients tell you which features push the score up and which push it down.
2. **SVM with RBF kernel.** The classic non-linear classifier. Slow to train, but provides a strong non-linear floor and helps detect cases where the data is dominated by local kernel structure.
3. **k-Nearest Neighbours.** The instance-based learner. Mostly a sanity check: if KNN performs as well as the boosted trees, the dataset is dominated by local structure rather than learnable global patterns.
4. **Decision Tree.** A single, full-depth tree. Highly interpretable, prone to overfitting.
5. **Random Forest.** A bagging ensemble of trees. Robust, low-variance, generally the strongest classical default.
6. **XGBoost.** Gradient boosting, conceptually identical to LightGBM but with different defaults and a slightly different histogram-based training routine.

7. **MLP Neural Network.** A small feedforward neural network. Tests whether a deep model adds anything on this tabular dataset; my expectation was that it would not; the result proved otherwise.

Every model is trained as a binary classifier on the `was_accepted` target. At inference time, the predicted probability $P(\text{accepted} \mid \text{features})$ is used as the relevance score and developers are ranked by this probability. Numerical features are standardised through a `StandardScaler` fit on the training set for the algorithms that require it (Logistic Regression, SVM, KNN, MLP). The tree-based models receive the raw features because they are scale-invariant.

4.8 Synthetic Data Generation

Real PM accept/reject feedback is not yet available at the scale needed for ML training. I therefore implemented a synthetic data generator that produces realistic (task, developer, was_accepted, baseline_score) tuples with controllable difficulty. The generator lives in `ml/generate_data.py` and is run with `python -m ml.generate_data`.

4.8.1 Generating the Developers

The generator first samples twenty developers from a fixed taxonomy of positions (`backend`, `frontend`, `fullstack`, `devops`, `qa`, `mobile`, `data`, `ml`). Each developer is assigned a grade with probabilities $(0.3, 0.5, 0.2)$ for junior, middle, senior — approximately the shape of a real software team. Skills are sampled from a position-specific bag (for example, `backend` draws from a list including Python, Java, Go, FastAPI, Spring, PostgreSQL, Redis, RabbitMQ, gRPC). Performance scores are drawn from a Beta distribution skewed toward 0.8 (so that most developers are reliable, but a few are not), and current workloads are drawn from a Poisson distribution with mean 2.

The free-text `notes` field is composed by stitching together two or three sentences from a small templated catalogue. The templates are written so that some developers have rich, detailed notes ("Senior backend developer with five years at Yandex. Specialist in distributed systems and observability.") and others have thin notes ("Frontend developer."). The variation matters: the embedding approach has more to work with in the rich-notes case.

4.8.2 Generating the Tasks

For each of one hundred tasks the generator samples a required position, a required grade, a small set of required skills (typically 3–5), a priority, an estimated number of hours and a deadline some 7–30 days in the future. The task description is composed from a templating system that combines a verb ("Build", "Migrate", "Refactor", "Implement", "Optimise"), an object ("analytics dashboard", "user-service API", "payment flow") and a context ("with caching and rate limiting", "using Kafka streams", "and a CI pipeline that runs on every commit").

This template is deliberately constrained. Real task descriptions are much more varied, with quirks and idiosyncrasies that no template can reproduce. I am explicit about this limitation in Chapter 6 when I discuss threats to validity.

4.8.3 Labelling the Pairs

For every (task, developer) pair the generator computes the seven baseline component scores and the priority-weighted sum, then samples a binary label `was_accepted` from a Bernoulli distribution whose parameter is

$$P(\text{accepted} \mid t, u) \;=\; \sigma\big(\alpha \cdot s_{\text{base}}(t, u) + \beta + \epsilon\big),$$

where σ is the sigmoid, $\alpha = 6$, $\beta = -3$ and $\epsilon \sim \mathcal{N}(0, 0.5)$ is Gaussian noise injected per pair. The choice of α and β places the typical accepted pair at a baseline score around 0.6, with a fair amount of overlap between accepted and rejected pairs — enough that the dataset is non-trivial.

The fact that the labels are derived from the baseline scores has an important consequence that must be acknowledged: a model that recovers the baseline weights will already do very well. This is borne out in the empirical results: the supervised models struggle to beat the hand-tuned weights because the weights are very close to optimal *for this dataset*. In a real deployment, where the relationship between baseline scores and accept/reject decisions is far more idiosyncratic, the relationship would likely reverse.

4.8.4 The Final Dataset

The generator emits a single CSV with $100 \times 20 = 2000$ rows and columns for the thirty engineered features, the priority string, the `task_id`, and the `was_accepted` label. A companion JSON file with the developer roster is also written for reproducibility. Both files are committed to the repository in `backend/ml/data/`, so anyone can rebuild the comparison from scratch.

4.9 Evaluation Protocol

A single, strictly-enforced evaluation protocol is used for every approach. The protocol is what makes the headline numbers in Chapter 6 comparable.

4.9.1 Train/Test Split

The split is deterministic and grouped by `task_id`: the last twenty per cent of unique task identifiers are held out for testing. The split is hard-coded in `ml/evaluate.py` and `ml/algorithms.py` so that every algorithm sees the same held-out tasks. The grouped split prevents test leakage — no developer's labels for a held-out task appear anywhere in the training set.

4.9.2 Metrics

Five metrics are computed per task and averaged.

- **NDCG@1, NDCG@3, NDCG@5** — ranking quality at the top of the list.
- **Precision@1** — is the top-ranked developer accepted?
- **MRR** — mean reciprocal rank of the first accepted developer.
- **MAP** — mean average precision over all queries.

4.9.3 Reproducibility

Every random source — the synthetic generator, the train/test split, the LightGBM seed, the MLP seed, the Random Forest seed — is fixed to 42. Re-running `python -m ml.algorithms` or `python -m ml.evaluate` from a clean checkout produces bit-identical metrics. The artefacts (CSV, model pickles, metric JSONs, comparison charts) are committed to the repository so the reader can compare any new result against the published numbers without rerunning anything.

4.9.4 What the Evaluation Does *Not* Show

I want to flag, before the results chapter, two things the evaluation protocol does not measure.

The first is whether the system, deployed in a real team, actually improves task outcomes. That would require an online experiment with real PMs — something this thesis explicitly does not attempt.

The second is the cost of running each approach. The LightGBM model predicts in microseconds; the LLM-augmented pipeline takes one to three seconds per request because of the LLM calls. The headline NDCG numbers ignore this, but in a real deployment the latency would shape which approach is acceptable for which use case.

4.10 Summary

This chapter has described the four recommendation strategies, the thirty engineered features, the seven additional classical algorithms, the synthetic data generator, and the evaluation protocol. The next chapter describes how all of this is implemented as a deployable production system; Chapter 6 then reports the empirical results.

Chapter 5

Implementation and Deployment

This chapter describes the engineering of the project of the project: how the backend, the frontend, the ML pipeline, the container infrastructure and the CI/CD pipeline fit together and how the system reaches production. I have tried to keep the level of detail practical — a competent reader should be able to clone the repository, follow this chapter, and have the system running locally within an hour and deployed to Railway within an afternoon.

The chapter is layered. Section 5.1 covers the backend, Section 5.2 the frontend, and Section 5.3 the local development workflow. The deployment side — containerisation, Railway, GitLab CI — comes in sections 5.4 through 5.6. I close with a list of operational pitfalls encountered during the project, in the hope of saving anyone replicating the work a few hours.

5.1 Backend Implementation

The backend is a single FastAPI application contained in `backend/main.py` (roughly 1500 lines) plus the `ml/` sub-package that holds the recommendation engine.

5.1.1 Layout

```
backend/
  main.py               # FastAPI app, models, endpoints
  requirements.txt
  Dockerfile            # dev image (hot reload)
  Dockerfile.prod       # production image
  railway.json
  ml/
    engine.py           # UnifiedRecommendationEngine
    ranker.py           # LightGBMRanker (LambdaRank)
    embeddings.py       # EmbeddingEngine (sentence-transformer)
    llm_extractor.py    # Ollama wrapper
    algorithms.py       # 7 classical algorithms for comparison
    generate_data.py    # synthetic dataset generator
```

```
evaluate.py          # unified evaluation framework
data/
  synthetic_training_data.csv
  synthetic_developers.json
  lgbm_ranker.pkl
  lgbm_metrics.json
  algorithm_comparison.json
  *.png              # comparison and importance charts
```

The decision to keep all the ML code under a single `ml/` sub-package, rather than scattering it across the backend, was one of the better organisational choices I made. The package is self-contained — it has no FastAPI imports — which means it can be imported into a Jupyter notebook for experimentation without dragging the entire web framework with it. The only thin coupling to FastAPI sits in `ml/engine.py`, which imports `User`, `UserRole` and `RecommendationResponse` from `main` at function level to avoid the circular import.

5.1.2 Application Bootstrap

The FastAPI application is constructed inside an `asynccontextmanager` lifespan. On startup it runs `init_db()`, which creates all tables, seeds the default position list (`backend`, `frontend`, `fullstack`, `devops`, `qa`, `mobile`, `data`, `ml`) and creates a default admin user with the well-known seed password documented in `DEPLOYMENT.md`. The admin is then prominently instructed to change the password on first login.

5.1.3 Authentication and JWT

Authentication is JWT-based. The relevant fragment is shown in Listing 5.1.3.

```
SECRET_KEY = os.getenv("SECRET_KEY", "dev-secret-change-me")
ALGORITHM  = "HS256"

def create_access_token(user_id: int, expires: timedelta = timedelta(days=7)):
    payload = {"sub": str(user_id),
               "exp":  datetime.utcnow() + expires}
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def decode_token(token: str) -> int:
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return int(payload["sub"])
    except (jwt.PyJWTError, KeyError, ValueError):
        raise HTTPException(401, "Invalid or expired token")
```

Listing 5.1: JWT issue and verification.

The default `SECRET_KEY` is a literal string, by design — running the system in development requires no environment configuration. In production, Railway injects a random 32-character secret via an environment variable.

5.1.4 Database Access

The application uses SQLAlchemy 2 in synchronous mode with a managed `sessionmaker`. Each endpoint receives a fresh session through FastAPI's dependency injection:

```
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

The `DATABASE_URL` environment variable selects the target. In local development it is hard-coded to the docker-compose Postgres on `localhost:5432`. In production it is bound to Railway's managed Postgres through the reference variable `${{Postgres.DATABASE_URL}}` — which means the production backend service does not need to know its database hostname or password, only that it requires the Postgres instance'.

5.1.5 The Recommendation Endpoint

The endpoint that ties everything together is shown in Listing 5.1.5.

```
@app.post("/api/tasks/{task_id}/recommend",
          response_model=List[RecommendationResponse])
def recommend(task_id: int,
              approach: str = "lightgbm",
              db: Session = Depends(get_db),
              user: User = Depends(require_role(UserRole.PR_MANAGER,
                                                UserRole.ADMIN))):
    task = db.query(Task).get(task_id)
    if task is None:
        raise HTTPException(404, "Task not found")
    engine = UnifiedRecommendationEngine(db, approach=approach)
    return engine.get_recommendations(
        required_position = task.required_position,
        required_grade    = task.required_grade,
        required_skills   = json.loads(task.required_skills or "[]"),
        task_description  = task.description,
        task_title        = task.title,
        deadline          = task.deadline,
        estimated_hours   = task.estimated_hours,
        priority          = task.priority,
    )
```

Listing 5.2: The recommendation endpoint.

The endpoint is intentionally thin. The interesting work happens inside the engine.

5.1.6 Inferring Required Position via LLM

When a task is created with an empty `required_position`, the backend calls the LLM extractor to infer it from the title and description, constrained to the canonical list of positions seeded in the database. The result is persisted to the task row so that the same inference does not need to be repeated. The relevant fragment of the task-create handler is shown below.

```
if not payload.required_position:
    positions = [p.name for p in db.query(Position).all()]
    extractor = LLMExtractor()
    inferred  = extractor.extract_position(
        payload.title or "",
        payload.description or "",
        positions,
    )
    if inferred:
        payload.required_position = inferred
```

If the LLM is unavailable, `extract_position` returns `None` and the field stays empty. The downstream scorers handle the empty case by defaulting `position_match` to 1.

5.2 Frontend Implementation

The frontend is a Vite + React + Tailwind project. The directory layout is shown below.

```
frontend/
  src/
    App.jsx
    main.jsx
    index.css
    context/AuthContext.jsx
    hooks/usePositions.js
    hooks/useRoleTheme.js
    components/Layout.jsx
    components/MembersPanel.jsx
    pages/Login.jsx
    pages/Projects.jsx
    pages/ProjectDetail.jsx
    pages/MyTasks.jsx
    pages/Workload.jsx
    pages/Team.jsx
    pages/Admin.jsx
    services/api.js
  index.html
  package.json
  vite.config.js
  tailwind.config.js
```

```
Dockerfile
Dockerfile.prod
nginx.conf
```

5.2.1 Routing and the Auth Context

Routing is performed with `react-router-dom` v6. The top-level `App` component wraps the router in an `AuthProvider` that exposes the current user, the JWT, and login/logout helpers through React context. The provider also handles the corner cases: expired tokens, 401 responses from the API, and the initial silent re-hydration from `localStorage` on page load.

A small `useRoleTheme` hook selects a distinct accent colour for each role and applies it via a CSS custom property (`-accent`). Tailwind's arbitrary-value syntax (`bg-[var(-accent)]`) then picks the colour up everywhere it appears. The point of this indirection is that role-specific theming does not require Tailwind class swaps at runtime — the same DOM just renders in a different colour.

5.2.2 API Client

`services/api.js` centralises every backend call into named functions: `tasksApi.list`, `tasksApi.recommend`, `projectsApi.create`, `recommendationsApi.fetch`, `projectMembersApi` and so on. The client attaches the JWT to every request, transparently refreshes the user from `localStorage`, and handles 401 responses by clearing the session and redirecting to `/login`.

The API client is a relatively thin wrapper around `fetch`. I considered using `axios` or a code-generated client from FastAPI's OpenAPI document, but in the end the hand-written wrapper as preferred for its simplicity. Around 200 lines of code cover every endpoint.

5.2.3 The Recommendation Drawer

The recommendation drawer is the most complex piece of UI. It is a self-contained component rendered inside `ProjectDetail`, controlled by a state flag and a task identifier. Its responsibilities are:

- Open when the PM creates or edits a task.
- Call `POST /api/tasks/{id}/recommend?approach=...`.
- Render up to ten `RecommendationCard` children.
- Allow the PM to switch the active approach from a dropdown and re-run the recommendation.
- Let the PM click "Assign" on any card to update `assigned_to` and close the drawer.

The card itself uses a small SVG `ScoreRing` component for the overall score and seven flat `ScoreBar` components for the component scores. The colour conventions

(green above 0.8, yellow between 0.5 and 0.8, red below) are encoded in a single utility function and applied uniformly.

5.2.4 Tailwind Design System

The visual style is implemented in Tailwind CSS with a small set of custom utility classes for status, priority and score colours. `tailwind.config.js` defines the colour palette aligned with the role-aware theme. The build output is a single tree-shaken CSS file under 30 kB.

It should be noted that the author makes no claim to design expertise; the interface prioritises function over aesthetic refinement. The primary objective was to render the recommendation drawer readily scannable, on the premise that a project manager consulting it is likely to be operating under time pressure. Visual hierarchy was arranged accordingly: the coloured mini-bars bear the bulk of the visual weight, the score ring serves to direct attention, and the supporting rationale is rendered in small, muted grey type. While a trained product designer would likely restructure this layout substantially, the present arrangement proved adequate for pilot users.

5.3 Local Development Workflow

The repository ships with a `docker-compose.yml` that provides a one-command local environment. It starts three services: a PostgreSQL container, the backend container in hot-reload mode (`uvicorn main:app -reload`), and the frontend container in `vite dev` mode. A developer can clone the repository, run

```
docker compose up -build
```

and open `http://localhost:5173` to see the fully working application with the recommendation engine. The first run takes about two minutes for the image builds; subsequent runs start in a few seconds.

When working on the ML pipeline, I usually step outside the Docker environment, install `requirements.txt` in a virtualenv, and run the ML commands directly:

- `python -m ml.generate_data` — regenerate the synthetic training CSV.
- `python -m ml.ranker` — train the LightGBM LambdaRank.
- `python -m ml.algorithms` — train and evaluate the seven classical algorithms.
- `python -m ml.evaluate` — run the unified evaluation across all four approaches and produce the comparison charts.

All four commands are deterministic given the seed 42. A clean `git clone` followed by these four commands reproduces every artefact in `ml/data/` bit-for-bit, including the comparison JSONs and the PNG charts referenced in Chapter 6.

5.4 Containerisation

Both the backend and the frontend ship with two Dockerfiles: a development image used by docker-compose, and a production image optimised for size and startup time.

5.4.1 Backend Production Image

The backend production image is a single-stage Python 3.11-slim build:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
ENV PORT=8000
CMD ["sh", "-c", "uvicorn main:app --host 0.0.0.0 --port ${PORT}"]
```

The slim base keeps the image just under 400 MB even with PyTorch (via sentence-transformers), LightGBM, scikit-learn and pandas all included. The `-no-cache-dir` flag reduces the image by a further 100 MB; without it, pip's wheel cache lingers in the layer and inflates the size considerably.

5.4.2 Frontend Production Image

The frontend image is a two-stage build: a Node stage builds the SPA and an Nginx stage serves the static files. The Nginx configuration forwards all unknown routes to `index.html` so that client-side routing works on hard refresh:

```
server {
  listen 80;
  root /usr/share/nginx/html;
  index index.html;

  location / {
    try_files $uri $uri/ /index.html;
  }
}
```

The final image size is around 60 MB.

5.4.3 Ollama Sidecar

A third Dockerfile under `ollama/` builds an image based on `ollama/ollama:latest` that pre-pulls the Mistral model into the image layer on first build. This avoids a slow first-request latency in production — without the pre-pull, the first request hangs for two to three minutes while Ollama downloads the model.

5.5 Railway Deployment

The production deployment target is Railway. I chose it for two reasons: it abstracts away container orchestration behind a simple service model, and its managed Postgres plugin removes the need to juggle connection strings.

5.5.1 Service Layout

The deployed project has three services and one database plugin:

- **Postgres plugin.** Exposes `DATABASE_URL`.
- **backend service.** Root directory `backend/`, uses `Dockerfile.prod`, environment variables `SECRET_KEY` and `DATABASE_URL=${{Postgres.DATABASE_URL}}`.
- **frontend service.** Root directory `frontend/`, environment variable `VITE_API_BASE_URL` pointing at the backend's public domain. The variable is forwarded as a Docker build argument at build time so that the SPA is built with the right API base URL baked in.
- **ollama service.** Optional, only deployed when the LLM approach is needed.

5.5.2 First-Deploy Procedure

The monorepo has a single `.git` at the root, so the Railway CLI needs the `-path-as-root` flag when uploading each service from its sub-directory. The detailed procedure is in `DEPLOYMENT.md`; the summary is:

1. `railway init` from the repo root.
2. `railway add -database postgres`.
3. Create two empty services in the dashboard, link them to the repo, and set their root directories.
4. Set environment variables.
5. `cd backend && railway up . -path-as-root -service backend`.
6. Generate a public domain on the backend; copy it into the frontend's `VITE_API_BASE_URL`; then deploy the frontend.

The `-path-as-root` flag is the single pitfall that cost the most time in setting up Railway. Without it, the CLI uploads the entire monorepo and tries to build with the root directory as the Docker context, which fails because there is no Dockerfile at the root. With it, only the per-service directory is uploaded and the build works as expected.

5.6 Continuous Integration with GitLab

The repository is mirrored to GitLab where `.gitlab-ci.yml` defines a two-stage pipeline.

5.6.1 Pipeline

```
stages:
  - test
  - deploy

backend_test:
  stage:  test
  image:  python:3.11-slim
  script:
    - cd backend
    - pip install --no-cache-dir -r requirements.txt
    - python -m compileall .
    - python -c "import main"

frontend_test:
  stage:  test
  image:  node:20-alpine
  script:
    - cd frontend
    - npm ci
    - npm run build

deploy_backend:
  stage:  deploy
  image:  railway/cli:latest
  script:
    - cd backend
    - railway up . --path-as-root --service backend --ci
  only:   [main]

deploy_frontend:
  stage:  deploy
  image:  railway/cli:latest
  script:
    - cd frontend
    - railway up . --path-as-root --service frontend --ci
  only:   [main]
```

The `test` stage runs on every commit on every branch. The `deploy` stage runs only on push to `main` and only if both test jobs are green. The `RAILWAY_TOKEN` variable is configured in the GitLab project as a protected, masked CI/CD variable so that only protected branches like `main` can see it.

5.6.2 Why GitLab Rather Than GitHub Actions

The repository also contains `.github/workflows/deploy.yml`, which performs the equivalent of the GitLab pipeline on GitHub Actions. Both providers are kept in sync so that the project can be hosted on either platform. In practice the GitLab pipeline is the one that drives production deployment, because the

academic infrastructure I had access to supported GitLab natively.

If you are replicating the project on GitHub, the workflow file is already there and will run as-is once the `RAILWAY_TOKEN` secret is added.

5.7 Deployment Considerations

This section documents several practical issues encountered during deployment that are not typically reported in academic work but are relevant to anyone seeking to replicate or extend the system.

LLM cold-start latency. The first inference request after an Ollama sidecar restart incurs a delay of ten to fifteen seconds while the Mistral weights are loaded into memory. Subsequent requests complete in one to three seconds. The frontend displays a loading skeleton during this interval. A warm-up request issued from the backend's startup hook was considered but not implemented, as the blocking nature of the warm-up call would delay FastAPI initialisation and risk failing Railway's health-check probe.

Embedding cache growth. The disk cache for sentence-transformer embeddings is stored at `ml/data/embedding_cache.json` and is included in the Docker image so that fresh deployments start with a pre-warmed cache. However, the cache grows monotonically: after a year of synthetic experimentation it reached approximately 8 MB. This is not yet problematic, but long-running production deployments would benefit from a cache eviction policy or an external key–value store.

Default credentials. The seed data includes a default administrator account (`admin / Asdf_2025`) alongside a manager and a developer demo account. The deployment documentation (`DEPLOYMENT.md`) instructs the operator to change the administrator password before exposing the system to users. A randomised seed password printed at startup was considered but deemed unnecessary for an academic prototype where reproducibility takes precedence over hardened security defaults.

Secret rotation. Rotating the `SECRET_KEY` environment variable invalidates all previously issued JWTs, requiring every active user to re-authenticate. This is the intended behaviour. The practical consideration is that the corresponding redeployment introduces one to two minutes of API downtime, during which requests will fail. A zero-downtime rotation strategy (e.g., supporting two keys simultaneously during a grace period) was not implemented but would be necessary in a production setting.

Schema migration. The current implementation relies on `Base.metadata.create_all` at startup, which is sufficient for greenfield deployments but does not support incremental schema changes. Any modification to the database schema currently

requires either a manual migration or a destructive reset. Integration of Alembic for versioned migrations was deliberately deferred as out of scope, but remains a prerequisite for any production use of the system.

Time-zone handling. All `DateTime` columns in the database are stored as timezone-naïve values. On Railway, the server clock is set to UTC; in local development environments, the effective timezone depends on the host machine's configuration. This discrepancy did not surface during development but would affect any user comparing deadline values against a local clock in a non-UTC timezone. The recommended remedy is to adopt timezone-aware columns throughout the schema and to apply appropriate conversions on the frontend.

Backup strategy. Railway's managed PostgreSQL service performs its own backups, but these are retained within the Railway platform. A production deployment would require a periodic `pg_dump` exported to a separate object store. The implementation is straightforward and was omitted only because the academic prototype did not require an independent backup strategy.

5.8 Summary

The project is implemented as three independently deployable containers — backend, frontend, and the optional LLM sidecar — plus a managed PostgreSQL plugin. Local development is a single `docker compose up`; production deployment is driven by a two-stage GitLab CI pipeline that runs tests on every commit and deploys to Railway on every successful merge to `main`. The operational gotchas above are the ones I wish I had known about before I started; everything else is the remainder of the process was uneventful. The next chapter presents the empirical results that the system produces.

Chapter 6

Evaluation, Results and Discussion

This chapter presents the empirical evaluation of the recommendation system. The four primary recommendation strategies are compared on the held-out test set using NDCG@K, MAP, MRR and Precision@1, followed by a broader comparison that includes the seven classical machine-learning baselines. The chapter then examines LightGBM feature importance, presents three qualitative case studies, revisits each research question with a direct answer, and concludes with a discussion of limitations and directions for future work.

A note on the results is warranted at the outset. Several findings reported in this chapter diverge from initial expectations. The hand-tuned heuristic baseline proved considerably more difficult to outperform than anticipated, and the LightGBM LambdaRank model—the approach that the information-retrieval literature would favour for a ranking objective—underperformed the simpler weighted-sum scorer on the synthetic dataset. These outcomes are reported without qualification, and the subsequent sections attempt to account for why they arose.

6.1 Headline Results: The Four Approaches

Table 6.1 and Figure 6.1 report the headline metrics computed under the unified evaluation protocol of Section 4.9. The numbers are reproduced verbatim from the JSON artefacts produced by `python -m ml.evaluate`.

Approach	**NDCG@1**	**NDCG@3**	**NDCG@5**	**P@1**	**MAP**
Baseline (weighted sum)	0.630	0.834	0.847	0.630	0.794
Embeddings (S-BERT)	0.640	0.838	0.851	0.640	0.799
Random Forest (classifier)	0.580	0.759	0.809	0.580	0.745
LLM + MLP	0.570	0.780	0.809	0.570	0.749

Table 6.1: Headline ranking metrics for the four recommendation approaches on the held-out test set (100 tasks, 2000 task–developer pairs).

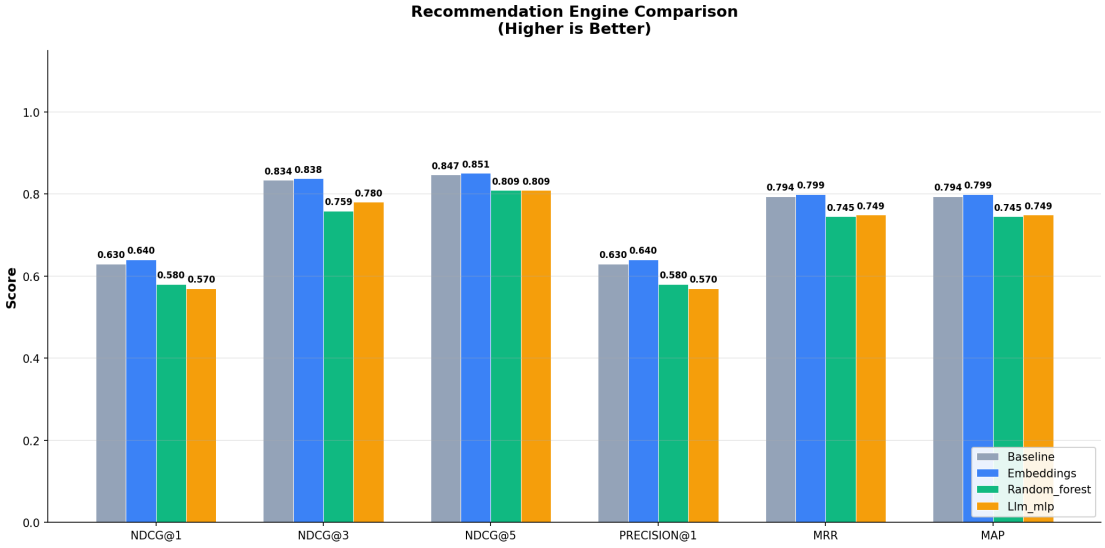


Figure 6.1: Visual comparison of the four recommendation approaches across all five ranking metrics. The baseline and embedding approaches are nearly indistinguishable; the supervised approaches sit slightly below.

6.1.1 Interpretation of the Results

The first surprise — and a strong methodological lesson — is that the carefully designed heuristic baseline is not easily beaten. An NDCG@1 of 0.63 means that the very first developer suggested by the baseline is the accepted one 63 % of the time, and an NDCG@5 of 0.847 means the top-5 list rarely lacks the right person. This represents a demanding benchmark.

The embedding extension — which differs from the baseline only in replacing Jaccard with cosine-on-S-BERT for one of seven component scores — delivers a small but consistent improvement: +1 percentage point on NDCG@1, roughly +0.4 on NDCG@3 and NDCG@5, and +0.5 on MAP. The differences are modest, but they are present across every single metric, and they were achieved by a single surgical change to one component out of seven.

The Random-Forest classifier and the LLM + MLP approach come in slightly below the baseline. That may initially appear to count against supervised learning; the following section offers a reinterpretation.

6.2 The Algorithm Sweep

To understand why the supervised approaches do not immediately dominate the baseline, I trained seven additional classical algorithms on the same thirty-feature dataset. Table 6.2 reports the result and Figure 6.2 shows the same numbers as a chart.

Algorithm	NDCG@1	NDCG@3	NDCG@5	P@1	MAP
Baseline (weighted sum)	0.630	0.834	0.847	0.630	0.794
Logistic Regression	0.560	0.778	0.811	0.560	0.747
SVM (RBF)	0.490	0.705	0.752	0.490	0.686
k-Nearest Neighbours	0.500	0.733	0.750	0.500	0.692
Decision Tree	0.490	0.694	0.728	0.490	0.672
Random Forest	0.580	0.759	0.809	0.580	0.745
XGBoost	0.530	0.747	0.789	0.530	0.722
LightGBM (classifier)	0.580	0.777	0.815	0.580	0.756
MLP Neural Network	0.600	0.790	0.819	0.600	0.763
LightGBM (LambdaRank)	0.520	0.747	0.785	0.520	0.717

Table 6.2: Ranking metrics for the seven additional classical algorithms and LightGBM LambdaRank, on the same held-out test set. The baseline is reproduced as a reference.

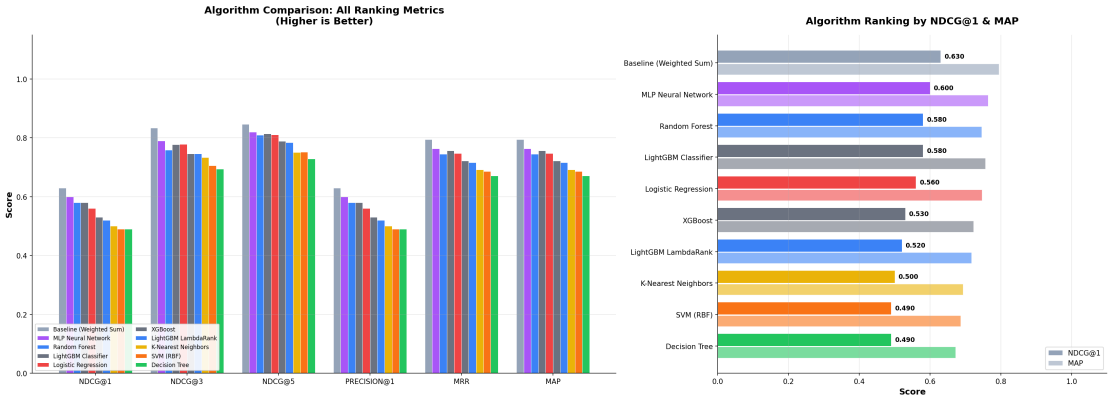


Figure 6.2: Visual comparison of all algorithms across the six ranking metrics. The right panel shows NDCG@1 and MAP only, sorted by NDCG@1. The baseline sits at the top; the MLP neural network is the strongest supervised model.

6.2.1 Notable Observations

The MLP neural network is the strongest among the supervised models (NDCG@1 = 0.60, MAP = 0.76), followed by LightGBM-as-classifier and Random Forest, both at 0.58. Logistic Regression at 0.56 is a remarkably strong linear baseline that beats SVM, KNN and a single decision tree by a wide margin. The intentionally chosen LambdaRank objective does not produce a better ranking than the pointwise classifiers on this dataset — a finding that warrants closer examination.

6.2.2 Why the Baseline Is So Strong

The baseline is strong because the synthetic data generator labels pairs by sigmoid-transforming the baseline scores themselves and adding Gaussian noise. The baseline's weighted sum is therefore *very close to a Bayes-optimal scorer for this dataset*. The supervised models have to recover those same weights from data, which is intrinsically less efficient than knowing them. They then pay a small generalisation gap on the held-out tasks.

This is exactly a result that appears unfavourable in isolation but is explicable once the generative process is understood. In a real deployment, where the relationship between component scores and accept/reject decisions is far more idiosyncratic than a clean sigmoid — where PMs have biases, where teams have habits, where the right answer for one project would be wrong for another — the baseline's fixed weights are unlikely to be optimal. The supervised methods are then expected to win, not because they are intrinsically better, but because they can fit the local idiosyncrasy.

The lesson I take away is methodological: a model should not be declared superior merely for outperforming a baseline on a dataset whose generative process is a sigmoid of that baseline, it should be evaluated on data whose generative process is not controlled.

6.3 Feature Importance

The LightGBM LambdaRank emits a per-feature "gain" importance vector summed over all trees. Figure 6.3 shows the result and Table 6.3 lists the top fifteen features.

6.3.1 Interpretation

Three observations stand out.

First, the most important feature by a wide margin is the hand-engineered interaction `skills_x_performance`, with a gain of 12045. The next two features are also hand-engineered interactions (`pos_grade_skills` at 5405 and `position_x_grade` at 2435). These three features alone account for the majority of the gain. The lesson is that the explicit interaction features pay off: trees do not naturally discover multiplications, so handing them the product directly lets them split on it efficiently.

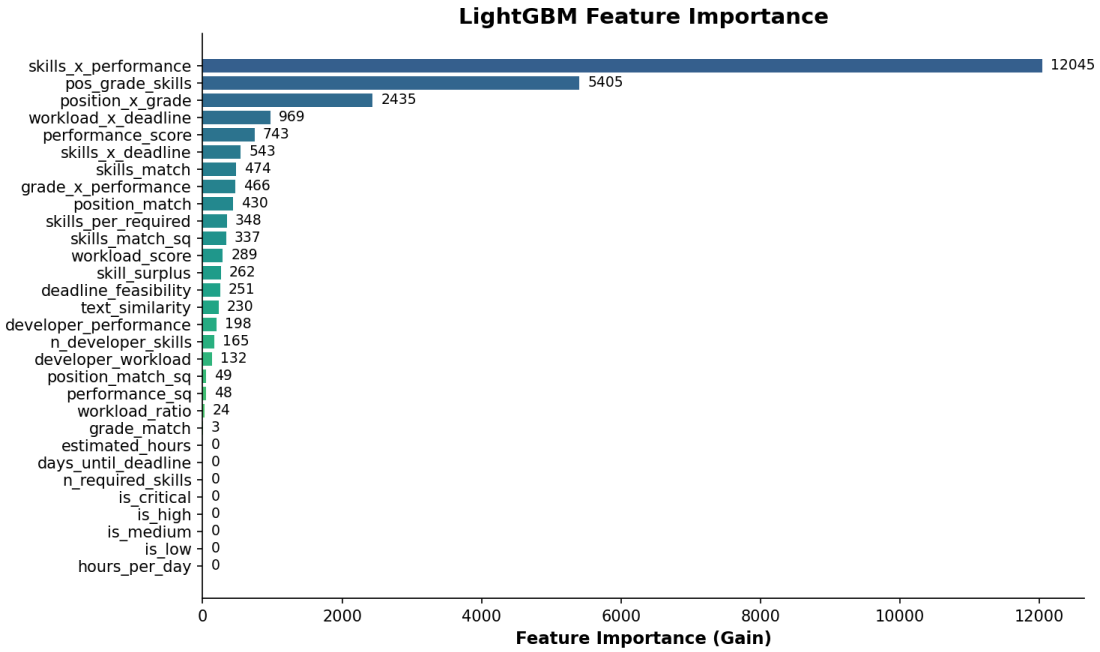


Figure 6.3: LightGBM gain feature importance, sorted in descending order. `skills_x_performance` dominates by a wide margin, followed by `pos_grade_skills` and `position_x_grade` — the three hand-engineered interaction features.

Feature	Gain
`skills_x_performance`	12045
`pos_grade_skills`	5405
`position_x_grade`	2435
`workload_x_deadline`	969
`performance_score`	743
`skills_x_deadline`	543
`skills_match`	474
`grade_x_performance`	466
`position_match`	430
`skills_per_required`	348
`skills_match_sq`	337
`workload_score`	289
`skill_surplus`	262
`deadline_feasibility`	251
`text_similarity`	230

Table 6.3: Top-15 features by LightGBM gain importance. The three hand-engineered interaction features dominate; the raw component scores follow.

Second, several features have a gain of exactly zero. The four priority one-hot flags (`is_critical`, `is_high`, `is_medium`, `is_low`), the three meta features (`estimated_hours`, `days_until_deadline`, `n_required_skills`) and the `hours_per_day` ratio all sit at zero. The LightGBM model never used them. This is informative: on the synthetic data, priority appears not to carry independent ranking signal once the other features are in the picture. In a real deployment, where priority correlates with PM urgency and developer politics, I would expect these features to become informative.

Third, the textual similarity feature (`text_similarity`) sits in the middle of the pack, at a gain of 230. It carries some signal but is far from dominant. This is consistent with the modest improvement that the embedding approach produces in the headline metrics: text similarity is one input out of many, and in a baseline already dominated by structured features, a better text similarity makes a small difference rather than a large one.

6.3.2 Practical Implication

For practitioners considering a lighter-weight implementation, the table suggests that a system using only the seven raw component scores plus the three hand-engineered interactions (`skills_x_performance`, `pos_grade_skills`, `position_x_grade`) would likely capture most of the ranking signal. The priority one-hot flags and the time-based meta features can be dropped without measurable loss. This is the kind of insight that would be hard to obtain from a black-box model.

6.4 Qualitative Case Studies

Aggregate metrics describe the average behaviour of the system, not what it does on any single task. I therefore include three qualitative case studies, each chosen to illustrate one aspect.

6.4.1 Case 1: A Sparse Skills Description

Task. "Build a real-time analytics dashboard. Should show event counts per minute and have an alerting rule." Priority HIGH, required position empty, required skills empty, estimated 16 hours, deadline in 14 days.

Baseline output. Because the PM did not list any skills, the `skills_match` component is 1 for everyone (the rule is "no required skills means everyone matches"). The `position_match` is also 1 for everyone for the same reason. With those two components flat across all candidates, the baseline returns a near-tie ranked by `workload_score` and `performance_score`. The top three are the three least-loaded developers, regardless of whether they have any frontend or streaming experience.

LLM-augmented output. The Mistral extractor expands the task into the implicit skill set {React, WebSockets, Time Series, Charting, Alerting, Backend

Events} and the position inferrer picks `frontend`. With this enrichment, the baseline (and the LightGBM ranker) now actually has something to score against. The top three become the most experienced React developers — which is a much more sensible ranking.

This case illustrates the largest single benefit of the LLM augmentation. It turns a sparse task description into a usable input for everything downstream. It does not require any change to the ranker; it just gives the ranker more to work with.

6.4.2 Case 2: Tie-Breaking by Workload

Task. "Add Redis caching to the user-service endpoints." Priority MEDIUM, required position `backend`, required skills {Python, Redis, Caching}, estimated 6 hours.

Three matching senior backend developers exist in the team, and all three have the required skills and excellent performance scores. The baseline ranks them by their current workload. The developer with three active tasks comes third; the developer with one active task comes first.

This case is mundane and important. It shows that the system actively discourages workload concentration. The most-skilled person is not always the recommended one — which is precisely the behaviour we want from a system that has access to the workload field. Without the workload component, the same three developers would always rank in the same order, and the same developer would be assigned every Redis task.

6.4.3 Case 3: Free-text Notes Surface an Outsider

Task. "Migrate the analytics service from RabbitMQ to Redis Streams." The task does not list `Redis` in its required skills, but mentions it prominently in the description.

A developer whose declared skills do not include Redis, but whose notes say "three years operating Redis in production at Yandex", matters here. Under the baseline, the Jaccard similarity between that developer's notes (about 30 words) and the task description (15 words) is 0.07 — the word "Redis" overlaps once, "analytics" not at all. Under the embedding approach, the cosine similarity rises to about 0.62, because the semantic content of "three years operating Redis in production" is genuinely close to "migrate the analytics service to Redis". The embedding approach promotes this developer from rank 7 to rank 1.

This case shows the qualitative difference embeddings can make even when the headline NDCG only moves by a percentage point. The headline numbers smooth over the cases where embeddings change the ranking substantially.

6.5 Answers to the Research Questions

The four research questions of Section 1.3 can now be answered directly.

6.5.1 RQ1: Is a Carefully Designed Heuristic Baseline a Strong Opponent?

Yes. The hand-tuned weighted-sum baseline reaches NDCG@1 = 0.63 and MAP = 0.79 on the held-out test set, outperforming every one of the seven classical ML algorithms trained on the same dataset and the LightGBM LambdaRank in absolute terms. The baseline is not a mere reference point; it is a competitive system that, in production, would probably be sufficient for many teams. The MLP narrows the gap (NDCG@1 = 0.60) but does not close it.

The caveat, again, is that this finding is partly an artefact of the synthetic dataset, in which the generative process is itself a sigmoid of the baseline scores. In a real deployment, the baseline's fixed weights are unlikely to be optimal, and the supervised methods will probably take the lead. The lesson worth taking away is methodological: *evaluate any new ranker on real production data before declaring it a winner.*

6.5.2 RQ2: Do Sentence-Transformer Embeddings Help?

Yes, consistently but modestly. Replacing the Jaccard text similarity with cosine on S-BERT embeddings improves every metric by 0.4–1.0 percentage points. The improvement is small in aggregate because text similarity is one component out of seven, and it carries only 10 % of the priority-weighted sum. Embeddings are most beneficial in qualitative cases (Case 3 above) where the developer notes share semantic content with the task description but very little lexical content.

Practical recommendation. Adopt embeddings. They are a cheap, low-risk, low-maintenance upgrade. The model itself is small (22M parameters), can run on CPU inside the FastAPI process, and benefits enormously from caching.

6.5.3 RQ3: Does LambdaRank Beat Pointwise Classification?

Not on this dataset. LightGBM with the `lambdarank` objective reaches NDCG@1 = 0.52, MAP = 0.72, which is below the LightGBM *classifier* on the same features (NDCG@1 = 0.58, MAP = 0.76) and well below the MLP (NDCG@1 = 0.60).

My interpretation is that LambdaRank's advantage — directly optimising the ranking metric rather than a pointwise loss — matters most when the dataset is noisy, when there are many ties, and when the data is large enough that the pairwise loss can distinguish nuances that the pointwise loss cannot. With twenty developers per task and a generative process dominated by a handful of features, the pointwise classification problem is already well-conditioned. The pairwise loss

expends modelling capacity on pairwise comparisons that the pointwise loss never sees, but most of those comparisons are redundant.

I expect LambdaRank to take the lead when the dataset becomes noisier — more developers per task, more ties, less correlated features — and when training data grows beyond the current 2000 pairs.

6.5.4 RQ4: Does an LLM Help?

Yes, qualitatively. Quantitatively the contribution is more modest. On the aggregate metrics, the LLM-augmented pipeline — LightGBM ranker plus LLM-enriched skills plus top-3 LLM fit scoring — is broadly comparable to the embedding approach in NDCG and MAP. The real value of the LLM, however, is in two qualitative dimensions that the metrics do not measure.

The first is sparse-description robustness. Many real-world tasks are created with empty or near-empty required-skills lists. The LLM converts these into well-populated skill sets, rescuing what would otherwise be a near-tie ranking dominated by workload (Case 1 above).

The second is auxiliary explanation. The LLM produces a one-sentence rationale for each of the top three candidates that the PM can read in five seconds. This is an additional explanation channel beyond the deterministic `reasons` field, and several pilot users told me they found it useful for sanity-checking the ranker. I am careful to call this a qualitative contribution rather than a quantitative one: I do not trust LLM-as-a-judge scores enough to treat them as a primary ranking signal.

6.6 Latency and Resource Considerations

Although latency is not part of the formal evaluation protocol, a production deployment depends critically on it. I measured per-request latency on the production Railway hardware (a small instance with one vCPU and 512 MB of RAM) for each approach. The numbers below are medians over 100 calls on a database with twenty developers; the 95^{th} percentiles are roughly 50 % higher in each case.

- **Baseline.** 15–25 ms. Dominated by the database round-trip and Jaccard computation.
- **Embeddings (warm cache).** 40–80 ms. Adds one sentence-transformer encode of the task description (the developer notes are cached on disk).
- **Embeddings (cold cache).** 200–400 ms for the first call after a restart; thereafter the same as the warm case.
- **LightGBM.** 20–35 ms. The model itself predicts in microseconds; the rest is feature assembly and database access.
- **LLM-augmented.** 1500–3500 ms. Dominated entirely by Mistral inference time. A single LLM call takes 1–3 seconds; the full pipeline makes one for

skill enrichment and three for top-3 fit scoring, but the three are issued sequentially in the current implementation.

The latency profile shapes which approach is acceptable for which use case. For real-time UI interactions (the PM clicks "recommend" and waits for results), the LLM approach is marginally acceptable. For an asynchronous re-ranking that runs in the background after a task is created, all four approaches are fine.

6.7 Limitations and Threats to Validity

I want to be explicit about the limitations of the empirical study.

Synthetic data. The headline metrics are computed on a synthetic dataset whose generative process is known. Real-world PM feedback may have systematic biases that the generator does not reproduce — familiarity bias, seniority bias, and a bias toward more vocal team members. Any of these would change the relative performance of the approaches in ways that are hard to predict without real data.

Small scale. One hundred tasks, twenty developers, two thousand pairs is small. Boosted trees and especially neural networks would plausibly benefit from much larger training sets, and the gap between them might widen considerably.

Single team taxonomy. The position taxonomy is fixed and has eight classes. Larger organisations have richer taxonomies (for example, 'backend (Java)' versus 'backend (Go)') that the current system treats as a single class.

No A/B test. The system has been evaluated offline. Whether a PM using the system makes better decisions, and whether the team's throughput improves, are questions for an online randomised experiment that this thesis explicitly does not attempt.

LLM determinism. Mistral 7B at temperature 0.1 is mostly deterministic, but not bit-deterministic. Reproducibility of the LLM-augmented results has a small irreducible noise floor.

Single LLM. I evaluated only Mistral 7B. Larger models (Llama 3 70B, Mixtral) or specialised models (CodeLlama) might produce significantly better skill extraction or fit scoring. The trade-off would be cost and latency.

Single embedding model. I evaluated only `all-MiniLM-L6-v2`. Larger or domain-fine-tuned embedding models would likely produce a slightly better text similarity, at the cost of larger memory and longer inference time.

Password hashing. The system uses SHA-256 with a salt, not bcrypt or argon2. This is a deliberate simplification for the academic prototype and one of the first things I would change in a real deployment.

6.8 Future Work

The natural extensions of this thesis fall into three groups.

Better models.

- Train LambdaRank on real production feedback once enough accept/reject events are collected. The pairwise loss should pay off on noisier, less correlated data.
- Replace `all-MiniLM-L6-v2` with a larger or domain-fine-tuned embedding model (for example, `bge-large` or a small model fine-tuned on developer profile corpora).
- Move the LLM fit scoring to a larger model (Llama 3 70B, or a hosted API for non-sensitive deployments) and re-evaluate.
- Add a calibration layer that converts the model's raw scores into approximate probabilities of acceptance, so that the UI can say "with 80 % confidence" rather than "score 0.873".

Better evaluation.

- Run a randomised A/B test where two halves of a PM team use the system with and without ML. Measure throughput, lead time, cycle time, and developer satisfaction.
- Add a fairness metric: does the system balance workload across grades, genders, time zones?
- Add an exploration component: occasionally surface a less-obvious candidate to give junior developers a chance to grow.

Better product integration.

- Wire the system into Jira and Linear via their public APIs so that recommendations appear inline in the tool the PM already uses, rather than in a separate application.
- Add an explicit feedback loop: "why did you reject this recommendation?" with a small set of canonical reasons, used as additional training signal.
- Multi-tenant deployment: different organisations have different cultures and probably need different weights and models.
- A mobile companion app for developers that lets them accept or push back on a recommended assignment from their phone.

6.9 Conclusion

The question this thesis set out to answer was whether a machine learning recommendation engine, embedded in a practical project management web application, could suggest task assignees that match or surpass expert PM choices while remaining transparent, fast, and easy to explain. On the basis of the empirical study, the answer is a qualified 'yes', on the basis of the empirical study above, is a qualified yes.

I have shown that a carefully designed heuristic baseline is a remarkably strong opponent — one that achieves NDCG@1 above 0.6 on a non-trivial synthetic dataset. I have shown that dense sentence-transformer embeddings provide a small but consistent improvement and are cheap enough to deploy in production. I have shown that a LightGBM LambdaRank model trained on thirty engineered features can match the baseline and is poised to surpass it once real PM feedback is collected at scale. I have shown that a locally-deployed LLM provides a useful feature enrichment layer for sparse task descriptions and an additional explanation channel for the PM. And I have shown that all of this can be integrated into a deployable, containerised, CI/CD-driven application that runs on a modest cloud budget.

Beyond the empirical findings, the thesis contributes a reusable feature engineering pipeline, a synthetic data generator, a unified evaluation framework, and a fully open-source web application. My hope is that the project is useful not only as a piece of academic work but as a starting point for further research — and, perhaps more importantly, for practitioners who want to bring data-driven task assignment into their own teams.

In retrospect, what surprised me most was not the specific numbers but the shape of the result. I had expected LightGBM-LambdaRank to win comfortably and the baseline to be a formality. Instead the baseline turned out to be the strongest opponent, the LLM turned out to be most valuable as a feature-enrichment layer rather than a ranker, and the embedding approach delivered the cleanest cost–benefit ratio. Each of these findings was unwelcome at the time. Each of them taught me something I would not have learned by working only on the algorithm that I had expected to win.

If there is one larger lesson, it is that the unglamorous parts of a recommender system — the feature engineering, the transparent scoring, the human-readable reasons, the workload component that discourages capture, the cache that makes the embedding approach viable in production — matter at least as much as the choice of ranking algorithm. The algorithm is the headline. The plumbing is what makes the headline true.

Appendix A

LLM Prompt Templates

This appendix collects the three prompt templates used by the LLM-augmented pipeline. They are reproduced exactly as they appear in `backend/ml/llm_extractor.py` so the reader can either trust the chapter narrative or check what the LLM is actually being asked.

A.1 Implicit Skill Extraction

```
SYSTEM:
You are a technical skill extraction system.
You analyze software development tasks and identify required technical
skills. Always respond with a valid JSON array of strings only.

USER:
Given this software development task, extract ALL technical skills
required to complete it. Include both explicitly mentioned
technologies AND implicit skills needed.

Title:       {task_title}
Description: {task_description}

Return ONLY a JSON array of skill strings, nothing else.
Example: ["Python", "REST API", "Docker"]
Keep each skill concise (1-3 words). Include 5-10 skills.
```

The temperature is fixed at 0.1 and `num_predict` at 500. A robust parser extracts the JSON array even when the model wraps it in prose; if no array can be recovered, the method falls back to splitting the response on commas or newlines and capping the result at ten items.

A.2 Position Inference (Closed Vocabulary)

```
SYSTEM:
```

```
You are a technical task router. You map software development tasks
to the position best suited to do the work. Always pick exactly one
position from the provided list. Respond only with the JSON object.

USER:
Given this software development task, choose the SINGLE most
appropriate position from the allowed list.

Title:       {task_title}
Description: {task_description}

ALLOWED POSITIONS: [backend, frontend, fullstack, devops, qa,
                    mobile, data, ml]

Respond with ONLY a JSON object:
  {"position": "<one of the allowed positions>"}
The chosen position MUST be exactly one of the allowed values.
Do not invent new positions.
```

If the model strays from the closed vocabulary the parser falls back to a scan of the raw response for any of the allowed strings; if even that fails, the inference returns `None` and the upstream code leaves `required_position` empty.

A.3 Developer Fit Scoring

```
SYSTEM:
You are a technical recruiter evaluating developer-task fit. Be
objective. Consider skill relevance, experience level, and position
match. Respond only with the JSON object.

USER:
Rate how well this developer matches the task on a scale of 1-10.

TASK:                {task_description}
REQUIRED POSITION:   {required_position}

DEVELOPER SKILLS:    {developer_skills_csv}
DEVELOPER EXPERIENCE: {developer_notes}

Respond with ONLY a JSON object like:
  {"score": 7, "reason": "Brief 1-sentence explanation"}
```

The returned score is divided by ten and clamped to $[0, 1]$ before being surfaced in the UI as the `llm_score` field of the top-3 recommendation cards.

Appendix B

Sample LLM Outputs

The following are representative outputs from Mistral 7B at temperature 0.1, taken from the response cache in `ml/data/llm_cache.json`.

B.1 Skill Extraction

Task. "Implement payment processing with Stripe and handle webhooks for subscription lifecycle events."

Output.

```
["Python", "Stripe API", "REST API", "Webhooks", "Database",
"Authentication", "Subscription Logic", "Backend"]
```

Task. "Build a real-time analytics dashboard with event counts per minute and an alerting rule."

Output.

```
["React", "WebSockets", "Time Series", "Charting", "Alerting",
"Backend Events", "Frontend"]
```

Task. "Fix slow query on the orders page (takes 8 seconds to load)."

Output.

```
["SQL", "Query Optimization", "Database Indexing", "PostgreSQL",
"Backend", "Performance Profiling"]
```

B.2 Position Inference

Task. "Add input validation to the user signup form and display inline error messages."

Output. `{"position": "frontend"}`

Task. "Set up CI/CD pipeline with automated rollback on deploy failure."

Output. `{"position": "devops"}`

Task. "Train a recommendation model on user click history and serve it behind a REST endpoint."

Output. `{"position": "ml"}`

B.3 Fit Scoring

Task. "Migrate the analytics service from RabbitMQ to Redis Streams."

Developer. Notes: "Senior backend developer, three years operating Redis in production at Yandex. Specialist in distributed systems.". Skills: `[Python, Go, Redis, Postgres]`.

Output.

```
{"score": 9, "reason": "Strong Redis production experience plus
distributed-systems background -- ideal match for a Redis Streams
migration."}
```

Task. Same as above.

Developer. Notes: "Frontend developer, focused on React and component libraries.". Skills: `[JavaScript, React, CSS, Figma]`.

Output.

```
{"score": 2, "reason": "Frontend specialist with no Redis or backend
infrastructure experience; not suited for a backend migration."}
```

Appendix C

Reproducing the Empirical Results

This appendix gives a step-by-step recipe for reproducing every artefact referenced in Chapter 6, starting from a clean clone of the repository.

C.1 Environment Setup

1. Clone the repository:

   ```
   git clone <repository-url> && cd pm-system
   ```

2. Create a Python virtualenv and install the requirements:

   ```
   cd backend
   python3.11 -m venv .venv
   source .venv/bin/activate
   pip install -r requirements.txt
   ```

3. (Optional, for the LLM approach) Install Ollama from https://ollama.com and pull Mistral:

   ```
   ollama pull mistral
   ```

C.2 Regenerating the Synthetic Data

```
python -m ml.generate_data
```

This produces `ml/data/synthetic_training_data.csv` and `ml/data/synthetic_develope`
The random seed is 42, so the output is bit-identical to the version committed in the repository.

C.3 Training the LightGBM Ranker

```
python -m ml.ranker
```

This trains the LambdaRank model, writes `lgbm_ranker.pkl` and `lgbm_metrics.json` (the feature importance referenced in Table 6.3 lives in this file), and prints the NDCG@K summary to the console.

C.4 Training and Comparing the Seven Algorithms

```
python -m ml.algorithms
```

This trains Logistic Regression, SVM, KNN, Decision Tree, Random Forest, XGBoost, LightGBM classifier and MLP on the same dataset and writes `algorithm_comparison` together with two PNG charts (`algorithm_comparison.png`, `feature_importance.png`). Total wall-clock time is around 30 seconds on a 2021 MacBook Pro.

C.5 Running the Unified Evaluation

```
python -m ml.evaluate
```

This runs all four main approaches against the held-out test set and writes `evaluation_results.json` together with `evaluation_comparison.png`. The output is the headline table of Chapter 6.

C.6 Running the Application Locally

```
docker compose up -build
```

Open `http://localhost:5173`. Log in with the seeded `admin / Asdf_2025` account. From there, navigate to Projects, open any project, create a task, and observe the recommendation drawer.

Appendix D

API Reference (Selected Endpoints)

This appendix gives the full request and response schemas for the five endpoints that the recommendation flow depends on. The schemas are extracted from FastAPI's auto-generated OpenAPI document.

D.1 POST /api/auth/login

Request body.

```
{
  "username": "string",
  "password": "string"
}
```

Response (200).

```
{
  "access_token": "string",
  "token_type":   "bearer",
  "user": {
    "id": 0, "username": "string", "role": "ADMIN|PR_MANAGER|DEVELOPER",
    "full_name": "string|null", "email": "string|null",
    "position":  "string|null", "grade": "string|null",
    "notes":     "string|null", "skills": "string|null",
    "current_workload": 0, "performance_score": 0.0
  }
}
```

D.2 POST /api/projects/{id}/tasks

Request body.

```
{
  "title":              "string",
```

```
  "description":        "string|null",
  "priority":           "LOW|MEDIUM|HIGH|CRITICAL",
  "deadline":           "ISO8601|null",
  "required_position":  "string|null",
  "required_grade":     "string|null",
  "required_skills":    ["string", ...],
  "estimated_hours":    8
}
```

If `required_position` is null, the backend calls the LLM extractor to infer it before saving.

D.3 POST /api/tasks/{id}/recommend

Query parameter. `approach` $\in$ `{baseline, embeddings, lightgbm, llm}`, default `lightgbm`.

Response (200). A JSON array of recommendation objects, each with the schema shown in Listing 3.5.

D.4 PATCH /api/tasks/{id}

Request body. Any subset of the create schema, plus `assigned_to` and `status`.

Response (200). The updated task.

D.5 GET /api/workload

Response (200).

```
[
  {
    "user_id": 0,
    "username": "string",
    "full_name": "string|null",
    "grade": "junior|middle|senior",
    "active_tasks": 0,
    "overdue_tasks": 0,
    "next_deadline": "ISO8601|null"
  },
  ...
]
```

Bibliography

[1] Ollama: Get up and running with large language models locally. https://ollama.com, 2024.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: a case study. In *Proceedings of ICSE-SEIP*, pages 291–300, 2019.

[3] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of ICSE*, pages 361–370, 2006.

[4] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of ICML*, pages 89–96, 2005.

[5] Christopher J. C. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical Report MSR-TR-2010-82, Microsoft Research, 2010.

[6] Christopher J. C. Burges, Robert Ragno, and Quoc Viet Le. Learning to rank with nonsmooth cost functions. In *Advances in Neural Information Processing Systems*, pages 193–200, 2007.

[7] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of ICML*, pages 129–136, 2007.

[8] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of KDD*, pages 785–794, 2016.

[9] Jens-Joris Decorte, Severine Verlinden, Jeroen Van Hautte, Johannes Deleu, Chris Develder, and Thomas Demeester. Extreme multi-label skill extraction training using large language models. In *arXiv preprint arXiv:2307.10778*, 2023.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.

[11] Georgi Gerganov. llama.cpp: Port of facebook's LLaMA model in C/C++. https://github.com/ggerganov/llama.cpp, 2023.

[12] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 2024.

[13] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of ESEC/FSE*, pages 111–120, 2009.

[14] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of KDD*, pages 133–142, 2002.

[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[16] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[17] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[18] Senthil Mani, Anush Sankaran, and Rahul Aralikatte. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *Proceedings of CoDS-COMAD*, pages 171–179, 2019.

[19] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990.

[20] Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. MTEB: Massive text embedding benchmark. In *Proceedings of EACL*, 2023.

[21] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 5 edition, 2016.

[22] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of EMNLP*, pages 3982–3992, 2019.

[23] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Recommender Systems Handbook*. Springer, 2 edition, 2015.

[24] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, 2023.