National College of Ireland

Configuration Manual

MSc Research Project
Artificial Intelligence

Shoban Ravichandran
Student ID: 23272040

School of Computing
National College of Ireland

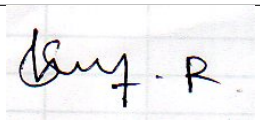
Supervisor: Paul Stynes

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Shoban Ravichandran
Student ID:	23272040
Programme:	Artificial Intelligence
Year:	2018
Module:	MSc Research Project
Supervisor:	Paul Stynes
Submission Due Date:	20/12/2018
Project Title:	Configuration Manual
Word Count:	2257
Page Count:	13

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	15th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Shoban Ravichandran
23272040

1 System Requirements and Dependencies

1.1 Hardware Requirements

The experimental setup requires the following minimum hardware specifications to ensure optimal performance during model evaluation and optimization based on computational requirements for multi-objective optimization and transformer model inference:

- **Memory (Random Access Memory - RAM):** 16GB minimum, 32GB recommended for optimal performance during embedding generation and vector storage operations
- **Storage:** 10GB free disk space for datasets, models, and vector storage, with additional space recommended for logging and temporary files
- **Central Processing Unit (CPU):** Multi-core processor (8+ cores recommended) for efficient embedding generation and parallel processing during optimization
- **Graphics Processing Unit (GPU):** Optional but recommended for faster embedding computation (Compute Unified Device Architecture (CUDA)-compatible graphics card with 6GB+ Video Random Access Memory (VRAM))

1.2 Reference System Specification

The experimental validation was conducted on a system with the following specifications, which provides excellent performance for all research components:

Table 1: Reference System Configuration

Component	**Specification**
Processor	AMD Ryzen 5 7235HS (3.20 GHz)
Installed RAM	24.0 GB (23.7 GB usable)
Graphics Card	NVIDIA GeForce RTX 3050 6GB Laptop GPU
Storage	954 GB total (589 GB of 954 GB used)

Performance Characteristics:

- **Processing Capability:** AMD Ryzen 5 7235HS provides sufficient multi-core performance for embedding generation and Non-dominated Sorting Genetic Algorithm II (NSGA-II) optimization

- **Memory Capacity:** 24GB RAM exceeds recommended specifications, enabling efficient processing of large datasets and multiple model orchestration
- **GPU Acceleration:** NVIDIA GeForce RTX 3050 with 6GB VRAM provides CUDA acceleration for embedding computations and transformer operations
- **Storage Capacity:** 954GB total storage provides ample space for datasets, vector indices, and system logs

This configuration demonstrates that the experimental framework can operate effectively on modern laptop hardware, making the research reproducible on widely available computing resources.

1.3 Software Prerequisites

The system has been tested and validated on the following operating system following established software engineering practices:

- Windows 10/11

Required Software Components:

- Python 3.8 or higher with pip package manager
- Git for version control and repository management
- Docker (optional, for local Qdrant vector database instance)

Graphics Processing Unit Driver Requirements (for NVIDIA systems):

- NVIDIA Driver 470.0 or later
- CUDA Toolkit 11.2 or later (for GPU-accelerated operations)
- CUDA Deep Neural Network library (cuDNN) 8.1 or later (for optimized deep learning operations)

1.4 Core Python Dependencies

The complete dependency specification is maintained in `requirements.txt` following standard dependency management practices. Critical dependencies include:

Core Data Processing:

- `numpy>=1.21.0` - Numerical computations and array operations
- `pandas>=1.3.0` - Data manipulation and analysis
- `scikit-learn>=1.0.0` - Machine learning utilities and metrics

Vector Database and Embeddings:

- `qdrant-client>=1.6.0` - Vector similarity search engine
- `sentence-transformers>=2.2.0` - Semantic embeddings generation

- `transformers>=4.21.0` - Hugging Face transformer models
- `torch>=1.12.0` - PyTorch deep learning framework with CUDA support

Large Language Model Providers:

- `groq>=0.4.0` - Primary Large Language Model (LLM) provider for fast inference
- `openai>=1.0.0` - OpenAI GPT model access (optional)
- `anthropic>=0.7.0` - Claude model integration (optional)

Multi-objective Optimization:

- `pymoo>=0.6.0` - NSGA-II optimization algorithm implementation

Text Processing and Evaluation:

- `nltk>=3.8` - Natural language processing toolkit
- `rouge-score>=0.1.2` - Recall-Oriented Understudy for Gisting Evaluation (ROUGE) evaluation metrics

Portable Document Format (PDF) Processing:

- `PyPDF2>=3.0.0` - PDF text extraction (fallback)
- `PyMuPDF>=1.20.0` - Advanced PDF processing (primary)

Code Analysis:

- `radon>=5.1.0` - Code complexity analysis
- `astunparse>=1.6.3` - Abstract Syntax Tree (AST) processing

Development and Analysis Tools:

- `python-dotenv>=0.19.0` - Environment variable management
- `rich>=13.0.0` - Enhanced console output formatting (optional)
- `pathlib>=1.0.1` - Path manipulation utilities
- `tqdm>=4.64.0` - Progress bars for long operations

Optional Visualization Dependencies:

- `matplotlib>=3.5.0` - Plotting and visualization
- `seaborn>=0.11.0` - Statistical data visualization
- `plotly>=5.10.0` - Interactive plotting

Note: All system diagrams and performance charts utilize minimum 12-point fonts for optimal readability following visualization best practices.

1.5 Performance Optimization Notes

The performance of the experimental system benefits significantly from targeted optimizations applied across memory management, storage, Input/Output (I/O), and processing efficiency. In terms of memory management, a batch size of 32 is used for embedding generation to ensure that available RAM is utilized effectively without excessive memory pressure. Vector database insertions are performed in batches of 200 chunks, striking a balance between throughput and memory stability. The NSGA-II optimization algorithm employs a population size of 150 individuals as established by Deb et al. (2002), providing sufficient search space for exploration while remaining within memory constraints. Additionally, intelligent skip logic for PDF processing prevents the re-generation of embeddings for documents that have already been processed, avoiding unnecessary memory consumption.

Storage and I/O operations are optimized to improve system responsiveness and reduce resource overhead. Vector indices are stored in a compressed format to minimize disk space usage, while intelligent caching mechanisms retain embeddings and model responses for reuse across experiments. A PDF processing status tracker ensures that files are not reprocessed unnecessarily, and logging is configured with rotation policies to prevent excessive disk usage from long-running operations. Furthermore, temporary files generated during processing are automatically deleted upon completion, ensuring a clean and lightweight working directory.

Processing efficiency is achieved through multiple strategies. Advanced code chunking methods following Fowler (2018) are applied to maximize the extraction of meaningful content, thereby improving the quality of downstream processing. Embedding generation for large datasets is performed in a multi-threaded manner to leverage parallelism and reduce execution time. Application Programming Interface (API) calls incorporate rate-limiting controls and robust error-handling routines to ensure stable, uninterrupted operation even under heavy load. For large document collections, the system employs progressive loading and processing techniques, allowing continuous throughput without overwhelming memory or storage resources.

1.6 Installation and Setup

Installation of all dependencies can be performed using standard Python package management:

```
pip install -r requirements.txt
```

Environment Configuration: Create a `.env` file or `key.env` file with required API keys following security best practices:

```
GROQ_API_KEY=your_groq_api_key
OPENAI_API_KEY=your_openai_api_key  # Optional
ANTHROPIC_API_KEY=your_anthropic_api_key  # Optional
QDRANT_URL=your_qdrant_cloud_url  # Optional for cloud deployment
QDRANT_API_KEY=your_qdrant_api_key  # Optional for cloud deployment
```

System Verification Commands: After installation, verify system capability using diagnostic scripts:

```
python -c "import torch; print(f'CUDA available: {torch.cuda.is_available()}')"
python -c "import sentence_transformers; print('Dependencies verified')"
python -c "from qdrant_client import QdrantClient; print('Qdrant client available')"
```

Quick Start:

```
python main.py  # Run complete system with evaluation and optimization
```

The system includes comprehensive health checks and status reporting to ensure all components are properly configured and operational before beginning the refactoring analysis workflow.

2 Application Programming Interface Configuration and Access Keys

2.1 Large Language Model Application Programming Interface Configuration

The transformer-based framework in this research supports multiple Large Language Model (LLM) providers to ensure robustness and facilitate comparative evaluation. While the system can integrate with different Application Programming Interfaces (APIs), at least one API key must be configured for operational use, with Groq designated as the primary and recommended provider. Groq is selected because of its extremely fast inference speeds and compatibility with five language models employed in this work: `llama3-70b-8192`, a high-performance general-purpose model; `gemma2-9b-it`, an efficient instruction-tuned model; `qwen/qwen3-32b`, which offers advanced reasoning capabilities within a 6K token limit; `moonshotai/kimi-k2-instruct`, designed for specialized instruction following; `deepseek-r1-distill-llama-70b`, a distilled large model. To access Groq, users must first register at `https://groq.com`, generate an API key from the dashboard's API Keys section, and then configure it within the environment variables. The platform enforces token limits ranging from 4,000 to 6,000 tokens per model, a default temperature of 0.1 for consistent outputs, and rate limits managed via built-in exponential backoff with 15-second retry delays. Requests have a 30-second timeout with automatic retries, ensuring reliability in production workloads. Groq's combination of speed, cost-effectiveness, and broad model support makes it the preferred choice for this research framework.

2.2 Vector Database Configuration

For high-performance vector similarity search, the system employs Qdrant with cosine distance as the similarity metric. Two deployment options are supported, optimized for the `jinaai/jina-embeddings-v2-base-code` embedding model. This research primarily uses Qdrant Cloud, which offers managed infrastructure with strong performance and scalability characteristics. Users can register at `https://cloud.qdrant.io`, create a new cluster with default settings, and retrieve the cluster Uniform Resource Locator (URL) and API key from the dashboard. Upon initialization, the system automatically creates the `code_refactoring_chunks` collection, configured with cosine distance and dynamic embedding dimensions. Qdrant Cloud is particularly advantageous for large-scale workloads as it can efficiently manage datasets containing over 2,200 examples and

more than 15,000 indexed code chunks. Its batch processing is optimized for inserting 200 chunks per operation, which enhances throughput. Additionally, the service handles transient delays with a 60-second timeout threshold, offering a robust, production-ready vector search solution.

2.3 Environment Variables Setup

All sensitive configuration parameters are stored in a `key.env` file located in the project root directory. This file is automatically loaded at runtime using the `python-dotenv` package. At minimum, the `GROQ_API_KEY` must be defined to enable the primary LLM provider, with optional keys available for secondary providers such as OpenAI and Anthropic. If Qdrant Cloud is used, the `QDRANT_URL` and `QDRANT_API_KEY` must also be specified. The system also supports traditional `.env` files and will detect either `key.env` or `.env` automatically during initialization.

2.4 Security and Configuration Requirements

To maintain security best practices, API keys must never be committed to version control. Instead, a `.env.example` template should be provided to indicate required variables, while ensuring that both `key.env` and `.env` are listed in `.gitignore`. Keys should be validated before executing experiments, rotated periodically in production deployments, and ideally stored within a secure environment variable management service. The system automatically validates API keys during startup, performing connection tests for all configured providers. Invalid or expired credentials trigger detailed error messages, and any unavailable APIs are gracefully bypassed in favor of remaining functional providers. This ensures a resilient configuration that can operate even with partial provider availability.

2.5 System Initialization and Validation

During startup, the system conducts comprehensive API key validation for all integrated LLMs and the Qdrant vector store. Users can test LLM connectivity through the `get_default_llm_configs` method and validate Qdrant access via the `QdrantVectorStore` class. A built-in health check system monitors the real-time status of all components, automatically retries transient failures, and logs configuration issues along with recommended solutions. The system also supports interactive status reporting, ensuring users are aware of operational readiness before large-scale processing begins. If certain providers are unavailable, the framework intelligently selects the best available models, applies exponential backoff for rate-limit handling, and continues running with reduced capacity where necessary.

2.6 Deployment Considerations

For workloads exceeding 10,000 code chunks, Qdrant Cloud is strongly recommended due to its scalability. The framework supports multiple API key rotation to maintain throughput under heavy loads and incorporates batch processing for efficient handling of large datasets. Memory usage during embedding generation is minimized through configurable batch sizes, and temporary files or cached embeddings are automatically cleaned up to preserve system resources. The system's built-in monitoring captures logs

at adjustable verbosity levels, tracks operational statistics for performance analysis, and prevents redundant operations during repeated PDF processing. Token consumption is monitored for cost optimization, and the framework dynamically adapts to changes in model availability, maintaining continuous and cost-effective operation in production environments.

3 Experimental Setup and Dataset Configuration

3.1 Project Structure Requirements

The experimental framework requires a specific directory structure to ensure proper component initialization and data processing:

```
python-refactoring-rag/
 config/                    # Configuration management modules
    __init__.py           # Package initialization
    settings.py           # Core system parameters and configurations
    model_configs.py      # LLM provider configurations
 data/                      # Data processing components
    __init__.py           # Package initialization
    generators/           # Dataset generation utilities
       __init__.py
       legacy_code_generator.py
    processors/           # Code and PDF processing
       __init__.py
       code_chunker.py
       pdf_processor.py
    embeddings/           # Vector embedding generation
        __init__.py
        code_embedder.py
 models/                    # Core algorithmic components
    __init__.py           # Package initialization
    llm_providers.py      # Multi-LLM orchestration
    evaluation/           # RAG evaluation metrics
       __init__.py
       rag_evaluator.py
    optimization/         # NSGA-II implementation
        __init__.py
        nsga2_optimizer.py
        nsga2_optimizer_2.py
 services/                  # System orchestration services
    __init__.py           # Package initialization
    rag_service.py        # Main RAG system controller
    vector_store.py       # Qdrant integration
    retrieval_service.py  # Enhanced retrieval logic
    query_processor.py    # Query enhancement
    interactive_service.py # Interactive mode
    interactive_display_manager.py # Enhanced display
```

```
utils/                      # Utility modules
   __init__.py
   logging_utils.py      # Enhanced logging configuration
Inputs/                     # Data directories (case-sensitive)
   Dataset/              # Generated synthetic datasets
      python_legacy_refactoring_dataset.json
      dataset_statistics.json
   Expert_Knowledge/     # PDF knowledge sources
       clean-code-in-python.pdf
       the-clean-coder-a-code-of-conduct-for-professional-programmers.pdf
main.py                    # Full system with optimization
requirements.txt           # Python dependencies
LICENSE                    # MIT License
README.md                  # Comprehensive documentation
.gitattributes            # Git configuration
key.env                    # API credentials configuration
```

3.2 Dataset Generation Configuration

Following the refactoring pattern taxonomy established by Fowler (2018), this research employs a comprehensive synthetically generated dataset of Python code refactoring examples. Dataset generation is performed using the integrated `LegacyCodeGenerator` class:

```
# Direct execution of the generator
python -c "from data.generators.legacy_code_generator import LegacyCodeGenerator;
           generator = LegacyCodeGenerator();
           dataset = generator.generate_dataset(2200)"

# Or run the generator module directly
python data/generators/legacy_code_generator.py
```

3.2.1 Dataset Specification Parameters

The dataset comprises a total of 2,200 original and refactored code pairs, each accompanied by comprehensive metadata to facilitate detailed analysis. Code chunking is performed aggressively, with a maximum of four chunks generated per example using the `process_code_aggressively` method. This results in approximately 8,800 indexed code chunks enriched with semantic metadata, providing a granular level of detail for downstream tasks. Cyclomatic complexity within the dataset spans a range from 5 to 53 prior to refactoring, which is significantly reduced to a range of 3 to 26 after refactoring. On average, complexity is reduced from 18.7 to 10.1, reflecting a 46% improvement in code simplicity and maintainability. The dataset and its statistics are stored in `Inputs/Dataset/python_legacy_refactoring_dataset.json` and `Inputs/Dataset/dataset_statistics.json`, respectively.

The dataset covers ten distinct refactoring patterns based on Fowler (2018), each evenly represented with 220 examples to ensure balanced analysis across different code transformation strategies. These patterns include Extract Method for function decomposition and modularization; Extract Class for object-oriented restructuring; Replace

Conditional with Polymorphism to implement design patterns; Introduce Parameter Object for better parameter management; Replace Loop with Comprehension reflecting Pythonic idioms; Add Type Hints for type safety and Integrated Development Environment (IDE) support; Improve Error Handling for enhanced robustness; Eliminate Code Duplication following the Don't Repeat Yourself (DRY) principle; Improve Naming to boost readability and self-documentation; and Optimize Performance focusing on algorithmic efficiency and resource usage.

To ensure realistic representation, the dataset is distributed across twelve application domains with varying coverage, ranging from 162 to 210 examples per domain. File Management holds the highest coverage with 210 examples, followed by E-commerce (191), Data Processing (190), Machine Learning (189), Scientific Computing and Financial Calculations (each 184), Web Development (183), Game Development (182), System Administration (179), User Management (175), Database Operations (171), and Application Programming Interface (API) Integration with the lowest coverage at 162 examples.

Complementing the dataset, a comprehensive metadata structure integrates multiple code quality metrics. Cyclomatic complexity is measured both before and after refactoring, alongside maintainability scores that range from 3.0 to 9.5, providing a quantitative index of code health. The dataset also tracks 20 distinct categories of code smells with frequency statistics, as well as 26 categories of refactoring benefits along with their occurrence data. Additional context information includes domain classification, function purpose descriptions, and legacy indicators. Temporal metadata, such as generation timestamps and refactoring pattern classification, further enriches the dataset, enabling thorough temporal and categorical analysis.

3.3 Expert Knowledge Base Configuration

The framework incorporates authoritative refactoring literature through advanced PDF processing capabilities with intelligent skip logic to prevent redundant processing.

3.3.1 Required Knowledge Sources

Primary Literature Sources:

1. **Clean Code in Python:** `clean-code-in-python.pdf`
2. **The Clean Coder:** `the-clean-coder-a-code-of-conduct-for-professional-programmers.pdf`

PDF Processing Configuration: The system processes PDF documents located in the case-sensitive input directory `Inputs/Expert_Knowledge/`. It supports only text-based PDFs, excluding scanned images, to ensure reliable text extraction. The primary processing engine is PyMuPDF, enabled via the `use_pymupdf=True` flag, with PyPDF2 configured as a fallback option to maximize robustness. Intelligent processing is implemented through file hash-based duplicate detection, which prevents redundant reprocessing of documents. Text extraction is performed in chunks ranging from 100 to 1,000 characters with a 50-character overlap to maintain context continuity. Additionally, the system automatically detects Python code blocks within the PDFs by recognizing backtick delimiters and indentation patterns. Both textual chunks and embedded code examples are extracted and enriched with metadata for downstream usage.

PDF Processing Optimization: To optimize processing efficiency, the system employs skip logic based on file hash comparisons, automatically detecting and excluding PDFs that have already been processed. Incremental processing ensures that only new or modified PDFs are handled during subsequent runs, significantly reducing unnecessary computation. A comprehensive status tracking mechanism maintains detailed summaries of PDF processing activities, complete with timestamps for traceability. Each extracted chunk is enriched with metadata including the source file name, processing timestamp, and content type, facilitating organized data management. In cases of corrupted or inaccessible PDFs, the system gracefully handles errors to maintain overall pipeline stability without interruption.

3.4 Vector Database Initialization

3.4.1 Embedding Model Configuration

The primary embedding model employed is `jinaai/jina-embeddings-v2-base-code`, specifically optimized for code representations. This model produces vectors with dynamic dimensions, typically around 768. In cases where the primary model is unavailable, fallback options such as `microsoft/codebert-base`, `all-mpnet-base-v2`, and `all-MiniLM-L6-v2` are utilized to maintain system robustness. To improve semantic richness, advanced query expansion techniques incorporate refactoring terminology and contextual information. Embedding generation is performed efficiently in batches of 32, with L2 normalization applied to ensure consistent similarity computations.

Text processing enhancements further enrich embeddings by detecting function and class types, capturing code complexity context, and distinguishing between original and refactored code versions. The system automatically identifies refactoring pattern types based on Fowler (2018) and extracts semantic intent from both code snippets and accompanying descriptions, thereby enabling a deeper contextual understanding.

For vector storage, the Qdrant collection named `code_refactoring_chunks` is configured to use cosine similarity as the distance metric, leveraging the Hierarchical Navigable Small World (HNSW) algorithm to optimize similarity search efficiency. Batch insertion is optimized at 200 chunks per batch, with a connection timeout set to 60 seconds to accommodate large operations. The configuration supports auto-scaling for both local and cloud-based Qdrant deployments, ensuring flexibility and performance under varying workloads.

3.4.2 Retrieval Configuration Parameters

Retrieval from the vector database defaults to a similarity threshold of 0.3, adjustable through the Retrieval Augmented Generation (RAG) Config settings. Queries retrieve the top 5 relevant chunks, which are further refined via enhanced reranking that incorporates multiple scoring factors such as version preference, intent matching, and complexity alignment. The maximum context length per retrieval is capped at 3,000 characters to maintain relevance and efficiency. Rich metadata filters are applied during retrieval, including attributes such as refactoring types, complexity metrics, and code version indicators. To improve query effectiveness, semantic expansion techniques inject Python-specific context into the retrieval queries, enhancing precision and recall for code-related information.

3.5 Multi-Large Language Model Provider Configuration

The system orchestrates five state-of-the-art language models through the Groq provider, employing optimized configuration parameters to ensure robust, efficient, and reliable performance.

3.5.1 Provider Configuration Settings

Key settings across all models include a low temperature value of 0.1, promoting deterministic and reproducible outputs. Each API call enforces a 30-second timeout complemented by retry mechanisms to handle transient failures. Rate limiting is managed via intelligent exponential backoff with 15-second intervals between retries, ensuring compliance with provider constraints while minimizing delays. The system features automatic failover between providers to maintain service continuity and graceful degradation in case of outages. Advanced prompt engineering techniques enable grounded generation with explicit knowledge references and contextual awareness. Additionally, model availability is automatically validated during system initialization to ensure readiness.

3.5.2 Enhanced Prompt Engineering

Prompt design follows a grounded structure to maximize relevance and accuracy. Contextual information is automatically organized by refactoring type based on Fowler (2018), with examples clearly numbered for easy reference. Responses are explicitly grounded in provided examples, requiring mandatory citations to source material. The system enforces a structured response format encompassing analysis, application, recommendations, and benefits to maintain consistency and clarity. When available, user-provided code context is seamlessly integrated into the prompt. To uphold response faithfulness, prompts include explicit instructions that outputs must be based solely on the retrieved examples, preventing hallucinations or unsupported assertions.

3.6 Non-dominated Sorting Genetic Algorithm II Optimization Configuration

3.6.1 Improved Algorithm Parameters

Following the NSGA-II methodology established by Deb et al. (2002), the system employs an enhanced NSGA-II optimization approach tailored for multi-variable optimization. It utilizes a population size of 150 individuals to ensure sufficient diversity, with a minimum threshold of 60 individuals. The optimization runs for 250 generations, although at least 100 iterations are required to guarantee convergence. The crossover operation is performed using Simulated Binary Crossover (SBX) with a probability of 0.9 and a distribution index (η) of 15. Mutation is adaptively applied at a rate of $1/n_{\text{variables}}$ using Polynomial Mutation (PM) with $\eta = 20$. The selection strategy incorporates elite preservation alongside duplicate elimination to maintain solution quality. Constraint handling includes model weight normalization with a tolerance threshold of 10%.

3.6.2 Enhanced Multi-Objective Optimization Framework

The decision variables in the optimization are dynamic, adapting to the number of models in the ensemble. These variables include ensemble weights for each model, which are

normalized to sum to one, a similarity threshold ranging between 0.1 and 0.8, a Top-K retrieval multiplier varying from 1.0 to 3.0, and four importance weights corresponding to core quality metrics.

The framework optimizes four competing objectives simultaneously. The primary objective is overall performance, which maximizes a weighted combination of core quality metrics. Consistency aims to enhance model reliability and reduce output variance. Complexity is minimized to favor simpler and more interpretable ensembles, while efficiency targets retrieval parameter optimization by minimizing deviations from their optimal values.

3.6.3 Comprehensive Metrics Integration

To comprehensively evaluate solutions, multiple metrics are integrated into the optimization process. Answer relevance is assessed through enhanced semantic alignment with code-specific terminology. Faithfulness is verified by grounding knowledge at the statement level. Response completeness is evaluated based on the presence of detailed explanations and code examples. The CodeBLEU score established by **?**, a specialized Bilingual Evaluation Understudy (BLEU) variant incorporating Abstract Syntax Tree (AST) awareness, measures code similarity. Efficiency is quantified by latency (inversely weighted) and token usage, ensuring optimized resource consumption.

3.7 Evaluation Framework Configuration

3.7.1 Enhanced Retrieval Augmented Generation-Specific Evaluation Metrics

The evaluation framework prioritizes core quality metrics, which collectively contribute 80% of the total evaluation weight. These include Answer Relevance, measured through multi-dimensional relevance scoring enhanced by concept detection, Faithfulness assessed via statement-level grounding verification, Response Completeness gauged using structured completeness indicators alongside code example detection, and the CodeBLEU Score which employs AST-aware similarity measurements considering both token-level and structural matching as established by **?**. The remaining 20% weight is allocated to efficiency metrics, with Latency Optimization targeting response times under a 5-second threshold and Token Usage focusing on maintaining efficiency within a 2,000-token limit.

3.7.2 Comprehensive Test Case Configuration

The system employs a comprehensive suite of 10 test cases designed to cover all major refactoring patterns established by Fowler (2018). These scenarios include complex function decomposition under Extract Method, object-oriented restructuring with Extract Class, polymorphism implementation for Replace Conditional, long parameter list management via Parameter Object, Pythonic loop transformation in Loop Comprehension, type safety improvements with Type Hints, robustness enhancements through Error Handling, configuration-driven duplication removal for Code Duplication, readability gains by Naming Improvement, and algorithmic efficiency advances in Performance Optimization. Each test case is evaluated across all five LLM models, benchmarked against expert-crafted refactoring solutions complete with detailed explanations. Evaluation integrates a combination of software engineering metrics and advanced natural language

processing techniques, supported by statistical validation comparing before and after optimization states. Token usage is closely monitored throughout to facilitate cost optimization analysis.

3.7.3 Improved Multi-Criteria Decision Making

For solution selection, the framework applies a Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS)-based multi-criteria decision-making approach. Objective weighting is distributed as 40% for Performance, 30% for Consistency, 20% for Complexity, and 10% for Efficiency. Normalization of metric values is performed using min-max scaling to ensure comparability across objectives. The ideal point analysis identifies the optimal solution based on distance metrics, while compromise selection balances the trade-offs between competing objectives. This rigorous evaluation configuration supports reproducible validation of multi-objective optimization results, providing detailed before and after comparisons alongside statistical significance analysis across all measured dimensions.

References

Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii, *IEEE Transactions on Evolutionary Computation* **6**(2): 182–197.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*, 2nd edn, Addison-Wesley Professional.