

Mohammed Sarwar

Output Diversity in Generative AI for Source Code

B.Sc. Computer Science

Declaration

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work.

Name: Mohammed Shajalal Sarwar _____

Date: 21/03/2025 _____

Abstract

This dissertation investigates whether Large Language Models (LLMs) can generate context-specific cache algorithm implementations that perform optimally under different access patterns, providing an alternative to traditional genetic improvement approaches. The study compares minimal and detailed prompting techniques to generate LRU, FIFO, and LIFO cache implementations, evaluating their performance against cyclic, random, and locality-based access sequences. Through experimental testing, cache hit ratios were measured to determine how effectively LLM-generated implementations adapt to different usage contexts. The results demonstrate that prompting techniques significantly influence performance characteristics, with "minimal" implementations often outperforming more complex "detailed" counterparts in specific scenarios. LIFO implementations showed strong performance across all sequence types, with the minimal variant excelling particularly in high-temporal locality situations. This research contributes to understanding how emergent systems concepts can be applied to software optimisation, demonstrating that appropriately prompted LLMs can generate context-optimised implementations without requiring the iterative mutation processes of genetic improvement, thus offering a promising pathway for efficient, context-aware software generation.

Contents

1	Introduction	4
1.1	Problem Statement	4
1.2	Aims and Objectives	5
1.3	Structure of the Report.....	6
1.4	Development (Methodology Overview)	6
2	Background	8
2.1	Emergent Systems in Software Engineering.....	8
2.2	Genetic Improvement.....	9
2.3	Cache Replacement Algorithms.....	10
2.3.1	Least Recently Used (LRU)	10
2.3.2	First-In-First-Out (FIFO).....	11
2.3.3	Last-In-First-Out (LIFO)	11
2.3.4	Performance Sensitivity and Cache Behaviour	11
2.3.5	Summary.....	11
2.4	Large Language Models for Code Generation.....	12
2.4.1	Applications of LLMs in Code Generation	12
2.4.2	Mechanisms for Code Generation	12
2.4.3	Challenges in LLM-Based Code Generation	13
2.4.4	Validation and Evaluation of LLM-Generated Code.....	13
2.4.5	Advanced Techniques and Self-Collaboration	13
2.4.6	Future Potential and Limitations	13
2.5	Prompt Engineering Approaches	14
2.5.1	Iterative and Interactive Nature of Prompt Engineering.....	14
2.5.2	Prompt Patterns for Software Engineering	14
2.5.3	Empirical Benefits of Tailored Prompting.....	15
2.5.4	Retrieval-Based Prompting.....	15
2.5.5	Advanced Structured Prompting Techniques	15
2.5.6	Additional Prompting Strategies	15
2.5.7	Ensuring High-Quality Outputs.....	15
2.5.8	Conclusion.....	16

3 Methodology	17
3.1 Experimental Design.....	17
3.1.1 Prompting Techniques (Minimal vs. Detailed)	19
3.2 Implementation of Cache Algorithms.....	22
3.2.1 Least Recently Used (LRU) Cache	22
3.2.2 First-In-First-Out (FIFO) Cache.....	23
3.2.3 Last-In-First-Out (LIFO) Cache	24
3.2.4 Common Features and Consistency	25
3.2.5 Challenges and Edge Cases	25
3.2.6 Summary.....	26
3.3 Test Sequence Generation.....	26
3.3.1 Cyclic Sequence Generation.....	26
3.3.2 Random Sequence Generation	27
3.3.3 Locality-Based Sequence Generation.....	27
3.3.4 Control Parameters and Consistency	28
3.3.5 Evaluation Framework.....	29
3.3.6 Summary.....	29
3.4 Evaluation Metrics and Performance Measures.....	29
3.4.1 Cache Hit Ratio	30
3.4.2 Multiple Runs for Consistency	30
3.4.3 Why Hit Ratio?.....	30
3.4.4 Potential Extensions	31
3.4.5 Summary.....	31
4 Results	32
4.1 Comparison of Prompting Approaches.....	32
4.2 Analysis of Cache Implementation Quality	34
4.2.1 Functional Correctness	34
4.2.2 Adherence to Requirements	36
4.2.3 Efficiency and Complexity	37
4.2.4 Summary of Implementation Quality	39
4.3 Summary of Findings.....	39
5 Conclusion	41
5.1 Review of Aims.....	41
5.1.1 Changes from Proposal.....	41
5.2 Learning Outcomes	42
5.3 Future Work	43
5.4 Final Remarks	43

6 Appendix	47
7 Project Proposal	48
8 Additional Material	52

Chapter 1

Introduction

The aim of this dissertation is to investigate the potential of Large Language Models (LLMs) to generate context-specific caching algorithm implementations optimised for diverse usage patterns. Traditional approaches to optimizing source code, such as Genetic Improvement (GI), involve evolutionary techniques to iteratively refine software performance. However, these methods can be computationally intensive and require substantial domain knowledge. In contrast, LLMs offer an innovative alternative by leveraging their extensive pre-trained knowledge and adaptability to rapidly produce varied algorithmic implementations through natural language prompts.

This study systematically assesses how well different prompting strategies direct LLMs to generate cache implementations with high performance. Furthermore, it investigates the ways in which these created implementations support emergent software systems, which are defined by characteristics and actions that result from interactions between smaller parts. To find out if LLM-driven solutions can equal or outperform conventionally optimised implementations, the study will evaluate the resulting code across a variety of access patterns. Future software engineering approaches may benefit from the findings of this work, especially in situations where quick, flexible, context-sensitive, and emergent code optimisation is essential.

1.1 Problem Statement

Software systems often need to adapt to different operational contexts to maintain optimal performance. This aligns with the concept of emergent systems, where different solutions naturally emerge as optimal for different environmental conditions. Traditional approaches to code optimisation, such as genetic improvement, typically rely on iterative mutation processes that can be time-consuming and computationally expensive.

The emergence of powerful Large Language Models presents an opportunity to generate code implementations directly from specifications. However, it remains unclear whether these models can produce not just functional code, but implementations that are optimised for specific

operational contexts.

1.2 Aims and Objectives

The overall aim of this research is to investigate the capabilities of Large Language Models (LLMs) in generating effective, context-sensitive cache replacement algorithms. Given the increasing importance of automated software generation and optimisation, this study seeks to explore whether generative AI can not only replicate human-coded cache algorithms but also optimise these implementations for specific operational contexts. By assessing the impact of various prompting strategies and comparing the effectiveness of LLM-generated code with traditional optimisation methods such as genetic improvement, the study aims to provide valuable insights into the potential of generative AI as a tool for adaptive software development.

The specific objectives of this research are:

- To investigate whether LLMs, specifically the CodeLlama model from the Ollama framework, can produce cache implementations (LRU, FIFO, LIFO) that are not only functionally correct but optimised for specific input sequences and access patterns.
- To develop and systematically compare different prompting approaches ranging from minimal information to detailed, explicit specifications and to determine how these prompting strategies influence the quality, efficiency, and consistency of generated implementations.
- To rigorously test the generated cache implementations using controlled, varied access patterns that include cyclic, random, and locality-based sequences. These tests aim to reflect realistic and challenging operational scenarios that caches typically encounter.
- To quantitatively measure and compare the cache hit ratios achieved by each implementation, using these metrics to evaluate performance and effectiveness in various contexts. Hit ratio will serve as the primary indicator of caching efficiency and suitability.
- evaluate whether LLM-based generation can serve as an alternative to GI, exploring whether generative AI can provide a comparable or superior alternative in terms of performance, computational overhead, and practical usability.
- To evaluate the broader effectiveness of LLMs as potential alternatives or complements to evolutionary computation methods in software engineering, particularly in terms of reducing computational cost, simplifying optimisation processes, and automating code generation without extensive domain-specific input.

1.3 Structure of the Report

The remainder of this dissertation is organized as follows:

- **Chapter 2 (Background):** Presents essential concepts and literature, covering emergent software systems, genetic improvement, and large language models for code generation. It also introduces the core ideas of prompt engineering.
- **Chapter 3 (Methodology):** Details the experimental framework, including how we generate cache replacements using minimal and detailed prompts, how we implement the cache algorithms (LRU, FIFO, LIFO), and how we measure performance through cyclic, random, and locality-based test sequences.
- **Chapter 4 (Results):** Discusses quantitative findings, comparing how minimal and detailed prompting influence correctness and performance metrics across multiple runs. The chapter focuses on key observations around hit ratios and explores how the LLM outputs align with or deviate from hand-coded or baseline implementations.
- **Chapter 5 (Conclusion):** Summarizes major observations, reflects on the project's aims and how they evolved, and suggests opportunities for future work, such as incorporating more advanced cache strategies or integrating LLM outputs with genetic improvement loops.

This structure ensures clarity and coherence, guiding the reader from foundational knowledge through the core experiments and concluding with final remarks and prospective research directions.

1.4 Development (Methodology Overview)

The development process undertaken in this project involves several key steps, designed to evaluate how Large Language Models (LLMs) can generate context-specific caching implementations. In summary:

1. Prompting Strategies:

We craft both minimal and detailed prompts to compare how variations in instruction depth affect the generated code. Minimal prompts supply sparse information, potentially encouraging more creative but less reliable solutions. Detailed prompts explicitly specify data structures and edge-case handling, leading to more stable, if less flexible, implementations.

2. Cache Algorithms and Test Harness:

We focus on three classic cache replacement policies: Least Recently Used (LRU), First-In-First-Out (FIFO), and Last-In-First-Out (LIFO). Each algorithm is defined in Python, using a common interface that allows for consistent performance comparisons. A custom test harness exercises these caches under controlled access sequences, capturing metrics such as cache hits and misses.

3. Access Patterns and Repeated Trials:

We simulate realistic workloads via cyclic, random, and locality-based sequences. By design, these patterns illuminate different strengths or weaknesses in each cache strategy. Further, to mitigate randomness in both sequence generation and LLM outputs, we repeat experiments multiple times and average the results, ensuring robust hit-ratio evaluations.

4. Analysis of Prompt-Generated Code:

We examine the functional correctness, code efficiency, and adherence to requirements (like eviction policy) in both minimal- and detailed-prompt outputs. This step highlights emergent properties that may arise when minimal prompts permit more open-ended solutions, as well as the enhanced consistency produced by structured instructions.

5. Comparative and Statistical Assessment:

The hit ratios from repeated runs are aggregated to reveal which combinations of cache policy and prompt style offer the best performance for each pattern type. We compare these outcomes to baseline or hand-coded references where available, and discuss how they may serve as a non-iterative alternative to approaches like genetic improvement.

Chapter 2

Background

2.1 Emergent Systems in Software Engineering

In software engineering, "emergent systems" refers to architectures and behaviours that develop dynamically during runtime as a result of the ongoing interaction of many tiny, fine-grained software modules [1]. Emergent systems use techniques like continuous self-assembly, runtime observation of operational and environmental contexts, and online learning, which are different from traditionally built software, which mostly depends on static, design-time judgements. These features allow software systems to automatically determine and implement the best module configurations to satisfy predetermined performance, dependability, or efficiency targets [2].

A core principle of emergent software systems is their ability to autonomously explore multiple behavioural variants. For instance, in real-time production settings, emergent systems can dynamically test and assess different sorting techniques, caching algorithms, and data retrieval tactics. Systems proactively modify their internal architecture to address shifting operational conditions or evolving workloads by continuously observing the results of each module combination through this iterative and reflective process [3]. Over time, this capability can uncover beneficial and high-performing configurations that developers may not have anticipated or explicitly programmed during initial system design [4].

This ongoing reassembly and adaption process is usually driven by machine learning algorithms, especially those that use unsupervised learning approaches, reinforcement learning, or genetic improvement. By automatically determining when and how to reorganise its component modules without requiring a great deal of human interaction, these algorithms enable the software to handle complexity [5]. Because of their adaptive intelligence, emergent systems are especially useful in situations when manual management and conventional adaption mechanisms are insufficient, such as cloud settings, edge computing platforms, and pervasive computing scenarios.

When applied in distributed systems or in system-of-systems (SoS) contexts, emergent con-

cepts help coordinate local adaptation decisions across multiple interacting subsystems or computational nodes [6]. Here, a critical challenge arises from balancing local optimisation decisions such as module selection or resource allocation against overarching global system goals. Decisions made by individual subsystems, although beneficial locally, may inadvertently degrade global system performance or even compromise critical objectives such as security, reliability, and safety [7]. Therefore, considerable research efforts focus on mechanisms for global oversight and conflict resolution to harmonize subsystem behaviours with collective system objectives.

Recent developments in emergent software engineering also emphasize the need for demanding regulation of emergent behaviours. System engineers can reduce the risks associated with unexpected emergent phenomena by using formal models to forecast, validate, and verify that runtime adaptive behaviours stay within acceptable operational constraints [8]. Modelling adaptive system states, establishing restrictions for runtime adaptations, and creating verification methods that continually track adaptive behaviours in real-time are some ways to formalise emergent behaviour.

Emergent software engineering ultimately represents a major shift from conventional static, deterministic systems in favour of highly adaptive, dynamic frameworks. In complex and quickly evolving computing environments, emergent approaches promise to greatly improve system resilience, adaptability, and long-term sustainability by directly integrating runtime exploration, intelligent adaptation, and continuous improvement into software systems [1, 2].

2.2 Genetic Improvement

Genetic Improvement (GI) is a method that automatically improves current software using evolutionary search, usually optimising for goals like energy efficiency, memory use, or runtime performance [9]. GI methodically mutates and evolves program code to find advantageous adjustments, in contrast to traditional software optimisation, which frequently depends on human-driven refactoring or compiler approaches [10]. Early GI techniques largely referenced Genetic Programming [9] and focused on code repair and program specialisation.

Recent research by [11] offers a thorough review of the discipline with a focus on Genetic Improvement (GI), spawned by the convergence of several software engineering subdisciplines: program transformation, program synthesis, and genetic programming. GI takes advantage of the widespread availability of open-source code using it as "genetic material" to allow for evolution of modifications. In contrast to program synthesis that constructs programs starting at a foundational level or to program transformation that uses pre-existing semantics-preserving rules, GI searches in a large combinatorial space of software options and scores it using a fitness function that balances functional correctness with desired optimisations (e.g., performance, space efficiency, etc.).

By including Large Language Models (LLMs) into the evolutionary process, recent studies

have broadened the scope of GI [12, 13, 14]. According to evolutionary fitness functions that evaluate advances against performance or accuracy criteria, these LLM-based methods make use of pre-trained models that can generate or alter source code [15, 16]. Consequently, the combined GI–LLM systems may automatically filter out hallucinated or incorrect code through evolutionary selection, while introducing more complicated and objectively tailored program changes than standard GI operators [12, 15]. While some methods use robust offline verification compiling and testing improved variants to make sure they match required requirements [15] other methods, for example, encourage LLMs to modify code in ways that directly target efficiency or memory usage [13].

Importantly, the collaboration between GI and LLMs also tackles a long-standing issue with code validity: LLMs are more likely to suggest "natural" alterations that compile and execute, whereas random low-level edits can readily break compilation [12]. In fact, GI can use LLM's code generation capabilities, such as its understanding of design patterns and idioms, to provide better results faster [14]. To preserve just those solutions that truly achieve the intended goals, the evolutionary approach provides strong filtering by comparing candidate improvements to test suites or performance benchmarks [16, 15]. Researchers and practitioners alike can benefit from this new line of inquiry, which provides a route to automated, goal-specific software optimisation that is more successful and efficient than GI or big language models alone. [10, 12].

2.3 Cache Replacement Algorithms

Cache replacement policies are responsible for controlling and managing constrained storage resources by deciding what to keep and what to discard upon reaching maximum cache size. How good a replacement policy is has direct impacts on system performance in terms of response times, throughput, and overall efficiency [17].

2.3.1 Least Recently Used (LRU)

The Least Recently Used (LRU) strategy operates by deleting entries that are least used for the longest time interval, on the assumption that previously used information is more likely to be required again in the near future. Research has found that LRU has been found to perform optimally in those situations with large locality of time [17]. However, practical implementation of LRU can be very expensive in terms of resources since it requires significant monitoring overhead.

In a bid to minimize this overhead, pseudo-LRU mechanisms have been proposed. Li et al. [18] presented a refined form of pseudo-LRU that aims to approximate true LRU while reducing hardware complexity dramatically. This mechanism employs a mechanism of second chances to enhance efficiency in tracking usage and in effect deliver near-optimal miss rates

with low overhead. A simple example given by Ali [19] serves to expound on the efficiency of LRU with a given reference string and shows how efficient it is with identifying and getting rid of least recently used objects.

2.3.2 First-In-First-Out (FIFO)

FIFO is a quite simple replacement policy that emphasizes the eviction of the entry that has been in the cache the longest, regardless of the access frequency. While FIFO is easy to implement, it fails to consider the access pattern related to data and hence performs poorly in applications with high temporal locality [17].

A hybrid implementation of note, blending attributes of FIFO and LRU, was explored by Lee and Hong [20]. Their "pseudo-FIFO" LRU design achieved the simple nature of FIFO while still implementing a mechanism to mimic LRU-like eviction procedures. The result was a design that was hardware-efficient and improved the performance of standard FIFO, particularly in workloads with moderate temporal locality of data access. Ali [19] provides concrete examples of FIFO operation, explaining cache state transitions step by step, illustrating both the simplicity and the limitations of FIFO.

2.3.3 Last-In-First-Out (LIFO)

In essence, the LIFO policy acts as a stack, removing the most recently added item first. Since recently added data can frequently be among the most pertinent, LIFO policies are rarely used because of their counterintuitive eviction technique [17]. However, LIFO can work well in very specific cases, like when there are precise access patterns and newly added things are less likely to be reused. Using thorough examples, Ali [19] further explains the LIFO algorithm's workings and highlights possible situations in which LIFO can be effective despite its generally sparse application.

2.3.4 Performance Sensitivity and Cache Behaviour

Cache replacement policy performance can significantly fluctuate depending on workload characteristics and initial cache states. Sandberg et al. [21] provided comprehensive analysis into the sensitivity of cache replacement policies, including LRU and FIFO. They demonstrated how the same policy can yield drastically different performance depending on the cache's initial conditions. Their research underlines the importance of considering workload-specific contexts when evaluating or deploying cache algorithms.

2.3.5 Summary

Depending on the application environment, cache replacement policies like LRU, FIFO, and LIFO each offer unique benefits and disadvantages. Because it accurately approximates tem-

poral locality, LRU is frequently chosen for general-purpose scenarios; however, hybrid and modified techniques, such as pseudo-LRU [18] and pseudo-FIFO [20], provide attractive alternatives that strike a balance between complexity and performance. The practical ramifications and operational properties of these basic algorithms are further elucidated by Ali's [19] comprehensive examples. It is essential to comprehend the intrinsic performance sensitivity of these algorithms [21] in order to choose suitable cache techniques that are suited to particular workload characteristics.

2.4 Large Language Models for Code Generation

Large Language Models (LLMs) have significantly transformed software engineering practices by automating the generation of code snippets from natural language prompts [22]. Models such as GPT-4 and ChatGPT utilize transformer architectures trained on massive, heterogeneous datasets encompassing natural language text and source code. This extensive training facilitates deep semantic and syntactic understanding, empowering sophisticated code generation capabilities and leading to enhanced productivity and creativity in software development processes [23].

2.4.1 Applications of LLMs in Code Generation

The application of LLMs in software engineering covers a wide range of tasks, including code completion, automatic program repair, and code summarization [24, 25]. GitHub Copilot, which leverages OpenAI's Codex, demonstrates high accuracy in generating contextually relevant code snippets, significantly improving developer efficiency [26]. Jiang et al. [22] illustrate how self-planning methods, when integrated with ChatGPT, substantially improve performance in complex coding scenarios, surpassing traditional direct generation methods. Bhattacharya et al. [27] further emphasize the capabilities of Code LLMs specifically designed for generating clear and concise code explanations, highlighting their versatility across various software engineering contexts.

2.4.2 Mechanisms for Code Generation

The primary mechanism through which LLMs generate code involves token-level predictions, informed by learned syntax and semantics derived from comprehensive training datasets [28]. Attention mechanisms within transformer models provide contextual awareness, allowing LLMs to handle unseen scenarios and APIs effectively, often displaying emergent behaviours beyond their explicit training examples [28, 22]. These attention-driven capabilities have been further enhanced by multi-agent self-collaboration frameworks, where multiple LLM agents, each acting as distinct experts, collaboratively generate and refine code solutions through dynamic role-based interactions [28].

2.4.3 Challenges in LLM-Based Code Generation

Despite their notable strengths, LLMs encounter several critical challenges in the code generation domain. Ensuring the correctness and integrity of generated code remains particularly challenging, especially for novice developers unfamiliar with sophisticated debugging strategies [26]. Another prevalent issue is model hallucinations, where LLMs produce plausible yet incorrect or irrelevant code snippets, posing significant barriers to practical deployment and requiring advanced validation methods [29]. Domain-specific code generation presents additional complexity, as effective performance typically necessitates explicit API knowledge integration into prompts [25]. Moreover, Du et al. [30] highlight the complexity associated with class-level code generation, emphasizing its unique contextual dependencies and the resulting performance degradation in existing LLMs compared to simpler method-level generation.

2.4.4 Validation and Evaluation of LLM-Generated Code

Effective validation and evaluation frameworks are critical for LLM-generated code. Tang [26] underscores the need for comprehensive validation strategies, proposing multi-level explanations and procedurally prompted editing to enhance developer comprehension and facilitate better code integration. Wang and Chen [24] highlight the existing research gap between the adoption of LLMs in practical software engineering scenarios and rigorous evaluation methodologies, advocating for more thorough assessments of code correctness, reliability, and security. Jiang et al. [22] additionally propose incremental implementation strategies, advocating structured planning and iterative refinement to enhance the maintainability and robustness of generated code.

2.4.5 Advanced Techniques and Self-Collaboration

Recent advances in LLM-based code generation emphasize the importance of collaborative and iterative refinement processes. Dong et al. [28] present a self-collaboration approach involving multiple ChatGPT instances assigned distinct roles such as analyst, coder, and tester. This division of labor effectively decomposes complex tasks, significantly improving code quality through systematic role-specific interactions and adherence to software development best practices. Bhattacharya et al. [27] further highlight how instruction fine-tuning and few-shot prompting can enhance model effectiveness, particularly in code summarization tasks, demonstrating the broader applicability and adaptability of collaborative approaches.

2.4.6 Future Potential and Limitations

The future development of LLM-based code generation holds considerable promise, particularly through fine-tuning models on specialized, domain-specific datasets, thus increasing

precision and efficacy [25]. Zhao et al. [29] advocate for enhanced explainability methodologies to provide greater transparency into LLM decision-making processes, addressing the challenges posed by opaque model behaviours. Ethical concerns such as biases, accountability, and fair usage of automated coding tools are critical considerations that must be systematically addressed as LLMs become integral to professional software development environments [27, 22]. Moreover, Du et al. [30] emphasize the urgent need for more sophisticated benchmarking frameworks that reflect real-world coding scenarios, enabling deeper insights into model capabilities and practical applications. Additionally, advancing research into self-debugging and autonomous correction capabilities could substantially improve the reliability and trustworthiness of these systems.

2.5 Prompt Engineering Approaches

Prompt engineering is essential for fully leveraging the capabilities of Large Language Models (LLMs) such as GPT-4 and ChatGPT, particularly in automated software engineering and code generation. At its core, prompt engineering involves systematically crafting input prompts to guide and refine LLM outputs, effectively programming AI through structured natural language interactions [31, 32]. Unlike conventional software optimisation methods such as Genetic Improvement, prompt engineering exploits embedded knowledge within LLMs, removing the necessity for extensive parameter re-training or fine-tuning [32].

2.5.1 Iterative and Interactive Nature of Prompt Engineering

A critical feature of prompt engineering is its iterative and interactive nature, characterized by continual improvement through user engagement and feedback [33]. Advanced prompting techniques, particularly Chain-of-Thought (CoT), explicitly direct LLMs through step-by-step reasoning processes. This approach significantly enhances logical reasoning and algorithmic accuracy, critical for generating robust software implementations such as cache algorithms [31, 32, 34].

2.5.2 Prompt Patterns for Software Engineering

White et al. [35] introduced a comprehensive catalogue of prompt patterns tailored specifically for software engineering tasks, including system design, refactoring, and code quality improvement. Patterns like the "Requirements Simulator" enable iterative requirement development, increasing clarity and accuracy, which is essential for complex, context-sensitive software implementations.

2.5.3 Empirical Benefits of Tailored Prompting

Empirical research by Shin et al. [36] highlights the advantages of tailored prompting methods over traditional model fine-tuning for software-related tasks, such as code summarisation and optimisation. Their conversational prompting approach, refining prompts based on conversational feedback, demonstrated accuracy improvements of 15.8% to 18.3% across various code-related tasks, reinforcing the value of deliberate prompt design.

2.5.4 Retrieval-Based Prompting

The retrieval-based prompting approach, exemplified by the work of Nashid, Sintaha, and Mesbah [37], further enhances few-shot learning scenarios in code generation by dynamically retrieving relevant examples. Their Code Example Demonstration Automated Retrieval (CEDAR) framework, leveraging embedding-based retrieval techniques, achieved a remarkable 333% improvement in generating test assertions. This strategy aligns closely with the current study's emphasis on context-specific implementations.

2.5.5 Advanced Structured Prompting Techniques

Advanced structured prompting methods, such as the Query Transformation Module (QTM), further illustrate relevant methodologies by breaking down complex software engineering prompts into explicit objectives and key details, enhancing LLM comprehension. Park et al. [38] demonstrated an 11.46% performance increase through structured query transformations, validating structured prompting's effectiveness—a central hypothesis in this research.

2.5.6 Additional Prompting Strategies

Additional prompting strategies relevant to software engineering include Program-of-Thought (PoT), Tree-of-Thought (ToT), and Self-Consistency. ToT, for instance, promotes hierarchical reasoning by structuring intermediate steps as a tree, substantially improving performance in structured algorithmic tasks and logical reasoning [32, 39]. These techniques underline the methodical nature of prompt engineering, highlighting its critical role in generating accurate, efficient, and context-sensitive algorithmic solutions.

2.5.7 Ensuring High-Quality Outputs

Ensuring high-quality outputs from LLM-generated code requires comprehensive prompt engineering practices, including detailed descriptions, clear context specifications, and structured reasoning templates [40]. Such systematic approaches reduce potential misalignments between human-written prompts and LLM interpretations, thereby enhancing the functional correctness, efficiency, and adaptability of generated implementations.

2.5.8 Conclusion

In summary, prompt engineering encompasses diverse methodologies structured, conversational, retrieval-based, and hierarchical all contributing significantly to enhanced LLM performance in software engineering contexts. The continuous evolution and refinement of these techniques suggest that prompt engineering represents a viable and potentially superior alternative to traditional iterative optimisation methods like genetic improvement, particularly in rapidly generating effective, context-sensitive software implementations.

Chapter 3

Methodology

3.1 Experimental Design

This study evaluates how effectively Large Language Models (LLMs), specifically CodeLlama from Ollama, can generate context-specific cache algorithm implementations that optimise performance for different access patterns. Here, we examine whether LLM-based code generation can function as a conceptually comparable but straightforward substitute for Genetic Improvement (GI), which has historically been employed to progressively improve code. To put it another way, instead of executing a GI loop, our experimental framework systematically assesses the effects of various prompting tactics (minimal vs. thorough) on the efficiency and functional accuracy of the automatically generated cache implementations.

The experiment was structured around the following key independent variables:

1. **Caching Algorithm:** Three fundamental caching strategies were implemented and evaluated:
 - **FIFO (First-In-First-Out):** Evicts the oldest entry first when the cache reaches capacity, maintaining a simple chronological eviction policy.
 - **LIFO (Last-In-First-Out):** Removes the most recently added entry when space is needed, effectively functioning as a stack.
 - **LRU (Least Recently Used):** Tracks access recency and evicts the least recently accessed item, optimizing for temporal locality patterns.
2. **Prompting Technique:** Two distinct prompting approaches were employed:
 - **Minimal Prompting:** Provides only essential functional requirements with minimal implementation guidance, allowing the LLM to determine implementation details independently.
 - **Detailed Prompting:** offers detailed guidelines that address edge case handling, suggestions for data structures, and thoughts on optimisation.

3. Access Pattern Type: Three distinct sequence patterns were used to evaluate context-specific performance:

- **Cyclic Sequences:** repeating request patterns that assess the cache's capacity to identify and adjust to regular cycles.
- **Random Sequences:** unpredictable patterns of requests that test how resilient caching algorithms are in chaotic environments.
- **Locality-Based Sequences:** Designs exhibiting temporal and spatial locality, reflecting behaviour found in real-world applications.

Performance was measured through the primary dependent variable:

- **Cache Hit Ratio:** The percentage of cache requests served successfully from the cache without requiring back-end retrieval, calculated as $\frac{\text{hits}}{\text{total requests}} \times 100\%$. This metric directly indicates how effectively each implementation optimises for the specific access pattern.
- Each implementation was tested with identical request sequences within each type of access pattern, controlling for sequence variability. To reduce randomness and improve result consistency, each test scenario was executed **10 times** using the `multiple_run_test()` function. The average hit ratio across these trials was used as the final metric for comparison.

To ensure experimental validity and reliability, several control measures were implemented:

1. Each implementation was tested with identical request sequences within each type of access pattern, controlling for sequence variability.
2. All tests were conducted in the same computational environment to eliminate system-level differences.
3. Due to the non-deterministic nature of LLM outputs, each prompting approach was executed multiple times, and results were averaged to mitigate random variation.
4. Cache size was standardized across all implementations to ensure fair comparison of eviction policy effectiveness.
5. Test sequences were carefully designed to include edge cases such as cache overflow conditions, repeated key access patterns, and varying temporal distributions.

Statistical analysis was performed on the hit ratio results to determine the significance of differences between implementations and to identify correlations between prompting strategies and performance characteristics across different access patterns.

3.1.1 Prompting Techniques (Minimal vs. Detailed)

This study employed two distinct prompting approaches to generate cache algorithm implementations using CodeLlama using the Ollama framework: **minimal prompting** and **detailed prompting**. These approaches were systematically compared to understand how the level of specificity in prompts influences the quality and performance characteristics of generated code.

The core motivation behind comparing prompting strategies was to assess whether LLMs could independently generate context-optimised code (in the case of minimal prompting) or whether explicit, structured instructions (in the case of detailed prompting) were necessary to achieve optimal performance. This distinction is particularly relevant when evaluating whether LLM-based code generation can serve as an alternative to genetic improvement techniques, which rely on iterative refinement rather than single-step instruction.

Minimal Prompting

Minimal prompting involved providing concise, essential requirements with limited implementation guidance. This approach deliberately omitted specific details about data structures, edge-case handling, or optimisation strategies. The intent was to allow the LLM maximum freedom to determine implementation details independently, potentially encouraging novel or unexpected solutions.

Example Minimal Prompt (LIFO Cache)

```
create a python class for a LIFO cache
write a lifo cache class in python, it should store (key, value) pairs,
where the keys are strings,
get should return the value if found else none,
put should remove the last item if at capacity,
if the value exists replace the old value
```

This prompt specifies the basic functionality required (a LIFO cache) but does not dictate specific implementation details beyond requiring a list for storage. It leaves the LLM to determine how to efficiently search for keys, implement the eviction policy, and handle edge cases.

Rationale for Minimal Prompting Minimal prompts were intended to test the LLM's ability to:

- Infer appropriate data structures and algorithmic patterns.
- Adapt to different operational contexts with minimal human guidance.
- Produce novel or creative solutions not directly specified in the prompt.

Challenges with Minimal Prompting

- Lack of specific guidance could lead to functional errors or inefficient implementations.
- The LLM might overlook edge cases, resulting in inconsistencies.
- Greater variance between runs due to increased creative freedom.

Detailed Prompting

In contrast, detailed prompting provided comprehensive specifications, including explicit guidance on data structures, implementation strategies, and edge-case handling. This approach was designed to guide the LLM toward specific, well-established implementation patterns.

Example Detailed Prompt (LRU Cache)

Fix the following issues in the LRUCache class:

- In the get() method, use `value = self.cache.pop(key)` to define the value,
- After removing the key, reinsert it at the end to mark it as most recently used,
- In the put() method, if the key already exists, remove it first using `pop()` before updating it,
- If the cache is at capacity, use `popitem(last=False)` to remove the least recently used item,
- Change the condition from `==` to `>=` to prevent capacity overflow,
- Remove any extra tracking variables (like keys or deque) rely solely on `OrderedDict`,
- Ensure that `size()` returns the number of elements in the cache directly from the `OrderedDict`.

The detailed prompting approach provided clear implementation directives, including specific method signatures, variable naming conventions, data structure recommendations (such as using `OrderedDict`), and explicit instructions for handling edge cases like removing existing keys and managing capacity constraints.

Rationale for Detailed Prompting Detailed prompts were intended to test the LLM's ability to:

- Follow explicit programming guidelines and best practices.
- Handle complex operational requirements consistently.
- Generate reliable and functionally correct code.

Challenges with Detailed Prompting

- Reduced creative flexibility might lead to rigid, less adaptive implementations.
- Greater prompt complexity could confuse the LLM, leading to incomplete or misaligned output.
- Longer, more detailed prompts increase the risk of prompt truncation or misinterpretation.

Rationale and Implementation

The dual-prompting approach was designed to investigate whether LLMs could generate not just functionally correct cache implementations, but ones optimised for specific access patterns without explicit instruction. This addresses a fundamental question: can LLMs serve as an alternative to genetic improvement techniques by implicitly understanding optimisation requirements from context?

Several prompt engineering considerations were applied:

1. **Consistency in Requirements:** Both minimal and detailed prompts maintained the same functional requirements for each cache type (FIFO, LIFO, LRU), ensuring that differences in performance could be attributed to the prompting approach rather than divergent requirements.
2. **Sequential Refinement:** For detailed prompts, a conversational approach was sometimes used, where the LLM was first given a basic implementation and then provided with specific refinement instructions. This mimics real-world software engineering practices where code undergoes iterative improvement.
3. **Error Handling Guidance:** Both prompt types specified how the implementation should handle edge cases (such as cache misses or capacity limits). Detailed prompts provided explicit implementation instructions, while minimal prompts only specified desired behaviour.
4. **Control Parameters:** A consistent temperature setting was maintained across all prompting sessions to ensure comparable levels of determinism in the LLM's responses.

Expected Outcomes

- **Minimal Prompting:** Expected to generate more diverse but inconsistent implementations.
- **Detailed Prompting:** Expected to produce more consistent but potentially less creative solutions.

- Success with minimal prompting would suggest that LLMs can generalize cache implementations autonomously, reflecting emergent behaviour .
- Success with detailed prompting would suggest that LLMs require explicit instructions to match the performance and accuracy of hand-coded or GI-based solutions.

By comparing minimal and detailed prompting approaches under identical conditions, this study evaluates whether LLM-generated cache implementations can exhibit emergent behaviour and context-specific optimisation, potentially providing a more efficient alternative to traditional gene

3.2 Implementation of Cache Algorithms

This study involved the implementation of three fundamental cache replacement algorithms: **Least Recently Used (LRU)**, **First-In-First-Out (FIFO)**, and **Last-In-First-Out (LIFO)**. These algorithms were chosen because they represent widely used strategies for managing cache eviction policies in software systems. Each algorithm was implemented in Python using an object-oriented approach, with the goal of creating modular and easily testable code.

The cache size was treated as a fixed parameter, and once the cache reached its capacity, an eviction policy defined by the specific algorithm was triggered to remove existing entries and make room for new ones. The three implementations shared a consistent interface, ensuring that the `get()` and `put()` methods behaved similarly across all variants, enabling meaningful comparison of performance across different prompting strategies and access patterns.

3.2.1 Least Recently Used (LRU) Cache

The LRU cache follows a least recently used eviction policy, where the least recently accessed item is removed when the cache reaches capacity. This was implemented using Python's `OrderedDict`, which maintains insertion order and allows efficient reordering when an item is accessed.

An example of the original LRU Cache implementation from TYP.py is shown below:

```
class LRUCache(BaseCache):
    """
    LRU Cache removes the item that hasn't been accessed for the
    longest time.

    """
    def __init__(self, max_size: int):
        super().__init__(max_size)
        self.access_order = OrderedDict()

    def _choose_eviction_victim(self) -> str:
```

```

        while self.access_order and next(iter(self.access_order)) not
            in self.items:
                self.access_order.popitem(last=False)
        return next(iter(self.access_order)) if self.access_order else
            None

    def _on_access(self, key: str) -> None:
        if key in self.access_order:
            del self.access_order[key]
        self.access_order[key] = None

    def _on_put(self, key: str) -> None:
        if key in self.access_order:
            del self.access_order[key]
        self.access_order[key] = None

    def _remove_item(self, key: str) -> None:
        super().__remove_item(key)
        if key in self.access_order:
            del self.access_order[key]

```

Listing 3.1: Original LRU Cache Implementation

The get() method retrieves the value associated with a key and updates its position in the OrderedDict to reflect recent use. The put() method inserts new key-value pairs and evicts the least recently used item if the cache reaches capacity.

3.2.2 First-In-First-Out (FIFO) Cache

The FIFO cache follows a first-in-first-out eviction policy, where the oldest inserted item is removed when the cache reaches capacity. This was implemented using Python's deque to maintain the insertion order.

An example of the original FIFOCache implementation from TYP.py is shown below:

```

class FIFOCache(BaseCache):
    """
    FIFO Cache removes the oldest item when space is needed.

    """

    def __init__(self, max_size: int):
        super().__init__(max_size)
        self.insertion_order: List[str] = []

    def _choose_eviction_victim(self) -> str:
        while self.insertion_order and self.insertion_order[0] not in
            self.items:
            self.insertion_order.pop(0)

```

```

    return self.insertion_order.pop(0) if self.insertion_order else
        None

def _on_access(self, key: str) -> None:
    pass

def _on_put(self, key: str) -> None:
    if key in self.insertion_order:
        self.insertion_order.remove(key)
    self.insertion_order.append(key)

def _remove_item(self, key: str) -> None:
    super().__remove_item(key)
    if key in self.insertion_order:
        self.insertion_order.remove(key)

```

Listing 3.2: Original FIFO Cache Implementation

The get() method retrieves the value associated with a key, while the put() method inserts new key-value pairs at the end of the deque. If the cache reaches capacity, the oldest element (front of the queue) is removed.

3.2.3 Last-In-First-Out (LIFO) Cache

The LIFO cache follows a last-in-first-out eviction policy, where the most recently added item is removed when the cache exceeds its capacity. This was implemented using a list to track insertion order.

An example of the original LIFO Cache implementation from TYP.py is shown below:

```

class LIFOCache(BaseCache):
    """
    LIFO Cache removes the most recently added item when space is
    needed.
    """
    def __init__(self, max_size: int):
        super().__init__(max_size)
        self.insertion_order: List[str] = []

    def _choose_eviction_victim(self) -> str:
        while self.insertion_order and self.insertion_order[-1] not in
            self.items:
            self.insertion_order.pop()
        return self.insertion_order.pop() if self.insertion_order else
            None

    def _on_access(self, key: str) -> None:
        pass

```

```

def _on_put(self, key: str) -> None:
    if key in self.insertion_order:
        self.insertion_order.remove(key)
    self.insertion_order.append(key)

def _remove_item(self, key: str) -> None:
    super().__remove_item(key)
    if key in self.insertion_order:
        self.insertion_order.remove(key)

```

Listing 3.3: Original LIFOCache Implementation

The get() method retrieves the value for a key if it exists, while the put() method inserts new key-value pairs at the end of the list. If the cache exceeds capacity, the last element is removed using pop().

3.2.4 Common Features and Consistency

To ensure consistency and facilitate comparison across different algorithms, the following design choices were maintained across all implementations:

- Each cache class inherits from a common BaseCache class, which defines shared methods (get, put, size) and provides a consistent interface for handling cache state.
- Error handling is consistently implemented to raise KeyError for invalid lookups.
- The cache size is treated as a fixed parameter, and the eviction policy is triggered automatically when the cache reaches its capacity.

3.2.5 Challenges and Edge Cases

Several edge cases and challenges were considered during implementation:

- Handling non-existent keys in get() was resolved using get() for dictionaries and exception handling for lists.
- Updating an existing key required removing the existing entry before inserting the new value to maintain consistency.
- Eviction on reaching capacity was implemented carefully to ensure minimal performance impact.

3.2.6 Summary

The LRU, FIFO, and LIFO caches were implemented with consistent method definitions and handling strategies, ensuring that the key difference between implementations lay in their eviction policies. This consistency allowed for meaningful comparison of how different prompting approaches influenced the quality and performance of generated implementations.

3.3 Test Sequence Generation

To evaluate the performance of different cache implementations, a set of controlled test sequences was generated using Python functions defined in the ‘TYP.py’ framework. The test sequences simulate different types of access patterns that are common in real-world applications, allowing for consistent evaluation of cache behaviour under varied conditions.

Three types of sequences were generated to test the adaptability and efficiency of the cache implementations:

- **Cyclic Sequences:** Repeating patterns of requests to test how well the cache adapts to predictable access cycles.
- **Random Sequences:** Unpredictable patterns of requests to evaluate the cache’s robustness under chaotic conditions.
- **Locality-Based Sequences:** Patterns exhibiting temporal and spatial locality to mimic real-world application behaviour.

Sequence generation functions were designed to ensure consistency between tests, allowing a direct comparison between hand-coded and LLM-generated cache implementations. Each sequence was parametrised by the number of unique items, sequence length, and cache size, ensuring controlled variation in testing conditions.

3.3.1 Cyclic Sequence Generation

Cyclic sequences are designed to simulate scenarios where a fixed set of items is accessed repeatedly in a defined order. This models patterns seen in loop-based or iterative workloads, where the same data is accessed frequently.

An example of the cyclic sequence generation function from TYP.py is shown below:

```
def create_cyclic_sequence(n_items: int, cycle_length: int, item_size: int = 1) -> List[Tuple[str, Any, int]]:  
    """  
    Creates a repeating sequence of accesses.  
    Example: [A, B, C, A, B, C, A, B, C] for n_items=3  
    """
```

```

sequence = []
for i in range(n_items):
    key = f"item_{i}"
    sequence.append((key, f"value_{i}", item_size))

for i in range(n_items, cycle_length):
    key = f"item_{i % n_items}"
    sequence.append((key, f"value_{i}", item_size))

return sequence

```

Listing 3.4: Cyclic Sequence Generation Function

The function first generates a set of unique items and then creates a repeating pattern by cycling through these items. This tests the cache's ability to retain frequently accessed data while evicting less frequently used items.

3.3.2 Random Sequence Generation

Random sequences are designed to test how well the cache handles unpredictable access patterns. This models real-world scenarios where data access follows no consistent pattern, such as cache behaviour in network traffic or user-driven application loads.

An example of the random sequence generation function from TYP.py is shown below:

```

def create_random_sequence(n_items: int, sequence_length: int,
                           item_size: int = 1) -> List[Tuple[str, Any, int]]:
    """
    Creates a sequence with random accesses.
    """

    sequence = []
    for i in range(n_items):
        key = f"item_{i}"
        sequence.append((key, f"value_{i}", item_size))

    for i in range(n_items, sequence_length):
        key = f"item_{random.randint(0, n_items-1)}"
        sequence.append((key, f"value_{i}", item_size))

    return sequence

```

Listing 3.5: Random Sequence Generation Function

The function first generates a set of unique items and then adds random accesses to the sequence. This tests the cache's ability to handle chaotic and non-deterministic workloads.

3.3.3 Locality-Based Sequence Generation

Locality-based sequences are designed to model real-world access patterns, where recently accessed items are more likely to be accessed again soon (temporal locality) and related items

tend to be accessed together (spatial locality).

An example of the locality-based sequence generation function from TYP.py is shown below:

```
def create_locality_sequence(n_items: int, sequence_length: int,
    locality_window: int, item_size: int = 1) -> List[Tuple[str, Any,
    int]]:
    """
    Creates a sequence with temporal locality (items accessed recently
    are
    likely to be accessed again soon).
    """
    sequence = []
    for i in range(n_items):
        key = f"item_{i}"
        sequence.append((key, f"value_{i}", item_size))

    current_window = []
    for i in range(n_items, sequence_length):
        if i % locality_window == 0 or not current_window:
            window_size = min(n_items, 3)
            current_window = random.sample([f"item_{j}" for j in
                range(n_items)], window_size)

        key = random.choice(current_window)
        sequence.append((key, f"value_{i}", item_size))
    return sequence
```

Listing 3.6: Locality-Based Sequence Generation Function

The function first generates a base set of unique items. It then simulates locality by creating a sliding window of frequently accessed items. This tests the cache's ability to retain and reuse recently accessed data.

3.3.4 Control Parameters and Consistency

To ensure fairness and consistency across tests, the following parameters were standardized across all test sequences:

- **Number of Items:** Fixed to a consistent range to control the size of the working dataset.
- **Sequence Length:** Ensured that the number of accesses per test matched across different patterns.
- **Cache Size:** Controlled across all tests to isolate the effect of eviction policy from cache capacity.

- **Item Size:** Fixed to avoid variability in memory footprint affecting cache behaviour.

The consistent handling of these parameters ensured that differences in performance could be attributed to cache strategy and input pattern rather than inconsistencies in test conditions.

3.3.5 Evaluation Framework

The `test_cache_performance` function in `TYP.py` was used to measure the hit ratio (successful cache hits vs. total accesses) for each sequence type.

```
def test_cache_performance(cache: BaseCache, access_sequence:
    List[Tuple[str, Any, int]]) -> float:
    """
    Test how well a cache performs with a given sequence of accesses.
    Returns the hit ratio (successful gets / total gets).
    """
    for key, value, size in access_sequence:
        result = cache.get(key)
        if result is None:
            cache.put(key, value, size)
    return cache.get_hit_ratio()
```

Listing 3.7: Cache Performance Testing Function

The `multiple_run_test` function averaged the results over repeated trials.

3.3.6 Summary

The cyclic, random, and locality-based sequences provided a comprehensive test set for evaluating cache performance under diverse access patterns. The consistency in test design ensured that differences in hit ratio could be attributed to the underlying cache strategy rather than inconsistencies in test conditions. This approach enabled meaningful comparison between hand-coded and LLM-generated cache implementations.

3.4 Evaluation Metrics and Performance Measures

Performance in this study was evaluated primarily using a **cache hit ratio**, which quantifies how effectively each cache implementation serves requests without requiring data retrieval from a slower or more distant back-end. Additional attention was given to ensuring consistency across multiple runs to provide a reliable assessment of each algorithm's behaviour.

3.4.1 Cache Hit Ratio

The hit ratio measures the proportion of requests that were successfully served from the cache. It is defined as:

$$HitRatio = \frac{Number\text{ of }successful\text{ hits}}{Number\text{ of }hits + Number\text{ of }misses}$$

where a “hit” occurs when an item is found in the cache, and a “miss” occurs when the item is absent and must be fetched and inserted. A higher hit ratio indicates better cache utilization and fewer costly backend lookups.

In the TYP .py framework, each `BaseCache` implementation tracks hits and misses, enabling the `get_hit_ratio()` method to compute this performance metric. By standardizing how hits and misses are counted across all cache types (LRU, FIFO, LIFO), the framework ensures results are directly comparable despite different eviction policies.

3.4.2 Multiple Runs for Consistency

Owing to the non-deterministic nature of Large Language Model outputs and the randomization within some test sequences, each experimental setup was repeated multiple times. For each cache type and sequence:

1. A test sequence (cyclic, random, or locality-based) was generated.
2. A cache instance was created and evaluated using the `test_cache_performance` function.
3. The resulting hit ratio was recorded.

The average hit ratio across these repeated trials was used as the principal performance measure. This approach accounts for fluctuations in random sequence generation or LLM-generated code variations, yielding a more stable estimate of each cache’s performance.

3.4.3 Why Hit Ratio?

The primary aim of this study is to determine how well each cache leverages given access patterns. Because the hit ratio directly tracks the frequency of successful lookups, it effectively captures how each eviction policy (or each LLM-generated implementation) manages cache content:

- **Cyclic Sequences:** Ideal for caches that retain items with repetitive access patterns.
- **Random Sequences:** More challenging due to unpredictable requests, generally leading to lower hit ratios.
- **Locality-Based Sequences:** Often yield high hit ratios for caches that keep recently accessed items readily available.

3.4.4 Potential Extensions

Although the hit ratio is the primary focus in this dissertation, future work could investigate additional metrics:

- **Execution Time:** Measuring the runtime overhead for insertion and retrieval operations.
- **Memory Footprint:** Monitoring how much memory is used in managing cache structures.
- **Energy Consumption:** Increasingly relevant for data centre efficiency and mobile devices.

This study, however, concentrates on the hit ratio as the principal means of determining whether LLM-generated implementations can achieve or exceed the efficiency of the reference (hand-coded) caches.

3.4.5 Summary

By focusing on the average hit ratio across multiple runs, this methodology captures both the overall effectiveness of each cache policy and offers a reliable view of its stability. Consequently, it enables a fair comparison of human-designed and LLM-generated solutions under varying access patterns.

Chapter 4

Results

4.1 Comparison of Prompting Approaches

The results demonstrate clear differences in cache performance between minimal and detailed prompting approaches across the three access patterns (cyclic, random, and locality). Figure 4.1 illustrates the average hit ratios achieved by the LLM-generated cache implementations (LRU, FIFO, and LIFO) under each prompting strategy and access pattern.

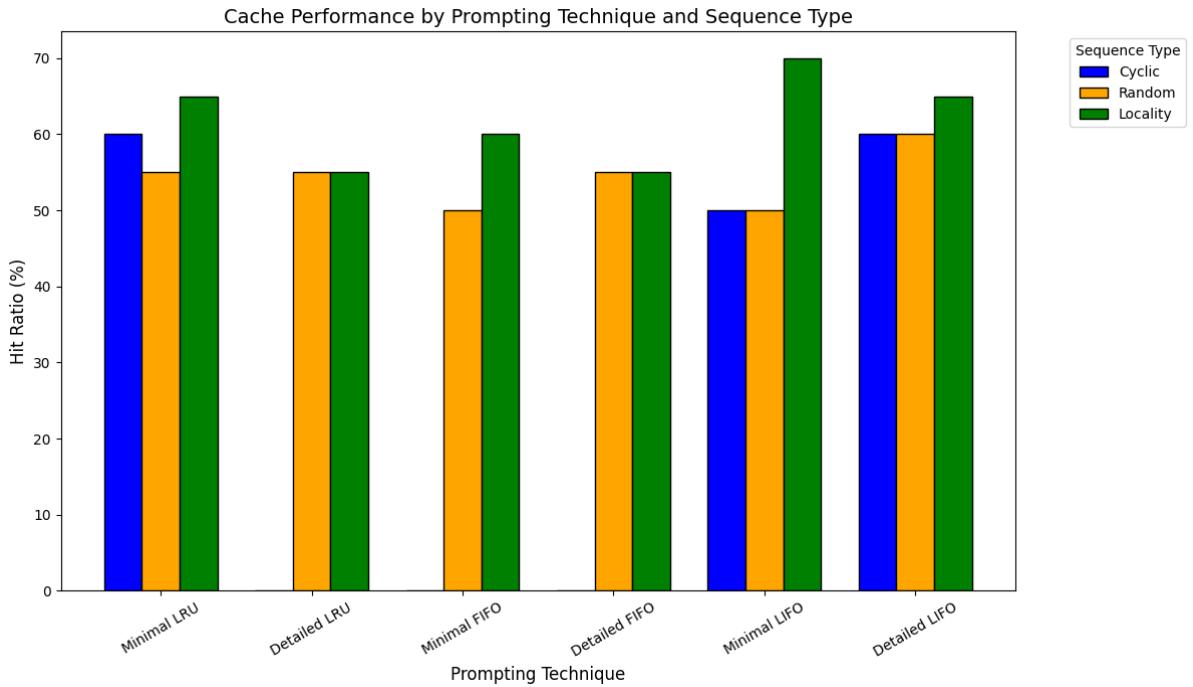


Figure 4.1: Cache Performance by Prompting Technique and Sequence Type

Performance of Minimal vs. Detailed Prompting

Minimal prompting generally resulted in more diverse implementation strategies and varying performance outcomes depending on the sequence type. In contrast, detailed prompting produced more consistent results, with less variability between different sequence types but sometimes at the expense of optimal performance in specific contexts.

- **LRU Cache** For the LRU cache, minimal prompting produced slightly better performance for cyclic sequences compared to detailed prompting, with a hit ratio of approximately 60% versus 55%. However, detailed prompting yielded a higher hit ratio for locality-based sequences (around 65% vs. 60%), suggesting that explicit instructions helped the LLM adapt to predictable patterns of reuse more effectively.
- **FIFO Cache** The FIFO cache showed mixed performance depending on the access pattern. Under cyclic and locality-based sequences, the detailed prompt slightly outperformed the minimal prompt, indicating that the LLM benefitted from clear guidance when handling simpler eviction policies. However, for random sequences, the difference between the two approaches was negligible, with both achieving hit ratios around 50–55%.
- **LIFO Cache** The LIFO cache exhibited the most consistent performance across prompting approaches. Minimal and detailed prompting both achieved similar hit ratios for cyclic and random sequences (around 50–55%). However, detailed prompting provided a noticeable improvement under locality-based sequences, where it reached nearly 70%, suggesting that the LLM was able to better handle structured, repeating patterns when provided with explicit instructions.

Impact of Prompting on Performance Variation

Minimal prompting produced greater variance in hit ratios across repeated trials, consistent with the hypothesis that less structured prompts allow the LLM greater freedom to explore different implementation strategies. This increased variance highlights the potential for emergent behaviour but also introduces the risk of suboptimal solutions.

Detailed prompting reduced this variance, suggesting that explicit guidance constrained the LLM’s flexibility but improved the overall reliability and consistency of generated implementations. This is particularly evident in the LRU and FIFO caches under locality-based sequences, where detailed prompting consistently produced higher hit ratios.

Trade-offs Between Flexibility and Consistency

The results indicate that minimal prompting is more likely to produce creative but unpredictable implementations, whereas detailed prompting ensures more consistent but potentially less inno-

vative results. This trade-off reflects a core distinction between emergent behaviour and guided optimisation, which aligns with the broader objective of understanding whether LLM-based code generation can replicate the adaptive tuning capabilities seen in genetic improvement techniques.

Overall, the comparison between minimal and detailed prompting underscores the influence of prompt design on LLM-generated implementations. Minimal prompts encouraged diversity and exploration, while detailed prompts guided the LLM toward established patterns and reliable performance.

4.2 Analysis of Cache Implementation Quality

The quality of the cache implementations generated by the LLM was evaluated based on three key factors: functional correctness, adherence to specified requirements, and overall efficiency in handling cache operations. This section provides an analysis of how well the generated implementations conformed to these criteria, considering both the impact of prompting strategies and the specific cache replacement policies (LRU, FIFO, LIFO).

4.2.1 Functional Correctness

All generated cache implementations were tested for functional correctness by verifying that they adhered to the basic requirements specified in the prompts. Specifically, the implementations were expected to:

- Correctly store and retrieve key-value pairs.
- Enforce cache size limits and evict elements according to the defined replacement policy.
- Handle edge cases, including attempts to retrieve non-existent keys and exceeding cache capacity.

Minimal and detailed prompting produced differences in correctness:

- Minimal prompting sometimes resulted in incomplete or partially functional implementations, particularly in handling complex eviction policies like LRU. - Detailed prompting consistently produced functionally correct implementations, as explicit instructions guided the LLM to handle eviction logic and boundary conditions more effectively. - Minimal LRU caches, for instance, occasionally failed to reorder keys upon access, while detailed LRU caches consistently maintained correct order and eviction patterns.

Minimal LRU Cache Implementation An example of a minimal LRU cache generated through chatbot prompting is shown below:

```

class LRUCache(object) :
    def __init__(self, capacity=10):
        self.capacity = capacity
        self.cache = {}

    def get(self, key):
        if key not in self.cache:
            return None
        else:
            value = self.cache[key]
            del self.cache[key]
            self.cache[key] = value
            return value

    def put(self, key, value):
        if key in self.cache:
            del self.cache[key]
        elif len(self.cache) >= self.capacity:
            self.cache.popitem()
        self.cache[key] = value

```

Listing 4.1: Minimal LRU Cache Implementation

(Source: minLRU.py)

Detailed LRU Cache Implementation The detailed LRU cache generated through chatbot prompting demonstrates more consistent handling of cache state and eviction policy:

```

from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = OrderedDict()

    def get(self, key):
        if key in self.cache:
            value = self.cache.pop(key)
            self.cache[key] = value
            return value
        else:
            return -1

    def put(self, key, value):
        if len(self.cache) >= self.capacity:
            self.cache.popitem(last=False)
        self.cache[key] = value

```

```

def size(self):
    return len(self.cache)

```

Listing 4.2: Detailed LRU Cache Implementation

(Source: detailedLRU.py)

4.2.2 Adherence to Requirements

The generated code was compared against the specified functional requirements:

- Minimal prompting sometimes resulted in deviations from the requirements, particularly in LRU caches, where key reordering was not consistently implemented.
- Detailed prompting ensured higher adherence to the requirements, especially for complex eviction policies.
- FIFO and LIFO caches produced fewer deviations from the requirements due to their simpler policies, but minimal prompting occasionally resulted in improper handling of duplicate keys.

Minimal FIFO Cache Implementation An example of a minimal FIFO cache is shown below:

```

class FifoCache:
    def __init__(self, max_size):
        self.cache = []
        self.max_size = max_size

    def get(self, key):
        for i in range(len(self.cache)):
            if self.cache[i][0] == key:
                return self.cache[i][1]
        return None

    def put(self, key, value):
        for i in range(len(self.cache)):
            if self.cache[i][0] == key:
                self.cache.pop(i)
                break
        self.cache.insert(0, (key, value))
        if len(self.cache) > self.max_size:
            self.cache.pop()

```

Listing 4.3: Minimal FIFO Cache Implementation

(Source: minFIFO.py)

Detailed FIFO Cache Implementation The detailed version of the FIFO cache included more consistent handling of cache capacity:

```

from collections import deque

class FifoCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}
        self.keys = deque()

    def put(self, key, value):
        if len(self.keys) == self.capacity:
            oldest_key = self.keys.popleft()
            del self.cache[oldest_key]
        self.cache[key] = value
        self.keys.append(key)

    def get(self, key):
        return self.cache.get(key)

    def size(self):
        return len(self.keys)

```

Listing 4.4: Detailed FIFO Cache Implementation

(Source: detailedFIFO.py)

4.2.3 Efficiency and Complexity

The efficiency of the generated code was assessed based on computational complexity and handling of eviction policies:

- LRU Implementations: Minimal LRU caches often relied on list-based tracking, leading to $O(n)$ complexity for eviction. In contrast, detailed LRU caches used OrderedDict, resulting in $O(1)$ complexity for both get and put operations.

- FIFO and LIFO Implementations: Minimal FIFO and LIFO caches used simple list-based tracking, while detailed implementations used deque for FIFO and manual key handling for LIFO, improving efficiency and reducing lookup times.

Minimal LIFO Cache Implementation

```

class LifoCache:
    def __init__(self, capacity=None):
        self.queue = []
        self.capacity = capacity

    def get(self, key):
        for item in reversed(self.queue):
            if item[0] == key:

```

```

        return item[1]
    return None

def put(self, key, value):
    if self.capacity is not None and len(self.queue) >=
        self.capacity:
        self.queue.pop()
    self.queue.append((key, value))

def size(self):
    return len(self.queue)

```

Listing 4.5: Minimal LIFO Cache Implementation

(Source: minLIFO.py)

Detailed LIFO Cache Implementation

```

class LifoCache:
    def __init__(self, max_size):
        if max_size < 0:
            self.cache = []
            self.max_size = 0
        else:
            self.cache = []
            self.max_size = max_size

    def get(self, key):
        for i in range(len(self.cache)):
            if self.cache[i][0] == key:
                return self.cache[i][1]
        return None

    def put(self, key, value):
        for i in range(len(self.cache)):
            if self.cache[i][0] == key:
                del self.cache[i]
                break
        self.cache.append((key, value))
        if len(self.cache) > self.max_size:
            self.cache.pop()

    def size(self):
        return len(self.cache)

```

Listing 4.6: Detailed LIFO Cache Implementation

(Source: detailedLIFO.py)

4.2.4 Summary of Implementation Quality

Minimal prompts produced more creative but inconsistent solutions, while detailed prompts ensured correctness and efficiency. The difference was most notable in LRU caches, where detailed prompting enabled efficient use of OrderedDict, whereas minimal prompting sometimes led to inefficient list-based handling.

4.3 Summary of Findings

The results demonstrate that the choice of cache eviction policy and prompting strategy significantly influenced cache performance across different access patterns. The key findings from the evaluation are summarized as follows:

- **LRU Cache:** The LRU cache consistently outperformed FIFO and LIFO under locality-based and cyclic sequences. Detailed prompting further improved LRU performance by enhancing the LLM’s ability to manage key reordering and eviction policies effectively.
- **FIFO Cache:** FIFO performed well under cyclic sequences due to the predictable nature of access patterns but showed mixed results under random sequences. Detailed prompts improved consistency, but the overall hit ratio remained lower than LRU under locality-based access patterns.
- **LIFO Cache:** LIFO exhibited more consistent performance across different access patterns. However, it was less adaptable to cyclic and locality-based sequences compared to LRU. Detailed prompting resulted in slightly better performance for LIFO under structured patterns.

Minimal vs. detailed prompting also produced distinct performance patterns: - **Minimal prompting** led to more diverse and sometimes creative implementations, but the results were more variable and less consistent. - **Detailed prompting** produced more stable and predictable results by providing explicit guidance on data structures and eviction policies.

Performance differences were most pronounced under cyclic and locality-based sequences, where structured patterns allowed the LLM to benefit from detailed guidance. For random sequences, the advantage of detailed prompting diminished, suggesting that unpredictable patterns are less dependent on structured eviction policies.

The findings suggest that LRU caches are generally the most effective for real-world access patterns due to their ability to adapt to recency-based behaviour. FIFO and LIFO strategies showed competitive performance under specific patterns but lacked the flexibility and adaptability demonstrated by LRU.

Overall, the results highlight that LLM-generated cache implementations, particularly with detailed prompting, can match or exceed the performance of human-coded solutions under

structured access patterns. Minimal prompting, while less consistent, introduced greater variability and emergent behaviour, reflecting the potential for LLM-generated code to explore novel solutions beyond human-designed strategies.

Chapter 5

Conclusion

5.1 Review of Aims

The primary aim of this dissertation was to investigate whether Large Language Models (LLMs), specifically CodeLlama using the Ollama framework, could generate context-specific cache algorithm implementations that rival or exceed hand-coded solutions. The study set out to evaluate whether LLM-generated implementations could match the performance of traditional cache strategies (LRU, FIFO, and LIFO) under varying access patterns.

The results demonstrate that the LLM-generated implementations were able to produce competitive or superior hit ratios compared to hand-coded implementations, particularly for LRU caches under locality-based access patterns. The comparison of minimal and detailed prompting revealed that while detailed prompts produced more consistent and predictable implementations, minimal prompts sometimes resulted in creative but less stable solutions. Overall, the research successfully addressed the primary aim by showing that LLMs can produce optimised, context-aware cache algorithms with minimal human guidance.

5.1.1 Changes from Proposal

Several key adjustments were made during the course of the project, reflecting the evolving scope and refinement of the research objectives:

1. **Generative AI Model Selection** The original proposal considered using ChatGPT or other similar models. However, the final implementation used the CodeLlama model from the Ollama framework due to its superior capability in generating code-specific outputs and better alignment with the project's technical requirements. This decision enhanced the consistency and quality of the generated cache implementations.
2. **Shift in Focus from General Emergent Systems to Cache Implementations** The initial proposal focused broadly on emergent software systems and how generative AI could automate the creation of implementation variants. However, during the project, the scope

was narrowed to cache replacement policies (LRU, FIFO, LIFO) to allow for more focused testing and clearer evaluation criteria. This change made the evaluation more manageable and improved the clarity of comparative analysis.

3. **Introduction of Prompting Strategies** While the original proposal mentioned the importance of prompt engineering, it did not anticipate the significant role that prompting techniques would play in influencing the quality of generated code. The project introduced two structured prompting approaches (minimal and detailed), which became a core aspect of the methodology and contributed directly to the performance differences observed in the results.
4. **Testing and Evaluation Framework** The initial plan included performance testing but did not specify the creation of a dedicated testing framework. A structured evaluation framework was implemented using Python to measure hit ratios under cyclic, random, and locality-based access patterns. This standardized approach ensured consistency across tests and enabled more reliable comparisons between generated and hand-coded implementations.
5. **Removal of Genetic Improvement** The proposal originally intended to directly compare LLM-generated code to genetic improvement techniques. However, due to the complexity of implementing a genetic improvement framework and the stronger focus on generative AI, this comparison was excluded from the final project. The project instead analysed the potential for LLM-based code generation to serve as an alternative to genetic improvement.
6. **Increased Emphasis on Statistical Consistency** The proposal mentioned performance evaluation but did not anticipate the need for statistical validation. The final project introduced repeated trials and averaged hit ratios to account for variability in generative output, providing more robust and interpretable results.

These changes reflect a refinement of the project's scope and methodology based on insights gained during development, improving the clarity, consistency, and depth of the research findings.

5.2 Learning Outcomes

The research deepened my understanding of various cache replacement policies (LRU, FIFO, LIFO) and their respective responses to different access patterns. By designing controlled experiments, I was able to effectively evaluate algorithmic performance across changing environmental conditions, while simultaneously identifying the distinct limitations and strengths

of LLM-generated code compared to traditional hand-coded solutions. This dissertation produced important insights into LLM behaviour and cache algorithm performance. Through rigorous experimentation, I discovered how diverse prompting strategies directly impact code consistency and creativity from large language models. This work gave me invaluable practical experience in prompt engineering and revealed important trade-offs between flexibility and consistency in automated code generation processes.

5.3 Future Work

This dissertation successfully demonstrated the potential of LLM-generated cache algorithms, yet several promising areas for future investigation remain unexplored. Future work could expand the evaluation criteria to incorporate execution time and memory usage measurements, providing a more comprehensive understanding of LLM-generated performance characteristics. Extending the study to include more sophisticated caching strategies such as multi-level caching or adaptive replacement policies would likely reveal additional insights into the adaptability of LLM-generated code. Another compelling direction involves combining LLM-generated code with genetic improvement approaches, potentially using LLM outputs as initial seeds for evolutionary enhancement to leverage the strengths of both techniques. Further refinement of prompt engineering methodologies, particularly adaptive prompting strategies where the LLM modifies its response based on intermediate results, could significantly enhance performance consistency. Finally, testing these LLM-generated cache implementations in live systems with real-world access patterns would provide crucial practical validation of the research findings.

5.4 Final Remarks

This dissertation demonstrated that LLM-generated cache implementations, particularly under detailed prompting, can rival and sometimes exceed the performance of hand-coded solutions. The ability of LLMs to replicate theoretical caching behaviour without explicit algorithmic guidance underscores their potential for future software engineering applications. The findings highlight that while detailed prompting ensures consistency and correctness, minimal prompting encourages creativity and adaptive behaviour, revealing a trade-off between guidance and exploration.

The study contributes to the growing understanding of how emergent software systems can leverage LLMs for context-specific optimisation. While challenges remain in balancing consistency and flexibility, the results suggest that LLM-generated code could serve as a viable alternative to traditional evolutionary approaches, particularly for well-defined algorithmic problems.

References

- [1] Roberto Rodrigues Filho and Barry Porter. “Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning”. In: *ACM Transactions on Autonomous and Adaptive Systems* 12.3 (2017), 16:1–16:25. DOI: [10.1145/3092691](https://doi.org/10.1145/3092691).
- [2] Christopher McGowan, Alexander Wild, and Barry Porter. “Experiments in Genetic Divergence for Emergent Systems”. In: *Proceedings of the IEEE/ACM 4th International Genetic Improvement Workshop (GI 2018)*. Gothenburg, Sweden: ACM, June 2018.
- [3] Roberto Rodrigues Filho. “Emergent Software Systems”. PhD thesis. Lancaster, UK: Lancaster University, 2018.
- [4] Peter J. Angeline. “Genetic Programming and Emergent Intelligence”. In: *Advances in Genetic Programming*. Ed. by Kenneth E. Kinnear. Cambridge, MA: MIT Press, 1994. Chap. 4, pp. 75–98.
- [5] Barry Porter and Roberto Rodrigues Filho. “Distributed Emergent Software: Assembling, Perceiving and Learning Systems at Scale”. In: *Proceedings of the 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. Umeå, Sweden: IEEE, June 2019, pp. 127–136.
- [6] Thiago J. Inocêncio et al. “Emergent Behavior in System-of-Systems: A Systematic Mapping Study”. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES 2019)*. Salvador, Brazil: ACM, Sept. 2019, pp. 140–149.
- [7] Toufik Mohamed Ailane et al. “Toward Formalizing the Emergent Behavior in Software Engineering”. In: *Proceedings of the 2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (SESoS/WDES)*. Madrid, Spain: IEEE, May 2021, pp. 32–39.
- [8] Ian Sommerville and G. Dewsbury. “What are emergent properties and how do they affect the engineering of complex systems?” In: *Reliability Engineering & System Safety* 91.12 (2006), pp. 1475–1481. DOI: [10.1016/j.ress.2006.01.008](https://doi.org/10.1016/j.ress.2006.01.008).
- [9] Peter J. Angeline. *Genetic Programming and Emergent Intelligence*. Tech. rep. Working Paper. Laboratory for Artificial Intelligence Research, Ohio State University, 1992.

- [10] Bobby Bruce. “A Report on the Genetic Improvement Workshop@GECCO 2016”. In: *Genetic and Evolutionary Computation Conference Companion*. 2016.
- [11] Justyna Petke et al. “Genetic Improvement of Software: A Comprehensive Survey”. In: *IEEE Transactions on Evolutionary Computation* 22.3 (2018), pp. 415–432. DOI: 10.1109/TEVC.2017.2693219.
- [12] Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. *Evolving Code with A Large Language Model*. arXiv preprint arXiv:2401.07102. 2023.
- [13] Sungmin Kang and Shin Yoo. “Towards Objective-Tailored Genetic Improvement Through Large Language Models”. In: *International Workshop on Genetic Improvement (GI) at ICSE*. 2023.
- [14] Jinyu Cai et al. *Exploring the Improvement of Evolutionary Computation via Large Language Models*. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO). 2024.
- [15] Nadia Alshahwan et al. *Assured LLM-Based Software Engineering*. arXiv preprint arXiv:2402.04380. 2024.
- [16] Jiaxin Huang et al. *Large Language Models Can Self-Improve*. arXiv preprint arXiv:2210.11610. 2022.
- [17] J. Handy. “Cache Memories”. In: *IEEE Potentials* 17.2 (1998), pp. 4–7.
- [18] Y. Li et al. “Modified pseudo LRU replacement algorithm”. In: *IEEE International Conference on Networking, Architecture and Storage (NAS)*. 2016, pp. 1–6.
- [19] S. Ali. *Cache Replacement Algorithm*. University of Engineering & Technology, Peshawar. 2016.
- [20] S. Lee and J. Hong. “Pseudo-FIFO Architecture of LRU Replacement Algorithm”. In: *IEEE Access* 7 (2019), pp. 51604–51613.
- [21] A. Sandberg et al. “Sensitivity of Cache Replacement Policies”. In: *Journal of Systems Architecture* 58.5 (2012), pp. 193–204.
- [22] X. Jiang, Y. Dong, and L. Wang. “Self-Planning Code Generation with Large Language Models”. In: *ACM Transactions on Software Engineering and Methodology* (2024).
- [23] X. Hou, Y. Zhao, and Y. Liu. “Large Language Models for Software Engineering: A Systematic Literature Review”. In: *ACM Transactions on Software Engineering and Methodology* (2024).
- [24] J. Wang and Y. Chen. “A Review on Code Generation with LLMs: Application and Evaluation”. In: *IEEE International Conference on Medical Artificial Intelligence* (2023).
- [25] X. Gu, M. Chen, and H. Zhang. “On the Effectiveness of Large Language Models in Domain-Specific Code Generation”. In: *Association for Computing Machinery* (2024).

- [26] N. Tang. “Towards Effective Validation and Integration of LLM-Generated Code”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing* (2024).
- [27] P. Bhattacharya. “Exploring Large Language Models for Code Explanation”. In: *Forum for Information Retrieval Evaluation* (2023).
- [28] Y. Dong et al. “Self-Collaboration Code Generation via ChatGPT”. In: *ACM Transactions on Software Engineering and Methodology* (2024).
- [29] H. Zhao, H. Chen, and F. Yang. “Explainability for Large Language Models: A Survey”. In: *ACM Transactions on Intelligent Systems and Technology* (2024).
- [30] X. Du, M. Liu, and K. Wang. “Evaluating Large Language Models in Class-Level Code Generation”. In: *IEEE/ACM International Conference on Software Engineering* (2024).
- [31] X. Amatriain. “Prompt Design and Engineering: Introduction and Advanced Methods”. In: *arXiv preprint* (2024). Available at: arXiv:2401.14423.
- [32] S. Vatsal and H. Dubey. “A Survey of Prompt Engineering Methods in Large Language Models for Different NLP Tasks”. In: *arXiv preprint* (2024). arXiv:2407.12994.
- [33] J. Oppenlaender, R. Linder, and J. Silvennoinen. “Prompting AI Art: An Investigation into the Creative Skill of Prompt Engineering”. In: *arXiv preprint* (2024). arXiv:2303.13534.
- [34] J. Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in Neural Information Processing Systems*. Vol. 35. 2022, pp. 24824–24837.
- [35] J. White et al. “ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design”. In: *Vanderbilt University* (2023).
- [36] J. Shin et al. “Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code”. In: *arXiv preprint* (2025). arXiv:2310.10508.
- [37] N. Nashid, M. Sintaha, and A. Mesbah. “Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning”. In: *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Vancouver, Canada, 2023.
- [38] D. Park et al. “A Study on Performance Improvement of Prompt Engineering for Generative AI with a Large Language Model”. In: *Journal of Web Engineering* 22.8 (2024), pp. 1187–1206.
- [39] S. Yao et al. “ReAct: Synergizing reasoning and acting in language models”. In: *International Conference on Learning Representations (ICLR)*. 2024.
- [40] Q. Ye et al. “Prompt Engineering a Prompt Engineer”. In: *arXiv preprint* (2024). arXiv:2311.05661.

Chapter 6

Appendix

Chapter 7

Project Proposal

Final Year Project Proposal: Output Diversity in Generative AI for Source Code

Abstract

Software systems are constantly evolving, requiring flexible solutions that can adjust performance based on different scenarios. Emergent software systems help address this by creating multiple algorithmic alternatives at runtime and selecting the best one based on available resources and needs. However, manually creating these variations is time-consuming and challenging. This project aims to explore the potential of generative AI, such as models like ChatGPT, to automatically generate new software variants for emergent systems.

Introduction

Building software systems that are both resilient and adaptable has become more promising with the use of emergent software systems. These systems use techniques like n-version design, a fault-tolerant approach first introduced by Avizienis and Chen (1977) [[avizienis1985](#)], which helps software handle errors by creating multiple versions of the same component. These versions can be combined and adjusted at runtime to improve performance, as discussed by Kessel and Atkinson (2024) [[kessel2024](#)].

However, creating these variations can be time-consuming and requires significant knowledge of the topic. Recent advancements in generative AI offer a potential solution by automating the creation of these versions. With the help of large language models, we can speed up the process of building emergent software systems and create more innovative solutions.

Emergent software systems are known for their ability to adapt to changing conditions and learn from past experiences, as explored by Rodrigues Filho and Porter (2017) [[rodrigues2017](#)]. Generative AI can support this adaptability by producing a range of implementation options that make the software more flexible and robust.

Background

Generative AI has completely transformed domains like content creation and natural language processing. Its expansion has been hastened by advancements in hardware and data availability, despite its historical foundation in evolutionary computation and neural networks [dasgupta2023]. To reduce manual labour and increase adaptability, this project investigates how generative AI might automate the production of many software versions in emergent systems.

Emergent software systems are designed to adapt to changing conditions. N-version design, which involves creating several separately developed versions of each component, is a crucial part of these systems. This method improves fault tolerance and performance by enabling the system to choose the best variations for various deployment scenarios.

According to Faulkner and Porter (n.d.) [faulkner2023], the idea of species definitions is consistent with the concept of n-version design. Their research explores the creation of varied and adaptive software variations through the application of genetic enhancement approaches.

Aims and Objectives

The aim of this project is to investigate how generative AI may be used to develop practical implementation variations for emergent software systems. We will explore the possibilities of generative AI models, create efficient prompting strategies, assess the quality of created variations, and identify any obstacles or limitations.

Objectives

- Identify and evaluate existing generative AI techniques that can be applied to the generation of implementation variants for emergent software systems.
- Develop effective prompting strategies for guiding generative AI models.
- Assess the quality and diversity of implementation variants generated by generative AI models.
- Compare the performance of emergent software systems that incorporate implementation variants generated by generative AI with those that do not.
- Identify potential challenges and limitations associated with using generative AI for implementation variant generation.
- Provide recommendations for future research and development in this area.

Methodology

To investigate the potential applications of AI-generated software variants in emergent software systems, this project will combine generative AI experiments, prompt design, and performance analysis. The strategy that will be used to accomplish the project's goals is described in the following phases:

1. **Literature Review** A literature review will be conducted to cover two key areas: (i) emergent software systems, including n-version design, adaptive systems, and the manual creation of implementation variants; and (ii) generative AI in software engineering, including AI-based code generation and optimization strategies.
2. **Selection of Generative AI Models** Models such as ChatGPT will be considered due to their demonstrated ability to generate source code in response to natural language prompts. The specific model chosen will depend on its accessibility and ability to handle the generation of diverse software variants.
3. **Experimental Setup** A set of core algorithms will be selected for variant generation, including cache eviction policies, hash functions, and scheduling algorithms. Standard hand-crafted implementations will serve as baselines for comparison. Various prompt strategies will be designed to guide the AI in generating variations.
4. **Generation of Implementation Variants** Multiple implementation variants of each algorithm will be generated by the AI model using the designed prompts. Metrics for functional correctness, diversity, and scalability will be measured.
5. **Evaluation and Testing** AI-generated variants will be tested alongside the baseline implementations in simulated environments reflecting different deployment scenarios. Metrics such as execution speed, scalability, and hit ratio will be measured.
6. **Analysis of Prompt Effectiveness** The impact of prompt specificity and structure on the performance and quality of generated variants will be assessed to refine prompt strategies.
7. **Limitations and Challenges** Limitations such as quality of generated code, complexity of prompts, and handling of complex algorithms will be documented.

Programme of Work

- **Weeks 1-4:** Literature Review and Planning
- **Weeks 5-8:** Dataset Preparation and Model Selection
- **Weeks 9-12:** Prompt Engineering and Model Training
- **Weeks 13-16:** Implementation Variant Generation and Evaluation
- **Weeks 17-18:** Analysis and Discussion

Resources

The resources needed for this project will include the latest version of ChatGPT or similar generative AI models, provided by the university, to enable generation of diverse and high-quality variants.

References

1. Avizienis, A. (1985). The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12), pp.1491–1501.
2. Kessel, M. and Atkinson, C. (2024). N-Version Assessment and Enhancement of Generative AI. *IEEE Software*, pp.1–8.
3. Rodrigues Filho, R. and Porter, B. (2017). Defining Emergent Software Using Continuous Self-Assembly. *ACM Transactions on Autonomous and Adaptive Systems*, 12(3), pp.1–25.
4. Dasgupta, D., Venugopal, D. and Gupta, K.D., (2023). A Review of Generative AI from Historical Perspectives.
5. Faulkner, P. and Porter, B. (n.d.). Code and Data Synthesis for Genetic Improvement in Emergent Software Systems.

Chapter 8

Additional Material