Experiences on Using Large Language Models to Re-engineer a Legacy System at Volvo Group

Vanshika Singh
Department of Computer Science
North Carolina State University
Raleigh, USA
Volvo Group
Greensboro, USA

Caglar Korlu
Onur Orcun
Volvo Group
Greensboro, USA

Wesley K. G. Assunção
Department of Computer Science
North Carolina State University
Raleigh, USA

***Abstract*—Digital processes driven by data are the basis for industry operations nowadays. Despite relying on software for such processes, the industry faces significant challenges because of legacy systems. Legacy systems are pieces of software that, despite being vital for the industry operations, have limitations in terms of performance and scalability needs. Thus, to ensuring that these systems can still continue to deliver business value, there is a need for re-engineering such legacy system. This is a situation faced by Volvo Group, with their system called SCORE, initially implemented in 2017, and used by HR teams and managers to manage team structures, employee data, and operational workflows. The SCORE system faces challenges related to performance issues, database inefficiencies, outdated user interface, lack of flexibility, and lack of modern software engineering practices. As a strategy to keep the business value of SCORE, Volvo has started using Large Language Models (LLMs) to speed up it re-engineering. This paper present the experiences of using LLMs at Volvo Group. More specifically, we describe how GPT-4 and Claude AI were applied, with three learning strategies (i.e., zero-shot, one-shot, and few-shot), to address the challenges of the legacy system. The prompts and examples of the responses given by the Foundations Models are presented and discussed. By adopting the insights provided by the LLMs, we were able to reduce API response times from 20-30 seconds to 3 seconds, improve UI usability by restructuring elements for easier navigation, and enhance scalability with better database queries and code modularization. The CI/CD pipeline was also streamlined, enabling faster and more reliable deployments. As an additional contribution, we report six lessons learned, allowing other industries and researchers to comprehend the strategic value of integrating LLMs into legacy system modernization.**

***Index Terms*—Foundation Models, Maintenance and Evolution, Software Modernization, Industry Report.**

I. Introduction

In today's digital landscape, industries such as finance, automotive, healthcare, and manufacturing are increasingly driven by data and digital processes to improve operations, enhance customer experience, and remain competitive [1]–[3]. Despite advancements, many of these sectors face significant challenges in upgrading outdated software systems that no longer meet modern performance and scalability needs [4], [5]. These legacy systems—built to address the demands of earlier eras—are often deeply integrated into the organization's core operations, making them difficult to replace or upgrade [6], [7]. Consequently, companies in these industries are confronted with the complex task of ensuring that these systems can continue to deliver value, while also adapting to new technological requirements and higher user expectations, hence there is a need for re-engineering such legacy systems [6], [8], [9].

The automotive industry, in particular, must balance the demands of real-time data processing, complex supply chains, and global customer networks, all of which require robust and responsive software systems [10]. Automotive companies rely heavily on platforms for everything from HR management to supply chain coordination and real-time vehicle diagnostics, making it critical that these systems operate seamlessly [1]. Systems in the automotive sector need to handle high transaction volumes, ensure security, and meet compliance requirements [11], [12]. The limitations of legacy systems are performance bottlenecks, lack of scalability, and outdated interfaces that impact the user experience, efficiency, and ultimately, the organization's overall agility.

Volvo Group encountered similar challenges with its SCORE system, an internal platform used by HR teams and managers to manage team structures, employee data, and operational workflows. Initially developed to support Volvo's HR functions, the SCORE platform became a critical resource over time, but it began to exhibit substantial limitations. Performance issues—such as slow API response times, inefficient database interactions, and an outdated user interface—were hindering its utility and negatively impacting the user experience. These issues not only led to delays in accessing essential data but also impacted Volvo's ability to make timely HR and organizational decisions. Furthermore, the SCORE system's lack of scalability and continuous integration/continuous deployment (CI/CD) support made it increasingly difficult to maintain and evolve in line with Volvo's growing needs.

To address these challenges, we proposed a solution to re-engineer leveraging Large Language Models (LLMs), specifically GPT-4[1] and Claude AI,[2] to explore optimization strategies for improving the platform's API performance, UI usability, scalability, and CI/CD pipeline. The idea was to apply the problem-solving capabilities of LLMs to identify and im-

[1]https://openai.com/index/gpt-4/
[2]https://www.anthropic.com/claude

plement targeted enhancements in these different tasks. LLMs offer significant advantages in modern software engineering by understanding complex code structures, identifying performance bottlenecks, and generating actionable recommendations that might not be readily apparent through traditional methods [13], [14].

In this project, we used LLMs to analyze and optimize multiple facets of the SCORE system. We experimented different learning techniques to systematically guide GPT-4 and Claude AI through the optimization process, namely zero-shot, one-shot, and few-shot [15], [16]. In zero-shot learning, the models were provided with general descriptions of the system issues to generate broad optimization strategies, enabling us to understand the models' baseline suggestions without specific code input. For one-shot learning, we gave the models a single code snippet to address a particular API performance issue, which allowed them to provide more focused recommendations. Lastly, few-shot learning involved presenting the models with the major codebase, enabling them to suggest more comprehensive, context-sensitive optimizations across API performance, UI design, and backend scalability.

The insights and optimizations generated by GPT-4 and Claude AI were helpful to re-engineer SCORE system. Specific improvements included reducing API response times for one of the APIs from 26 seconds to 3 seconds through query optimization, improving UI usability by restructuring elements for easier navigation, and enhancing scalability with better database handling and code modularization. The CI/CD pipeline was also streamlined, enabling faster and more reliable deployments, ultimately reducing the time and resources required to maintain the system.

This experience report demonstrates the strategic value of integrating LLMs into legacy system modernization, highlighting their ability to drive tangible improvements across multiple aspects of a system. By harnessing GPT-4 and Claude AI, Volvo Group not only addressed immediate performance concerns but also established a framework for continued optimization and scalability. These experiences underscore the potential of AI-driven solutions in enabling organizations to overcome the limitations of aging software systems, achieve operational efficiency, and future-proof their digital infrastructure in an increasingly competitive environment.

The remainder of this paper is structured as follows. Section II provides an overview of the SCORE system and the challenges faced at Volvo Group because of its limitations. Section III describes the strategies for the re-engineering SCORE, and Section IV details the experiences with the different learning strategies. The lessons learned with our study are described in Section V. Section VI addresses the threats to the validity of our study. Section VII reviews related work, and Section VIII presents the concluding remarks.

II. Background and Problem Statement

Volvo Group[3] is a global leader in the manufacturing and distribution of heavy-duty vehicles, including trucks, buses, and construction equipment, with an established reputation for quality and innovation. In addition to producing advanced vehicles, Volvo's operations rely on an intricate network of warehousing and logistics centers that manage parts and materials essential to vehicle maintenance, repair, and delivery. These centers support Volvo's extensive fleet and help ensure that customers receive reliable and timely service. One such critical hub in the Volvo network is the Byhalia warehouse in Mississippi, USA, where a dedicated team of industrial workers handle complex tasks such as inventory management, storage, packing, and the movement and delivery of truck spare parts. The warehouse plays a vital role in supporting Volvo's supply chain and ensuring that parts are available for customers and service centers when needed.

[3]https://www.volvogroup.com/en/

To manage this vast operation, Volvo developed the SCORE web system in 2017 as a centralized management and human resources (HR) platform. The SCORE system was designed with two main purposes in mind: (i) enabling managers to track and monitor industrial workers' performance, and (ii) assisting HR teams with core functions related to workforce management. For managers, SCORE provides a single platform to view key data on worker attendance, daily performance metrics, and efficiency levels, empowering them to make informed decisions that directly impact productivity and warehouse operations. For HR, SCORE supports essential functions such as onboarding new employees, managing terminations, tracking attendance and leaves, and handling other administrative tasks that are fundamental to workforce management.

Despite SCORE importance, this system has faced several limitations since its creation. After its development, the system quickly became outdated, suffering from a lack of maintenance and architectural degradation. Over time, these issues compounded, hindering its effectiveness and user satisfaction. The current state of SCORE reveals five technical and usability challenges (C) that impact both managerial efficiency and HR operations, described in what follows.

C1. Severe API Performance Issues. One of the most pressing issues with the SCORE system was its slow API response times, which can extend up to multiple seconds to minutes per request. For managers who rely on real-time data in UI using these APIs to monitor industrial workers' performance and attendance, these delays create substantial obstacles. The inability to access timely information negatively impacts decision-making, often resulting in missed opportunities to improve productivity, allocate resources efficiently, or address problems promptly. In a high-paced environment like the Byhalia warehouse, such delays are not just an inconvenience, as they disrupt workflow continuity and reduce the operational agility that is crucial for meeting daily goals.

C2. Database Inefficiencies and Bottlenecks. The backend of the SCORE system is built on a monolithic database structure that has proven inadequate for handling the high volume of data generated by the warehouse's operations. Issues such as slow query performance and poor database indexing lead to lengthy wait times when retrieving or updating worker records, attendance logs, and other critical data. This bottleneck is especially problematic when managers or HR personnel attempt to pull comprehensive reports or access multiple records simultaneously, causing the system to lag or even crash. Consequently, these inefficiencies limit the platform's scalability, making it incapable of handling additional data or expanding to accommodate Volvo's evolving needs.

C3. Outdated User Interface and Poor User Experience. The SCORE system's user interface, developed several years ago, lacks the intuitiveness and user-centric design expected in a modern software application. The interface had a poor user experience due to delay in API response as well, making it difficult for users to access the data they need efficiently. This outdated UI impacts both managers, who need quick insights into worker performance; and HR personnel, who rely on the platform for various administrative functions. The poor user experience leads to decrease in users, reducing their engagement with the system. Also, some users resort to alternative manual processes, which are time-consuming and lead to inconsistencies in data management.

C4. Architectural Limitations and Lack of Flexibility. SCORE was built on a monolithic architecture that restricts the system's adaptability and scalability. Adding new features or modifying existing ones is complex and risky, as changes to one part of the codebase can impact other areas of the system (i.e., Ripple effect [17]). This rigid structure has made it challenging to introduce updates or improvements, trapping the platform in its initial state and limiting Volvo's ability to evolve SCORE in response to emerging operational needs. The lack of architectural flexibility is particularly problematic given the growing complexity of the Byhalia warehouse's operations and the increasing demand for new functionalities that could enhance productivity and resource management.

C5. Absence of Continuous Integration/Continuous Deployment (CI/CD). The SCORE system lacks a CI/CD pipeline, which is essential for modern software systems to manage frequent updates, bug fixes, and feature deployments seamlessly. Without automated deployment, each update or fix requires significant manual effort, increasing the risk of errors and creating long delays in implementing necessary changes. This lack of CI/CD has contributed to SCORE's stagnation, as even minor improvements are postponed or dismissed due to the labor-intensive nature of manual deployments. The absence of a CI/CD pipeline also limits the development team's ability to respond quickly to critical bugs or introduce optimizations that could improve system performance and user experience.

While these challenges represent areas for improvement, the SCORE system remains a foundational tool in supporting workforce management at Volvo's Byhalia warehouse. This context motivated a re-engineering initiative focused on enhancing API performance, optimizing the database, redesigning the UI, improving scalability, and implementing CI/CD processes. The goal was to offer an opportunity to future-proof SCORE, ensuring it continues to meet the needs of both management and HR teams.

III. Using LLMs to Re-engineer SCORE

The proposed solution involves leveraging LLMs, specifically GPT-4 and Claude AI, to generate and implement optimization strategies. These AI-driven technologies were invaluable in analyzing complex code structures, identifying performance bottlenecks, and recommending actionable improvements across various components of SCORE. By using zero-shot, one-shot, and few-shot learning techniques, these models enabled us to evaluate different aspects of the platform (e.g., API efficiency, UI usability, and scalability) and provide targeted recommendations that address specific system needs. This approach allowed Volvo to harness the latest AI-driven advancements, applying intelligent optimizations that can significantly enhance SCORE's functionality and user experience.

Through the AI-driven re-engineering of SCORE, Volvo Group aims to revitalize its operational and HR management capabilities, ensuring that the platform continues to deliver reliable, efficient, and scalable support for its Byhalia warehouse operations. By addressing the identified areas for improvement, Volvo can maintain its commitment to operational excellence and workforce management innovation, positioning the SCORE platform as a resilient tool that can adapt to the dynamic requirements of the automotive and logistics industries.

IV. Experiences on Using LLMs

In this work, we utilized GPT-4 and Claude AI to address the challenges within Volvo's SCORE system. The first goal was to improve the speed and efficiency of four critical APIs whose SQL queries were creating severe performance bottlenecks. These APIs were essential for retrieving critical data from the SCORE system, including attendance, daily performance metrics, and efficiency reports for the workforce. However, certain APIs experienced response times measured in seconds or even minutes due to poorly optimized SQL queries and inefficient backend structures.

To systematically identify and implement improvements, we employed GPT-4 and Claude AI with the aim of generating optimized code and redesign suggestions for these API functions. Additionally, we sought to explore how LLMs could support similar re-engineering tasks, such as UI improvements, scalability enhancements, and CI/CD pipeline integration. To achieve this, we implemented a methodology allowing GPT-4 and Claude AI to independently analyze, suggest, and validate improvements across different levels of guidance. This process was organized into prompt design, exploring the zero-shot, one-shot, and few-shot learning techniques to systematically test how the models could handle different amounts of contextual information.

Prompt Design. We iteratively refine our prompts and use progressive learning techniques to enhance the specificity and effectiveness of the LLM-generated optimizations. The process began with generic descriptions of the performance issues and gradually moved to more detailed prompts, including specific code snippets and full code contexts, allowing us to evaluate how well each model could provide relevant optimizations based on varying levels of input information. For each API, we created three types of prompts, each utilizing a different learning approach:

1) *General Problem Description (Zero Shot Learning):* Starting with a high-level prompt that described the API performance issues and the general need for SQL optimization.
2) *Specific Code Example (One Shot Learning)*: Providing the SQL query and specific sections of code from an underperforming API to prompt more targeted improvements.
3) *Full Code Context (Few Shot Learning)*: Supplying the entire API function along with contextual information to prompt comprehensive recommendations for optimizing each SQL query and the API's logic.

Use of GPT-4 and Claude AI for Optimization. In this experimentation phase, GPT-4 and Claude AI were used independently to optimize underperforming APIs in the SCORE system. The primary issue identified in these APIs was inefficient SQL queries that led to extensive response times due to factors such as poor query structure, unnecessary loops, and repetitive data retrieval. We provide below a comprehensive analysis of the prompts used in each learning technique and details the outputs, effectiveness, model adaptability, and impact on the overall system performance.

A. Zero-Shot Learning Approach

The first experiment involved using Zero-shot learning, where neither GPT-4 nor Claude AI was given any specific code examples. The idea was to allow the models to infer optimization strategies based on a high-level description of the task, without additional context or specific code snippets. The objective was to understand how well the models could propose solutions purely based on their internal knowledge of Node.js, SQL Queries, and API performance, and general software optimization principles. The Zero-shot prompt used was:

> *The Node.js APIs for our management platform experience severe performance delays. Some internal API SQL queries take up to a minute to execute, causing major response time issues. Please provide general recommendations to optimize these APIs for better efficiency and faster response times.*

Recommendations. Both LLMs provided broad recommendations, as follows:

- *GPT-4* Recommended various optimization techniques to improve Node.js API performance, focusing on SQL query efficiency. Suggestions included optimizing query indexes, reducing unnecessary data retrieval, limiting complex joins, and using in-memory caching with tools like Redis for frequently accessed data. It emphasized asynchronous processing, connection pooling, and database load distribution, as well as monitoring performance to identify specific bottlenecks.
- *Claude AI* provided a practical, example-rich guide with specific Node.js code implementations for optimizing database interactions. Techniques included improving query selectivity, connection pooling, and query caching with examples for in-memory caching, query batching, pagination, and request queuing. Additionally, it emphasized SQL indexing, error logging for slow queries, and a rate-limiting approach to prevent database overload.

The insights were valuable into improving Node.js performance and SQL efficiency. While GPT-4 delivered a high-level optimization strategy, Claude AI included actionable code examples, making its recommendations more applicable to real-world Node.js applications.

Effectiveness. The zero-shot learning approach yielded valuable high-level recommendations but lacked the granularity needed for direct implementation. Both LLMs provided relevant insights into optimizing SQL queries and API performance, yet the absence of code-specific guidance limited their immediate applicability to SCORE APIs.

Learning and Adaptability. Without specific code examples, neither model demonstrated contextual adaptability based on the unique SQL queries or API structures used. Instead, both relied on general best practices for enhancing performance, missing the opportunity to address specific bottlenecks within the SCORE APIs.

Differences in Output by each LLM. GPT-4 emphasized SQL-centric improvements, such as indexing and query optimization, while Claude AI focused on infrastructure-level strategies like connection pooling, caching, and batch processing. Claude's suggestions were more aligned with high-load, real-time application needs, but neither model offered detailed query-level fixes that could be applied directly.

Prompt Iterations and Changes. In this step, prompt iteration was unnecessary, as both models provided consistent outputs across attempts. However, the general insights gathered informed the design of future prompts, where specific code examples were incorporated to target more precise solutions.

Impact on Performance. No immediate performance gains were realized due to the generalized nature of the recommendations. However, the Zero-shot learning insights offered a foundation for refining prompts in subsequent phases, guiding the approach toward more targeted, actionable suggestions.

B. One-Shot Learning Approach

The second experiment involved one-shot learning, where the models were provided with a small code snippet with an inefficient SQL query to offer more focused and context-specific recommendations. This approach allowed the models to analyze a specific, real-world instance of a poorly performing API and generate targeted optimizations. The one-shot prompt was:

Optimize the following Node.js API for reduced response time. This API takes 20-30 seconds to fetch data due to SQL query inefficiency:

<code in Listing 1>

Recommendations. In response, both models provided more tailored suggestions:

- *GPT-4* provided a comprehensive analysis of the query's inefficiencies, targeting issues that commonly impact performance, such as unnecessary complexity in sub-queries, redundant SQL variables, and multiple joins across tables. Its response emphasized restructuring these elements by simplifying the query to enhance readability and streamline processing. It suggested reducing joins by consolidating key information into fewer steps, thus eliminating nested sub-queries and improving overall clarity. The use of *NULLIF* in aggregations helped prevent division errors, leading to smoother handling of null values in calculation-heavy operations. Additionally, GPT-4 optimized the JavaScript layer by moving from Promise chaining to async/await, enhancing readability and reducing asynchronous complexity. These improvements addressed specific points of inefficiency while maintaining the query's original purpose, providing practical changes that resulted in a noticeable, though modest, improvement in query execution time.
- *Claude AI* Claude AI delivered a thorough breakdown of performance issues and offered more infrastructure-based solutions aimed at SQL efficiency. Key recommendations included converting unnecessary unions to more optimal alternatives and adding indexes to accelerate frequently queried columns. By organizing the SQL structure into logical segments, Claude's optimization minimized table scans, avoided excessive nested sub-queries, and handled date conversions more efficiently with CONVERT instead of CAST. Additional indexes on frequently used fields like `UserID`, `GroupCode`, and `TransactionTime`, enhanced the query's response time, making it better suited for large datasets with heavy read operations. Claude AI also emphasized readability, using well-defined naming conventions and a clean structure that facilitates easier debugging and maintenance in the future. Together, these changes improved SQL performance and slightly reduced execution time, making the code more maintainable and efficient.

Effectiveness. The one-shot responses from both GPT-4 and Claude AI were effective in providing clear, actionable optimizations that were easy to implement. Each model focused on enhancing the query's structure, with GPT-4 zeroing in on query simplification and async processing, while Claude AI provided a balanced approach between indexing and structure-based optimizations. This approach made the recommendations directly implementable and beneficial, leading to a small but measurable improvement in performance.

Listing 1. Code and SQL query used with the One-Shot prompt (changed for compliance purpose)

```
const calculateAverageTransactions = (request, response,
      next) => {
  // Query data for specific transaction types
  const environment = request.userData.environment;
  const source = request.query.source;
  const operatorId = request.query.operatorId;
  const includeTypes = request.query.includeTypes ===
      "false" ? false : true;
  const actionType = request.query.actionType;
  let activeUserId = request.query.operatorId;
  let transactionTypes = [];

  const sqlFetchCorrectUser = `
    SELECT
    MappedUserID
    FROM UserMapping
    WHERE UserID = @operatorId
  `;

  const applyUserMapping = async () => {
    return await new Promise((resolve, reject) => {
      let sqlRequest = new Request(sqlFetchCorrectUser,
            (error) => {
        if (error) {
          console.error(error);
          reject(error);
        }
      });
      sqlRequest.addParameter("operatorId",
            TYPES.NVarChar, operatorId);
      let mappedUserId = false;

      sqlRequest.on("row", (column) => {
        if (column[0].value) {
          mappedUserId = column[0].value;
        }
      });
      sqlRequest.on("requestCompleted", () => {
        if (mappedUserId) {
          activeUserId = mappedUserId;
        }
        resolve();
      });
      connection.execSql(sqlRequest);
    });
  };

  const sqlFetchTransactionSummary = `
    SELECT
        CAST(TransactionTime AS DATE) AS Date,
        (SUM(CASE WHEN (Operators.UserID = @operatorID)
             THEN 1 ELSE 0 END) /
             NULLIF(COUNT(DISTINCT(CASE WHEN
             Operators.UserID = @operatorID THEN
             DATEPART(HOUR, TransactionTime) END)), 0)) AS
             'Operator Avg',
        (COUNT(Action)/
        (COUNT(DISTINCT(SystemRecords.UserID))
        *COUNT(DISTINCT(DATEPART(HOUR, TransactionTime)))))
             AS 'Group Avg'
    FROM (SELECT * FROM Operators UNION SELECT * FROM
          TempOperators) AS Operators
        LEFT JOIN UserMapping
        ON Operators.UserID = UserMapping.UserID
        RIGHT JOIN SystemRecords
        ON CASE WHEN (UserMapping.MappedUserID IS NOT NULL)
             THEN UserMapping.MappedUserID ELSE
             Operators.UserID END = SystemRecords.UserID
    WHERE Operators.GroupCode = (SELECT
        Operators.GroupCode
        FROM (SELECT * FROM Operators UNION SELECT * FROM
             TempOperators) AS Operators
        WHERE Operators.UserID = @operatorID
    )
    GROUP BY CAST(TransactionTime AS DATE)
    ORDER BY CAST(TransactionTime AS DATE)`;
};
```

Learning and Adaptability. Although both models offered useful insights, neither fully adapted to the unique SQL nor data structures specific to the SCORE APIs. Instead, they provided solutions rooted in best practices for SQL optimization. This approach helped address generic inefficiencies, although some SCORE-specific performance challenges were not entirely addressed, leaving room for further context-based tuning.

Differences in Output by each LLM. GPT-4 focused on SQL query efficiency through simplification and async operations in JavaScript, suggesting a holistic optimization approach that reduced overall query complexity. Meanwhile, Claude AI placed more emphasis on robust indexing, structuring queries logically, and using inner joins when possible to streamline data access. Claude's output offered a more balanced solution for high-volume query optimization, while GPT-4's recommendations were beneficial for improving code readability and reducing JavaScript complexity.

Prompt Iterations and Changes. The responses from both models were consistent, yielding useful insights on the first attempt. The clear recommendations allowed prompt iteration to focus on more context-specific optimizations in subsequent phases. These outputs confirmed the effectiveness of one-shot prompting with code-based examples for quickly identifying SQL and API improvements.

Impact on Performance. The one-shot learning suggestions led to a moderate performance boost, with query execution showing slight improvement after the recommended indexing and restructuring were implemented. Though these gains were not transformative, the optimizations established a solid foundation for further performance tuning and helped shape a more refined approach for future prompts.

C. Few-Shot Learning Approach

The third experiment was with the few-shot learning, where the LLMs were provided with the major API codebase. This included the API logic, database interaction details if required, and context about how different modules interacted. By presenting the models with the major system, we aimed to evaluate their capacity to analyze and optimize a more complex, integrated system. In this research case prompt, we are giving query based on one shot learning approach to improve the API that we are considering here, The Few-shot prompt was:

> *Here is the code with 2 faulty queries for the Node.js API, which is performing poorly:*
>
> <code in Listing 2>

Recommendations. In response, both LLMs provided more tailored suggestions:

- *GPT-4* highlighted several key inefficiencies in the original SQL queries (`sqlFetchTotalTransactions` and `sqlFetchTransactionsByType`). Notable issues included repeated logic, where both queries exhibited similar structures with redundant sub-queries and joins, leading to unnecessary complexity and potential performance degradation. The use of the `UNION` operator was criticized for requiring complete scans of both the `Operators` and `TempOperators` tables, which is computationally expensive. Additionally, complex `JOIN` conditions, particularly the conditional joins involving the `UserMapping` table, were identified as areas for simplification. The nested sub-query for retrieving the `GroupID` was also flagged for being repetitively executed, adding overhead without necessity. To optimize these queries, GPT-4 recommended an enhanced version that employed a Common Table Expression (CTE) called `OperatorData`, which consolidated the union

Listing 2. SQL queries used with the Few-Shot prompt (changed for compliance purpose)

```
const sqlFetchTotalTransactions = `
   SELECT
      CAST(EventTime AS DATE) AS Date,
      (SUM(CASE WHEN (Operators.OperatorID = @operatorID)
           THEN 1 ELSE 0 END) /
           NULLIF(COUNT(DISTINCT(CASE WHEN
           Operators.OperatorID = @operatorID THEN
           DATEPART(HOUR, EventTime) END)), 0)) AS
           'Operator Avg',
      (COUNT(Event)/(COUNT(DISTINCT(Recordings.OperatorID))*
      COUNT(DISTINCT(DATEPART(HOUR, EventTime))))) AS
           'Group Avg'
   FROM (SELECT * FROM Operators UNION SELECT * FROM
        TempOperators) AS Operators
      LEFT JOIN UserMapping
      ON Operators.OperatorID = UserMapping.OperatorID
      RIGHT JOIN Recordings
      ON CASE WHEN (UserMapping.MappedOperatorID IS NOT
           NULL) THEN UserMapping.MappedOperatorID ELSE
           Operators.OperatorID END =
           Recordings.OperatorID
   WHERE Operators.GroupID = (SELECT
         Operators.GroupID
         FROM (SELECT * FROM Operators UNION SELECT *
              FROM TempOperators) AS Operators
         WHERE Operators.OperatorID = @operatorID
      )
   GROUP BY CAST(EventTime AS DATE)
   ORDER BY CAST(EventTime AS DATE)`;

const sqlFetchTransactionsByType = `
   SELECT
      CAST(EventTime AS DATE) AS Date,
      (SUM(CASE WHEN (Operators.OperatorID = @operatorID)
           THEN 1 ELSE 0 END) /
           NULLIF(COUNT(DISTINCT(CASE WHEN
           Operators.OperatorID = @operatorID THEN
           DATEPART(HOUR, EventTime) END)), 0)) AS
           'Operator Avg',
      (COUNT(Event)/(COUNT(DISTINCT(Recordings.OperatorID))*
      COUNT(DISTINCT(DATEPART(HOUR, EventTime))))) AS
           'Group Avg'
   FROM (SELECT * FROM Operators UNION SELECT * FROM
        TempOperators) AS Operators
      LEFT JOIN UserMapping
      ON Operators.OperatorID = UserMapping.OperatorID
      RIGHT JOIN Recordings
      ON CASE WHEN (UserMapping.MappedOperatorID IS NOT
           NULL) THEN UserMapping.MappedOperatorID ELSE
           Operators.OperatorID END =
           Recordings.OperatorID
   WHERE Operators.GroupID = (SELECT
         Operators.GroupID
         FROM (SELECT * FROM Operators UNION SELECT *
              FROM TempOperators) AS Operators
         WHERE Operators.OperatorID = @operatorID
      )
      AND
      Recordings.EventType = @eventType
   GROUP BY CAST(EventTime AS DATE)
   ORDER BY CAST(EventTime AS DATE)`;
```

of `Operators` and `TempOperators` for reuse in both queries. This significantly reduced redundancy and improved clarity. The use of `COALESCE` in `JOIN` conditions was introduced to streamline the logic and enhance readability. By implementing `UNION ALL`, the optimized version bypassed the overhead of duplicate checks since there were no expected duplicates between the two tables. Additionally, the recommendation to multiply counts by 1.0 ensured that division operations yielded floating-point results, which are more suitable for averaging calculations. Overall, GPT-4's proposed changes aimed to enhance the performance and maintainability of the queries by leveraging SQL best practices.

- *Claude AI* conducted a similar analysis and identified key areas for optimization. The issues included redundant sub-queries in both the `FROM` and `WHERE` clauses, multiple casting operations on the `EventTime`, unnecessary `UNION` operations, and complex `CASE` statements that could be simplified. To tackle these problems, Claude AI proposed creating two CTEs: (i) `CombinedOperators` for consolidating the `Operators` and `TempOperators` tables to avoid repeated `UNION` operations, and (ii) `UserGroup` to streamline the retrieval of the user's group identifier. In the optimized version, Claude AI suggested using the `DATE()` function instead of repeated `CAST` operations on `EventTime`, which enhanced performance. The queries were designed to compute operator and group transaction averages over time, with simplified joins replacing complex conditions. The join structure was refined by utilizing `COALESCE` for better clarity in operator ID mapping and removing redundant operations. Suggested indexes were also provided to improve execution speed, focusing on commonly queried fields across the involved tables. Claude AI's formatting and organization of the code improved readability, which facilitates maintenance and debugging.

Effectiveness. Both GPT-4 and Claude AI produced optimized SQL queries that addressed the identified performance issues, such as redundant sub-queries and complex join conditions. While GPT-4 provided a more streamlined approach using a Common Table Expression (CTE) to consolidate employee data, Claude AI also leveraged CTEs but focused on improving readability and maintainability through a clearer structure. The optimized versions offered by both models resulted in reduced complexity and improved clarity, enhancing the overall effectiveness of the queries in calculating operators and group averages.

Learning and Adaptability. Despite the strengths of both LLMs, neither demonstrated the ability to learn nor adapt based on specific SQL or API structures provided in the prompts. Instead, both relied on general best practices and optimizations common to SQL query performance, as it was doing in the one shot learning approach. While the suggestions were insightful, they lacked customization to the unique context of the queries, indicating a limitation in the adaptability of the LLMs to specific codebases or frameworks.

Differences in Output by each LLM. The outputs from GPT-4 and Claude AI exhibited notable differences in their approaches. GPT-4 emphasized refactoring through the use of CTEs to reduce redundancy and improve the efficiency of joins, while Claude AI focused on enhancing readability and maintaining a structured format, also utilizing CTEs. Additionally, Claude AI proposed suggested indexes for performance improvement, while GPT-4 concentrated more on optimizing the existing query structure. This divergence in focus highlights the varying strengths of each model in tackling similar problems.

Prompt Iterations and Changes. In this phase, there were three multiple iterations of prompts done, to have both LLMs generate effective output in response to the provided queries. The insights from the initial prompts helped shape subsequent designs, but both LLMs offered a cohesive response that met the requirements of the task.

Impact on Performance. The optimizations proposed by both LLMs resulted in marginal improvements in query performance. The elimination of redundancy and simplification of join logic should lead to better execution times. Although specific performance metrics were assessed for a particular API and were improved considerably post-implementation, there is still scope to quantify these improvements accurately. The refined queries were designed to be more efficient, contributing to an overall enhanced performance in processing transaction data while maintaining the original functionality of the queries. Listing 3 presents the optimized query provided by GPT-4:

D. Performance Monitoring and Evaluation.

To evaluate the effectiveness of the LLMs' optimizations, we implemented their suggestions and monitored the API's performance. Key metrics included API response time reduction, resource utilization improvements, and database efficiency.

GPT-4's optimizations resulted in a significant reduction in API response times, with the most noticeable improvement in database query performance due to the implementation of pagination and batch processing. In our research API, ITS response times dropped from 26 seconds to approximately 3 seconds, a 300% improvement and more APIs too had the similar impact. Claude AI's architectural suggestions, while more ambitious, laid the groundwork for future scalability. Although Claude AI's changes were more complex to implement, they offered valuable insights into how the system could be restructured to handle higher traffic volumes and improve long-term performance. In conclusion, GPT-4 proved to be highly effective in providing granular, context-specific optimizations that were easy to implement and resulted in immediate performance gains. Claude AI, on the other hand, offered a more strategic, architecture-level view, which would be beneficial for future scalability and system-wide improvements. Both LLMs demonstrated their potential in optimizing

Listing 3. Optimized Code Provided by GPT-4 (changed for compliance purpose)

```
-- Optimized Common Table Expression for employees
WITH OperatorData AS (
   SELECT
      OperatorID,
      GroupID,
      DATEPART(HOUR, EventTime) AS EventHour,
      EventTime
   FROM
      (SELECT * FROM Operators UNION ALL SELECT * FROM
          TempOperators) AS Operators
)

SELECT
   CAST(rec.EventTime AS DATE) AS Date,
   (SUM(CASE WHEN op.OperatorID = @operatorID THEN 1 ELSE
       0 END) * 1.0 /
    NULLIF(COUNT(DISTINCT CASE WHEN op.OperatorID =
         @operatorID THEN op.EventHour END), 0)) AS
         'Operator Avg',
   (COUNT(rec.Event) * 1.0 /
    NULLIF(COUNT(DISTINCT rec.OperatorID) *
         COUNT(DISTINCT op.EventHour), 0)) AS 'Group Avg'
FROM
   Recordings rec
JOIN
   OperatorData op ON op.OperatorID =
       COALESCE(um.MappedOperatorID, op.OperatorID)
LEFT JOIN
   UserMapping um ON op.OperatorID = um.OperatorID
WHERE
   op.GroupID = (SELECT GroupID FROM OperatorData WHERE
       OperatorID = @operatorID LIMIT 1)
GROUP BY
   CAST(rec.EventTime AS DATE)
ORDER BY
   CAST(rec.EventTime AS DATE);

-- For transactions by event type
SELECT
   CAST(rec.EventTime AS DATE) AS Date,
   (SUM(CASE WHEN op.OperatorID = @operatorID THEN 1 ELSE
       0 END) * 1.0 /
    NULLIF(COUNT(DISTINCT CASE WHEN op.OperatorID =
         @operatorID THEN op.EventHour END), 0)) AS
         'Operator Avg',
   (COUNT(rec.Event) * 1.0 /
    NULLIF(COUNT(DISTINCT rec.OperatorID) *
         COUNT(DISTINCT op.EventHour), 0)) AS 'Group Avg'
FROM
   Recordings rec
JOIN
   OperatorData op ON op.OperatorID =
       COALESCE(um.MappedOperatorID, op.OperatorID)
LEFT JOIN
   UserMapping um ON op.OperatorID = um.OperatorID
WHERE
   op.GroupID = (SELECT GroupID FROM OperatorData WHERE
       OperatorID = @operatorID LIMIT 1)
   AND
   rec.EventType = @eventType
GROUP BY
   CAST(rec.EventTime AS DATE)
ORDER BY
   CAST(rec.EventTime AS DATE);
```

API performance through a structured, step-by-step approach, each excelling in different aspects of the optimization process.

V. Lessons for Practitioners

From our experiences on conducting this work, we derive practical lessons valuable for practitioners and researchers on exploring LLMs for re-engineering and optimization tasks.

Prompt Strategies. Using a range of prompt strategies, namely zero-shot, one-shot, and few-shot learning, proved beneficial in guiding the models to produce increasingly actionable and context-aware recommendations:

- *Zero-Shot Learning* was useful for high-level suggestions, but yielded generalized advice that was less tailored to specific scenarios, requiring further refinement to be directly implementable.
- *One-Shot Learning*, by incorporating a single example, improved relevance and specificity, allowing the model to understand the query structure better and generate recommendations with some contextual alignment.
- *Few-Shot Learning* was particularly effective, providing the model with multiple examples to draw from, leading to detailed, highly contextualized optimizations. This strategy resulted in concrete improvements in query efficiency and execution times when applied to our SQL and API queries. Practitioners should invest time in creating well-structured prompt sequences (with several examples), which greatly enhance the model's understanding and output quality.

Exploring Different LLMs. Experimenting with GPT-4 and Claude AI highlighted the strengths and distinct focuses each model brought to query optimization. This diversity in LLMs allowed for a broader exploration of performance solutions. GPT-4 provided deeper insights into SQL-specific improvements, with a focus on query refinement techniques such as indexing, join optimization, and sub-query reduction. These solutions were particularly useful for addressing SQL bottlenecks at the code level. Claude AI offered complementary perspectives, including infrastructure-level suggestions and efficient use of database features like connection pooling, making it especially suited for high-load scenarios. Recommendation: Practitioners should consider using multiple models in tandem to harness a broader range of optimization techniques, especially for complex tasks where code-level and infrastructure improvements are both crucial.

These strategies underscore the importance of prompt design and model selection in achieving meaningful performance improvements. By carefully choosing prompt strategies and exploring various models, practitioners can leverage LLMs effectively for tailored and impactful optimization.

Refining LLM Solutions. The solutions generated by LLMs, while insightful, often required manual refinement and iterative adjustments to fit seamlessly into the existing codebase. For instance, when the models recommended SQL optimizations, we had to manually integrate these solutions into the code, addressing any discrepancies with the system's architecture or business logic. Furthermore, LLMs sometimes produced overly generalized or impractical solutions that didn't fully align with the constraints of the SCORE system's legacy infrastructure. In such cases, we refined the solutions by adjusting the models' outputs and re-testing the modified code multiple times. This iterative process ensured that the LLM-generated suggestions were tailored to meet the specific requirements of the platform while enhancing overall performance. Practitioners should anticipate this manual refinement process, as it bridges the gap between AI recommendations and real-world implementation.

LLMs for Different Tasks in Legacy System Modernization. Our experiences demonstrate the potential for LLMs in various aspects of legacy system modernization. While this project focused on API optimization, the insights extended into other areas, such as UX improvement, scalability planning, and database efficiency. Here are some key observations:

- *API Optimization*: LLMs provided effective, actionable suggestions for improving API response times, especially in query optimization and data retrieval techniques. Techniques such as pagination, query restructuring, and indexing recommendations were immediately useful and directly impacted performance.
- *UX Improvements:* LLMs' suggestions in UI modularization and interface simplification showed potential. By refining UI-related prompts, developers can obtain design recommendations for user engagement, accessibility, and efficiency, improving the UX of legacy systems.
- *Scalability*: LLMs were effective in providing high-level recommendations for architectural improvements related to scalability. For instance, Claude AI suggested moving to a microservice architecture for handling high data volumes and traffic, though this was outside the scope of our legacy system's immediate needs. However, these insights serve as valuable guidelines for planning long-term scalability in legacy systems.
- *Database Efficiency:* Database optimizations suggested by the LLMs, such as implementing indexing strategies and refining query structures, directly contributed to performance gains. The models also suggested adopting asynchronous processing for batch operations, which would support larger-scale data processing while minimizing impact on the system's overall performance.

Each of these areas shows how LLMs can be applied to several re-engineering tasks. Practitioners looking to modernize other legacy system components can leverage LLMs for broader guidance, from back-end performance enhancements to front-end design strategies.

Constraints Related to Token Size and Request Limits. A significant limitation encountered was the token size constraint imposed by GPT-4 and Claude AI, which limits the amount of code and context that can be processed in a single request. These restrictions required adjustments, especially during the Few-shot learning phase, where the models were presented with full API codebases and SQL queries. Both GPT-4 and Claude AI have input length limitations, with GPT-4 accepting fewer tokens than Claude AI, impacting how much information could be provided for comprehensive analysis.

To work around this, we split the code into segments and adjusted prompts to focus on one aspect of the API at a time. Although this process allowed us to stay within the token limits, it also increased the time spent on prompt refinement, as multiple iterations were often needed to yield a cohesive solution. Practitioners should be prepared for these token limitations, particularly when dealing with large codebases, and should consider structuring prompts in manageable segments to ensure effective LLM analysis.

Data Privacy and Security Concerns in Prompts. One of the essential considerations when using LLMs, especially for business-critical systems, is data privacy and security. In this project, we ensured that sensitive information, such as employee records or proprietary company data, was excluded from prompts. Instead, anonymized or generalized data was used to illustrate API and database structures without exposing personal information.

Practitioners should exercise caution when sharing data in prompts, as LLMs process requests through external APIs that may store or analyze inputs for model improvement. Organizations handling sensitive or proprietary data should explore options for fine-tuning LLMs on private datasets or consider deploying on-premise models, when possible, to maintain data security. Establishing clear guidelines on data handling in LLM interactions can prevent inadvertent exposure of confidential information.

VI. Threats to Validity

Several potential threats to the validity of our findings must be considered when interpreting the results of this study. First, the use of LLMs, such as GPT-4 and Claude AI, for API optimization is still an emerging field, and the effectiveness of their suggestions may vary depending on the complexity and specificity of the task. Although both LLMs provided useful insights for improving the performance of the Node.js APIs, their responses were highly dependent on the quality and clarity of the prompts given. This introduces a potential threat of bias in the prompt engineering process, as the way we framed the optimization tasks could have influenced the LLMs' outputs.

Although we implemented and evaluated many of the optimizations suggested by the models, we did not conduct a large-scale comparative study across multiple projects or domains. The optimizations were applied solely to the SCORE system's APIs, which may limit the generalizability of our findings. The results may not fully extend to other types of APIs or systems with different architectures, workloads, or performance bottlenecks.

Another threat is related to the evaluation metrics used to assess the models' performance. We primarily focused on API response times and database efficiency, but other factors such as code maintainability, scalability, and security were not deeply explored. The optimizations proposed by the LLMs may have unintended consequences in these areas that were not captured in our performance monitoring.

The black-box nature of LLMs poses a threat to understanding the rationale behind some of their recommendations. While GPT-4 and Claude AI provided useful suggestions, the reasoning behind certain architectural changes, such as the adoption of microservices or asynchronous queues, may require deeper domain knowledge that was not directly verifiable within the scope of this study. This lack of transparency could lead to trust issues in critical system environments.

Finally, the evaluation timeframe was limited, and long-term monitoring of the optimizations was not conducted. The improvements observed in the short term, such as reduced response times, may not hold up under sustained or increased load, which introduces a threat to the internal validity of the results. Future work should include a more comprehensive evaluation over time to assess the robustness and sustainability of the applied optimizations.

VII. Related Work

Large Language Models (LLMs) have been explored in various aspects of software engineering, such as code understanding, code generation, review, requirements engineering, and optimization [18]–[20]. Tools like GitHub Copilot and Amazon CodeWhisperer illustrate the power of LLMs to generate code suggestions, automate syntax correction, and expedite debugging [21], [22]. By interpreting natural-language prompts, these tools help developers by quickly generating relevant code snippets and addressing common coding issues, which in turn reduces time spent on repetitive tasks and accelerates early development stages [23]. This capability has proven especially useful for both novice and experienced developers in improving productivity [24].

Sauvola et al. [19] explores the application of AI-driven automation in legacy software maintenance, traditional software operations, and networked applications. Their study shows that generative AI, including LLMs, can improve productivity, cut maintenance costs, and enable developers to handle complex maintenance and modernization tasks dynamically. By supporting parallelized development processes and handling repetitive updates, generative AI tools show promise for large-scale software modernization, allowing legacy systems to be updated more efficiently and effectively.

In the industry, Ericsson's Llama-based chatbot employs RAG to answer complex CI/CD-related questions, retrieving relevant documentation and technical details to guide engineers in deployment and troubleshooting [24]. Retrieval-Augmented Generation (RAG) frameworks further enhance LLM applications by incorporating document retrieval to respond to specific, context-driven queries [25]. Similarly, QAssist [26] uses RAG to assist software engineers in analyzing project requirements, retrieving pertinent information from specifications to support complex decision-making.

In team-based development and educational settings, LLMs have proven useful for collaborative tasks, including initial project setup, syntax resolution, and debugging. A study by Rasnayaka et al. [21] examined the role of LLMs in student-led software projects, where LLMs facilitated foundational code generation and expedited the debugging process. LLMs also supports requirements engineering. Krishna et al. [20] evaluated GPT-4 and CodeLlama in generating Software Requirements Specifications (SRS) documents, demonstrating that LLMs can produce draft requirements that closely align with those created by entry-level software engineers. However, in both cases, manual oversight and refinement were necessary to ensure that the results met specific project standards.

The COLLMS framework by Truong et al. [27] addresses the need for coordinating LLMs with platform-specific knowledge in complex development environments like edge-cloud systems. COLLMS integrates LLM-driven automation with platform knowledge to manage deployment policies, observability, and architectural configurations, especially for edge-cloud applications. This integration highlights a key limitation in using standalone LLMs for complex systems and emphasizes the need for contextual data and platform knowledge to make LLM-driven automation more effective.

These studies collectively highlight the expanding role of LLMs in automating various software development workflows, including legacy system modernization, API optimization, CI/CD integration, and database performance. By combining LLM insights with RAG frameworks, platform-specific knowledge, and role-based collaboration, these tools enable a new level of efficiency and scalability in software engineering, paving the way for comprehensive, AI-assisted software modernization.

VIII. Conclusion

This research demonstrates the potential of using LLMs to optimize Node.js APIs, showing significant improvements in both performance and efficiency. By employing different learning paradigms (i.e., zero-shot, one-shot, and few-shot learning) we were able to experiment the capabilities of GPT-4 and Claude AI under various levels of context and input details. The results indicated that both LLMs can provide valuable insights and actionable suggestions for improving API performance, with GPT-4 excelling in code-specific optimizations and Claude AI offering broader architectural recommendations. By the application of these optimizations, we achieved a dramatic reduction in API response times for one of our research APIs (from 26 seconds to 3 seconds) and improved the overall efficiency of the system.

For developers and teams considering LLMs for software optimization tasks, this study highlights several key lessons. First, prompt design is critical; clear and well-structured prompts that provide enough context can significantly improve the relevance of the LLMs' recommendations. Second, while LLMs can offer powerful optimizations, they should be seen as augmentative tools rather than standalone solutions. Developers must critically evaluate and implement the suggestions, as AI-generated optimizations may sometimes require deeper human intervention, especially when dealing with complex system dependencies or legacy architectures. Lastly, incorporating LLMs into regular re-engineering and optimization workflows can provide continuous benefits, offering fresh perspectives and reducing the cognitive load on developers during performance tuning or architectural decision-making.

In conclusion, our work suggests that AI-driven tools are valuable in software re-engineering and optimization. When used correctly, LLMs can substantially enhance the quality, scalability, and maintainability of applications.

References

[1] M. S. Bhatia and S. Kumar, "Critical success factors of industry 4.0 in automotive manufacturing industry," *IEEE Transactions on Engineering Management*, vol. 69, no. 5, pp. 2439–2453, 2020.

[2] M. Tian, Y. Chen, G. Tian, W. Huang, and C. Hu, "The role of digital transformation practices in the operations improvement in manufacturing firms: A practice-based view," *International Journal of Production Economics*, vol. 262, p. 108929, 2023.

[3] P. Gölzer and A. Fritzsche, "Data-driven operations management: organisational implications of the digital transformation in industrial practice," *Production Planning & Control*, vol. 28, no. 16, pp. 1332–1343, 2017.

[4] R. Cao and M. Iansiti, "Digital transformation, data architecture, and legacy systems," *Journal of Digital economy*, vol. 1, no. 1, pp. 1–19, 2022.

[5] Z. Irani, R. M. Abril, V. Weerakkody, A. Omar, and U. Sivarajah, "The impact of legacy systems on digital transformation in european public administration: Lesson learned from a multi case analysis," *Government Information Quarterly*, vol. 40, no. 1, p. 101784, 2023.

[6] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, "How do professionals perceive legacy systems and software modernization?," in *36th International Conference on Software Engineering*, pp. 36–47, 2014.

[7] W. K. Assunção, T. E. Colanzi, L. Carvalho, A. Garcia, J. A. Pereira, M. J. de Lima, and C. Lucena, "Analysis of a many-objective optimization approach for identifying microservices from legacy systems," *Empirical Software Engineering*, vol. 27, no. 2, p. 51, 2022.

[8] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. USA: Addison-Wesley, 2003.

[9] D. Wolfart, W. K. G. Assunção, I. F. da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, and D. d. N. Paza, "Modernizing legacy systems with microservices: A roadmap," in *25th Evaluation and Assessment in Software Engineering (EASE)*, p. 149–159, ACM, 2021.

[10] E. Y. Nakagawa, P. O. Antonino, F. Schnicke, T. Kuhn, and P. Liggesmeyer, "Continuous systems and software engineering for industry 4.0: A disruptive view," *Information and software technology*, vol. 135, p. 106562, 2021.

[11] C. C. Raţiu, C. Mayr-Dorn, W. K. G. Assunção, and A. Egyed, "Taming cross-tool traceability in the wild," in *2023 IEEE 31st International Requirements Engineering Conference (RE)*, pp. 233–243, 2023.

[12] R. Kasauli, E. Knauss, J. Horkoff, G. Liebel, and F. G. de Oliveira Neto, "Requirements engineering challenges and practices in large-scale agile system development," *Journal of Systems and Software*, vol. 172, p. 110851, 2021.

[13] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[14] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, "An empirical study on usage and perceptions of llms in a software engineering project," in *1st International Workshop on Large Language Models for Code*, LLM4Code '24, (New York, NY, USA), p. 111–118, Association for Computing Machinery, 2024.

[15] S. Rahman, S. Khan, and F. Porikli, "A unified approach for conventional zero-shot, generalized zero-shot, and few-shot learning," *IEEE Transactions on Image Processing*, vol. 27, no. 11, pp. 5652–5667, 2018.

[16] S. Kadam and V. Vaidya, "Review and analysis of zero, one and few shot learning approaches," in *Intelligent Systems Design and Applications: 18th International Conference on Intelligent Systems Design and Applications (ISDA)*, pp. 100–112, Springer, 2020.

[17] S. S. Yau, J. S. Collofello, and T. MacGregor, "Ripple effect analysis of software maintenance," in *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC'78.*, pp. 60–65, IEEE, 1978.

[18] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[19] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of software development with generative ai," *Automated Software Engineering*, vol. 31, no. 1, p. 26, 2024.

[20] M. Krishna, B. Gaur, A. Verma, and P. Jalote, "Using llms in software requirements specifications: An empirical evaluation," *arXiv preprint arXiv:2404.17842*, 2024.

[21] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, "An empirical study on usage and perceptions of llms in a software engineering project," in *1st International Workshop on Large Language Models for Code*, pp. 111–118, 2024.

[22] M. Boukhlif, N. Kharmoum, and M. Hanine, "Llms for intelligent software testing: a comparative study," in *7th International Conference on Networking, Intelligent Systems and Security*, pp. 1–8, 2024.

[23] J. Jahić and A. Sami, "State of practice: Llms in software engineering and software architecture," in *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, pp. 311–318, IEEE, 2024.

[24] D. Chaudhary, S. L. Vadlamani, D. Thomas, S. Nejati, and M. Sabetzadeh, "Developing a llama-based chatbot for ci/cd question answering: A case study at ericsson," *arXiv preprint arXiv:2408.09277*, 2024.

[25] A. Salemi and H. Zamani, "Evaluating retrieval quality in retrieval-augmented generation," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2395–2400, 2024.

[26] S. Ezzini, S. Abualhaija, C. Arora, and M. Sabetzadeh, "Ai-based question answering assistance for analyzing natural-language requirements," in *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1277–1289, IEEE, 2023.

[27] H.-L. Truong, M. Vukovic, and R. Pavuluri, "On coordinating llms and platform knowledge for software modernization and new developments," in *2024 IEEE International Conference on Software Services Engineering (SSE)*, pp. 188–193, IEEE, 2024.