DOI: 10.15849/IJASCA.250730.11

Int. J. Advance Soft Compu. Appl, Vol. 17, No. 2, July 2025
Print ISSN: 2710-1274, Online ISSN: 2074-8523
Copyright © Al-Zaytoonah University of Jordan (ZUJ)

Assessing ChatGPT's Capability in Extracting Architectural Elements from Software Systems

Hind Milhem[1], Neil B. Harrison[2]
[1]Information Technology Department, Faculty of Prince Al-Hussein Bin Abdallah II For Information Technology, The Hashemite University, Zarqa, Jordan, P. O. Box 330127
E-mail: hinda_is@hu.edu.jo
[2]Department of Computer Science, Utah Vally University, Orem, Utah, USA
E-mail: neil.harrison@uvu.edu

Abstract

Software architecture plays a critical role in ensuring system quality, yet manually identifying architectural patterns and tactics in codebases remains time-consuming and error prone. While traditional tools like Archie automate parts of this process, the emergence of large language models (LLMs) like ChatGPT offers new opportunities for scalable, intelligent analysis. This work aims to investigate the capability of ChatGPT, a large language model (LLM), for software design analysis, specifically in detecting architectural patterns and tactics. We evaluated ChatGPT-4o and Archie on five open-source systems (Apache Storm, Flink, Spark, Gradle, and Maven) using precision, recall, F1-score, and accuracy metrics. The methodology involved: (1) processing entire codebases (via zipped uploads) and architecturally significant snippets, (2) manually validating outputs against a ground truth, and (3) comparing results with Archie's established benchmarks. Results reveal that ChatGPT achieves 57% accuracy for pattern detection outperforming Archie (44.4%) in systems like Spark and Gradle, where modern patterns (e.g., Pipeline, Master-Slave) are prevalent. However, it achieves a precision of 23% vs. Archie's 75% for tactic detection, highlighting limitations in recognizing traditional architectural strategies. Both tools face challenges with false positives, but ChatGPT demonstrates superior adaptability to newer paradigms. These results suggest that LLMs can augment—but not yet replace—traditional tools for architectural analysis. ChatGPT's strength in pattern extraction bridges gaps in automation, while its tactic detection limitations underscore the need for hybrid approaches. This study provides empirical insights into LLMs' role in software engineering, paving the way for more intelligent, collaborative architecture analysis tools.

Keywords: *Software Engineering, Elements Extraction, Pattern, Tactic, ChatGPT Assessment, ChatGPT Evaluation.*

1 Introduction

Software architecture is crucial in determining software systems' maintainability, scalability, and overall quality [1]. It encompasses architectural patterns [1], tactics [2],

Received 6 May 2025; Accepted 5 July 2025

and quality attributes [3], which collectively define the system's structure, design, and behaviour. Identifying these elements in existing codebases is critical for architectural reviews, quality assessments, system re-engineering, and refactoring tasks. Traditionally, this identification process relies heavily on expert knowledge and manual analysis, which can be time-consuming and error-prone, such as the traditional tool Archie [4-6].

Recent advances in Natural Language Processing (NLP) [7] and the emergence of large language models [8] like ChatGPT have opened new avenues for automating software analysis tasks. ChatGPT [9], with its deep understanding of human language and pre-trained knowledge of programming constructs, can assist in extracting and reasoning about architectural patterns and tactics directly from the source code. However, the effectiveness of such language models in performing these specific tasks remains relatively unexplored.

This paper aims to evaluate the effectiveness of ChatGPT in identifying architectural patterns and tactics from software systems' source code. Specifically, the study addresses two key research questions:

Q1:How effective is ChatGPT in extracting architectural patterns from software systems' source code?

Q2:How effective is ChatGPT in extracting architectural tactics from software systems' source code?

By answering these questions, this research seeks to provide insights into the capabilities and limitations of using ChatGPT for architectural analysis, potentially informing the design of more intelligent, automated tools for software engineering tasks. Our contributions to this work are:

- Conducting experiments to extract architectural patterns and tactics from the source code of five open-source systems using ChatGPT which are: Apache Storm [10], Apache Flink [11], Apache Spark [12], Gradle [13], and Maven [14], [15].
- Performing a comparative analysis of ChatGPT's performance against Archie, a traditional architectural analysis tool. The evaluation results for Archie, applied to the same five open-source systems, are detailed in our previous studies [16], [17], [18], [19].
- Measuring and evaluating ChatGPT's performance through precision, recall, and accuracy metrics.
- Addressing the defined research questions by analyzing the experimental results.

With the success of AI models like ChatGPT in programming assistance, there is growing interest in applying these models to more abstract areas of software development, including architecture. The potential to leverage AI for automating architecture-related tasks, like extracting design patterns and tactics, could streamline processes that are currently manual and time-consuming, enhancing productivity in the software architecture domain.

While research has shown that ChatGPT can assist with coding and other technical tasks, limited work has been specifically focused on evaluating its ability to handle more conceptual elements of software architecture, such as identifying patterns or suggesting tactics for quality attributes. This gap represents an opportunity to extend the capabilities of AI models into new and impactful areas, thereby contributing novel insights to the field.

While there are anecdotal reports of ChatGPT's utility in various software development contexts, systematic, empirical studies evaluating its effectiveness in extracting and applying architectural knowledge are lacking. This research can provide quantitative and qualitative evidence to support or challenge the current assumptions about the usefulness of AI in architecture decision-making, guiding future development of tools and models.

The remainder of this paper is organized as follows: Section 2 provides an overview of the background and related work. Section 3 details the research methodology employed. Our results are presented, analyzed, and discussed in section 4. Section 5 addresses potential threats to validity. Finally, section 6 concludes the paper and outlines directions for future research.

2 Related Work

2.1 Background

In this work, we address several concepts from various domains of software engineering. We provide an overview of the key background terminology and related research, focusing on concepts such as the architecture drivers and architectural elements. We also introduce the tools used in this study, including Large Language Models (LLMs) like ChatGPT and the traditional tool Archie. Additionally, we present the five software systems—Apache Storm, Apache Flink, Apache Spark, Gradle, and Maven—selected as case studies for validation. These concepts and tools are discussed both in general and in the context of this study's specific objectives. Architecture drivers involve quality attributes and functional requirements [3], context, and constraints [3]. This paper only considers quality attributes from the architecture drivers. Architecture elements include architectural patterns and architectural tactics. While architectural patterns express high-level design decisions, an architectural tactic is a design strategy that addresses a quality attribute [2].

2.1.1 Architectural Drivers

Architectural drivers are considerations that need to be made for the architecturally significant software system. They include requirements that influence the overall architecture [2-3]. They drive and guide the software architecture design.

2.1.1.1 Quality Attributes (QAs) Quality attributes (QAs) are characteristics that are required by the system. They are qualifications of the functional requirements or the overall product, such as performance, security, usability, and reliability [2][3]. These qualifications should be considered with the functions of the system. For example, an NFR performance might describe how quickly that dialogue should appear. An NFR availability also might express how often this function should fail, and so on.

2.1.1.2 Functional Requirements (FRs) they are functions of a system or components of a system that the system should do [1][19]. Functionality is achieved by assigning responsibilities to architectural elements, resulting in one of the most basic of architectural structures. Functional requirements are supported by non-functional requirements. Generally, functional requirements are expressed in the form of "system must do something," while non-functional requirements take the form of "system shall be a quality."

2.1.2 Architectural Elements

Architecture elements include architectural patterns and architectural tactics.

2.1.2.1 Architectural Patterns they are solutions that apply to specific problems and their contexts. Examples of architectural patterns include the Broker pattern [20, 21], the Layers pattern [20, 21], and the Pipes and Filters pattern [20][21]. Architectural patterns express high-level design decisions and describe high-level structures and behaviors [1, 2].

2.1.2.2 Architectural Tactics they are design decisions that address quality attributes (NFRs). They implement strategies to achieve these requirements [2]. Examples of tactics are "Heartbeat" [2], "Ping/Echo" [2], "Authentication" [2], and "Authorization" [2]. In general, a tactic implementation includes structure and behavior and can influence architectural patterns in several ways when implemented together. Tactics can be implemented in the same structure as architectural patterns or require changes to the structure and behavior of architectural patterns.

The relationship between patterns, tactics, and quality attributes is shown in Figure 1.

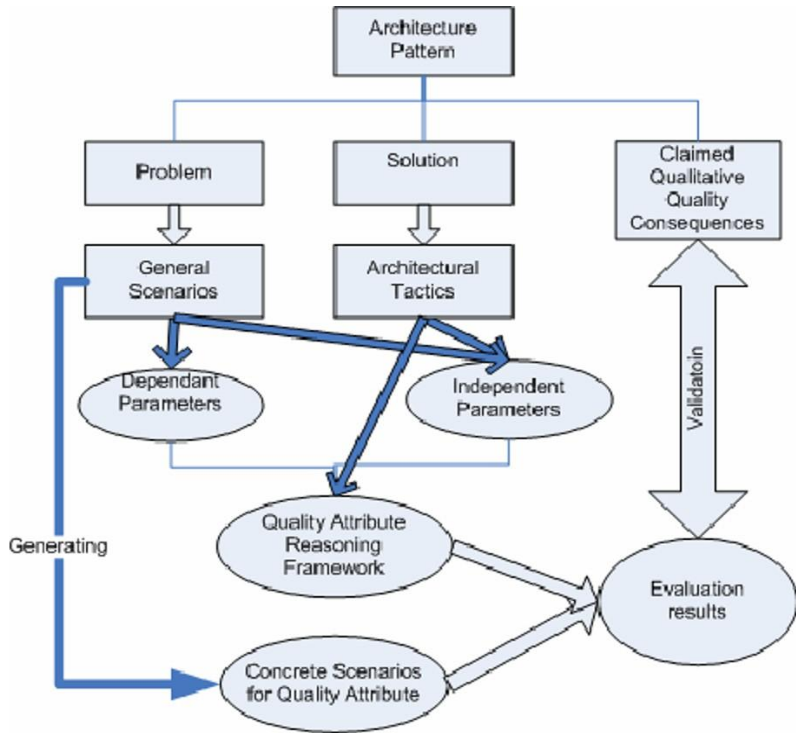


Figure.1: A skeleton of the relationship between patterns, tactics, and quality attributes

2.1.3 Large Language Models (LLMs)

Large Language Models (LLMs) [8] are a class of artificial intelligence models designed to understand, generate, and interact with human language. Built using deep neural networks, LLMs are typically trained on vast amounts of text data from diverse sources, enabling them to learn to indicate patterns, grammar, context, and even nuances within language. One of the most notable advancements in natural language processing, LLMs like GPT (Generative Pre-trained Transformer) and BERT (Bidirectional Encoder Representations from Transformers) utilize architectures based on transformers. This architecture, introduced by Vaswani et al. [22] in 2017, allows the models to process language in a parallelized manner, capturing both short- and long-term dependencies across words more efficiently than previous approaches like recurrent neural networks (RNNs). Through fine-tuning specific tasks or continued training on domain-specific data, LLMs can achieve impressive accuracy across applications such as machine translation, summarization, sentiment analysis, question-answering, and even code generation. Despite their power, LLMs also pose challenges, including high computational costs, potential biases, and difficulties in handling domain-specific or uncommon languages. Nonetheless, LLMs have fundamentally transformed how machines process human language, driving innovation across fields like customer support, software development, healthcare, and more.

2.1.3.1 ChatGPT-4

ChatGPT-4 [9], part of OpenAI's GPT-4 family, is an advanced Large Language Model (LLM) that builds upon the capabilities of its predecessors to offer improved language understanding, generation, and interaction. Released in 2023, GPT-4 is designed to handle complex prompts with greater accuracy and contextual awareness, making it effective for a wide array of applications, from creative writing to technical support and detailed analytical tasks. Figure 2 shows the architecture of ChatGPT. The architecture of ChatGPT-4, as depicted in Figure 2, consists of the following key components:

1. **Input Layer (blue color):** Processes user prompts (text or multimodal inputs).
2. **Transformer Layers (orange color):** Multiple layers of self-attention mechanisms for contextual understanding.
3. **Output Layer (green color):** Generates human-like responses based on learned patterns.
4. **Feedback Loop (purple color:** Optional fine-tuning via reinforcement learning from human feedback (RLHF).

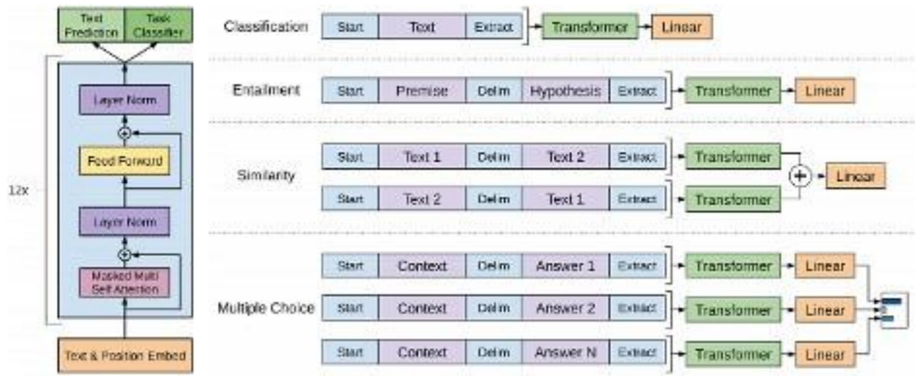
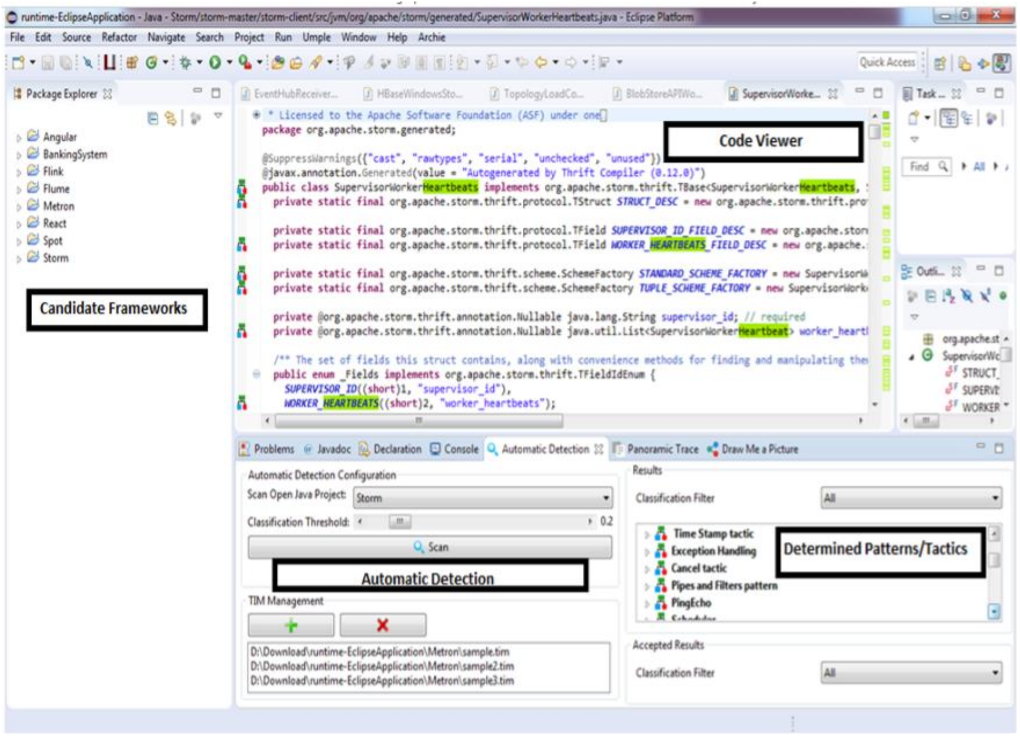


Figure 2: ChatGPT-4 Architecture, highlighting its transformer-based design and multimodal capabilities [9]

2.1.4 The Traditional Tool: Archie

Archie is an Eclipse plugin. It is used to determine architectural tactics, monitor related code, and notify developers when they modify architecturally significant parts of the code [4][5][6]. It is built to help automation the creation and maintenance of architecturally relevant trace links between code, architectural decisions, and related requirements. Figure 3 illustrates the Eclipse plugin interface of Archie, which includes:

1. **Code Editor Panel (gray color):** Displays source code with architecturally significant sections highlighted.
2. **Tactic Detection Panel (green color):** Lists detected tactics (e.g., "Heartbeat," "Ping/Echo") with confidence scores.
3. **Pattern Visualization (blue color):** Graphically represents patterns (e.g., "Layered Architecture") in the codebase.
4. **Notification System:** Alerts developers about architectural violations.



```
 * Licensed to the Apache Software Foundation (ASF) under one
package org.apache.storm.generated;

@SuppressWarnings({"cast", "rawtypes", "serial", "unchecked", "unused"})
@javax.annotation.Generated(value = "Autogenerated by Thrift Compiler (0.12.0)")
public class SupervisorWorkerHeartbeats implements org.apache.storm.thrift.TBase<SupervisorWorkerHeartbeats, 
  private static final org.apache.storm.thrift.protocol.TStruct STRUCT_DESC = new org.apache.storm.thrift.pro

  private static final org.apache.storm.thrift.protocol.TField SUPERVISOR_ID_FIELD_DESC = new org.apache.stor
  private static final org.apache.storm.thrift.protocol.TField WORKER_HEARTBEATS_FIELD_DESC = new org.apache.

  private static final org.apache.storm.thrift.scheme.SchemeFactory STANDARD_SCHEME_FACTORY = new Supervisor
  private static final org.apache.storm.thrift.scheme.SchemeFactory TUPLE_SCHEME_FACTORY = new SupervisorWork

  private @org.apache.storm.thrift.annotation.Nullable java.lang.String supervisor_id; // required
  private @org.apache.storm.thrift.annotation.Nullable java.util.List<SupervisorWorkerHeartbeat> worker_heart

  /** The set of fields this struct contains, along with convenience methods for finding and manipulating the
  public enum _Fields implements org.apache.storm.thrift.TFieldIdEnum {
    SUPERVISOR_ID((short)1, "supervisor_id"),
    WORKER_HEARTBEATS((short)2, "worker_heartbeats");
```

Automatic Detection Configuration
Scan Open Java Project: Storm
Classification Threshold: 0.2
Scan

TIM Management
D:\Download\runtime-EclipseApplication\Metron\sample.tim
D:\Download\runtime-EclipseApplication\Metron\sample2.tim
D:\Download\runtime-EclipseApplication\Metron\sample3.tim

Results
Classification Filter: All
Time Stamp tactic
Exception Handling
Cancel tactic
Pipes and Filters pattern
PingEcho

Accepted Results
Classification Filter: All

Figure 3: Archie tool: Eclipse plugin

2.1.5 Applied Software Systems

This section presents the five software systems used as case studies in this work: Apache Storm, Apache Flink, Apache Spark, Gradle, and Maven.

2.1.5.1 Apache Storm

Apache Storm [10] is a free, open source, distributed real-time computation system. It can be used with several programming languages. It consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation as needed. Figure 4 shows the architecture of Apache Storm. The interested reader may refer to [10] for more details.

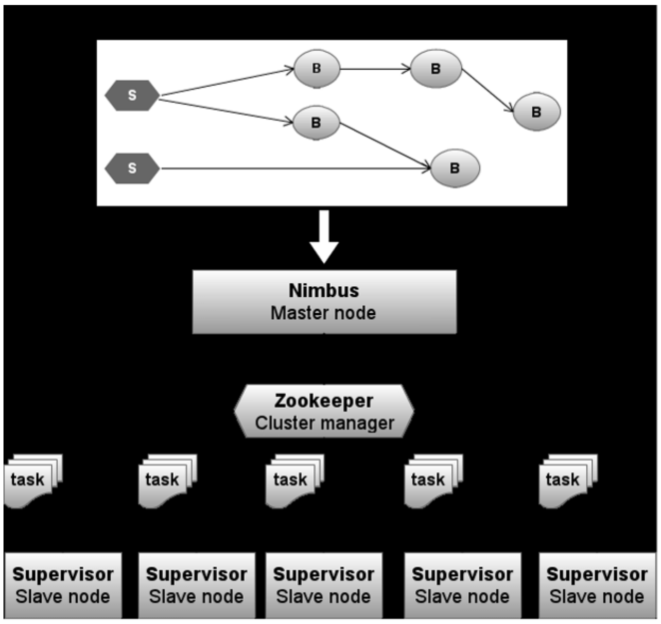


Figure 4: Apache Storm Architecture [10]

2.1.5.2 Apache Flink

Apache Flink [11] is an open-source framework for distributed stream processing. It uses machine learning for threat detection, investigation, and remediation applications. Flink is stateful, fault-tolerant, and performs on a large scale. It can run over thousands of nodes with excellent throughput and latency characteristics. Figure 5 shows the architecture of Apache Flink. The interested reader may refer to [11] for more details.

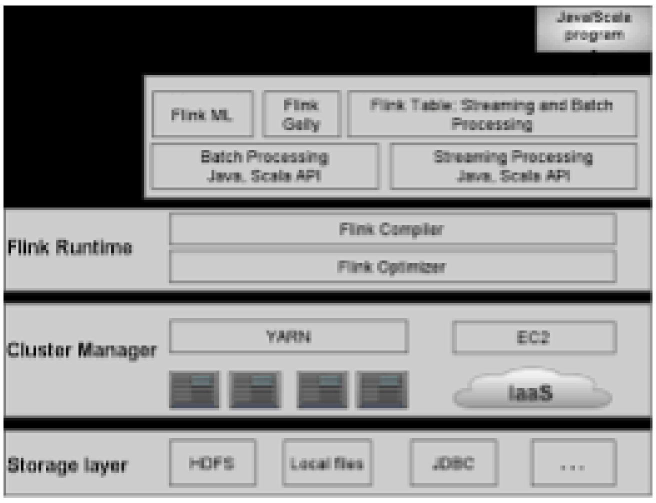


Figure 5: Apache Flink Architecture [11]

2.1.5.3 Apache Spark

Apache Spark [12] is an open-source framework for distributed and large-scale data processing. It provides an interface for entire programming clusters with implicit data parallelism and fault tolerance. It is a fast and general-purpose cluster computing system. Apache Spark provides distributed task dispatching, scheduling, and basic I/O functionalities. Figure 6 shows the architecture of Apache Spark. The interested reader can refer to [12] for more details.

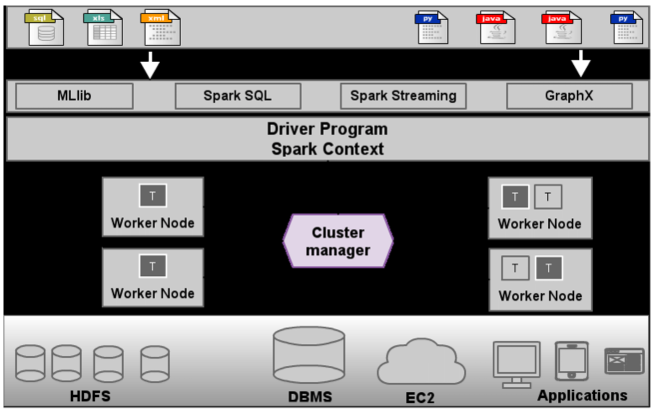


Figure 6: Apache Spark Architecture [12]

2.1.5.4 Gradle

Gradle [13] is an open-source build automation tool focused on flexibility and performance. Gradle is modelled in a customizable and extensible way in the most fundamental ways. It completes tasks quickly by reusing outputs from previous executions, processing only inputs that changed, and executing tasks in parallel. Gradle is the official build tool for Android and supports several languages and technologies. Figure 7 shows the architecture of Gradle [19].

2.1.5.5 Maven

Maven [14] is an open-source build automation tool. It is a software project management and comprehension tool. Maven is based on the concept of a project object model (POM) [15]. It can manage a project’s build, reporting and documentation from a central piece of information. Figure 8 shows the architecture of Maven [19].

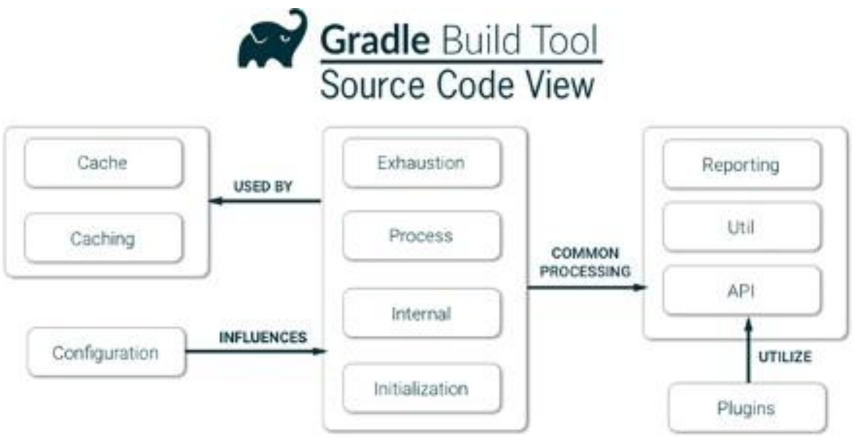


Figure 7: Gradle Architecture [13]

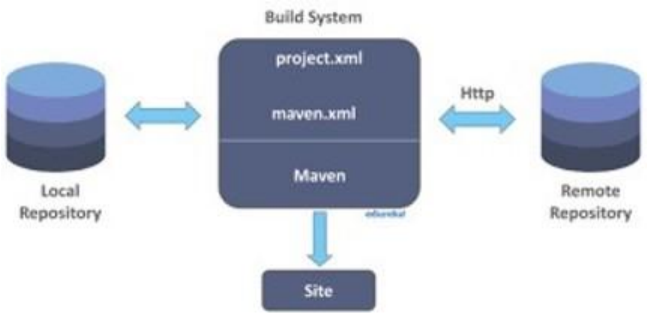


Figure 8: Maven Architecture [14]

2.2 Related Work

In a world increasingly reliant on technology, the exploration of generative AI's potential in creative and technical fields is both timely and transformative. Tan [23] investigates how ChatGPT can aid designers in identifying and extracting key design concepts from narrative stories. By leveraging the AI's natural language processing capabilities, designers can automate the extraction process, drawing inspiration from textual narratives. This method holds significant relevance in educational contexts, enhancing narrative-based learning in design education and showcasing the transformative power of generative AI in creative fields.

As the narrative unfolds, the focus shifts to the realm of software development. Gilson et al. [24] delve into the application of user stories within agile software development, aiming to extract essential quality attributes early in the architectural process. By employing natural language processing (NLP) techniques, they highlight the importance of recognizing non-functional requirements like performance and security, which are often overlooked in favour of functional ones. Their approach enables architects to make more informed decisions, thus avoiding costly adjustments later in development.

The story continues with Souvick et al. [25], who further emphasize the significance of NLP by automating the extraction of goals from unstructured natural language requirements. Their method employs predefined linguistic patterns to identify key goals and dependencies, streamlining the goal modeling process and ensuring an accurate representation of stakeholder objectives. This automated approach reduces the labour-intensive nature of manual goal modeling, allowing for a more efficient alignment of technical solutions with user needs.

In the healthcare sector, Huang et al. [26] highlight the efficiency of ChatGPT in extracting critical medical information. Their research shows the model's impressive accuracy across various datasets, indicating that well-structured prompts can significantly enhance its performance. By outperforming traditional NLP methods, ChatGPT offers the potential to revolutionize clinical data extraction, facilitating faster and more accurate decision-making in healthcare.

The authors in [27] examine how AI can assist software architects by fostering collaboration throughout the design process. Through a case study focused on a bike-sharing application, they demonstrate that while ChatGPT can support architectural tasks, human oversight remains vital for refining and validating outcomes. This collaboration between human expertise and AI tools emphasizes a future where large language models aid architects in navigating complex design challenges.

Pragyan et al. [28] focus on automating the extraction of use case components from user-authored scenarios related to mobile applications. They address the challenge of gathering requirements in early app development stages and propose a method using refined prompts to enhance extraction performance. By converting user scenarios into structured use case components, this approach aims to streamline requirements acquisition, helping developers refine app functionalities efficiently.

The exploration of ChatGPT's capabilities does not end there. Mohajer et al. [29] Investigate the effectiveness of ChatGPT in performing static analysis tasks, particularly focusing on detecting common bugs such as Null Dereference and Resource Leaks. Using Infer, a recognized static analysis tool, they compile a dataset from ten open-source projects, revealing that ChatGPT achieves notable precision—up to 68.37 percent for Null Dereference and 76.95 percent for Resource Leaks. Moreover, it surpasses Infer's existing performance, showcasing its ability to enhance static analysis processes and provide developers with improved accuracy in bug detection.

Terzi et al. [30] also investigate how developers interact with ChatGPT when using its code suggestions. The study analyzes a substantial dataset comprising over 267,000 lines of code, including commits, pull requests, and discussions. Key findings suggest that developers are more inclined to incorporate code provided by ChatGPT when they engage in multiple rounds of concise prompts that are specific to the problems at hand, rather than using longer, more detailed inputs. The research indicates that this targeted approach enhances the effectiveness of the interaction, as ChatGPT demonstrates the ability to efficiently address various types of programming issues across different languages.

Furthermore, Mahmoudi et al. [31] evaluate the potential of ChatGPT to automate systematic reviews (SRs), focusing on its capabilities in literature search, screening, data extraction, and content analysis. The authors developed a structured approach utilizing ChatGPT, divided into four modules: preparation (formulating Boolean search terms and collecting articles), screening (abstract screening and categorizing articles), filtering (full-text filtering and information extraction), and analysis (content analysis to identify trends and gaps in the literature). This structured methodology illustrates how generative AI can streamline complex processes in systematic reviews.

Sun et. al [32] use ChatGPT for extracting pharmacovigilance events, which involves identifying adverse events or potential therapeutic events from medical texts. The study evaluates the performance of ChatGPT using various prompt strategies and compares it to fine-tuned models.

Finally, KC et al. [33] use ChatGPT to extract use case (UC) components from user-authored scenarios, particularly in the context of mobile app development. The authors focus on refining the prompt strategies to enhance precision and recall in identifying UC components from these scenarios. Their approach involves constructing a dataset of 50 user-authored scenarios, which are manually labelled with UC components to serve as ground truth. They discovered that incorporating domain-specific knowledge into the prompts significantly improves the quality of the extracted UC components, indicating that large language models (LLMs) like ChatGPT benefit from more targeted prompts when handling such tasks. This work highlights the potential of leveraging LLMs in automating requirements acquisition processes, which is particularly valuable for small-scale development teams that might lack extensive resources for traditional requirements-gathering activities. The study demonstrates that although ChatGPT shows promise in extracting detailed UC components, further refinement in prompt engineering is crucial for optimizing its performance in real-world applications.

3. Methodology

3.1 Overview

Figure 9 provides an overview of the pipeline used in our work. Note that all the steps of Archie have been done previously in our studies [16], [17], [18], [19]. The process is structured into four main steps:

1. Input Source Code (yellow color):
    - Inputs: The pipeline begins with two types of input: the complete source code (zipped) and selected code snippets.
    - These inputs are used as prompts for ChatGPT, initiating the process of identifying architectural patterns and tactics. However, only the complete source code (zipped) is used as input for Archie.
2. Extract Patterns and Tactics (purple/blue for Archie/ChatGPT):
    - Archie:
        - The source code undergoes preprocessing, followed by training, and then detection proper. This step has been done previously in our studies [16], [17], [18], [19].
        - This results in three outputs: Extracted Patterns and Extracted Tactics.
    - ChatGPT:
        - The input is processed through pre-trained understanding, analysis, and recognition stages.
        - ChatGPT produces three outputs: Extracted Patterns and Extracted Tactics.
3. Compare Results (gray color):
    - The patterns and tactics identified by Archie and ChatGPT are compared to analyze their similarities, differences, and effectiveness in identifying these elements.
4. Calculate Accuracy (compass tool):
    - The final step involves calculating the accuracy of the outputs from both Archie, which has been done in previous studies [16], [17], [18], [19] and ChatGPT to evaluate their performance in extracting architectural patterns and tactics from the given source code.

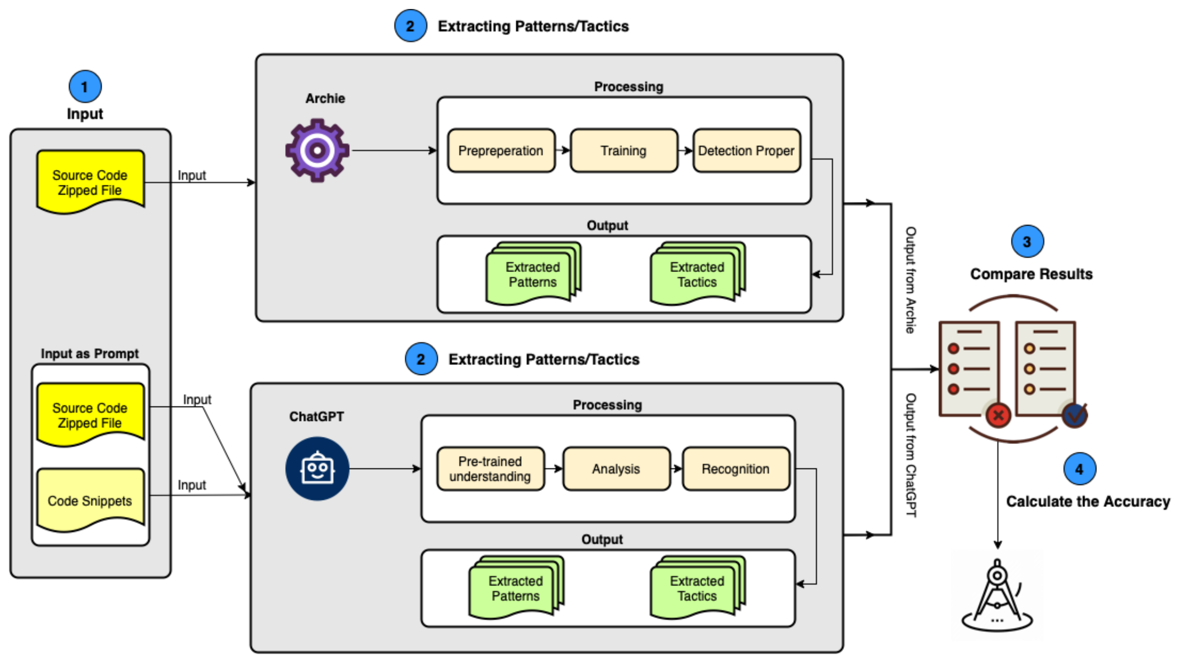


Figure 9: The overview of our work

3.2 Prompt Engineering Strategies

We used ChatGPT-4o (May 2024 release) via OpenAI's web interface (UI), as the API version did not support file uploads during our experiments. The following settings were applied to ensure consistency:

- ***Temperature***: 0.2 (to balance creativity and determinism in architectural analysis).
- ***Max Tokens***: 4096 (to accommodate long code snippets and detailed responses).
- ***Session***: Fresh chat sessions for each system to avoid context contamination.
- ***File Handling***: Source code files (zipped or snippets) were uploaded directly to the UI, with prompts explicitly requesting pattern/tactic extraction (Figures 10-12).

We applied the low temperature to reduce the risks, while max tokens allowed comprehensive responses. The UI was chosen over the API for its file-upload capability, critical for processing entire codebases.

In this work, we utilized specific prompts [34] to extract architectural patterns and tactics using ChatGPT. The first approach involved providing the entire source code file of each system, enabling ChatGPT to identify patterns and tactics from the entire codebase. In the second approach, we supplied selected code snippets and instructed ChatGPT to extract patterns and tactics based solely on these excerpts. Additionally, we tasked ChatGPT with extracting specific code snippets from the system's source code and subsequently determining the architectural patterns and tactics based on the extracted portions. Please refer to Figures 10, 11, and 12 in Section 3.5 for further clarification.

To handle the volume and complexity of the five open-source systems, we employed a hybrid input strategy:

1. **Full Codebase Upload (Zipped):**

Directly uploaded entire project directories (.zip) via ChatGPT's UI when possible (e.g., Gradle, Maven). The limitation of this strategy is that ChatGPT-4o's context window (128K tokens) restricted analysis depth for monolithic systems (e.g., Spark).

2. **Code Snippet Extraction:**

For systems exceeding practical context limits, we:

- Pre-filtered architecturally significant files (e.g., pom.xml for Maven, StormTopologyBuilder.java for Storm) based on both the file naming conventions (e.g., Manager.java, Strategy.java) and directory structure (e.g., /core/, /runtime/).
- Manually extracted critical snippets (≤5K tokens) reflecting patterns/tactics (e.g., heartbeat logic, pipeline interfaces).

3. **Prompt Chunking for Long Files:**

Files >10K tokens (e.g., FlinkStreamExecutionEnvironment.java) were splitted into logical segments (e.g., methods, configuration blocks), processed with iterative prompts (e.g., "Analyze this segment for the Pipes and Filters pattern"), and aggregated results post-hoc to avoid fragmentation.

4. **Validation of Input Reduction:**

We compared outputs from full uploads vs. snippets for consistency (e.g., confirmed Master-Slave detection in both approaches for Flink). The mitigation of this strategy is that Snippets were cross-checked with system documentation to ensure coverage of key architectural elements.

3.3 Evaluation Metrics

In this work, we evaluate the performance of ChatGPT using recall, precision, F1-score, and accuracy metrics to assess the effectiveness of our experiments. For this purpose, we calculate the True Positives (TP), False Positives (FP), and False Negatives (FN) required for these metrics. However, we do not compute the False Negatives (FN) as all patterns and tactics were evaluated and accounted for during detection. Specifically:

- **TP (True Positives):** The number of patterns and tactics correctly identified by ChatGPT or Archie.
- **FP (False Positives):** The number of patterns and tactics incorrectly identified by ChatGPT or Archie.
- **FN (False Negatives):** The number of patterns and tactics missed by ChatGPT or Archie.
- **Precision** = True Positives (TP) / (True Positives (TP) + False Positives (FP)).
- **Recall** = True Positives (TP) / (True Positives (TP) + False Negatives (FN)).
- **F1-score** = 2 * (Precision * Recall) / (Precision + Recall).
- **Accuracy** = (True Positives (TP) + True Negatives (TN)) / (TP + FP + FN + TN).

3.4 Manual Validation

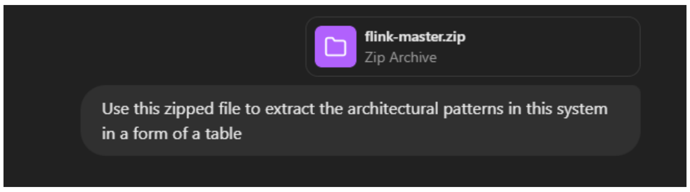
To validate the True Positives (TP), False Positives (FP), and False Negatives (FN) for both ChatGPT and Archie, we employed a manual, expert-driven process:

- Ground Truth Creation: For each of the five systems (Apache Storm, Flink, Spark, Gradle, Maven), we collected a gold-standard set of architectural patterns and tactics by:
  - Reviewing official documentation (e.g., Apache Flink's architectural guide [10]-[12]).
  - Consulting prior studies on these systems ([16]-[19]) and domain experts.
- Tool Output Comparison:
  - For ChatGPT, we extracted patterns/tactics from its responses to prompts (Figures 10-12) and mapped them to the ground truth.
  - For Archie, we reused its detections from prior work ([16]-[19]) and verified them against the same ground truth.
- Labeling Rules:
  - TP: A pattern/tactic detected by the tool and confirmed in the ground truth.
  - FP: A pattern/tactic detected by the tool but absent in the ground truth.
  - FN: A pattern/tactic missed by the tool but present in the ground truth.

3.5 Apache Flink

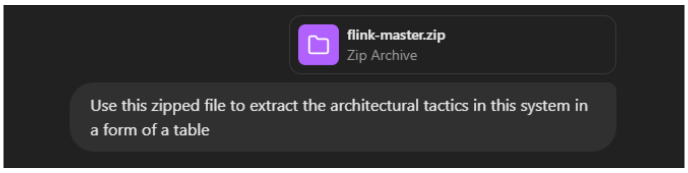
In this section, we use Apache Flink as an example to illustrate the methodology's steps outlined in Section 3.1.

Step 1: First, we present the prompts used to extract architectural patterns and tactics from the Apache Flink source code. Figures 10 and 11 demonstrate the prompts applied to the entire Flink source code for identifying its architectural patterns and tactics, respectively. Figure 12 showcases the prompts used with specific code snippets to extract patterns and tactics from Flink.


flink-master.zip
Zip Archive

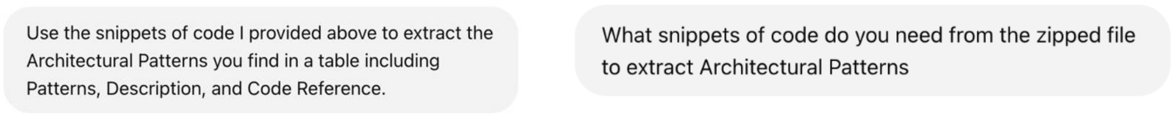
Use this zipped file to extract the architectural patterns in this system in a form of a table

Figure 10: Prompt utilizing the entire source code to identify architectural patterns


flink-master.zip
Zip Archive

Use this zipped file to extract the architectural tactics in this system in a form of a table

Figure 11: Prompt utilizing the entire source code to identify architectural tactics


Use the snippets of code I provided above to extract the Architectural Patterns you find in a table including Patterns, Description, and Code Reference.

What snippets of code do you need from the zipped file to extract Architectural Patterns

(a) (b)

Figure 12: Prompt using specific snippets of code (a) after providing snippets of code and (b) before providing sniptes of code

Steps 2 and 3: After extracting the patterns and tactics, we compared the results of ChatGPT with those from the traditional tool Archie for Apache Flink. For pattern detection, Table 1 summarizes the comparison of pattern detection outcomes between Archie and ChatGPT in Apache Flink, while Figure 13 illustrates the percentage of patterns detected by each tool for Apache Flink. As observed, ChatGPT achieved a pattern detection rate of 55.6%, compared to Archie's rate of 44.4%.

Pattern Detection:

Table 1: Comparison results of the patterns detection for Apache Flink

Architectural Pattern	**Also Known As**	**Archie**	**ChatGPT**
Pipeline Pattern	Streaming Pipeline	✗	✓
Master-Slave Pattern	—	✗	✓
Layered Architecture	Multitier Architecture	✗	✓
Event-Driven Architecture	Message-Driven Architecture	✗	✓
Service Component Pattern	—	✗	✓
Layers	Tiered System	✗	✓
Broker	Message Broke	✗	✓
Observer/Publish-Subscribe	—	✓	✗
Pipes and Filters	—	✓	✗
Shared- Repository	Common Data Repository	✓	✗
		✗	✗

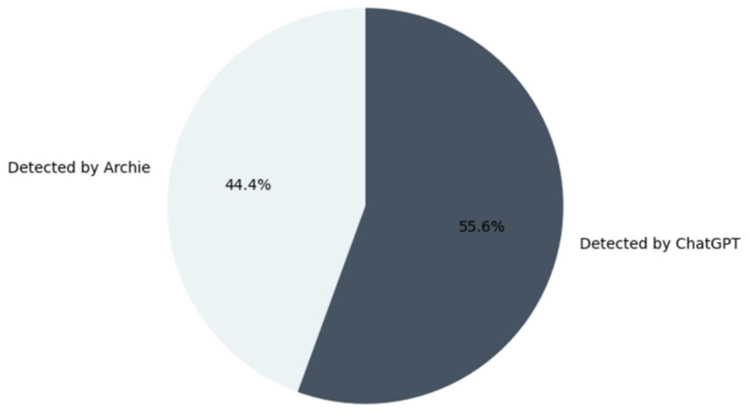


Figure 13: The patterns detection percentages for ChatGPT and Archie for Apache Flink

Tactic Detection: Table 2 summarizes the comparison of tactic detection outcomes between Archie and ChatGPT in Apache Flink, while Figure 14 illustrates the percentage of tactics detected by each tool. ChatGPT detected 25.0% of the tactics, whereas Archie achieved a higher detection rate of 75.0%. This is expected, as Archie was specifically designed to identify traditional tactics, whereas ChatGPT is more capable of discovering modern tactics.

Table 2: The comparison results of the tactics detection for Apache Flink

Architectural Tactic	**Also Known As**	**Archie**	**ChatGPT**
Kerberos	—	✓	✓
	—		
Heartbeat		✓	✓
Ping/Echo	Connectivity	✓	✗
Exception Handling	Probe		
	Error Handling/	✓	✗
	Fault Handling		
	—		
Authenticate	—	✓	✗
Time Stamp	Resource	✓	✗
Resource Pooling	Sharing	✓	✗
	—		
Audit Trail	Policy-Based		
PBAC	Access Control	✓	✗
	Role-Based	✓	✗
	Access Control	✓	✗
RBAC	Task Scheduling		
		✓	✗
	Load		
Resource Scheduling	Management	✓	✗
	—	✓	✗
Session Management	System Reboot		
Load Balancing	—		
Restart	—		✗
	Data Duplication	✓	✗
Time-out	—	✓	
Cancel	—	✓	✗
Active Redundancy	—		✗
Checkpoint			
Retry		✓	
Retry Logic	—	✓	✗

for Fault Tolerance		✓	✗
Data Partitioning for	Resource	✓	✗
Scalability	Sharing	✗	✓
Resource Pooling	—	✗	
for Efficient			✓
Resource Use		✗	
Circuit Breaker for			✓
Fault Isolation		✗	
			✓
		✗	

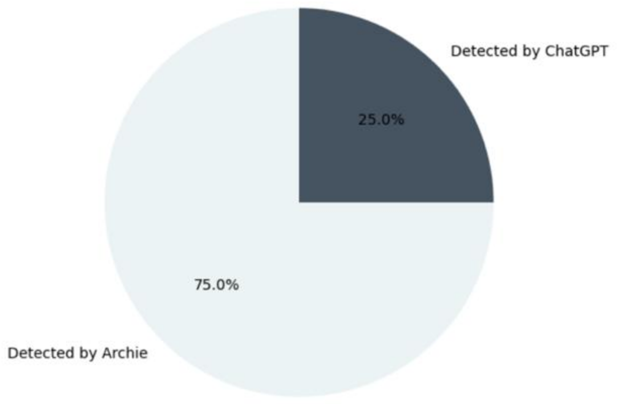


Figure 14: The tactics detection percentages for ChatGPT and Archie for Apache Flink

Step 4: we calculate the accuracy of ChatGPT using recall, precision, and accuracy metrics to assess the effectiveness of our experiments. For this purpose, we calculate the True Positives (TP), False Positives (FP), and False Negatives (FN) required for these metrics. However, we do not compute the False Negatives (FN) as all patterns and tactics were evaluated and accounted for during detection. Tables 3[*] and 4[*] present the results for True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) of pattern detection for both Archie and ChatGPT. ChatGPT identified five patterns, demonstrating its strength in detecting modern architectural patterns, such as fault-tolerance strategies. In contrast, Archie identified four patterns, showcasing its proficiency in recognizing traditional architectural elements.

ChatGPT missed 5 patterns (FN), whereas Archie missed 6 patterns (FP), highlighting its challenges in detecting modern or nuanced techniques. ChatGPT also falsely identified four patterns (FP), whereas Archie falsely identified five patterns. Since all patterns were evaluated, true negatives (TN) are not applicable in this context.

[*] TP/FP/FN counts were manually validated against the ground truth (see Section 3.4). TN was omitted as all patterns/tactics were evaluated for presence/absence.

Table 3: Metrics results of Archie for Flink Table 4: Metrics results of ChatGPT for Flink

Archie Detection Metrics	Architectural Patterns
True Positives (TP)	4
True Negatives (TN)	0
False Positives (FP)	5
False Negatives (FN)	6

ChatGPT Detection Metrics	Architectural Patterns
True Positives (TP)	5
True Negatives (TN)	0
False Positives (FP)	4
False Negatives (FN)	5

Tables * and 6* present the results for True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) of tactic detection for both Archie and ChatGPT. ChatGPT identified six tactics, demonstrating its strength in detecting modern architectural tactics. In contrast, Archie identified 18 tactics, showcasing its proficiency in recognizing traditional architectural elements.

ChatGPT missed 17 tactics (FN), underscoring its limitations in identifying foundational or traditional tactics. On the other hand, Archie missed five tactics, highlighting its challenges in detecting modern or nuanced techniques. ChatGPT also falsely identified 16 tactics (FP), whereas Archie falsely identified four tactics. Since all tactics were evaluated, TN are not applicable in this context.

Table 7 presents the precision, recall, and accuracy results for both pattern and tactic detection using Archie and ChatGPT in Apache Flink. As observed, Archie achieves higher accuracy than ChatGPT in Flink, which is expected since Archie is specifically designed to detect traditional tactics, whereas ChatGPT excels in identifying modern tactics. Additionally, the results indicate that ChatGPT demonstrates greater accuracy in detecting architectural patterns than tactics, as shown in Figure 13. We followed the same approach for the remaining systems. The results for these systems are available online. For more details, please refer to the data availability section below at the end of paper.

Table 5: Metrics results of Archie

Archie Detection Metrics	Architectural Tactics
True Positives (TP)	18
True Negatives (TN)	0
False Positives (FP)	4
False Negatives (FN)	5

Table 6: Metrics results of ChatGPT

ChatGPT Detection Metrics	Architectural Tactics
True Positives (TP)	6
True Negatives (TN)	0
False Positives (FP)	16
False Negatives (FN)	17

Table 7: Metrics of pattern and tactic detection of Flink

Metrics	Archie	ChatGPT
Precision	0.71%	0.35%
Recall	0.67%	0.33%
F1-score	0.69%	0.34%
Accuracy	52.38%	20.75%

4 Results, Analysis and Discussion

In this section we explained the results and the comprehensive discussion of our method. Results are shown in Section 4.1 and the discussion is shown in Section 4.2.

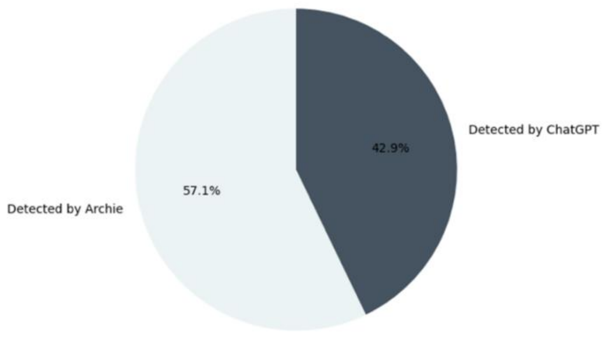
4.1 Results

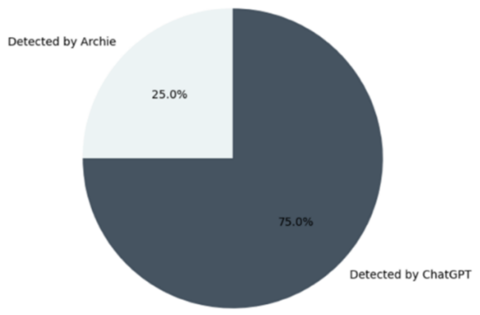
RQ1: How effective is ChatGPT in extracting architectural patterns from software systems' source code?

Approach: To address this question, we utilized ChatGPT to identify patterns from the complete codebase for all five systems. We began with two types of input: the complete source code (zipped) and selected code snippets. These inputs are used as prompts for ChatGPT, initiating the process of identifying architectural patterns. However, only the complete source code (zipped) is used as input for Archie. Once the results were obtained, they were organized into tables for each system, and the percentage of patterns detected by each tool was calculated.

Comparison: We compared the results of ChatGPT with those from the traditional tool Archie across all five systems. The comparison results of pattern detection for Apache Storm, Spark, Gradle, and Maven are available online, see Section 7. While the percentage of patterns detected by each tool across all five systems is shown in Figures 15, 16, 17, and 18.

Results: As observed in Figures 15, 16, 17, and 18, ChatGPT achieved a pattern detection rate of 42.9%, compared to Archie's rate of 57.1% in Apache Storm. In contrast, ChatGPT outperformed Archie in Spark, detecting patterns at a rate of 75.0% compared to Archie's 25.0%. Similarly, ChatGPT achieved a pattern detection rate of 72.7% in Gradle, surpassing Archie's 27.3%. ChatGPT also led with a 60.0% detection rate in Maven, while Archie achieved 40.0%. The F1-scores for these comparisons are provided in Tables 8-12, demonstrating the balance between precision and recall for

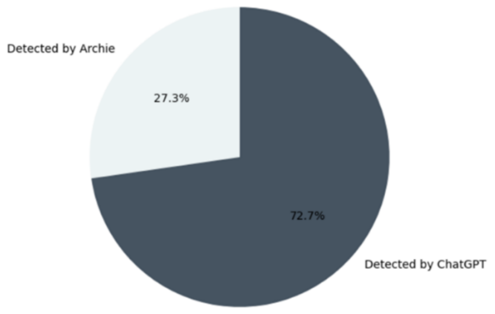


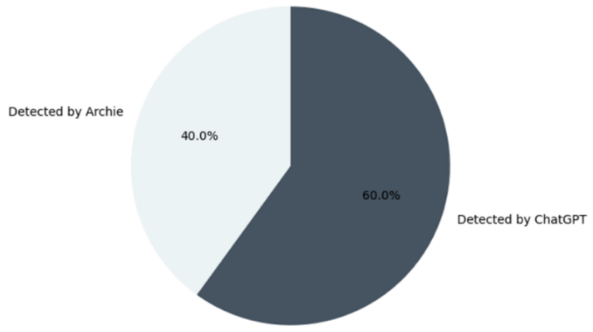


each tool.

Fig. 15. The pattern detection percentages for ChatGPT and Archie for Apache Storm

Fig. 16. The pattern detection percentages





for ChatGPT and Archie for Apache Spark

Fig. 17. The pattern detection percentages for ChatGPT and Archie for Gradle

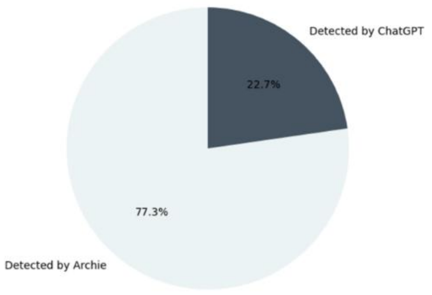
Fig. 18. The pattern detection percentages for ChatGPT and Archie for Maven

RQ2: How effective is ChatGPT in extracting architectural tactics from software systems' source code?

Approach: To address this question, we utilized ChatGPT to identify tactics from the complete codebase. We began with two types of input: the complete source code (zipped) and selected code snippets. These inputs are used as prompts for ChatGPT, initiating the process of identifying architectural tactics. However, only the complete source code (zipped) is used as input for Archie.

Comparison: We compared ChatGPT's results with those from the traditional tool Archie across all five systems. The tactics detection results for Apache Storm, Spark, Gradle, and Maven are available online; see Section 7. The percentage of tactics detected by each tool across all five systems is shown in Figures 19, 20, 21, and 22.

Results: As illustrated in Figures 19, 20, 21, and 22, ChatGPT achieved a tactic detection rate of 22.7% in Apache Storm, while Archie outperformed it with a rate of 77.3%. In Spark, both ChatGPT and Archie detected tactics at an equal rate of 50.0%. Likewise, ChatGPT identified tactics at a rate of 33.3% in Gradle, whereas Archie achieved 66.7%. However, in Maven, ChatGPT led with a detection rate of 66.7%, surpassing Archie's 33.3%. The F1-scores for these comparisons are provided in Tables 8-12, demonstrating the balance between precision and recall for each tool.



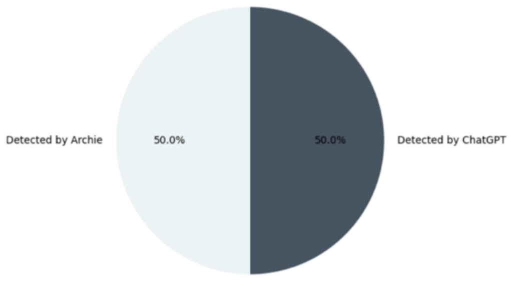
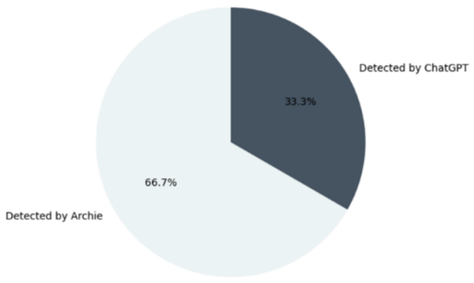


Fig. 19. The tactic detection percentages for
percentages ChatGPT and Archie for Apache Spark.
for Apache Storm

Fig. 20. The tactic detection
for ChatGPT and Archie



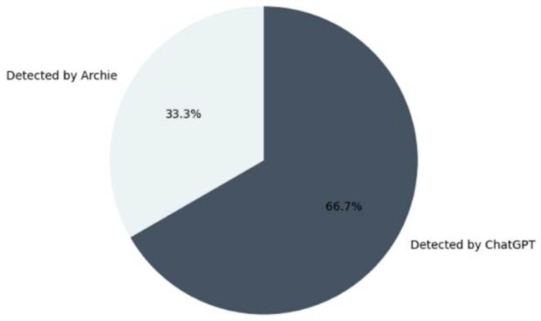


Fig. 21. The tactic detection percentages for
for
ChatGPT and Archie for Gradle

Fig. 22. The tactic detection percentages
ChatGPT and Archie for Maven

4.1.1 F1-score,

To illustrate ChatGPT’s challenges, we highlight representative failures from our experiments:

1. False Positive:
    - System: Apache Storm
    - ChatGPT Output: Detected "Microservices Architecture" in Storm’s topology.
    - Ground Truth: Storm uses the pipeline pattern (confirmed via documentation [10]).
    - Root Cause: ChatGPT merged distributed processing with microservices due to superficial similarities in decentralization.
2. False Negative (Missed Tactic):
    - System: Apache Flink
    - Missed Tactic: "Checkpointing" for fault tolerance (a well-documented Flink feature [11]).
    - ChatGPT Output: Ignored checkpointing code snippets unless explicitly prompted with the tactic name.
    - Root Cause: Over-reliance on explicit keyword matching in prompts.
3. Misclassified Pattern:
    - System: Gradle
    - ChatGPT Output: Labeled the build script DSL as "Interpreter Pattern."
    - Ground Truth: Gradle uses a declarative pipeline model, not an interpreter structure.
    - Root Cause: Misinterpretation of domain-specific language (DSL) flexibility as a design pattern.

These examples underscore ChatGPT’s limitations in contextual reasoning and domain-specific knowledge, suggesting opportunities for fine-tuning or hybrid approaches.

4.2 Discussion

Tables 8, 9, 10, and 11 present the pattern and tactic detection metrics across Storm, Spark, Gradle, and Maven. Table 12 summarizes the overall detection metrics for both Archie and ChatGPT across all systems. As detailed in Section 3.4, all metrics (precision, recall, F1-score, accuracy) were derived from manual validation against a ground truth. The results indicate that Archie performs better in tactic detection compared to pattern detection, with higher precision, recall, F1-score, and accuracy in most cases. ChatGPT generally performs better in pattern detection, especially in Gradle and Maven, with higher precision and recall. Both tools struggle with false positives and false negatives, particularly in tactic detection, where the number of missed tactics (FN) is high. ChatGPT has a higher precision in pattern detection, indicating fewer false positives, but its recall, F1-score, and accuracy vary significantly across different frameworks.

Table 8: Metrics of pattern and tactic detection of Storm

Metrics	Archie	ChatGPT
Precision	0.75%	0.29%
Recall	0.67%	0.26%
F1-score	0.71%	0.27%
Accuracy	55.0%	16.0%

Table 9: Metrics of pattern and tactic detection of Spark

Metrics	Archie	ChatGPT
Precision	0.45%	0.58%
Recall	0.45%	0.57%
F1-score	0.45%	0.58%
Accuracy	30.0%	40.0%

Table 10: Metrics of pattern and tactic detection of Gradle

Metrics	Archie	ChatGPT
Precision	0.57%	0.48%
Recall	0.50%	0.42%
F1-score	0.53%	0.45%
Accuracy	36.0%	29.0%

Table 11: Metrics of pattern and tactic detection of Maven

Metrics	Archie	ChatGPT
Precision	0.36%	0.67%
Recall	0.28%	0.52%
F1-score	0.31%	0.59%
Accuracy	19.0%	41.0%

Table 12: The final Metrics of pattern and tactic detection for both Archie and ChatGPT

Framework	Detection Type	Precision	Recall	F1-score	Accuracy
ChatGPT	Pattern detection	0.50-0.80	0.43-0.69	0.46-0.74	30%-57%
ChatGPT	Tactic detection	0.23-0.67	0.21-0.53	0.22-0.59	12%-41%
Archie	Pattern detection	0.25-0.67	0.23-0.57	0.24-0.62	14%-44%
Archie	Tactic detection	0.33-0.77	0.26-0.71	0.29-0.74	17%-59%

5. Threats to Validity

This section outlines the potential threats to the validity of the findings in this study. While the research explores the capabilities of ChatGPT in architectural analysis, certain limitations could impact the robustness, reliability, and generalizability of the results. These threats are categorized into three main types: construct validity, focusing on the design and measurement of the study; internal validity, addressing factors that could influence the interpretation of results; and external validity, concerning the applicability of findings to broader contexts. Each category highlights specific challenges and areas for improvement, ensuring a balanced evaluation of the study's strengths and limitations.

Construct Validity: One of the threats of this work is that the effectiveness of ChatGPT heavily depends on prompt quality, and variations in prompt design might influence results. The lack of detailed discussion about prompt optimization could affect reproducibility. We mitigate this by Using a standardized prompt evaluation framework to ensure consistency and reproducibility. The also paper might not cover all possible architectural patterns, tactics, and QAs comprehensively, potentially leading to biased results. We mitigate this threat by consulting domain experts to ensure the selected patterns, tactics, and QAs represent a comprehensive and balanced subset of architectural elements. Another limitation is that our reliance on ChatGPT's UI (not API) may limit automation potential, though settings like low temperature (0.2) mitigated variability. Future work should explore API-based batch processing**.**

Internal Validity: One of the internal threats of this work is that the choice of five open-source systems might not generalize to other types of software systems, limiting the scope of the findings. We mitigate this threat by selecting one project from a different domain, so we cover most of the software engineering domains. The method for manually verifying TP/FP/FN (detailed in Section 3.4) may introduce subjectivity, as ground truth labeling relies on expert interpretation of documentation and prior studies. We mitigated this by:

- Using multiple authoritative sources (e.g., system documentation, peer-reviewed studies) to establish ground truth.
- Cross-validating Archie's outputs with our prior work ([16]-19]).

External Validity: One of the generalizability threats is that the findings are based on specific open-source projects, and the results may not be applicable to proprietary or less-structured codebases. We have future work to expand the study to include proprietary systems and unstructured codebases to assess generalizability. Another threat is that since the study evaluates ChatGPT (a specific LLM), the results may not generalize to other LLMs or AI-based tools for architectural analysis. We have another task in the future to evaluate the performance of other LLMs and AI-based tools to provide a broader perspective.

6. Conclusion

This work explored the effectiveness of ChatGPT, a modern large language model, in identifying architectural patterns and tactics, within software systems, comparing its performance to that of the traditional tool Archie. The findings highlight the unique strengths and limitations of both tools. While ChatGPT demonstrated promise in detecting modern patterns (e.g., achieving **57% accuracy** in pattern detection), its performance lagged in tactic identification (**23% precision**), underscoring key limitations:

1. **Input Constraints**:
   - ChatGPT’s token limit necessitated fragmented code analysis, risking incomplete context.
   - Manual snippet selection introduced potential selection bias in large systems (e.g., Spark).
2. **Specialization Gap**:
   - Archie’s rule-based approach excelled in tactic detection (75% recall) but struggled with newer patterns.
   - ChatGPT’s generative nature led to false positives (e.g., misclassifying generic code as patterns).
3. **Validation Subjectivity**:
   - Ground truth reliance on documentation may omit undocumented tactics.

Future work could focus on enhancing ChatGPT's training data to improve its recognition of traditional tactics, as well as integrating the capabilities of both tools to create a hybrid solution. Such advancements could pave the way for more intelligent, automated systems that bridge the gap between natural language understanding and software engineering tasks, ultimately contributing to improved software maintainability and scalability.

References

[1] Carey, J. & Carlson, B. (2002). Framework Process Patterns: Lessons Learned Developing Application Frameworks, Addison-Wesley, (2002).
[2] Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice, Addison-Wesley, (2012).
[3] Johnson, R.E. (1997). How frameworks compare to other object-oriented reuse techniques, Communications of the ACM, 40(10), 39-42, (1997).
[4] Mirakhorli, M. (2014). Preserving the Quality of Architectural Tactics in Source Code, The Institutional Repository at DePaul University, College of Computing and Digital Media, (2014).

[5] Mirakhorli, M., and Cleland-Huang, J. (2016). Detecting, Tracing, and Monitoring Architectural Tactics in Code, IEEE Transactions on Software Engineering, Volume: 42, Issue 3, pp 205-220, (2016).
[6] Mirakhorli, M., Fakhry, A., Grecho, A., Wieloch, M., & Cleland-Huang, J. (2014). Archie: A Tool for Detecting, Monitoring, and Preserving Architecturally Significant Code, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 739-742, Hong Kong, China, (2014).
[7] https://www.datacamp.com/blog/what-is-natural-language-processing.
[8] https://www.ibm.com/topics/large-language-models.
[9] https://openai.com/index/hello-gpt-4o/
[10] https://storm.apache.org, (last accessed 2024/10/9).
[11] Apache Flink, flink.apache.org, (last accessed 2024/10/9).
[12] https://spark.apache.org, (last accessed 2024/10/9).
[13] https://docs.gradle.org/current/ userguide/userguide.html, (last accessed 2024/10/9).
[14] https://maven.apache.org/what-is-maven.html, (last accessed 2024/10/9).
[15] https://maven.apache.org/guides/introduction/ introduction-to-the-pom.html, (last accessed 2024/10/9).
[16] Milhem, H., Weiss, M., & Somé, S. (2020). Modeling and Selecting Frameworks in terms of Patterns, Tactics, and System Qualities. In: 32nd Proc of International Conference on Software Engineering and Knowledge Engineering (SEKE 2020), USA, July 2020. (Best Paper Award).
[17] Milhem, H., Weiss, M., & Somé, S. (2019). Extraction of Architectural Patterns from Frameworks and Modeling their Contributions to Qualities, In: 26th Proc. of Conf. on Patterns Language of Programs (PLoP), Ottawa, Canada, October 2019. http://www.hillside.net/plop/2019/index.php?nav=program.
[18] Milhem, H., Weiss, M., & Somé, S. (2020). Modeling and Selecting Frameworks in terms of Patterns, Tactics, and System Qualities, International Journal of Software Engineering and Knowledge Engineering (ijSEKE), 2020.
[19] Milhem, H. (2020). https://ruor.uottawa.ca/bitstream/10393/40831/1/Bani_Milhem_Hind_Ahmad_Ismail_2020_thesis.pdf, thesis, University of Ottawa, 2020.
[20] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern- Oriented Software Architecture: A System of Patterns, vol. 1, John Wiley and Sons, (1996).
[21] Buschmann, F., Henney, K., and Schmidt, D.C. (2007). Pattern-Oriented Software Architecture, vol. 4, Wiley, (2007).
[22] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L- ., & Polosukhin, I. (2017). Attention is all you need. In Advances in Neural Information Processing Systems (NeurIPS), 30.
[23] Tan, L. (2023) Using ChatGPT to extract design concepts from stories, in Derek Jones, Naz Borekci, Violeta Clemente, James Corazzo, Nicole Lotz, Liv Merete Nielsen, Lesley-Ann Noel (eds.), The 7th International Conference for Design Education Researchers, 29 November - 1 December 2023, London, United Kingdom[https://doi.org/10.21606/drslxd.2024.05].
[24] F. Gilson, M. Galster & F. Georis. (2019). Extracting Quality Attributes from User Stories for Early Architecture Decision Making, 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 2019, pp. 129-136, [doi: 10.1109/ICSA-C.2019.00031].

[25] Das, Souvick and Deb, Novarun and Cortesi, Agostino & Chaki, Nabendu. (2024). Extracting goal models from natural language requirement specifications. Journal of Systems and Software. [211.111981. 10.1016/j.jss.2024.111981].

[26] J. Huang, D. M. Yang, R. Rong, K. Nezafati, C. Treager, Z. Chi, S. Wang, X. Cheng, Y. Guo, L. J. Klesse, G. Xiao, E. D. Peterson, X. Zhan, & Y. Xie (2024). A critical assessment of using ChatGPT for extracting structured data from clinical notes. npj Digital Medicine, 7(1), 1-13. [https://doi.org/10.1038/s41746-024-01079-8].

[27] Ahmad, A., Waseem, M., Liang, P., Fehmideh, M., Aktar, M. S., & Mikkonen, T. (2023). Towards Human-Bot Collaborative Software Architecting with ChatGPT. arXiv. [https://doi.org/https://arxiv.org/abs/2302.14600v1].

[28] KC, Pragyan and Slavin, Rocky and Ghanavati, Sepideh and Breaux, Travis & Bokaei Hosseini, Mitra. (2024). An Analysis of Automated Use Case Component Extraction from Scenarios using ChatGPT. [10.48550/arXiv.2408.03395].

[29] Mohajer, Mohammad and Aleithan, Reem and Shiri Harzevili, Nima and Wei, Moshi and BoayeBelle, Alvine and Pham, Hung & Wang, Song. (2024). Effectiveness of ChatGPT for Static Analysis: How Far Are We?. [151-160. 10.1145/3664646.3664777].

[30] Terzi Anastasia, Bibi Stamatia, Tsitsimiklis Nikolaos, & amp; Angelidis Pantelis. (2024). Using code from ChatGPT: Finding patterns in the developers' interaction with ChatGPT [Data set]. https://b2share.eudat.eu.

[31] Mahmoudi, Hesam and Chang, Doris and Lee, Hannah and Ghaffarzadegan, Navid & Jalali, Mohammad. (2024). A Critical Assessment of Large Language Models for Systematic Reviews: Utilizing ChatGPT for Complex Data Extraction. SSRN Electronic Journal. 10.2139/ssrn.4797024.

[32] Z. Sun, G. Pergola, B. C. Wallace, and Y. He (2024). Leveraging ChatGPT in Pharmacovigilance Event Extraction: An Empirical Study. arXiv. [https://doi.org/https://arxiv.org/abs/2402.15663v1]

[33] KC, P., Slavin, R., Ghanavati, S., Breaux, T., & Hosseini, M. B. (2024). An Analysis of Automated Use Case Component Extraction from Scenarios using ChatGPT. arXiv. https://doi.org/https://arxiv.org/abs/2408.03395v1.

[34] White, Jules and Fu, Quchen and Hays, Sam and Sandborn, Michael and Olea, Carlos and Gilbert, Henry and Elnashar, Ashraf and Spencer-Smith, Jesse & Schmidt, Douglas. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. [10.48550/arXiv.2302.11382

Notes on contributors

Hind Milhem holds a Doctor of Software Engineering degree from Ottawa University, Canada in 2020.She also received her B.Sc. and M.Sc. (Computer Science) from Yarmouk University, Jordan in 2007 and 2009, respectively. She is currently an assistant professor in the Information Department in IT faculty at The Hashemite University, Zarqa, Jordan. Her research includes software architecture design, architectural patterns, architectural tactics, software evaluation, and artificial intelligence. She published six papers in international journals and conferences. She also has five papers under processing and is waiting for the final decision on their acceptance. From 2019 till now, she has been a reviewer (PC member) at the International Journal of Software Engineering and Knowledge Engineering (IJSEKE) and SEKE conference. She can contacted at email: hinda_is@hu.edu.jo.

Neil Harrison is a professor and chair of the Department in Computer Science Utah Valley University. He is the author of numerous articles on software patterns, software architecture, and agile software, and is the co-author of the seminal book, "Organizational Patterns of Agile Software Development." He is the namesake of the "Neil Harrison Shepherding Award", given at pattern conferences for outstanding shepherding. He is a Hillside Group member and has served on its board of directors. He was formerly a Distinguished Member of the Technical Staff at Bell Laboratories/Avaya Labs. He has computer science degrees from Brigham Young University, Purdue University, and the University of Groningen.