Mining Architectural Technical Debt through LLM-Assisted Issue Inspection: An Empirical Study with Apache Projects

Armando Sousa
Computer Science Department,
Federal University of Piauí
Teresina, Brazil
armando@ufpi.edu.br

Lincoln Rocha
Computer Science Department,
Federal University of Ceará
Fortaleza, Brazil
lincoln@dc.ufc.br

Ricardo Britto
AB, Ericsson
Karlskrona, Sweden
ricardo.britto@ericsson.com

Abstract

Architectural Technical Debt (ATD) is difficult to observe because architectural degradation emerges from dispersed evidence: recurrent changes, architectural smells, issue-tracker discussions, and design decisions that are rarely documented in a single artifact. This paper presents an empirical study on software architecture and issue inspection supported by Large Language Models (LLMs). We build on *ATDCodeAnalyzer*, an automated approach that identifies critical classes potentially affected by ATD using historical change metrics, co-change information, and architectural-smell evidence. We then investigate whether issues linked to commits that modify such critical classes exhibit architectural impact. The study analyzes four mature Apache Java systems—Cassandra, Kafka, ActiveMQ, and Hadoop—covering 75,809 commits, 51,163 issues, 9,471 commits touching critical classes, 4,570 issues linked to those commits, and 685 inspected issues. A semi-automatic LLM-assisted protocol labels issues as architecturally impactful or not, records textual justifications, and is compared against manual inspection on Apache Cassandra. Results show substantial agreement between manual and LLM-assisted labeling (Cohen's $\kappa = 0.721$) and indicate that 34.5% to 42.5% of inspected issues linked to critical classes have architectural impact. Critical classes also appear prominently among the classes with the highest number of architecture-impact issues. The findings suggest that LLM-assisted inspection can reduce the effort of mining issue trackers while preserving traceability and supporting evidence-based ATD management.

Keywords

Software Architecture, Architectural Technical Debt, Self-Admitted Technical Debt, Issue Trackers, Large Language Models, Empirical Software Engineering

1 Introduction

Technical Debt (TD) captures the long-term maintenance cost introduced by short-term software decisions [9, 12]. Architectural Technical Debt (ATD) is particularly harmful because it affects structural decisions, component dependencies, architectural constraints, and cross-cutting design assumptions that influence future evolution [2, 24]. While a local code smell may be mitigated with an isolated refactoring, ATD often requires coordinated changes across modules, interfaces, build pipelines, tests, deployment assumptions, and sometimes even operational practices. Therefore, ATD tends to remain hidden until a system evolves enough for the accumulated costs to become visible.

Self-Admitted Technical Debt (SATD) provides a complementary signal. Developers sometimes explicitly document limitations, workarounds, incomplete implementations, postponed design decisions, or risky assumptions in comments, commit messages, and issue discussions [14, 15, 20]. SATD is not equivalent to ATD, but it can reveal design discomfort and maintenance risk. For example, a developer note indicating that a synchronization mechanism is temporary may later correspond to a recurring architectural problem in a distributed system. Similarly, issue discussions may expose component coupling, scalability limitations, or configuration constraints that are not visible through static analysis alone.

Despite extensive work on TD, SATD, architectural smells, and software repository mining, validating whether automatically detected critical files are actually related to architectural issues remains challenging. Manual inspection requires architectural expertise and is costly for large repositories. Issue trackers provide rich evidence about design concerns, bug fixes, improvements, feature requests, component interactions, and developer reasoning, but their textual nature makes systematic inspection difficult at scale [1, 18, 21]. Recent LLMs can support classification, summarization, and reasoning over software artifacts [3, 4, 25]. This motivates their use as inspection assistants for architecture-related issue analysis, provided that the process remains traceable, verifiable, and aligned with empirical rigor.

This paper studies the relationship between software architecture and issue inspection supported by LLMs. We build on *ATDCodeAnalyzer*, which identifies source-code files potentially affected by ATD by combining historical evolution metrics and architectural-smell evidence [22]. The central research question is: **RQ.** *To what extent do issues associated with commits affecting critical classes demonstrate a relationship with ATD and SATD?*

To answer this question, we analyze four Apache Java projects [1]: Cassandra, Kafka, ActiveMQ, and Hadoop. These systems are large, long-lived, and architecturally complex. They also maintain public repositories and issue trackers, allowing us to connect commits, critical classes, issues, and textual discussions. We identify commits that modify critical classes, link them to issue-tracker tickets, select representative issue samples, and apply a semi-automatic LLM-assisted inspection model to classify architectural impact. The study follows an empirical pipeline designed to improve originality, relevance, rigor, verifiability, transparency, and quality.

The main contributions are: (i) an empirical protocol that links critical classes, commits, issue-tracker data, SATD evidence, and architectural-impact labels; (ii) a reusable LLM-assisted inspection process for classifying architecture-related issues with textual justifications; (iii) a cross-project dataset of 685 inspected issues from four Apache systems; and (iv) empirical evidence that issues linked

[1] https://www.apache.org

to commits touching critical classes frequently exhibit architectural impact and that LLM-assisted inspection can achieve substantial agreement with manual labels.

The remainder of this paper is organized as follows. Section 2 presents background concepts. Section 3 describes the methodology. Section 4 presents the dataset analyzed. Section 5 reports the results. Section 6 discusses implications. Section 7 reviews related work. Section 8 discusses threats to validity and Section 9 concludes the paper.

2 Background

2.1 Architectural Technical Debt

ATD arises when architectural decisions or structural problems make future maintenance and evolution more expensive [2, 24]. Examples include cyclic dependencies, erosion of modular boundaries, unstable abstractions, excessive coupling, duplicated architectural responsibilities, and classes that become coordination hubs. ATD differs from local code debt because its consequences propagate across modules and often require architectural refactoring rather than isolated patches. Prior studies emphasize the need to identify, prioritize, and manage ATD using both technical indicators and stakeholder information needs [16, 17, 23].

ATD is strongly related to software evolution because design decisions that were once acceptable may become problematic as requirements, performance constraints, or deployment environments change. Therefore, detecting ATD requires understanding how the system evolves, including recurrent co-changes, accumulated modifications, and architectural smells, which motivates combining repository mining with structural analysis.

2.2 Self-Admitted Technical Debt

SATD denotes debt explicitly acknowledged by developers, often through terms such as *TODO*, *FIXME*, *workaround*, or comments that signal incomplete or suboptimal design choices [14, 20]. SATD detection evolved from keyword-based heuristics to machine-learning and natural-language-processing techniques [11, 15]. Although SATD does not cover all technical debt, it provides valuable developer-generated evidence for identifying maintenance risks and refactoring opportunities.

In this study, SATD is treated as an evidence stream rather than the sole target variable. We search for SATD-related signals in commit messages, diffs, issue summaries, descriptions, and comments. Such signals can reveal whether developers themselves acknowledge debt around classes that *ATDCodeAnalyzer* identifies as critical. The combination is important: an architectural hotspot may be detectable from metrics, but SATD helps explain why maintainers perceive parts of the system as problematic or incomplete.

2.3 Issue Trackers as Architectural Evidence

Issue trackers provide structured and unstructured evidence about software evolution, including metadata, descriptions, comments, design discussions, workarounds, and links to commits or pull requests. They are valuable for architecture research because they can reveal the rationale behind changes, such as cross-component failures, new integration mechanisms, or extensions to core abstractions [18]. However, because they also contain noisy or mixed-content issues, architecture-impact inspection requires a clear definition: an issue is considered architecturally impactful when its implementation or resolution affects system-level design decisions, module responsibilities, communication mechanisms, data storage, concurrency, configuration, distributed coordination, or core abstractions; otherwise, localized fixes are classified as non-impactful.

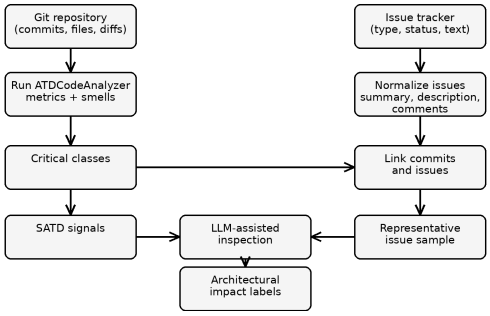

Figure 1: Evaluation pipeline linking repositories, issue trackers, critical classes, SATD signals, and LLM-assisted architectural-impact labels.

2.4 LLMs for Software-Engineering Inspection

LLMs have shown strong capabilities in code-related tasks, including code generation, completion, summarization, and reasoning over textual software artifacts [4, 6, 8]. In issue trackers, LLMs can assist with classification, triage, summarization, and justification because issue reports combine structured fields and unstructured developer discussions. However, LLM-supported empirical research must preserve traceability, avoid opaque labels, and compare outputs against human inspection when possible. Therefore, our protocol stores issue inputs, prompts, labels, and justifications to make the inspection auditable.

The use of LLMs in our study is deliberately conservative. The LLM is not asked to infer hidden architectural facts beyond the issue text. Instead, it receives definitions, examples, and the issue content, then returns a binary label and a textual rationale. The classification process is designed to be repeatable and inspectable by researchers. This design follows the principle that LLMs can reduce repetitive inspection effort while human researchers remain responsible for protocol design, validation, interpretation, and quality control.

3 Approach and Methodology

Figure 1 summarizes the evaluation pipeline. The method begins by extracting repository and issue-tracker artifacts. It then applies *ATDCodeAnalyzer* to detect critical classes, filters commits touching such classes, links those commits to issues, extracts SATD signals, selects representative issues, and uses LLM-assisted inspection to label architectural impact.

3.1 Overview of the Eight Phases

The pipeline[2] has eight phases. First, repository artifacts are collected and *ATDCodeAnalyzer* is executed. Second, commits that touch critical classes are retained. Third, issue fields such as type, status, priority, summary, description, and comments are extracted. Fourth, correlation metrics quantify how critical classes appear in commits and linked issues. Fifth, SATD keywords are searched in commit messages, diffs, and issue text. Sixth, a statistically representative sample of issues is selected for inspection. Seventh, the LLM-assisted inspector classifies each issue as *yes* or *no* for architectural impact and records a justification. Eighth, the final labels are analyzed by repository, issue type, class, and time-to-resolution.

This phased design makes the study traceable. Each issue label can be traced back to the issue text, its linked commits, the critical classes touched by those commits, and the prompt used during inspection. This traceability is important for a research setting in which LLM outputs must be auditable and where future researchers may want to reproduce, challenge, or extend the labels.

3.2 Critical-Class Identification

ATDCodeAnalyzer detects ATD-prone files by combining historical change information, co-change patterns, and architectural-smell indicators. The underlying assumption is that critical classes affected by ATD tend to concentrate recurrent changes, appear in commits linked to important issues, and participate in structural problems. Metrics include accumulated modified lines of code (AMLOC), file occurrence in commits (FOC), cyclomatic complexity (CC), and co-change/composition indicators. Quartile-based selection identifies files with high maintenance and architectural relevance [22].

The method is aligned with Design Science Research because it evaluates an artifact that supports a relevant software-engineering problem [19]. The artifact is not only a classifier but a mining pipeline that helps researchers identify candidate architectural hotspots. In this paper, the candidate hotspots are further validated through issue-tracker evidence. The validation does not assume that every critical class is architecturally problematic; instead, it asks whether commits involving such classes are disproportionately connected to issues that express architectural impact.

We study four widely-used Apache Java projects (Cassandra, Kafka, Hadoop, ActiveMQ) hosted on GitHub, chosen for complex architectures, long histories, and industrial relevance. Following Apache's development and issue-tracking standards, we executed *ATDCodeAnalyzer* to identify critical classes impacted by ATD and used this set to drive subsequent analyses. To characterize the repositories, we employed `cloc`[3] (LOC, files, Java files, comments at 2023/10/04). Detailed repository attributes and the evaluation steps are available in the replication kit.

We explain the evaluation with Apache Cassandra (RP1), including relationships between commits and issues and a step-by-step application of our method (Figure 2). The list of critical classes and extraction procedure are provided in the replication kit.

3.3 Commit and Issue Linkage

For each project, we extracted commits from the Git repository and issues from the corresponding Apache issue tracker. A commit was retained for downstream analysis when it modified at least one critical class. We then identified issue references in commit messages using project-specific issue-key patterns such as `CASSANDRA-XXXX`, `KAFKA-XXXX`, `AMQ-XXXX`, and `HADOOP-XXXX`. The linkage step produced the population of issues associated with commits touching critical classes.

This linkage strategy favors precision over recall. If a commit does not mention an issue key, the corresponding issue cannot be connected automatically. Conversely, when project conventions encourage issue references in commits, the link provides useful traceability from discussion to implementation. We therefore interpret the linked population as the set of issues observable through explicit repository-tracker references rather than the complete universe of all architecture-related discussions.

3.4 SATD Extraction

SATD-related signals were extracted from commit messages, diffs, issue summaries, descriptions, and comments. The keyword lexicon was compiled from prior SATD studies [11, 14, 20]. The goal was not to create a new SATD classifier, but to annotate the inspected population with developer-admitted debt evidence. SATD evidence helps distinguish cases where architectural impact is accompanied by explicit developer recognition of a design or implementation compromise.

The extraction also supports qualitative interpretation. For example, if an issue linked to a critical class contains terms such as *workaround*, *temporary*, or *refactor*, the issue may deserve closer inspection even if the label is ultimately non-architectural. Conversely, an issue may have architectural impact without SATD keywords, such as when it introduces a new distributed coordination mechanism or changes a core storage abstraction.

More specifically, we retained detailed Cassandra commits that touched at least one critical class, resulting in 4,522 commits, and used them for downstream correlation. From the Cassandra Jira[4], we analyzed 18,635 issues across ~14 years (2009/03/02–2023/10/04), organizing fields (type, status, priority, text) for subsequent linkage with commits.

3.5 Sampling and Inspection Protocol

For Cassandra, the inspected sample was selected from the 2,912 issues linked to commits that modified critical classes. We used a confidence level of 0.95, margin of error of 0.05, and expected proportion of 0.8, producing 226 issues. For each selected issue, we generated a structured text representation containing the issue key, type, status, priority, summary, description, and comments. The same representation was used for manual inspection and LLM-assisted inspection. The manual inspection process was done by 3 system engineers who know the four Apache projects.

We adopted a semi-automatic inspection using ChatGPT[5] (v3.5) with few-shot and chain-of-thought prompting [3, 26] to classify

[2] https://github.com/Technical-Debt-Large-Scale/my_validation/blob/main/imagens/a.jpg
[3] https://github.com/AlDanial/cloc
[4] https://issues.apache.org/jira/projects/CASSANDRA
[5] https://chat.openai.com

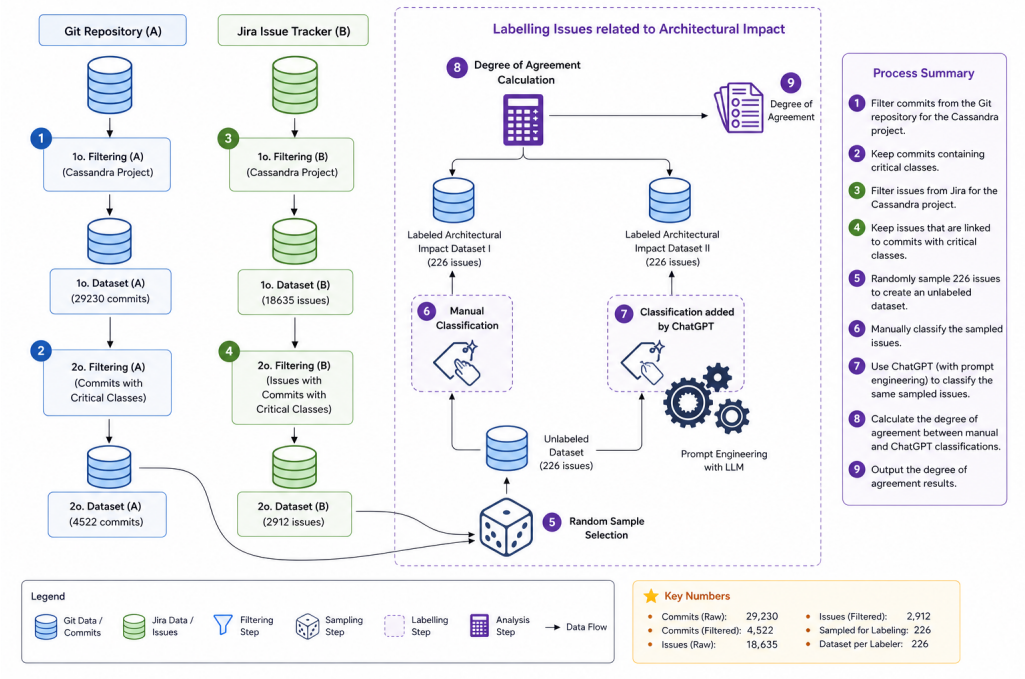

Figure 2: Empirical pipeline applied to Apache Cassandra

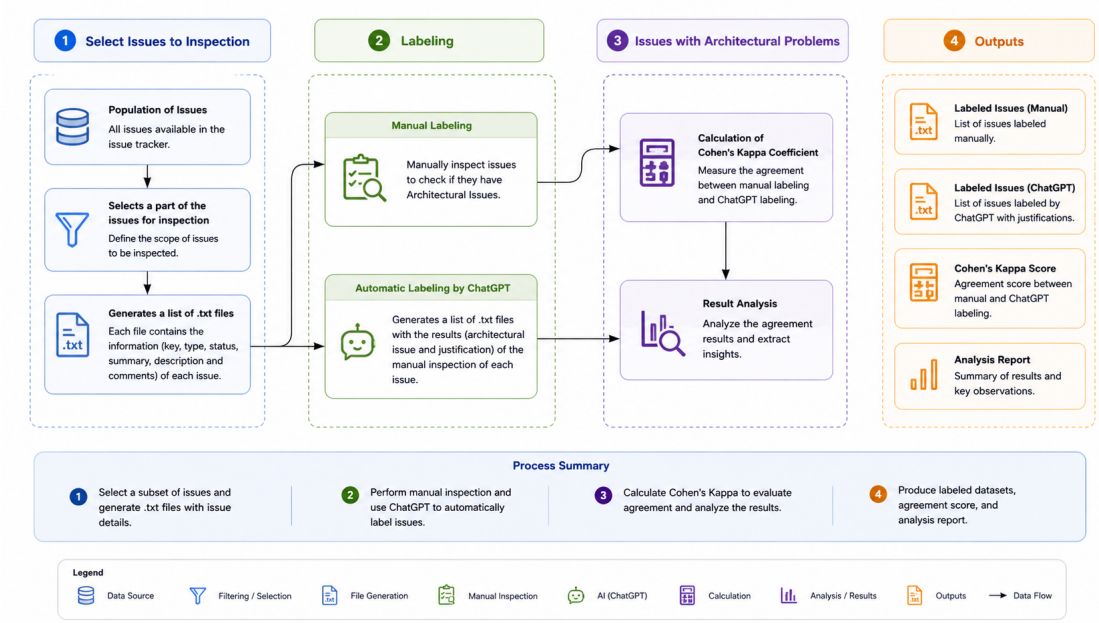

Figure 3: Issue labeling process

whether issues have architectural impact and to record justifications. The process (Figure 3) comprises six stages: (1) concepts and examples (ATD/SATD, commits, issues); (2) exemplar issues and *Architectural Impact Classifier*; (3) data preparation (226 `.txt` issue files); (4) manual inspection; (5) ChatGPT inspection (226 files); (6) agreement analysis via Cohen's Kappa between manual and automatic labels. Artifacts and prompts are provided in the replication kit related to inspection protocol.

The LLM-assisted inspection protocol receives a structured textual representation of each issue. The prompt defines architectural

impact, provides positive and negative examples, and requests a binary label plus a concise justification. The classification is not used as an unverified oracle. For Cassandra, labels are compared with manual inspection using Cohen's Kappa, precision, recall, accuracy, and F1-score. This design supports transparent, auditable use of LLMs rather than replacing expert judgment. The full script related to ispection protocol is available in this repository kit[6].

3.6 Architectural-Impact Definition

An issue was labeled as having architectural impact when the issue text indicated that its solution would affect system-level design. Examples include changes to storage engines, distributed coordination, replication, messaging, scheduling, configuration infrastructure, security subsystems, stream-processing topology, broker behavior, or core public abstractions. Issues were labeled as not having architectural impact when they described localized bug fixes, test adjustments, documentation changes, minor parameter corrections, or isolated feature requests without evident structural implications.

This definition intentionally focuses on impact rather than severity. A low-priority issue may still be architectural if it modifies a core abstraction. A high-priority issue may be non-architectural if it involves a localized fix. Therefore, priority and type are used as descriptive fields rather than classification rules.

4 The Architectural Impact Issues Dataset

We analyzed the four Apache Java repositories with long histories and industrial relevance. Table 1 summarizes the repository characteristics used in the empirical study.

This section analyzes our dataset to assess the relationship between architecturally impactful issues and commits touching *critical classes*. We summarize the study goal, research questions, Project Profiles and Cassandra as detailed running example.

4.1 Goal and Research Questions

We applied the evaluation method to Apache ActiveMQ, Apache Kafka, and Apache Hadoop (in addition to Cassandra) to measure architectural impact around critical files. Our **goal** is:

Quantify the number and proportion of issues with architectural impact associated with critical classes across four open-source Java systems.

Based on issues linked to commits that modify critical classes, we ask:

RQ1) *What proportion of issues in each repository are classified as impacting architectural design?*

RQ2) *Which classes are most frequently involved in architecturally impactful issues?*

Table 2 presents the collection pipeline from raw commits and issues to inspected architectural-impact labels. The final dataset contains 685 inspected issues: 226 from Cassandra, 179 from Kafka, 132 from ActiveMQ, and 148 from Hadoop.

4.2 Project Profiles

Apache Cassandra is a distributed database whose architecture-impact issues often involve storage, replication, compaction, schemas, SSTables, node operations, and messaging. Apache Kafka is a distributed event-streaming platform with issues related to consumers, producers, stream processing, coordinators, fetchers, and task management. Apache ActiveMQ focuses on message brokering, with issues involving queues, dispatching, broker services, connectors, and persistence. Apache Hadoop covers distributed computing and storage, with architectural concerns related to configuration, file systems, data nodes, block management, schedulers, and protocols.

The diversity of these systems strengthens the empirical setting. Although all projects are Java and Apache-based, their architectural domains differ: distributed databases, streaming infrastructure, messaging middleware, and distributed data processing/storage. This allows us to observe whether the relationship between critical classes and architectural-impact issues appears across different kinds of large infrastructure software. The full scripts related to evaluate commits and issues the Apache Projects are available in this repository[7] and this repository[8]. The next subsection explain more details related to each Apache Project evaluation.

Table 1: Analyzed repositories.

Property	Cassandra (RP1)	Kafka (RP2)	ActiveMQ (RP3)	Hadoop (RP4)
Files	4,998	5,648	5,174	14,970
LOC files	1,055,561	874,543	466,706	4,323,485
Java files	4,459	4,350	4,367	11,811
LOC Java files	680,827	649,612	417,291	1,889,967
Java comments	166,567	175,655	150,220	613,888
Commits	29,142	11,810	11,537	26,945
Issues	18,635	14,326	5,955	12,247
Releases	297	65	85	374
Lifespan (years)	14.53	12	14	14
Stars	8,200	26,100	2,200	13,900
Forks	3,500	13,000	1,400	8,600
Collaborators	426	1,062	130	552

Table 2: Dataset collection and inspected labels.

Description	Cassandra	Kafka	ActiveMQ	Hadoop	Total
Extracted commits	29,230	11,732	7,941	26,906	75,809
Commits with critical classes	4,522	1,452	721	2,776	9,471
Extracted issues	18,635	14,326	5,955	12,247	51,163
Issues with critical classes	2,912	939	480	239	4,570
Issues inspected	226	179	132	148	685
Architectural impact	96	72	55	51	274
No architectural impact	130	107	77	97	411

4.2.1 Evaluating the ATDCodeAnalyzer across Apache Projects. For Apache ActiveMQ, we filtered commits and issues, linked issues referenced by commits that modify *critical classes* identified by ATDCodeAnalyzer, and applied the semi-automatic LLM-assisted inspection protocol. Representative critical classes include *DemandForwardingBridgeSupport.java*, *BrokerService.java*, *TransportConnector.java*, *Queue.java*, and *TopicSubscription.java*. The resulting issues were labeled as having or not having architectural impact.

For Apache Kafka, we applied the same pipeline by selecting commits that touch critical classes, associating them with referenced issues, and classifying architectural impact with LLM assistance. Representative critical classes include *KafkaConsumer.java*, *KafkaProducer.java*, *KafkaStreams.java*, *StreamThread.java*, and *Fetcher.java*.

[6] https://github.com/armandossrecife/inspectionprocessevaluation

[7] https://github.com/Technical-Debt-Large-Scale/my_validation

[8] https://github.com/armandossrecife/IssuesRepositoriesApacheProjects

Table 3: Selected Cassandra critical Java files and issue-related metrics.

File	AIC	AICWI	AII	AIB	AIII	AINFI
StorageService.java	1317	1034	967	160	79	3
ColumnFamilyStore.java	1178	893	832	139	96	3
DatabaseDescriptor.java	854	697	659	90	62	4
StorageProxy.java	572	445	463	71	55	2
CompactionManager.java	668	506	406	72	47	5
Config.java	500	393	379	58	37	3
SSTableReader.java	457	399	352	47	48	1
MessagingService.java	328	241	288	50	31	1
NodeProbe.java	369	303	287	39	26	2
CassandraDaemon.java	386	316	235	28	34	5

For Apache Hadoop, we linked issues to commits that modify critical classes and inspected them using the same LLM-aided classification protocol. Example critical classes include *Configuration.java*, *FSNamesystem.java*, *DataNode.java*, *BlockManager.java*, and *CapacityScheduler.java*. As in the other projects, the output was a set of issues labeled according to the presence or absence of architectural impact.

Summary. Across ActiveMQ, Kafka, and Hadoop, the dataset preserves the trace from commits touching critical classes to linked issue-tracker entries and assigns a consistent Yes/No architectural-impact label to each inspected issue through a semi-automatic and reproducible protocol. This supports the analysis of **RQ1**, by measuring the proportion of architectural-impact issues per repository, and **RQ2**, by identifying the classes most frequently involved in such issues, while also enabling replication and extension in future studies.

4.3 Cassandra as Detailed Running Example

Cassandra is used as the detailed running example because it provides a rich issue history and a sufficiently large linked population. The Cassandra analysis (Figure 2) retained 4,522 commits touching at least one critical class and linked them to 2,912 issues. Those issues were then sampled and inspected. The critical classes with high metrics include storage, compaction, configuration, messaging, node management, and query-processing classes. These classes are plausibly architecture-relevant because they embody central responsibilities in Cassandra's distributed database architecture.

Table 3 summarizes selected Cassandra critical classes and their issue-related metrics[9]. AIC denotes appearances in commits, AICWI denotes appearances in commits with issues, AII denotes appearances in issues, AIB denotes appearances in bugs, AIII denotes appearances in improvement issues, and AINFI denotes appearances in new-feature issues.

The metrics show that critical classes are not only frequently changed; they are also frequently connected to issues. For instance, `StorageService.java` appears in 1,034 commits with issues and in 967 issue-related occurrences. `ColumnFamilyStore.java` shows similarly high values and is later observed among the top classes with architectural-impact issues. Such data motivates the deeper inspection stage.

[9]https://github.com/Technical-Debt-Large-Scale/my_validation/blob/main/ccim.md

5 Results

This section reports the empirical results. We first present the agreement between manual and LLM-assisted inspection in Cassandra. We then answer RQ1 by reporting the proportion of architectural-impact issues in each repository. Finally, we answer RQ2 by analyzing which classes are most frequently involved in architecture-impact issues.

5.1 Manual and LLM-Assisted Agreement

For Apache Cassandra, we compared manual inspection against LLM-assisted labels. Manual inspection labeled 76 of 226 issues (33.63%) as architecturally impactful and 150 (66.37%) as not architecturally impactful. The LLM-assisted process labeled 96 issues (42.48%) as *yes* and 130 (57.52%) as *no*. The agreement analysis produced Cohen's $\kappa = 0.721$, which is commonly interpreted as substantial agreement [5, 10]. The classifier reached precision 0.926, recall 0.833, accuracy 0.867, and F1-score 0.893. These results indicate that the LLM-assisted protocol can reliably support inspection while still requiring expert review for borderline cases.

We performed both manual and ChatGPT-assisted inspections and computed Cohen's Kappa to assess agreement. All scripts, data and results are in the replication kit[10].

Cohen's Kappa was used to interpret the level of agreement between manual and LLM-assisted inspection. Following the adopted interpretation scale, values below 0.00 indicate poor agreement, 0.00–0.20 slight agreement, 0.21–0.40 fair agreement, 0.41–0.60 moderate agreement, 0.61–0.80 substantial agreement, and 0.81–1.00 almost perfect agreement. Since the obtained value falls within the substantial range, the result suggests that the semi-automatic process is consistent enough to support large-scale inspection when combined with validation and quality-control mechanisms. However, this does not imply that the LLM labels are perfect, but rather that they can provide reliable first-pass classifications under the defined protocol.

A qualitative inspection of disagreements suggests that boundary cases often involve issue reports with mixed concerns. Some issues mention design elements but resolve with localized patches. Others describe implementation symptoms while requiring deeper design changes. These cases are precisely where a human-in-the-loop process is valuable: the LLM can provide an initial label and explanation, but researchers or maintainers can review cases with low confidence or high impact.

5.2 RQ1: Proportion of Architectural-Impact Issues

Figure 4 shows the proportion of inspected issues with and without architectural impact for each repository. Among issues linked to commits that modify critical classes, the proportion of architectural-impact issues ranges from 34.5% in Hadoop to 42.5% in Cassandra. Kafka and ActiveMQ show similar proportions, with 40.2% and 41.7%, respectively.

The results suggest that a substantial fraction of issues connected to critical classes involve architectural concerns. This supports the hypothesis that critical classes identified by *ATDCodeAnalyzer*

[10]https://github.com/Technical-Debt-Large-Scale/my_validation/blob/main/mllm.md

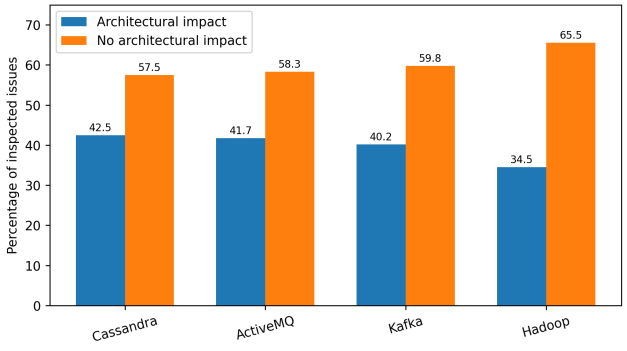

Figure 4: Percentage of inspected issues with and without architectural impact in commits that touch critical classes.

represent architectural hotspots that deserve additional inspection and prioritization. The results are also practically relevant: if a maintainer wants to prioritize a limited inspection budget, commits touching critical classes provide a focused entry point where the probability of finding architecture-relevant issues is high.

5.3 Resolution Effort and Temporal Patterns

Issues with architectural impact tend to demand more design reasoning, coordination, and regression analysis than local bug fixes. In the analyzed data, issues linked to critical classes frequently require longer resolution time[11] than issues not associated with architectural impact. Temporal trends also indicate that commits touching critical classes decrease over time in some projects, while architecture-impact issues present project-specific peaks. Such peaks can correspond to releases, feature introduction, refactoring cycles, or architectural stabilization periods.

Issues with architectural impact also take longer to resolve than those not involving critical classes (Figure 5). Over time, commits involving critical classes tend to decrease, while issues with architectural impact exhibit distinct temporal patterns. These graphics are available in result section of the main replication kit[12].

This pattern is consistent with the interpretation that architecture-impact issues represent more than ordinary churn. When a class lies at the center of storage, messaging, scheduling, or configuration behavior, changes to that class can require compatibility considerations, data-migration reasoning, concurrency analysis, distributed-system testing, and coordination among maintainers. Consequently, issue-resolution time becomes an indirect indicator of architectural complexity, even though it should not be used as a definitive measure on its own.

5.4 RQ2: Classes Most Involved in Architectural-Impact Issues

Table 4 reports representative top classes with more architectural-impact issues. In Cassandra, several top classes are critical classes,

[11]Resolution time is the difference in days between the ‘created‘ and ‘resolutiondate‘ fields from Apache JIRA metadata. Only issues with status *Resolved* or *Closed* and a populated ‘resolutiondate‘ were included.
[12]https://github.com/Technical-Debt-Large-Scale/my_validation

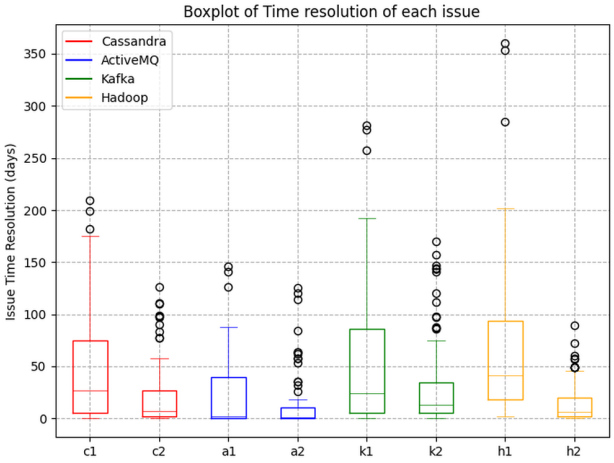

Figure 5: Boxplot of issue resolution time for issues related to architectural impact.

including *ColumnFamilyStore.java*, *StorageService.java*, *DatabaseDescriptor.java*, *SSTableReader.java*, and *CompactionManager.java*. Similar patterns appear in Kafka, ActiveMQ, and Hadoop, where critical classes related to stream processing, queue management, configuration, and distributed-storage coordination appear among the most frequent architecture-impact locations.

Some highly frequent files are not marked as critical, especially test classes or broad metadata files. This is expected: issue frequency alone does not imply ATD. The relevant observation is that many critical classes appear among the top architecture-impact classes, indicating that *ATDCodeAnalyzer* identifies files that are both historically change-prone and semantically connected to architectural discussions.

5.5 Issue-Type, Status, and Priority Analysis

The original dataset records issue type, status, and priority as contextual variables for interpreting architectural impact, but these fields are not used as deterministic classification rules. Architectural concerns may appear as bugs, improvements, tasks, or new features, and their distribution helps characterize each project's maintenance profile. In Cassandra, for example, many issues linked to commits touching critical classes are improvements, suggesting that architectural pressure is not limited to bug fixing but also involves refactoring, API evolution, storage-engine adjustments, and changes in distributed behavior.

Bug and new-feature issues also require semantic inspection. Architecture-impact bugs may expose design fragility in replication, compaction, stream scheduling, broker dispatching, or block management, while new features may be architectural when they extend subsystems, introduce integration points, or affect configuration, communication, or storage behavior. Conversely, issues of the same type may be non-architectural when they describe localized fixes or narrow interface changes. The issue type alone does not determine whether a change has architectural impact. You can see more details about each field Table 5.

5.6 Interpretive Examples of Architectural Impact

Although the inspection protocol operates at scale, the resulting labels are grounded in recurring qualitative patterns. Issues labeled as architectural typically mention changes to core behavior, component interactions, distributed coordination, data layout, scheduling mechanisms, security infrastructure, or configuration semantics. They often require maintainers to reason about backward compatibility, failure modes, concurrency, performance, or migration. For example, changes around Cassandra storage services can affect node behavior and data consistency. Changes around Kafka stream threads can affect execution lifecycle and fault handling. Changes around ActiveMQ queues can affect dispatch policies and persistence semantics. Changes around Hadoop configuration and block management can affect distributed execution and storage guarantees.

Issues labeled as non-architectural usually describe localized fixes, tests, documentation, small usability improvements, or isolated behavior changes. These issues may still be important, but they do not substantially alter module responsibilities or system-level design. The distinction is not always obvious: a test failure may indicate a deep architectural problem, while a class with an architectural name may be changed for a local reason. This reinforces the importance of reading the issue content rather than relying only on file names or issue metadata.

The LLM-assisted process helps by applying a consistent decision criterion across hundreds of issues. Human inspectors can then focus on validating difficult cases, analyzing patterns, and interpreting the results. In this sense, the LLM acts as a scaling mechanism for qualitative inspection. It does not eliminate the need for empirical rigor; instead, it requires additional rigor in prompt design, output storage, validation, and threat analysis.

5.7 Detailed Class-Level Findings

The class-level results provide a more concrete view of why critical classes are useful indicators for architectural inspection. Table 4 expands the representative class analysis by listing the top architecture-impact classes for each repository. The table preserves the main pattern observed in the original study: many of the highest-ranked classes are exactly those identified as critical by *ATDCodeAnalyzer*, while non-critical entries are often test harnesses, metadata files, or utility classes that appear frequently because they support broad validation activities.

In Cassandra, architecture-impact issues concentrate on storage and compaction responsibilities. *ColumnFamilyStore.java*, *StorageService.java*, *DatabaseDescriptor.java*, and *SSTableReader.java* are connected to persistence, configuration, and distributed operation mechanisms. In Kafka, *StreamThread.java*, *Fetcher.java*, *StreamTask.java*, and *KafkaConsumer.java* represent execution and consumption mechanisms that affect the stream-processing architecture. ActiveMQ shows a broker-oriented profile, with *Queue.java*, *QueueDispatchSelector.java*, and *BrokerService.java*. Hadoop exhibits a broader infrastructure profile, with *Configuration.java*, *Writable.java*, *BlockManager.java*, and *CapacityScheduler.java*.

These observations are relevant for architecture governance because they indicate where maintainers should look first when issue backlogs are large. A class that is both critical according to historical/smell metrics and repeatedly involved in architecture-impact issues is a strong candidate for deeper design review. The LLM-assisted labels do not remove the need for expert judgment, but they help reduce the search space by organizing issue evidence around architectural hotspots.

Table 4: Expanded examples of classes involved in architectural-impact issues.

Project	Class	Issues	Critical
Cassandra	ColumnFamilyStore.java	20	Yes
Cassandra	StorageService.java	14	Yes
Cassandra	DatabaseDescriptor.java	11	Yes
Cassandra	SSTableReader.java	11	Yes
Cassandra	CompactionManager.java	10	Yes
Kafka	StreamThread.java	14	Yes
Kafka	Fetcher.java	11	Yes
Kafka	StreamTask.java	10	Yes
Kafka	KStreamImpl.java	10	Yes
Kafka	KafkaConsumer.java	9	Yes
ActiveMQ	Queue.java	12	Yes
ActiveMQ	QueueDispatchSelector.java	5	Yes
ActiveMQ	BrokerService.java	5	Yes
ActiveMQ	QueueBrowserSubscription.java	3	Yes
ActiveMQ	TransportConnector.java	3	Yes
Hadoop	Configuration.java	15	Yes
Hadoop	Writable.java	9	Yes
Hadoop	BlockManager.java	7	Yes
Hadoop	ClientProtocol.java	7	Yes
Hadoop	CapacityScheduler.java	5	Yes

In Cassandra, the strongest signals are concentrated in storage, compaction, and configuration classes. This is coherent with Cassandra's architecture: storage and replication behavior influence performance, consistency, data layout, repair, compaction, and node operations. In Kafka, classes such as `StreamThread.java`, `Fetcher.java`, `StreamTask.java`, `ConsumerCoordinator.java`, and `KafkaConsumer.java` connect issue discussions to streaming execution, consumption, coordination, and task management. In ActiveMQ, queue and broker-service classes are central to message dispatching and broker behavior. In Hadoop, configuration, writable abstractions, block management, client protocols, node management, and scheduling appear as relevant architectural loci.

The cross-project consistency is important. The particular class names differ, but the pattern remains: classes that implement central architectural mechanisms are more likely to accumulate issues whose resolution affects architecture. This reinforces the value of combining automated hotspot detection with issue-level semantic inspection.

6 Discussion

The results support three main interpretations. First, critical classes identified from historical and architectural-smell indicators are useful entry points for architecture-aware maintenance. The fact that between one-third and nearly half of inspected linked issues exhibit architectural impact suggests that the approach captures more than local churn. Such proportions are high enough to justify focused inspection and low enough to show that the method is not trivially labeling everything as architectural. This balance is important for practical usefulness.

Second, LLM-assisted inspection is promising for scaling issue-tracker analysis. The substantial agreement observed in Cassandra indicates that LLMs can help classify architecture-related issue

reports and produce useful justifications. Nevertheless, the role of the LLM should be framed as inspection assistance, not automatic ground truth. Expert review remains important for ambiguous issues, domain-specific terminology, and cases where architectural impact depends on implicit project knowledge.

Third, combining issue trackers with repository history improves ATD observability. Static analysis and smell detection identify structural symptoms, while issues and commits provide developer intent and maintenance context. This combination helps maintainers prioritize files that are not only structurally problematic but also repeatedly involved in architecture-relevant evolution. In practice, maintainers could use this pipeline to identify candidate classes for architectural review before major releases, during refactoring planning, or when triaging long-standing issue backlogs.

6.1 Implications for Researchers and Practitioners

For researchers, this study provides a protocol for connecting ATD detection with issue-tracker evidence. Instead of evaluating ATD tools only through static smells or expert labels, researchers can examine whether detected hotspots are associated with architecture-impact issue discussions, creating a richer validation setting. The study also illustrates a controlled use of LLMs in empirical software engineering by relying on explicit definitions, stored prompts, textual justifications, and comparison with manual inspection. These results open directions for future work on prompt sensitivity, cross-domain replication, open-weight LLMs, active-learning workflows, and the integration of issue-level labels with code metrics for architecture-impact prediction.

For practitioners, the proposed pipeline can support architecture governance and technical-debt prioritization. By focusing on commits that touch critical classes and applying LLM-assisted triage, development teams can identify issues that are more likely to require architectural attention without manually inspecting large issue backlogs. The approach can also enrich technical-debt dashboards by highlighting classes that are both structurally critical and semantically connected to architecture-impact discussions. Since issue trackers preserve fragments of architectural rationale, LLM-assisted summarization and classification can help recover this knowledge, provided that final decisions remain part of a human-in-the-loop engineering workflow.

The empirical pipeline can be translated into a practical maintenance workflow in which repositories are periodically mined to update critical-class metrics, recent commits touching those classes are linked to issue-tracker entries, and issue texts are inspected with the LLM-assisted classifier. Maintainers can then review the resulting list of architecture-impact issues in relation to release plans, refactoring initiatives, and operational incidents, deciding whether critical classes require refactoring, additional documentation, ownership assignment, architectural decision records, or continued monitoring. This workflow is useful because architectural debt is rarely visible from a single indicator: by connecting quantitative repository evidence with qualitative issue evidence, the approach helps teams distinguish ordinary churn from architectural pressure and make more informed technical-debt management decisions.

Table 5: Structured fields used in the LLM-assisted issue inspection.

Field	Purpose in inspection
Issue key	Preserves traceability to the issue tracker
Issue type	Provides context such as bug, improvement, or task
Status and priority	Describe process state and reported importance
Summary	Gives a compact description of the requested change
Description	Provides the main technical evidence for classification
Comments	Capture design discussion and maintainer rationale
SATD signals	Highlight explicit debt-related vocabulary
LLM label	Stores the architectural-impact decision
Justification	Explains why the issue was labeled Yes or No

6.2 Interpretation of LLM Reliability

The agreement result should be interpreted carefully: a Cohen's Kappa of 0.721 indicates substantial agreement in the Cassandra sample, but not universal reliability across projects, models, or prompt settings. LLM outputs may vary with model updates, prompt design, issue length, missing context, and domain-specific terminology. Nevertheless, the strong precision and F1-score (0.893) suggest that, under the defined protocol, the LLM-assisted inspector can reduce repetitive work and provide useful first-pass labels. Therefore, LLM labels should be used as a triage layer rather than as ground truth: high-confidence positive cases can be reviewed first by architecture maintainers, while disagreements or uncertain cases should be routed to experts. The stored justifications help reviewers understand each decision, preserve accountability, and support the incremental refinement of prompts, examples, and definitions.

6.3 Prompt Schema and Inspection Output

The inspection prompt[13] was designed to make the LLM operate as a structured classifier rather than as a free-form evaluator. Each input file represented one issue and included the issue identifier, type, status, priority, summary, description, and comments. The prompt defined architectural impact, provided positive and negative examples, and requested a two-field output: a binary label and a justification. This structure enabled both quantitative analysis and qualitative review, since the binary label supported agreement and classification metrics, while the justification allowed researchers to audit the rationale behind each decision. In practice, the schema supports a human-in-the-loop workflow in which researchers can sort issues by label, inspect explanations, prioritize disagreements or high-risk cases, and reuse the protocol with different LLMs because it does not depend on provider-specific features.

7 Related Work

Technical debt research has produced systematic mappings and management frameworks that distinguish debt types, causes, and consequences [9, 12]. SATD studies introduced keyword-based exploration [20], taxonomy and quantification of debt types [14], and NLP-based automatic detection [11, 15]. Our work differs by using SATD as one evidence stream within a broader architecture-impact inspection pipeline.

ATD research focuses on identifying and managing architecture-level debt through design decisions, smell detection, modularity

[13] https://github.com/armandossrecife/inspectionprocessevaluation/blob/main/ip.md

indicators, and architectural debt indices [2, 13, 23, 24]. Architectural smells such as cyclic dependencies and unstable dependencies provide structural signals, while evolution patterns reveal how architectural degradation changes over time [7]. *ATDCodeAnalyzer* contributes to this line by using repository evolution and architectural-smell evidence to identify critical files [22]. Our paper extends this perspective by validating the architectural relevance of such files through issue-tracker inspection.

Mining issue trackers has long supported bug triage, dependency analysis, and maintenance analytics [1, 18, 21]. Issue trackers are attractive because they connect user-visible problems, developer discussions, priorities, and implementation traces. However, their unstructured text creates scalability challenges. Our study addresses this challenge by combining structured extraction, representative sampling, and LLM-assisted classification.

More recently, LLMs have enabled new forms of software artifact analysis, including code understanding and generation [4, 8, 25]. Prior work on code-oriented LLM evaluation shows both the promise and limitations of these models. Our contribution is to operationalize LLMs as assistants for architectural-impact labeling with agreement analysis and artifact-oriented traceability. The focus is not on replacing static analysis or human expertise, but on combining repository mining, architecture-aware definitions, and LLM-supported inspection.

Our study connects critical-class detection with commit–issue traceability and validates an LLM-assisted protocol for inspecting architectural impact, integrating the techniques aforementioned.

8 Threats to Validity

Although the study provides evidence about the relationship among critical classes and architecture-impact issues, several threats must be considered. **Construct validity** is affected by the fact that critical classes are identified through metrics and architectural-smell evidence, which may not capture all architectural problems, while SATD keywords may miss implicit debt or detect non-debt comments. In addition, the architectural-impact label depends on the operational definition adopted in the inspection protocol. To mitigate this threat, the prompt includes explicit definitions and examples, and the Cassandra labels are compared against manual inspection.

Internal validity concerns the issue-to-commit linkage and the LLM-assisted labeling process. The linkage depends on issue references in commit messages and project conventions, so missing or inconsistent references may affect the analyzed issue set. LLM outputs may also vary across model versions, prompts, and decoding parameters. We mitigate these risks by using structured prompts, storing justifications, and comparing LLM-assisted labels with manual inspection for Cassandra; however, future replications should record exact model versions and inference settings.

External validity is limited because the study analyzes four open-source Apache Java infrastructure systems with mature issue trackers. Results may differ in proprietary systems, smaller projects, other programming languages, mobile or web applications, embedded systems, or organizations with weaker traceability practices. Therefore, future work should replicate the protocol across additional ecosystems and industrial datasets.

Conclusion validity is related to the interpretation of the reported proportions, correlations, and agreement metrics. Although the results indicate strong patterns, they should not be interpreted as causal evidence that critical classes cause architecture-impact issues. Instead, the findings support an association useful for prioritization and inspection. Since only samples were inspected, larger replications and additional statistical models could further strengthen the conclusions.

LLM-specific validity concerns the possibility that LLMs may hallucinate justifications, overfit to examples, or infer architectural impact from superficial keywords. The study reduces this risk by requiring labels to be grounded in issue content and by validating the protocol against manual inspection. Nevertheless, LLM outputs should be treated as empirical artifacts subject to validation rather than authoritative truth, and future work should compare multiple LLMs and prompt variants.

9 Conclusions

This paper presented an empirical study on software architecture and issue inspection supported by LLMs. By combining *ATDCodeAnalyzer*, issue-tracker mining, SATD signals, and LLM-assisted labels, we analyzed whether issues linked to commits touching critical classes exhibit architectural impact. Across four Apache Java systems, 34.5% to 42.5% of inspected issues associated with critical classes were classified as architecturally impactful. In Cassandra, the LLM-assisted inspection achieved substantial agreement with manual inspection ($\kappa = 0.721$), with high precision (0.926) and F1-score (0.893).

The findings indicate that LLM-assisted issue inspection can help software-maintenance teams identify architecture-relevant issues at scale, while maintaining traceability through structured inputs, explicit labels, and justifications. Critical classes identified through repository evolution and architectural-smell evidence frequently appear among the classes most involved in architecture-impact issues. This supports the interpretation that combining static/evolutionary signals with issue-tracker semantics provides a stronger basis for ATD analysis than either source alone.

Future work should evaluate additional LLMs, compare prompting strategies, extend the protocol to non-Java systems, and investigate human-in-the-loop workflows for architectural debt prioritization. Another important direction is to integrate the approach into continuous software-engineering dashboards that warn maintainers when issue discussions, SATD signals, and critical-class changes converge around the same architectural components.

Artifact Availability

We provide a replication kit[14] containing the dataset, evaluation protocol, data-analysis scripts, result-generation scripts, and the prompts used in the study. The LLM-assisted inspection protocol was implemented using ChatGPT (3.5) with structured prompts, recorded responses, and a general set of question–answer interactions. Representative input/output examples are also included in the repository[15] to support transparency, auditability, and replication.

[14] https://github.com/Technical-Debt-Large-Scale/my_validation
[15] https://github.com/armandossrecife/inspectionprocessevaluation

References

[1] Sean Banerjee, Z. Syed, J. Helmick, M. Culp, K. Ryan, and B. Cukic. 2017. Automated triaging of very large bug repositories. *Information and Software Technology* 89 (2017), 1–13. doi:10.1016/j.infsof.2017.03.008

[2] Terese Besker, Antonio Martini, and Jan Bosch. 2018. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software* 135 (2018), 1–16. doi:10.1016/j.jss.2017.09.025

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[6] Yangruibo Ding, Vibhu Kumar, Yuchen Tian, Zeyu Wang, Rob Kwiatkowski, Xin Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, Saptarshi Sengupta, et al. 2023. A static evaluation of code completion by large language models. *arXiv preprint arXiv:2306.03203* (2023).

[7] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. 2017. Arcan: A tool for architectural smells detection. In *IEEE International Conference on Software Architecture Workshops*. 282–285. doi:10.1109/ICSAW.2017.16

[8] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).

[9] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2019. *Managing Technical Debt: Reducing Friction in Software Development.* Addison-Wesley Professional.

[10] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174. doi:10.2307/2529310

[11] Yikun Li, Mohamed Soliman, and Paris Avgeriou. 2023. Automatic identification of self-admitted technical debt from four different sources. *Empirical Software Engineering* 28, 3 (2023), 65. doi:10.1007/s10664-023-10293-5

[12] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220. doi:10.1016/j.jss.2014.12.027

[13] Zengyang Li, Peng Liang, Paris Avgeriou, Nicolas Guelfi, and Apostolos Ampatzoglou. 2014. An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures*. 119–128. doi:10.1145/2602576.2602581

[14] Everton da Silva Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the 7th International Workshop on Managing Technical Debt*. 9–15. doi:10.1109/MTD.2015.7332629

[15] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062. doi:10.1109/TSE.2017.2654244

[16] Antonio Martini and Jan Bosch. 2015. Towards prioritizing architecture technical debt: Information needs of architects and product owners. In *41st Euromicro Conference on Software Engineering and Advanced Applications*. 422–429. doi:10.1109/SEAA.2015.32

[17] Antonio Martini and Jan Bosch. 2017. On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. *Journal of Software: Evolution and Process* 29, 10 (2017), e1877. doi:10.1002/smr.1877

[18] Lloyd Montgomery, Clara Marie Lüders, and Walid Maalej. 2025. Mining Issue Trackers: Concepts and Techniques. In *Handbook on Natural Language Processing for Requirements Engineering*. Springer, 309–336. doi:10.1007/978-3-031-73143-3_11

[19] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. 2007. A design science research methodology for information systems research. *Journal of Management Information Systems* 24, 3 (2007), 45–77. doi:10.2753/MIS0742-1222240302

[20] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *IEEE International Conference on Software Maintenance and Evolution*. 91–100. doi:10.1109/ICSME.2014.31

[21] Mikko Raatikainen, Queralt Motger, Christoph M. Lüders, Xavier Franch, Lassi Myllyaho, Eero Kettunen, Jordi Marco, Juha Tiihonen, Mika Halonen, and Tomi Männistö. 2022. Improved management of issue dependencies in issue trackers of large collaborative projects. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2128–2148. doi:10.1109/TSE.2022.3213139

[22] A. Sousa, L. Rocha, R. Britto, and G. Avelino. 2024. An automated approach to identify source code files affected by architectural technical debt. In *International Conference on Product-Focused Software Process Improvement*. Springer, 100–115.

[23] Roberto Verdecchia, Patricia Lago, Ivano Malavolta, and Ipek Ozkaya. 2020. ATDx: Building an architectural technical debt index. In *Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering*. 531–539.

[24] Roberto Verdecchia, Ivano Malavolta, and Patricia Lago. 2018. Architectural technical debt identification: The research landscape. In *Proceedings of the IEEE/ACM International Conference on Technical Debt*. 11–20. doi:10.1145/3194164.3194176

[25] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A Survey of Large Language Models. *arXiv preprint arXiv:2303.18223* (2023).

[26] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).