

Agentic AKM

Rudra Dhar
IIIT Hyderabad
Hyderabad, Telangana, India
rudra.dhar@research.iiit.ac.in

Karthik Vaidhyanathan
IIIT Hyderabad
Hyderabad, India
karthik.vaidhyanathan@iiit.ac.in

Vasudeva Varma
IIIT Hyderabad
Hyderabad, India
vv@iiit.ac.in

Abstract

ICSE AGENT

Keywords

Agentic AI, Architecture Decision Record, Architecture Knowledge Management

ACM Reference Format:

Rudra Dhar, Karthik Vaidhyanathan, and Vasudeva Varma. 2018. Agentic AKM. In . ACM, New York, NY, USA, 5 pages. <https://doi.org/XXXXXXX.XXXXXX>

1 Introduction

Software architecture serves as the foundational blueprint for a system, yet keeping its documentation current and comprehensive is a persistent challenge in software engineering. This documentation, which captures the high level structure, components, and design principles, is crucial for a system's long-term maintenance and evolution. Effective Architecture Knowledge Management (AKM) aims to systematize the creation and preservation of this knowledge, enabling teams to build, scale, and adapt software effectively.

Despite its importance, the manual creation and maintenance of architectural documentation remain a significant bottleneck. This is especially true for crucial artifacts like Architecture Decision Records (ADRs), which capture the rationale behind significant design choices. The documentation process is often perceived as a secondary task, leading to records that are incomplete, outdated, or disconnected from the actual implementation. This knowledge gap introduces significant risks, making it difficult for teams to understand the system's design, onboard new developers, and make informed decisions during its evolution.

The recent advancements in Large Language Models (LLMs) present a compelling opportunity to automate the generation of architecture documentation by directly analyzing source code repositories, or other relevant documentation. However, a naive, single prompt approach, feeding an entire repository to an LLM is often ineffective. Such a method is constrained by practical limitations like context window size and struggles to comprehend the distributed and multi-faceted nature of architectural knowledge. This frequently results in outputs that are superficial, inaccurate, or lacking in essential context.

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted by ACM, provided that the copies are not made for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference '17, Washington, DC, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXX>

2025-10-18 07:15. Page 1 of 5.

To overcome these challenges, we propose a more sophisticated, agentic approach. Frameworks like AutoGen [9] from Microsoft and CrewAI¹ have demonstrated the power of multi-agent collaboration in various domains. Hence we move beyond a monolithic LLM call and introduce a structured, multi-agent system where autonomous agents collaborate to solve the complex problem of architecture recovery and documentation. Our framework is built upon four distinct types of agents, each responsible for a key stage of the process: information Extraction, relevant document Retrieval, artifact Generation, and iterative Validation. These specialized agents, each with specific roles and tools, are coordinated by a central orchestrator.

In this paper, we present the design and an instantiation of this agentic approach. In this instantiation our approach analyzes a code repository and produce a set of high-quality, context-aware ADRs. We validate our approach through a user study comparing our system against a baseline single-LLM method. The results demonstrate that our agentic system produces significantly more relevant, coherent, and complete documentation, establishing it as a promising and practical approach for automating architecture knowledge management.

2 Motivation and Overview to Agentic Approach

Large Language Models (LLMs) have recently shown remarkable capabilities in understanding and generating complex technical content, making them promising tools for automating software architecture documentation. By leveraging their ability to reason over code, configuration files, and natural language text, LLMs can assist in generating consistent, readable, and context-aware architectural artifacts. This opens new possibilities for maintaining up-to-date documentation, supporting Architecture Knowledge Management (AKM), and reducing the burden on developers.

However, directly applying LLMs to architecture documentation presents several challenges. Architectural knowledge is distributed and contextual, spanning multiple abstraction levels from high-level design principles to specific implementation details, which a single LLM prompt often struggles to capture. This is compounded by practical limitations, such as context windows that are too small for modern code repositories, forcing an incomplete analysis. Furthermore, without a structured process, an LLM may generate documentation that is only superficially correct, lacking a deep understanding of the underlying trade-offs and historical context. A single LLM call results in outputs that can be inconsistent and lack traceability, making it difficult to verify their reasoning. These limitations are particularly acute when attempting to record design

¹<https://github.com/crewAIInc/crewAI>

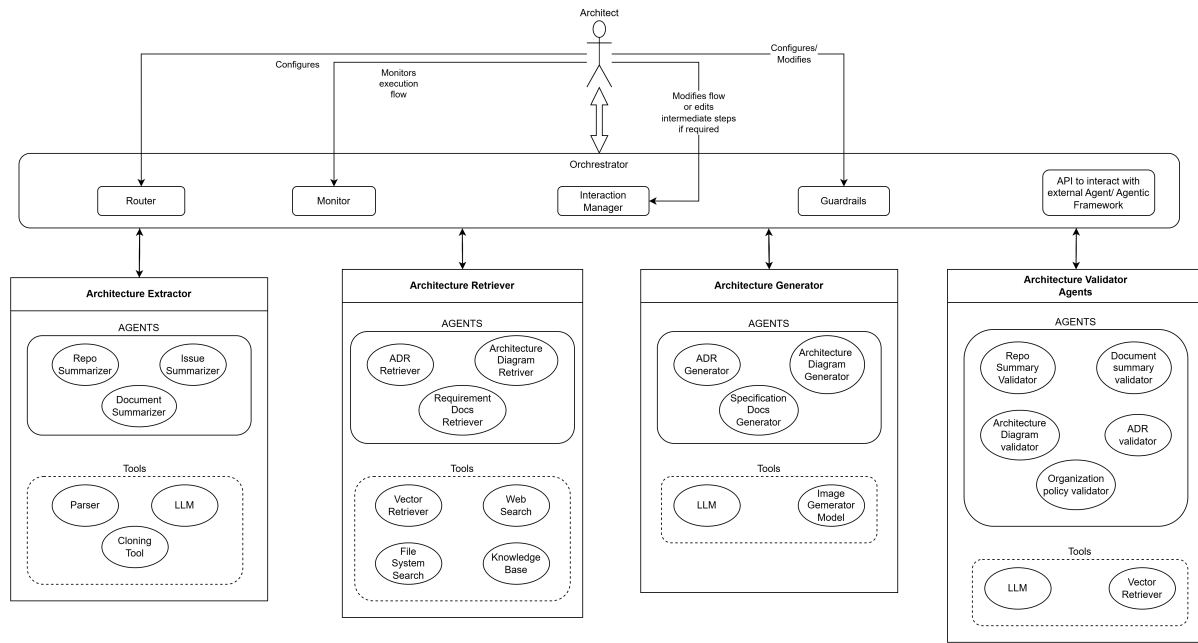


Figure 1: Agentic Approach

decisions, where precision, consistency, and the capture of design rationale are paramount.

To overcome these challenges, we propose an agentic approach. Instead of a single model, we employ a multi-agent system where the complex problem is decomposed into distinct, manageable sub-tasks. Each task is handled by a specialized AI agent with a specific role and toolset. These agents collaborate in a structured workflow, coordinated by a central Orchestrator, to produce accurate and context-aware documentation. This paradigm offers several key benefits:

Decomposition: It breaks down the monumental task of documenting an entire system into smaller, logical steps (e.g., summarize code, retrieve existing docs, generate new ADRs, validate output).

Specialization: It assigns each step to a specialized agent (e.g., a Repo Summarizer, an ADR Generator) that is optimized for that specific task.

Collaboration: Agents pass information and artifacts to one another, ensuring that each stage builds upon a validated foundation. For instance, the summary from the Architecture Extractor provides crucial context for the Architecture Generator.

Validation and Refinement: It incorporates dedicated validator agents that act as quality control checkpoints. This allows for an iterative refinement process, significantly improving the accuracy and reliability of the final output.

2.1 Sample Workflow in Action

To illustrate, consider a developer introducing a new message queue (e.g., RabbitMQ). The workflow begins with Extractor agents identifying this change, while Retriever agents confirm no existing ADR documents it, flagging it as a significant, undocumented decision.

The Orchestrator then tasks the ADR Generator to draft a record explaining the context and consequences of implementing RabbitMQ. This draft is immediately passed to Validator agents, which check it for technical accuracy against the code and for compliance with organizational policies. If the draft is approved, it is finalized; if not, it is returned to the generator with feedback for refinement, ensuring the final ADR is a robust and accurate document.

3 Agents

Our proposed system is a multi-agent framework coordinated by a central Orchestrator and supervised by a human Architect as shown in Figure 1. The Architect configures the initial workflow, monitors its execution, and can intervene to modify the flow if necessary. The Orchestrator manages the interaction between four specialized agent groups: Architecture Extractor, Architecture Retriever, Architecture Generator, and Architecture Validator. In the rest of the section, we describe some of the Agents. However its important to note, this paper doesn't exhaustively cover all possible agents in Agentic AKM, more agents can be added if required.

3.1 Architecture Extractor Agents

This group is responsible for parsing the code repository or other documentation, and creating high-level summaries that serve as the foundational architectural information for all other agents. Some of the agents can be:

Repo Summarizer: This agent performs a comprehensive analysis of the entire codebase to understand its overall architecture, primary components, and key functionalities. It uses an LLM tool to synthesize this analysis into a concise summary.

Issue Summarizer: This agent focuses on the project's issue tracker (e.g., GitHub Issues, Jira). It analyzes bug reports, feature requests, and developer discussions to extract architectural context and pain points, which can inform the ADR generation process.

Document Summarizer: This agent ingests existing documentation within the repository (e.g., READMEs, wikis) and uses an LLM to produce condensed summaries.

3.2 Architecture Retriever Agents

This group is tasked with finding and retrieving specific information from the project's internal knowledge base or external sources.

ADR Retriever: This agent specializes in searching for existing ADRs. It utilizes a Vector Retrieval tool to perform semantic searches over a vector database of past decisions, helping to avoid documenting the same decision multiple times.

Architecture Diagram Retriever: This agent searches for existing architectural diagrams within the project's documentation or file system.

Requirement Docs Retriever: This agent uses File System Search and Vector Retrieval to locate and pull information from requirements documents, ensuring that generated architectural decisions align with specified project goals.

3.3 Architecture Generator Agents

This group is responsible for creating new architectural artifacts based on the context provided by the Extractor and Retriever agents.

ADR Generator: This is the core generation agent. It uses an LLM tool to draft new ADRs based on the repository summary and retrieved information. Each ADR is structured with standard sections like Title, Context, Decision, and Consequences.

Diagram Generator: This agent creates new architectural diagrams (e.g., UML, C4 models) from scratch. It leverages a powerful Image Generator Model to visualize the architecture described in the repository summary or a specific ADR.

Specification Docs Generator: This agent drafts technical specification documents for new components or services, using an LLM to ensure the documentation is detailed, clear, and consistent with the established architecture.

3.4 Architecture Validator Agents

This group acts as a quality assurance layer, scrutinizing the generated artifacts for accuracy, coherence, and compliance with organizational standards.

Repo Summary Validator: This agent cross-references the summary created by the Extractor against the actual source code to ensure its accuracy and completeness. It uses an LLM to perform this comparative analysis.

Document Summary Validator: This agent checks the summaries of existing documents for factual correctness.

ADR Validator: This agent scrutinizes generated ADRs for logical consistency, format correctness, and overall quality. It uses a Vector Retrieval tool to compare the new ADR against existing ones to check for redundancy or contradiction.

Organization Policy Validator: This agent ensures that generated architectural document comply with organizational best

practices or predefined architectural principles stored in a knowledge base.

4 Experiments

To test the viability and effectiveness of our proposed approach, we made a instantiation of it with a multi-agent system to automate the creation of ADRs by dividing the task among specialized agents coordinated by a central Orchestrator.

4.1 Agentic ADR generation from repository

The workflow begins with a **Repository Summarizer Agent** analyzing the codebase to create a high-level summary. This summary is then validated by a **Summary Checker Agent**. If the summary is rejected, it is sent back to the Summarizer for refinement in a loop that runs up to three times.

Once the summary is approved, an **ADR Generator Agent** uses it to identify significant architectural decisions and draft corresponding ADRs. These drafts are scrutinized by an **ADR Checker Agent** for correctness and quality. Similar to the summarization step, this triggers a refinement loop with the generator for up to three iterations if the ADRs are rejected. Finally the approved ADRs are saved.

The Orchestrator Orchestrates the whole flow. It Orchestrates the iterative process and call the respective Agents when required. This agentic, iterative approach ensures that each stage builds upon a validated foundation, significantly improving the accuracy and relevance of the automatically generated architectural documentation.

4.2 Evaluation Setup

To evaluate the effectiveness of our proposed system, we conducted a comparative user study. We designed the experiment to assess the quality of ADRs generated by two distinct methodologies across two different LLMs. We compared two primary approaches for ADR generation:

Baseline Approach: This method involved extracting key files and components from a given repository and feeding this condensed information directly to an LLM in a single prompt to generate ADRs.

Agentic Approach: This is the multi-agent system detailed in Section 2, which uses a structured, iterative workflow involving summarizer, generator, and checker agents to produce the final ADRs.

For the underlying LLMs, we selected 'Gemini-2.5-pro' and 'gpt-5', which were the top-ranked models on the LmArena leaderboard at the time of our experiment (October 5th, 2025). This resulted in four distinct experimental configurations:

- Baseline with Gemini
- Baseline with GPT
- Agentic with Gemini
- Agentic with GPT

User Study Protocol We recruited participants and asked them to provide code repositories they were familiar with, resulting in a dataset of 20 unique repositories. For each repository, we generated four sets of ADRs, one from each of the four experimental configurations. The participants, who had expert knowledge of their

respective repositories, were then asked to evaluate the generated ADRs using a feedback form. The evaluation consisted of two parts:

Quantitative Ratings: Participants provided a star rating (from 1 to 5) for each set of ADRs based on five criteria: Relevance, Coherence, Completeness, Conciseness, and Overall Quality.

Qualitative Feedback: Participants also provided written comments detailing the strengths and weaknesses of the ADRs generated by each of the four configurations.

4.3 Results

The **quantitative** results clearly indicate that the Agentic framework outperformed the LLM-only approach across all evaluation metrics. Agent-based configurations (GPT-5 and Gemini) achieved higher scores in Relevance (4.5), Coherence (4.0–4.5), and Overall Quality (4.0–4.5), demonstrating their ability to produce more contextually aligned and well-structured ADRs. In comparison, the LLM-only configurations scored lower overall (3.0–3.5), particularly in Completeness (2.5–3.0), suggesting that they often omitted details or provided limited reasoning depth. The Agentic approach also maintained strong Conciseness (4.0–4.5), indicating an effective balance between brevity and richness. Overall, these findings reinforce that Agentic frameworks generate ADRs that are more relevant, coherent, and complete, providing consistently higher overall quality than single-step LLM generation.

The **qualitative** evaluation of the two approaches, baseline LLM call and the Agentic approach revealed distinct qualitative differences in both the structure and perceived usefulness of the generated ADRs. Users described the LLM generated ADRs as “accurate,” and “spot on,” yet often “too concise,” or “misleading.” While these configurations were praised for precision and readability, participants noted a tendency toward redundancy and insufficient exploration of the decision space, reflecting the limited contextual reasoning capacity of single LLM calls.

In contrast, ADRs generated via the Agentic framework were generally evaluated as more contextually grounded and complete. Users described these outputs as “Context is well written,” and “the best till now.” They highlighted that these configurations provided a broader and more nuanced understanding of architectural decisions, often distinguishing between positive and negative consequences. Nonetheless, some feedback indicated that the results were occasionally “too brief” or inconsistent in structure, reflecting a trade-off between completeness and conciseness.

Overall, the feedback indicates that while LLM-based methods deliver concise and correct ADRs, the Agentic framework provides superior contextual completeness and semantic coherence. These findings substantiate the hypothesis that agentic coordination enables more holistic architectural reasoning than isolated LLM inference, yielding outputs that are both decision-aware and contextually interpretable.

5 Related Works

Recent advances in GenAI have begun reshaping software architecture research. Esposito et al. [4] mapped the emerging use of LLMs in architectural reconstruction, documentation, and decision support, while Ivers and Ozkaya [5] examined which architectural activities are realistically automatable, stressing the enduring need

for human governance. Empirical studies such as Dhar et al. [2] explored LLMs’ ability to generate architectural decisions from context, revealing model- and prompt-dependent variability. Similarly, Manjula and Dube [7] demonstrated the use of LLMs for creating and interpreting architecture diagrams. Collectively, these efforts show that LLMs can support specific architecture-related tasks but lack integration into continuous, repository-aware workflows.

The agentic paradigm extends this line of work by coordinating multiple LLM agents to tackle complex engineering problems. Arora et al. [8] and Bouzenia et al. [1] analyzed distributed AI agents that decompose software development work into collaborative reasoning cycles, exposing both potential and coordination challenges.

ReArch. Díaz-Pace et al. [3] proposed ReArch, a reflective, LLM-based framework where autonomous agents explore design alternatives and reason about trade-offs. While innovative in design exploration, ReArch focuses on creating new architectural solutions rather than recovering existing ones. It does not operate on real repositories or preserve architectural rationale. In contrast, our approach applies agentic reasoning to extract and document architecture knowledge embedded within source code and project artifacts. MAAD. Li et al. [6] introduced MAAD (Multi-Agent Automated Architecture Design), a system where specialized agents collaborate to synthesize and assess new architectures. MAAD advances multi-agent collaboration for design generation, whereas our work centers on architecture extraction and documentation—automating how knowledge is identified, validated, and maintained from existing software systems.

Overall, prior studies in both LLM-assisted and agentic architectures mostly emphasize on design synthesis. Our work differs by embedding multi-agent reasoning into Architecture Knowledge Management (AKM), focusing on automated extraction and refinement of architectural artifacts—bridging generative AI and sustainable architecture documentation.

6 Conclusion and Future Works

Acknowledgments

Hiya et al in SA4S.

References

- [1] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. arXiv:2506.18824 [cs.SE] <https://arxiv.org/abs/2506.18824>
- [2] Rudra Dhar, Karthik Vaidhyanathan, and Vasudeva Varma. 2024. Can LLMs Generate Architectural Design Decisions? -An Exploratory Empirical study. arXiv:2403.01709 [cs.SE] <https://arxiv.org/abs/2403.01709>
- [3] J. Andrés Díaz-Pace, Antonela Tommasel, Rafael Capilla, and Yamid E. Ramírez. 2025. Architecture Exploration and Reflection Meet LLM-based Agents. In *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*. 1–5. doi:10.1109/ICSA-C65153.2025.00015
- [4] Matteo Esposito, Xiaozhou Li, Sergio Moreschini, Noman Ahmad, Tomas Cerny, Karthik Vaidhyanathan, Valentina Lenarduzzi, and Davide Taibi. 2025. Generative AI for Software Architecture. Applications, Challenges, and Future Directions. arXiv:2503.13310 [cs.SE] <https://arxiv.org/abs/2503.13310>
- [5] James Ivers and Ipek Ozkaya. 2025. Will Generative AI Fill the Automation Gap in Software Architecting?. In *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*. 41–45. doi:10.1109/ICSA-C65153.2025.00014
- [6] Ruiyin Li, Yiran Zhang, Xiyu Zhou, Peng Liang, Weisong Sun, Jifeng Xuan, Zhi Jin, and Yang Liu. 2025. MAAD: Automate Software Architecture Design through Knowledge-Driven Multi-Agent Collaboration. arXiv:2507.21382 [cs.SE] <https://arxiv.org/abs/2507.21382>

Source	Model	Relevance	Coherence	Completeness	Conciseness	Overall
Agent	GPT-5	3.8	3.8	3.6	3.6	3.2
Agent	Gemini	4.3	4.1	4.0	4.2	4.0
LLM	GPT-5	4.1	3.8	3.6	3.7	3.2
LLM	Gemini	4.4	4.0	3.4	3.5	3.6

Table 1: User study results comparing Agentic vs. Baseline (LLM) approaches across two models. Scores are averaged over 20 repositories on a 5-point scale.

[7] Nishchai Manjula and Akhilesh Dube. 2024. Harnessing generative AI to create and understand architecture diagrams. *International Journal of Science and Research Archive* 13 (12 2024), 3330–3336. doi:10.30574/ijrsra.2024.13.2.2601

[8] Nalin Wadhwa, Atharv Sonwane, Daman Arora, Abhav Mehrotra, Saiteja Utpala, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. MASAI: Modular Architecture for Software-engineering AI Agents. In *NeurIPS 2024 Workshop on Open-World Agents*. <https://openreview.net/forum?id=NSINt8LLYB>

[9] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI] <https://arxiv.org/abs/2308.08155>