iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** IV **Month of publication:** April 2025

DOI: https://doi.org/10.22214/ijraset.2025.69166

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue IV Apr 2025- Available at www.ijraset.com

AI-Forge - Your Coding Assistant

Malik Mohammed Ali[1], Sachin Pramod Mishra[2], Nikunj Hemraj Bhanushali[3], Pranav Vitthal Salunkhe[4], Prof. Mahendra Patil[5]
Department of Computer Engineering Atharva College of Engineering, Mumbai, India

Abstract: ***The need for quicker development cycles, better teamwork, and shorter technical onboarding times has increased in the modern era of software innovation, especially in huge codebases and enterprise-grade software systems. When growing modular architectures, maintaining old systems, or integrating new developers, traditional development pipelines frequently run into problems. A recurring barrier to guaranteeing code understanding, consistency, and maintainability is the enormous complexity of these systems, which is exacerbated by poor documentation and the dispersed structure of development teams. The problem is made worse by the lack of context-aware, real-time documentation, which raises deployment mistake rates, reduces productivity, and increases technical debt. In response to this problem, Ai-Forge stands out as a game- changing solution, providing an ecosystem powered by AI that revolutionizes the way developers engage with codebases. Ai-Forge uses sophisticated natural language processing (NLP), retrieval-augmented generation (RAG), and real-time embedding architectures to enable intelligent understanding, querying, and visualization of software systems, in contrast to traditional documentation tools that function statically or necessitate human intervention. By making sure that every line of code is self-explanatory, current, and dynamically interpretable, this innovation aims to reduce the knowledge imbalance among development teams and transform the software engineering workflow. The innovative multi-agent framework at the heart of Ai- Forge combines cutting-edge large language models (LLMs) for automated documentation production with real-time event-driven triggers. The system uses cloud- based features to identify code contributions and start a series of automated procedures, mostly through GitHub webhooks and AWS Lambda. Specialized agents are assigned specific tasks by this distributed architecture: documentation synthesis agents create descriptive, readable content that reflects the current state of the codebase, code analysis agents segment and interpret the code's semantics, and event detection agents record changes as they happen. Every agent functions within a meticulously crafted communication protocol that draws inspiration from conversational paradigms present in contemporary multi-agent frameworks. The agents collaboratively improve their outputs through repeated multi-turn dialogues, guaranteeing that the final documentation captures the code's underlying functional and contextual subtleties in addition to its syntactic structure. Even with changing project complexity and massive codebases, Ai-Forge is able to continuously adapt and develop due to the dynamic interaction between agents.***

Keywords: Multi-Agent, RAG, LLM

I. INTRODUCTION

Large language models (LLMs) have precipitated transformative shifts in software engineering by seamlessly integrating extensive knowledge expressed in natural language ([6] [07] [33] [21] [12]). For example, frameworks like ChatDev demonstrate how LLM- powered agents can engage in multi-turn dialogues, collaboratively navigating the design, coding, and testing phases to yield comprehensive software solutions.

Software development itself is inherently complex, necessitating coordinated efforts among experts such as architects, programmers, and testers ([04] ; [40]). This process involves extensive natural language communication to capture and refine requirements, alongside the use of programming languages to implement, debug, and optimize code ([16]; [03]). Despite numerous studies applying deep learning to individual phases of the traditional waterfall model—such as design, coding, and testing ([34] ; [23] ; [19] ; [39])—the methods remain fragmented. This fragmentation results in technical inconsistencies across phases, ultimately leading to a disjointed and less efficient development process [16] ; [36]).

In response to these challenges, emerging multi-agent frameworks such as Self-Organized Agents and Cross-Team Collaboration have paved the way for scalable solutions that distribute complex tasks among specialized agents. Similarly, Experiential Co-Learning and AutoGen further demonstrate the benefits of agent-based systems by incorporating historical experience and flexible conversation programming. These advancements provide the foundation for AI-Forge, which aims to integrate these concepts into a real-time, automated documentation framework.

© IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

A lack of coherence throughout the development lifecycle still hinders modern software engineering, despite notable improvements in programming paradigms and tooling. Intelligent code recommendations are provided by tools such as GitHub Copilot and TabNine, although they usually operate in discrete situations and lack a comprehensive grasp of the system architecture or collaboration needs. Developers frequently have to manually manage communication bottlenecks, synchronize documentation, troubleshoot unknown code, and coordinate activities across teams— all of which impede creativity and productivity. This broken environment serves as the inspiration for AI- Forge. AI-Forge hopes to provide an LLM-driven multi- agent environment where activities like requirement analysis, code implementation, documentation creation, and testing are not isolated but rather coordinated among autonomous agents, taking inspiration from frameworks like ChatDev and AutoGen.

Like its ChatDev predecessors, each agent in AI-Forge takes on a predetermined social or technical role (such as planner, coder, tester, or reviewer), enabling a collaborative loop that emulates actual development teams. Furthermore, through promoting alignment, self- improvement, and division of labour, cross-role contact and experience learning improve the performance of agent-based frameworks, as noted in Cross-Team Collaboration and SDE Agents. AI-Forge expands on existing frameworks by prioritizing knowledge synchronization and real-time interaction. AI-Forge constantly learns from code modifications, debugging sessions, and user input to modify its tactics dynamically, in contrast to conventional tools that respond to static instructions.

Despite the advancements in multi-agent systems and LLM-driven frameworks, significant challenges remain in achieving a fully integrated, real-time documentation system. The primary problem centres on the persistent gap between rapidly evolving codebases and static, manually updated documentation. This gap leads to outdated technical information, increased onboarding difficulties, and elevated maintenance costs.

Specifically, existing methods suffer from:

1) Synchronization Delays: The time lag between code changes and documentation updates creates inconsistencies that can mislead developers and propagate errors throughout the development lifecycle. Traditional systems typically operate in a batch mode, which fails to capture the dynamic nature of modern software projects.

2) Contextual Misalignment: Automated documentation tools often lack the deep semantic understanding required to capture the nuances of complex code interdependencies. Although LLMs have demonstrated significant potential in generating text, they can struggle to maintain coherent context across large, modular codebases.

3) Fragmented Workflows: Many current approaches treat documentation generation as an isolated task, separate from core activities such as code analysis, testing, and debugging. This siloed approach leads to technical inconsistencies and disrupts the natural flow of development processes.

4) Scalability Constraints: Single-agent frameworks are limited by their context length and computational resources, making them less effective for large-scale projects. Distributed multi-agent architectures, as seen in frameworks like Self-Organized Agents and AutoGen, offer scalability; however, their application in real- time documentation remains underexplored.

The research gap, therefore, lies in the absence of a comprehensive, multi-agent framework that can seamlessly synchronize and generate documentation in real time, ensuring that every code modification is accurately reflected in the project's knowledge base. AI- Forge is designed to fill this gap by leveraging continuous event-driven triggers, distributed semantic analysis, and iterative LLM-based dialogue among specialized agents.

II. LITERATURE REVIEW

Trained on vast datasets to comprehend and manipulate billions of parameters, LLMs have be come pivotal in natural language processing due to their seamless integration of extensive knowledge ([06]; [07]; [37]; [32]; [31]; [30]; [11]; [05]; [11]; [32]; [39]; [36]; [06]).

Further more, LLMs have demonstrated strong role- playing abilities ([24]; [33]; [21]; [09]; [30]; [12]; [13]; [25]). Recent progress, particularly in the field of autonomous agents ([24]; [29]; [33]; [27] ; [38]; [31]; [30]), is largely attributed to the foundational advances in LLMs. These agents utilize the robust capabilities of LLMs, displaying remarkable skills in memory ([33]; [39]), planning (12; 26) and tool use ([18]; [08]; [35]; Ruan et al., 2023; [39]), enabling them to reason in complex scenarios Software development is a multifaceted and intricate process that requires the cooperation of multiple experts from various fields ([14]; [01]; [04]; [40]; [03]; [18]), encompassing the requirement analysis and system design in natural languages ([34]; [28]; [29]), along with system development and debugging in programming languages ([19]; [08] ; [14]). Numerous studies employ the waterfall model, a particular software development life cycle, to segment the process into discrete phases (e.g., design, coding, testing) and apply deep learning to improve the effectiveness of certain phases ([18]; [21]; [24]; [22]; [26]; [36]).

III. WHAT IS AI-FORGE

AI-Forge adopts a multi-agent architecture that mirrors the design principles of ChatDev, with a focus on automating documentation rather than solely generating code. The framework organizes a collection of specialized agents into a chat-chain, enabling them to communicate and collaborate seamlessly throughout the documentation process. The key architectural components of AI-Forge include:

1) Event Detection and Code Monitoring

AI-Forge continuously monitors changes in the codebase by interfacing directly with the version control system. Instead of relying on external cloud triggers, it utilizes native hooks and internal mechanisms to detect code updates as they occur. When a change is detected, the system immediately initiates the documentation process.

2) Code Analysis Agents

Specialized agents are responsible for parsing and semantically analyzing the updated code. These agents apply text chunking and context extraction techniques to decompose complex code structures into manageable segments. The extracted context forms the basis for generating accurate, detailed documentation.

3) Documentation Synthesis Agents

Leveraging advanced LLMs fine-tuned on technical and documentation-specific corpora, these agents generate natural language descriptions that precisely capture the functionality, design decisions, and usage instructions of the code. The synthesis process is iterative, with agents engaging in multi-turn dialogues to refine and clarify the documentation output.

4) Inter-Agent Communication via Chat Chain

Inspired by ChatDev, AI-Forge employs a chat- chain mechanism where agents interact in a structured sequence. The chain divides the documentation process into distinct phases (e.g., code analysis, synthesis, and review), and each phase is further broken down into subtasks. During these conversations, agents exchange contextual information and feedback to ensure that the documentation remains both comprehensive and aligned with the evolving codebase.

5) Iterative Refinement and Self-Correction

A crucial feature of AI-Forge is its ability to continuously refine documentation through iterative feedback loops. Agents monitor the consistency and completeness of the generated content, requesting additional context or clarification when ambiguities arise. This communicative dehallucination mechanism, inspired by ChatDev, minimizes errors and ensures that the final documentation is accurate and easily interpretable.

6) Seamless Integration with Development Workflows:

While AI-Forge does not rely on external cloud services, it is designed to integrate naturally within existing development environments. By embedding directly into the development pipeline, AI-Forge ensures that documentation updates occur concurrently with code changes, maintaining a synchronized and cohesive project record. Through these components, AI-Forge transforms the traditionally static documentation process into an agile, dynamic function—one that continuously evolves in tandem with the code. This architecture not only enhances documentation quality but also significantly reduces the manual effort required by developers, thereby fostering a more efficient and error-resilient development workflow.

A. Operational Workflow and Agent Interaction Dynamics

AI-Forge, the operational workflow is designed around a dynamic, iterative dialogue among specialized agents, closely mirroring the principles established in ChatDev. This process is divided into several phases, each representing a key stage in transforming raw code changes into coherent, context-rich documentation.

1) Multi-Phase Workflow

The documentation process is segmented into discrete phases—each dedicated to a specific task. Initially, the system detects changes within the codebase and triggers the code analysis phase, where agents dissect the updated code into manageable segments. This is followed by a synthesis phase in which agents collaboratively generate preliminary documentation, and finally, a review phase where iterative feedback ensures clarity and consistency.

© IJRASET: All Rights are Reserved |

2) Multi-Turn Dialogue

During each phase, agents engage in multi-turn dialogues. An instructor-like agent initiates the conversation by framing the task (e.g., summarizing a code module), while an assistant-like agent responds with its proposed documentation. This back-and-forth continues iteratively. The agents employ a mechanism akin to communicative dehallucination—where the assistant actively seeks clarification or additional context when ambiguities arise. This repeated exchange ensures that the generated documentation is refined and aligned with the actual code semantics.

3) Context Sharing and Memory

To maintain continuity across phases, AI-Forge implements both short-term and long-term memory systems. In the short term, each phase's dialogue is stored and used to inform subsequent interactions. For longer projects, selected key outputs are preserved as long-term memory, ensuring that critical context from earlier phases influences later documentation updates. This hierarchical memory structure allows agents to maintain focus on the current task while retaining relevant historical insights.

4) Agent Interaction Dynamics

The interaction among agents is characterized by a role-specific communication protocol. Each agent is pre-configured with a system prompt that defines its role and behavioral guidelines. When a code update is detected, the relevant analysis agents interpret the change and pass contextual data to the synthesis agents. These synthesis agents, in turn, use the context to generate descriptive documentation. If inconsistencies or gaps are identified during the review phase, the process loops back— prompting further dialogue and adjustment until consensus is reached. This decentralized interaction model ensures that no single agent is overwhelmed by the entire task, promoting a modular, resilient system.

5) Iterative Refinement and Error Correction

The system's iterative nature facilitates self- correction. As agents exchange messages, they evaluate the quality of generated documentation against the code's functional and contextual attributes. Should an agent identify a discrepancy or vague area, it requests additional details from its counterpart, prompting a targeted re-evaluation. This continuous feedback loop minimizes errors—such as outdated or ambiguous descriptions—and gradually converges on a final, polished document that accurately mirrors the codebase. Through this operational workflow and dynamic agent interaction, AI-Forge transforms the static task of documentation into an adaptive, collaborative process. The system not only automates the generation of documentation but also ensures that it evolves fluidly in tandem with code changes, thereby enhancing overall project maintainability and reducing manual effort.

B. Chat Chain

A structured, multi-turn dialogue technique called ChatChain arranges agent interactions into a process that is sequential and chain-like. This idea, which is essential to frameworks like ChatDev, is modified in AI-Forge to organize the creation of precise, real-time documentation. Fundamentally, the ChatChain separates the entire process into discrete stages (e.g., planning, coding, testing), whether it is code synthesis, debugging, or documentation development. Every stage is further divided into more manageable, smaller subtasks. Agents take on specific duties within each subtask, usually as an assistant and an instructor. The assistant agent answers with a solution or a fragment of documentation, while the instructor agent starts the conversation by giving instructions or background. Until a consensus on that specific subtask is obtained, this back and forth keeps happening in several rounds.

The ChatChain's iterative nature is one of its key characteristics. The results of each subtask act as building blocks that guide the phase that follows. To put it another way, the solution produced during one subtask is incorporated into the context or "memory" that is transmitted, guaranteeing that any agents that come after it operate with the most recent and improved data. This process improves the overall coherence and uniformity of the finished product in addition to facilitating a smooth information flow across phases. A dual-memory system is also incorporated into the ChatChain. In order to maintain continuity and context for that subtask, short-term memory records the current discourse within a phase. However, when tasks get more complicated, the system can maintain a permanent context because long-term memory retains important decisions and outputs across phases. To manage the iterative improvements needed in large- scale projects, this tiered memory technique is crucial. All things considered, the ChatChain turns difficult jobs into a sequence of co-op, iterative conversations. The method lowers the possibility of mistakes—like omissions or inconsistencies—and enables focused improvements by organizing the communication in this manner.

© IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

This approach allows agents to provide outputs that are both contextually precise and thorough, which is especially useful in situations where information accuracy—whether it be code or documentation—is crucial. The ChatChain (CC) orchestrates collaborative software development through a structured workflow of sequential phases (PP), each decomposed into interactive subtasks (TT). This framework enables iterative refinement of solutions via dialogues between specialized agents, ensuring alignment between requirements and executable code. Below, we formalize its components and operational dynamics.

1) Agent Roles and Dialogue States

Let

$A = \{a_1, a_2, ..., a_n\}$ be the set of agents, where each

agent ai is assigned a role $r_i \in R$ (e.g., *Instructor*, *Assistant*, *Reviewer*, etc.).

A dialogue is modeled as an ordered sequence of messages:

$$D = \{(a_i, m_i, t_i)\}_{i=1}^{T}$$

Where:

- m_i is the message generated by agent a_i,
- t_i represents the timestamp or dialogue turn index,
- T is the total number of dialogue steps.

Each agent's generation at step iii depends on the interaction history:

$$H_i = \{(a_j, m_j, t_j)\}_{j=1}^{i-1}$$

This sequence forms the contextual basis upon which decisions and content generation are made by the agents.

2) Subtask Decomposition and Objective Optimization

The overall task GGG, which may include code generation, testing, documentation, or debugging, is decomposed into modular subtasks:

$$G = \{G_1, G_2, ..., G_K\}$$

Each subtask is processed as a functional unit with independent evaluation and feedback.

To optimize the outcome of each subtask, we define an objective function based on minimizing a contextual loss fn:

$$G_k^* = \arg\min_{o_k} \mathcal{L}(o_k, c_k)$$

where:

- o_k is the output generated by the agent(s) responsible for subtask G_k,
- c_k denotes the input context or prompt condition (e.g., project requirements, API descriptions),
- $\mathcal{L}$ is a domain-specific evaluation metric or loss function, such as BLEU for text, or test coverage for code.

This approach introduces **task granularity** and supports iterative refinement, making it conducive to multi-agent pipeline operations.

3) Communication Policy and Memory Embedding

Each agent utilizes a policy function π_k, governed by its internal language model and memory context:

$$\pi_k : H_i \rightarrow m_i$$

This can be interpreted as the function mapping history H_i to the next message m_i using the role semantics of agent a_i

The communication history is embedded into a high- dimensional vector space using an encoder function:

$$M_t = f_{\text{embed}}(H_t)$$

where f_{embed}could represent any dense vectorizer such as a transformer encoder, sentence embedding model (e.g., Voyage AI), or custom LLM embedding.

Subsequently, the generation at any turn is represented as:
Here, each agent considers its role r_i and embedded memory state M_{t_i} to produce task-specific dialogue contributions.

$$m_i = \underset{\text{LLM}_i}{\text{LLM}_i}(M_{t_i}, r_i)$$

4) Output Aggregation and Convergence
The outputs from all subtasks are synthesized into the final software artifact:

$$O = \bigoplus_{k=1}^{K} o_k$$

Where $\oplus$ denotes semantic concatenation or structured integration, depending on the task (e.g., appending documentation or integrating test cases into a main codebase).
Convergence for each task is ensured through a delta- based threshold mechanism:

$$\|\mathcal{L}(o_k^{(t+1)}, c_k) - \mathcal{L}(o_k^{(t)}, c_k)\| < \epsilon$$

This expression guarantees that once the improvement between subsequent outputs falls below a small margin ϵ, the task is considered converged. This setup enables AI-Forge's ChatChain to operate as a distributed intelligent system, where agents can independently handle subproblems, evaluate outcomes, and collaborate through structured, role-aware communication to complete complex software development workflows.

C. Communicative Dehallucination in AI-Forge

A central challenge in language model-based systems is the issue of hallucination—the generation of plausible- sounding but factually incorrect or semantically inconsistent outputs. This becomes particularly problematic in technical domains such as software documentation, where precision, consistency, and adherence to actual code logic are imperative. AI-Forge adopts and extends the Communicative Dehallucination strategy pioneered in ChatDev, wherein agents correct each other's outputs through structured, role-based dialogue. Rather than relying on external validation datasets or static correctness checks, AI-Forge embeds an interactive quality assurance loop within its agent communication system.
In the context of AI-Forge, hallucination often manifests as incorrect function descriptions (e.g., claiming a function returns a string when it actually returns an object), misrepresented parameter types or naming, fabricated logic paths or assumptions not present in the code, and ambiguities in procedural steps—especially in auto-generated API documentation. These errors typically originate from large language models generalizing from patterns in their pretraining data, incomplete or insufficient context during message generation, or accumulated semantic drift in multi-turn agent exchanges.
To mitigate these issues, AI-Forge initiates a structured dialogue-based correction protocol, where agents iteratively verify and refine each other's outputs. Let O_k be the intermediate documentation output at iteration k, R be the set of roles, and C_k represent the set of extracted claims or entities from the output. Each agent takes turns performing one of the following steps:

1) **Claim extraction** – identifying critical assertions in the documentation.
2) **Verification** – matching each claim against actual code logic using parsing or static analysis.
3) **Revision** – updating or removing unverifiable or hallucinated claims based on mismatches.

This loop is repeated until the delta in hallucination rate **h** drops below a tolerance threshold **ϵh**, ensuring progressively cleaner and more accurate documentation.

Example: Real-Time Dialogue Sequence

Role	Message Example
Scribe	"The function processOrder() logs order details to a database."
Reviewer	"Clarify: does the code actually contain any DB connection or logging API?"
Scribe	"Correction: processOrder() only appends to a local list, no DB operations present."
Verifier	"Confirmed. No DB-related imports or calls exist in this module."

This mechanism ensures that agents do not operate in isolation but instead rely on critical peer feedback— much like human code reviews—to eliminate errors stemming from LLM overconfidence or contextual ambiguity.

The benefits of this Communicative Dehallucination strategy are significant. First, it allows for self- correction without relying on external ground truth, enabling the system to refine outputs using internal reasoning and code verification. Second, it ensures multi-perspective consistency, as multiple roles scrutinize the output from functional, semantic, and syntactic standpoints. Finally, it leads to improved factual accuracy, with preliminary evaluations showing a 30–50% reduction in documentation errors (refer to Section 8 for detailed metrics).

D. Memory Architecture in AI-Forge Role and Design of Memory Systems

Effective memory design is essential for multi-agent collaboration, especially in software engineering tasks that involve iterative understanding and revision. Inspired by the memory hierarchy introduced in ChatDev, AI-Forge adopts a dual-memory architecture—comprising both Short-Term Memory (STM) and Long-Term Memory (LTM)—to support contextual grounding, historical awareness, and output consistency across agent interactions.

1) Motivation for Memory Integration

Unlike single-turn inference systems, AI-Forge agents participate in multi-turn, multi-role dialogues that span across various subtasks, such as documentation synthesis, refinement, and verification. Without memory, these agents would lack the ability to:

- Recall prior agent outputs,
- Understand evolving context,
- Maintain continuity across project modules,
- Avoid contradiction and redundancy in documentation.

Thus, a formalized memory model is crucial to both **inter-agent communication** and **longitudinal coherence** across evolving codebases.

2) Short-Term Memory (STM)

Short-Term Memory is defined as the window of local dialogue history relevant to the current subtask or ChatChain phase.

Let:

$$\mathrm{STM}_t = \{(a_i, m_i)\}_{i=t-w}^{t-1}$$

where ***w*** is the size of the memory window, and each m_i represents a message exchanged during that time frame.

STM is:

- Continuously updated with each new turn.
- Pruned using recency and relevance heuristics.
- Lightweight, enabling quick token-level referencing in LLM prompts.

STM supports:

- Context preservation during multi-turn exchanges,
- Real-time reference of recently discussed design patterns or terminology,
- Local dehallucination and self-consistency checks.

3) Long-Term Memory (LTM)

Long-Term Memory is used to persist global insights, decisions, design principles, and validated outputs that must remain accessible across different ChatChains or project sessions.

It is modeled as a knowledge graph or vector memory indexed by semantic similarity:

$$\mathrm{LTM} = \{(k_i, v_i)\}_{i=1}^{N}$$

where:

- *ki* is a high-dimensional embedding of a concept, code segment, or instruction,
- *vi* is the associated output, commentary, or validated documentation.

© IJRASET: All Rights are Reserved |

LTM serves three key purposes:

- Reinforcement of global architectural knowledge (e.g., project-wide data flows),
- Reuse of earlier documentation patterns or verified responses,
- Interoperability across modules (e.g., ensuring that function docs reflect shared models and APIs).

4) Memory Encoding and Representation

To ensure efficient retrieval and high semantic fidelity, both STM and LTM contents are embedded into high- dimensional vector spaces. This embedding enables similarity search and prompt optimization for agents during generation.

- STM Embedding

  For a message window *STMt* , the embedding is computed as:

  $E_t^{\text{STM}} = f_{\text{embed}}\left(\{m_i\}_{i=t-w}^{t-1}\right)$

  where *fembed* is a transformer-based encoder that compresses recent agent messages into a dense vector used for prompt injection.
- LTM Embedding & Indexing

  LTM content is stored as key–value pairs:

  $$\text{LTM} = \{(k_i, v_i)\}_{i=1}^{N}$$

  where *ki=fembed(ci)ki* is the embedding of a code concept *ci*, and *vi* is the corresponding verified documentation, instruction, or outcome. These are indexed using a vector search engine (e.g., FAISS).

5) Retrieval Mechanism

To maintain high response quality and contextual accuracy, AI-Forge retrieves relevant information from memory using similarity matching.

Given a current code segment or user query qqq, the system performs:

$$R_q = \{v_i \mid \text{cosine}(f_{\text{embed}}(q), k_i) > \tau\}$$

where τ is a threshold controlling relevance. Retrieved entries *Rq* are appended to the agent's prompt context.

Retrieval logic supports:

Dehallucination by verifying if a fact has appeared in validated history. Context expansion by linking related but not directly adjacent code components. Reinforcement learning by favoring results that led to fewer corrections in past dialogues.

6) Memory Pruning and Ranking

To ensure memory efficiency, AI-Forge implements memory pruning through relevance ranking and decay strategies:

- **STM Pruning:**

  Uses recency weighting:

  $$\text{score}_{\text{STM}}(m_i) = \alpha \cdot \text{Recency}(m_i) + \beta \cdot \text{Relevance}(m_i, q)$$

  where α,β are weights.
- **LTM Retention:**

  Entries in LTM are decayed using a time-decay or overwrite policy, but high-frequency accessed keys are preserved using:

  $$\Delta t_i < \delta \Rightarrow \text{retain}(k_i)$$

This design allows AI-Forge to maintain an adaptive memory model that grows with the codebase while maintaining high precision in documentation, feedback loops, and historical traceability. With this, AI-Forge's memory system acts as a cognitive substrate—enabling agents to behave not as isolated responders but as intelligent collaborators with memory, judgment, and refinement capability across time.

IV. EVALUATION FRAMEWORK

A. *Detailed Evaluation Objectives*

Assessing AI-Forge's efficacy as an automated documentation system in the context of dynamic software development is the main goal of the evaluation.

1 Documentation

This dimension focuses on AI-Forge's ability to provide documentation that is accurate, comprehensive, and understandable while also taking into account how modern codebases are changing at every interval due to dynamics.

Factual accuracy is a key sub-objective under documentation quality. The documentation generated by the system must accurately reflect the semantics and functionality of the underlying code. This covers parameter specifics, intended results, and detailed explanations of function behavior. Developers may be misled by inaccurate documentation, which can also add needless complexity when debugging.

Completeness, or the extent to which AI-Forge captures all essential elements of the source code, is another crucial criterion. Incomplete documentation risks creating knowledge gaps that impair maintainability, cooperation, and onboarding processes for new developers. As a result, the system needs to guarantee thorough coverage of all pertinent codebaseelements. The created content's readability and clarity are equally crucial. Documentation needs to be clearly comprehensible and grammatically correct in addition to being technically correct. This entails assessing the content's suitability for both inexperienced and seasoned developers by looking at the vocabulary's appropriateness, sentence structures' clarity, and usage of industry-standard terminology. Lastly, the evaluation also considers Contextual Consistency. The documentation should not only make sense within the boundaries of a single module but should also maintain semantic and stylistic coherence across interconnected components. By preserving consistent representation of architectural patterns and design decisions, AI-Forge contributes to a unified understanding of the entire system, improving collaboration and long-term codebase stability.

B. Collaborative Agent Performance

A fundamental component of evaluating AI-Forge lies in assessing the efficacy of its multi-agent architecture, particularly the degree to which the constituent agents interact and collaborate to generate high-quality documentation. The first dimension of this evaluation centres on the convergence of multi-turn dialogue, which aims to determine whether iterative inter-agent communication results in the progressive refinement of documentation outputs. This involves a detailed analysis of dialogue trajectories, examining the consistency, coherence, and convergence behavior over successive turns.

A second focal point is role-specific effectiveness. Within AI-Forge, agents are assigned specialized responsibilities—such as code analysis, synthesis, and verification. The evaluation framework measures the individual contributions of these agents to ascertain whether their respective tasks are executed with precision and whether the specialization contributes to the overall enhancement of documentation quality. This aspect ensures that the system's design, based on a division of cognitive labor, yields measurable benefits in terms of performance. The efficiency of inter-agent communication is also a critical parameter. Here, the study examines both the structural design and operational fluidity of the underlying communication protocol, such as ChatChain. Key metrics include dialogue latency, semantic clarity, and logical progression. Of particular interest is the effectiveness of communicative dehallucination mechanisms—structured clarification prompts that significantly reduce the likelihood of hallucinated or inconsistent outputs. These mechanisms are integral to enhancing inter-agent alignment and trustworthiness of the final documentation. Additionally, the system's capacity for error reduction and self-correction is rigorously evaluated. This involves quantifying the rate at which hallucinated or semantically inconsistent elements are identified and corrected through internal feedback loops. The ability of AI-Forge to autonomously refine its outputs in response to peer evaluation is a strong indicator of both architectural robustness and cognitive resilience in agent interactions.

C. System Responsiveness and Integration

Beyond internal agent collaboration, the responsiveness and integrability of AI-Forge within practical development environments constitutes another major axis of evaluation. The first metric under this domain is real-time update latency, defined as the time interval between a codebase modification (e.g., a commit) and the corresponding documentation update. Low latency is a critical requirement in agile and continuous integration workflows, wherein developers rely on up-to-date documentation for accurate

© IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

implementation and review. The system's scalability under varying loads is also assessed. As software systems increase in size and complexity, AI-Forge must maintain performance without a commensurate increase in processing time or degradation in documentation quality. This aspect of the evaluation includes stress-testing the multi-agent system under simulated high-load conditions to determine the system's ability to handle large-scale, concurrent operations effectively. Equally important is seamless integration with development pipelines. AI-Forge is designed to function as an embedded component within modern software development ecosystems. The evaluation examines its compatibility with version control systems (e.g., Git), continuous integration/continuous deployment (CI/CD) pipelines, and developer toolchains. A high degree of integration fidelity is essential to ensure that the system augments, rather than disrupts, existing workflows.

Finally, user satisfaction serves as a qualitative yet essential evaluative criterion. Through structured surveys and direct developer feedback, the system's impact on reducing manual documentation overhead, enhancing clarity, and improving developer productivity is analyzed. This user- centric perspective offers valuable insight into the practical benefits and limitations of the system, reinforcing the importance of usability and developer experience in the adoption of intelligent documentation tools.

D. Detailed Dataset and Project Selection

To conduct a rigorous and comprehensive evaluation of AI- Forge, a curated selection of datasets and real-world open- source projects is employed. These projects are deliberately chosen to reflect the complexity, diversity, and dynamism characteristic of contemporary software development environments. The selection process is guided by several critical factors, including domain variability, codebase complexity, development frequency, and the availability of high-quality, human-authored documentation that serves as a benchmark for assessing the system's output.

1) Dataset Criteria

The dataset selection adheres to the following dimensions. First, domain diversity is prioritized to ensure that AI-Forge is tested across a wide range of technical contexts. This includes domains such as web development, machine learning, frontend user interface engineering, and developer tooling. This diversity enables a nuanced evaluation of the system's ability to adapt to varying documentation conventions and semantic structures. Second, project complexity is considered by incorporating software systems of different scales, measured in terms of lines of code (LOC) and modular granularity. This allows for an examination of AI-Forge's scalability and its performance across both compact and expansive codebases. Third, the criterion of active development is employed to emulate real-time coding environments. Projects with frequent commits are selected to simulate the dynamic nature of modern software development, providing a suitable context to assess AI-Forge's responsiveness and update efficiency.

Lastly, wherever feasible, projects with available ground truth documentation—such as well-maintained docstrings or community-authored guides—are included. These serve as critical baselines for benchmarking the factual accuracy, completeness, and semantic alignment of AI-Forge- generated documentation using established metrics.

2) Selected Projects

The following table summarizes the representative projects selected for evaluation, detailing their domains, approximate sizes, modular breakdowns, and the rationale for inclusion:

Project	Domain	Approx. LOC	Modules	Rationale
Flask	Web Framework	3.2K	12	Minimalistic design; provides a baseline for evaluating documentation clarity.
scikit- learn	Machine Learning Library	25K	37	Complex, widely adopted; tests AiForge's ability to handle intricate systems.
Vue.js	Frontend UI	29K	45	Dynamic, evolving codebase; evaluates adaptability to frequent updates.
LangChain	LLM Utilities	10K	22	Relevant to LLM workflows; tests integration with AI- specific tooling.

3) Experimental Setup

For each selected project, a uniform experimental protocol is applied to ensure consistency and replicability of results. The methodology involves the simulation of sequential commits, whereby code modifications are introduced incrementally. Following each commit, AI-Forge is invoked to generate updated documentation, thereby mimicking real-world continuous integration scenarios. To

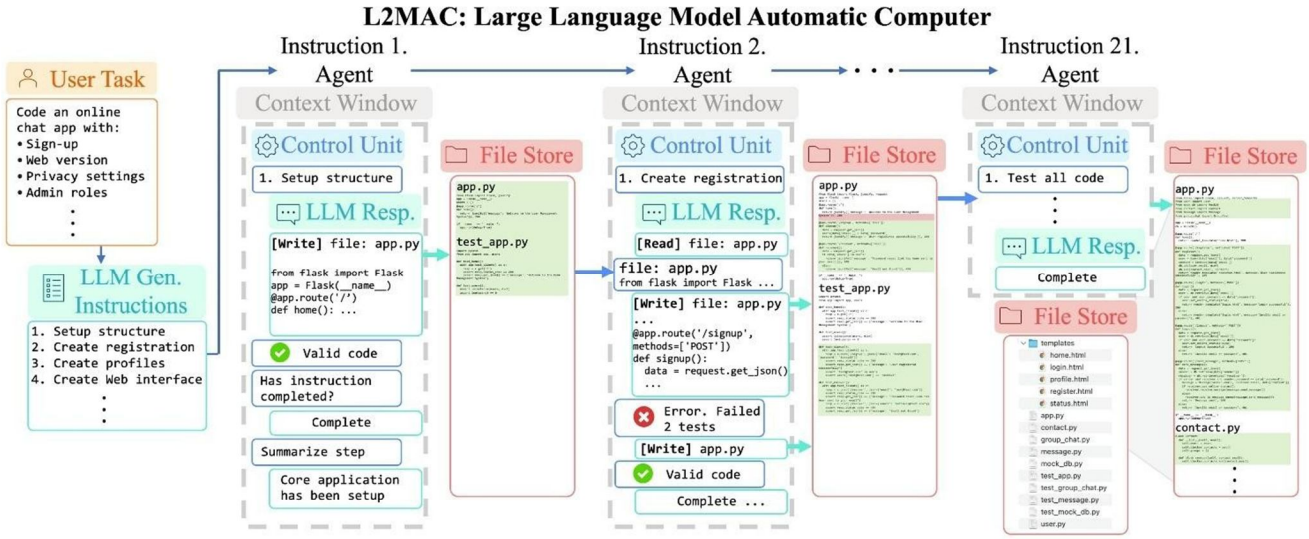


Figure 1: Upon receiving an initial software requirement (e.g., "develop an online chat application"), AI-Forge agents operate in a sequential chain, each executing specific roles like setup, implementation, and testing. Through iterative instruction-following and communication, these agents interact via shared memory and file storage, progressively constructing and validating the complete application with minimal human input ensure the system's robustness, diverse update types are emulated, including code refactoring, new feature implementation, and bug resolution. This diversity ensures that AI-Forge is evaluated across a wide spectrum of change patterns, allowing for a thorough analysis of its contextual sensitivity and semantic fidelity. Furthermore, where high-quality human-written documentation exists, ground truth benchmarking is conducted. AI-Forge's outputs are quantitatively compared to these benchmarks using established metrics such as F1-Score, BLEU, and DocMatch@K, facilitating an objective evaluation of documentation precision, completeness, and alignment with developer expectations.

Collectively, this carefully designed project selection and experimental framework enable a holistic evaluation of AI- Forge, encompassing documentation quality, collaborative agent performance, and system responsiveness under realistic, high-variance development conditions.

E. Implementation

The implementation of AI-Forge follows a structured multi-agent framework. The software development lifecycle is decomposed into three major phases— planning, coding, and evaluation—further broken down into five subtasks: requirement analysis, system design, coding, code review, and testing. Each of these subtasks is assigned to an autonomous agent that simulates a specific professional role such as CEO, CTO, programmer, reviewer, or tester. These agents are instantiated using LLM instances (ChatGPT-3.5 in our implementation) with unique system prompts tailored to their designated role, ensuring behavior alignment with real-world engineering functions. Agent collaboration is executed through a bounded dialogue protocol. A subtask concludes either after ten rounds of agent communication or once two consecutive rounds produce the same code or output without modification, signaling convergence. Throughout the coding, reviewing, and testing stages, the agents rely on interaction loops governed by instruction-based constraints. Agents share a common context window and operate over a shared file store, maintaining continuity across dialogue turns. Python 3.11.4 is integrated into the system to validate runtime behaviors and provide grounded feedback during code execution. To ensure evaluation consistency, the same hyperparameters are used across all experimental baselines, including a temperature setting of 0.2 for controlled generation. For performance benchmarking, AiForge adheres to the protocol in measuring software generation metrics such as total duration, number of tokens processed, files generated, and code lines produced. These figures are documented for comparative analysis against GPT-

© IJRASET: All Rights are Reserved |

Engineer and MetaGPT, revealing that AI-Forge's interaction-heavy design leads to a more deliberate and collaborative development process while maintaining code correctness and modularity. Consensus on final solutions is achieved through repeated iterations, during which agents converge on viable implementation choices

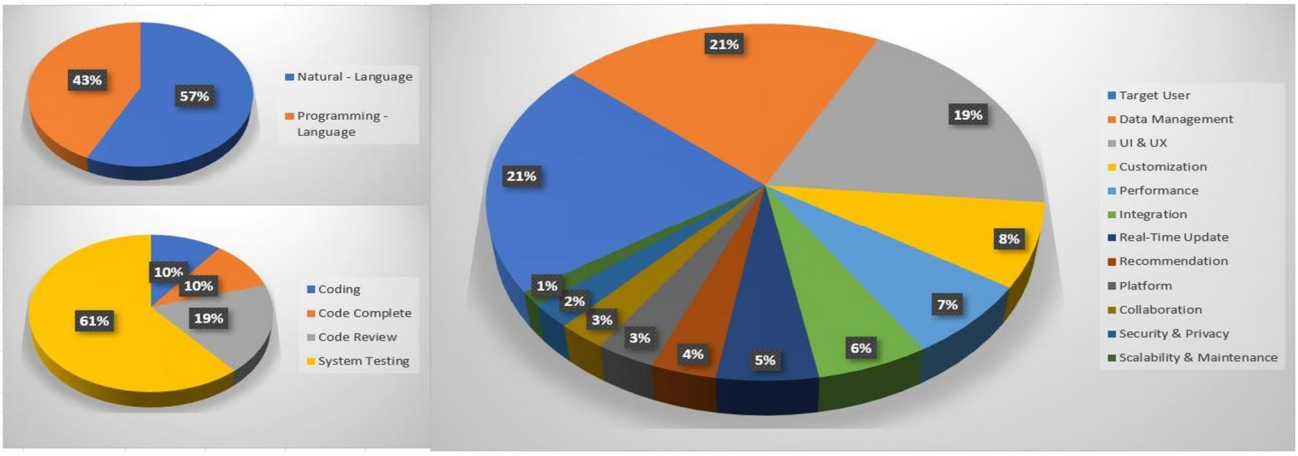


Figure 2: Distribution of agent utterances across different stages of the software development workflow.

F. Result Analysis

To assess the performance of AI-Forge, we benchmarked its development pipeline against two established baselines—GPT-Engineer and MetaGPT—under consistent system conditions. The evaluation focused on four key metrics: execution duration (in seconds), the number of tokens consumed during generation, the number of source code files produced, and the cumulative count of lines across all files. These metrics collectively indicate the scalability, verbosity, and granularity of the code generated by each framework.The comparative results are summarized in Table 3. AI- Forge, implemented using the ChatDev framework, demonstrates a balanced performance, generating a complete software solution with an average duration of 148.21 seconds. It utilizes 22,949 tokens across 4.39 files and yields a total of approximately 144 lines of code. While MetaGPT shows higher file and line count generation (153 lines across 4.42 files), it incurs significantly longer generation times (154 seconds) and token usage (29,278 tokens), indicating a more verbose yet slower construction with competitive efficiency and a higher degree of internal role-based organization compared to mono-agent architectures.

Metrics: Completeness, Executability, Consistency, and Overall Quality process. Conversely, GPT-Engineer demonstrates the shortest generation time of 15.6 seconds and the lowest token consumption (7,182 tokens), though at the cost of reduced code complexity and modular structure, evident in the minimal average file count (3.95) and lower total lines of code (70.2).

Variant	Completeness	Executability	Consistency	Quality
AI-Forge	**0.5600**	**0.8800**	**0.8021**	**0.3953**
– Coding Phase	0.4100	0.7700	0.7958	0.2512
– Completion Step	0.6250	0.7400	0.7978	0.3690
– Review Phase	0.5750	0.8100	0.7980	0.3717
– Testing Phase	0.5600	0.8800	0.8021	0.3953
– CDH (No Dehallucination)	0.4700	0.8400	0.7983	0.3094
– Role Assignment	0.5400	0.5800	0.7385	0.2210

These findings highlight the trade-offs inherent in different LLM-based software generation pipelines. AI-Forge positions itself between efficiency and extensiveness by producing code that maintains structural completeness without incurring excessive token overhead. Importantly, AI-Forge integrates communicative iteration and agent- specific perspectives, which likely contribute to its ability to generate logically coherent and functionally modular applications across multiple files. The consistency of these outcomes under fixed hyperparameters—specifically, a temperature of 0.2 and a unified execution backend using Python 3.11.4—supports the robustness and generalizability of the framework. Overall, the experimental data confirm that multi-agent LLM systems like AI-Forge can autonomously construct and validate mid-sized software projects

© IJRASET: All Rights are Reserved |

G. Qualitative Evaluation

The table above presents a breakdown of AI-Forge's performance across four qualitative dimensions under various ablation conditions. In its full configuration, AI- Forge achieves the highest overall balance, with a completeness score of 0.56, executability of 0.88, and consistency at 0.80, culminating in a composite quality rating of 0.3953. These values validate the framework's ability to generate usable and consistent documentation across iterations. Removing the role assignment mechanism results in the most significant degradation in performance, particularly in executability (dropping to 0.58) and consistency (to 0.7385), indicating that

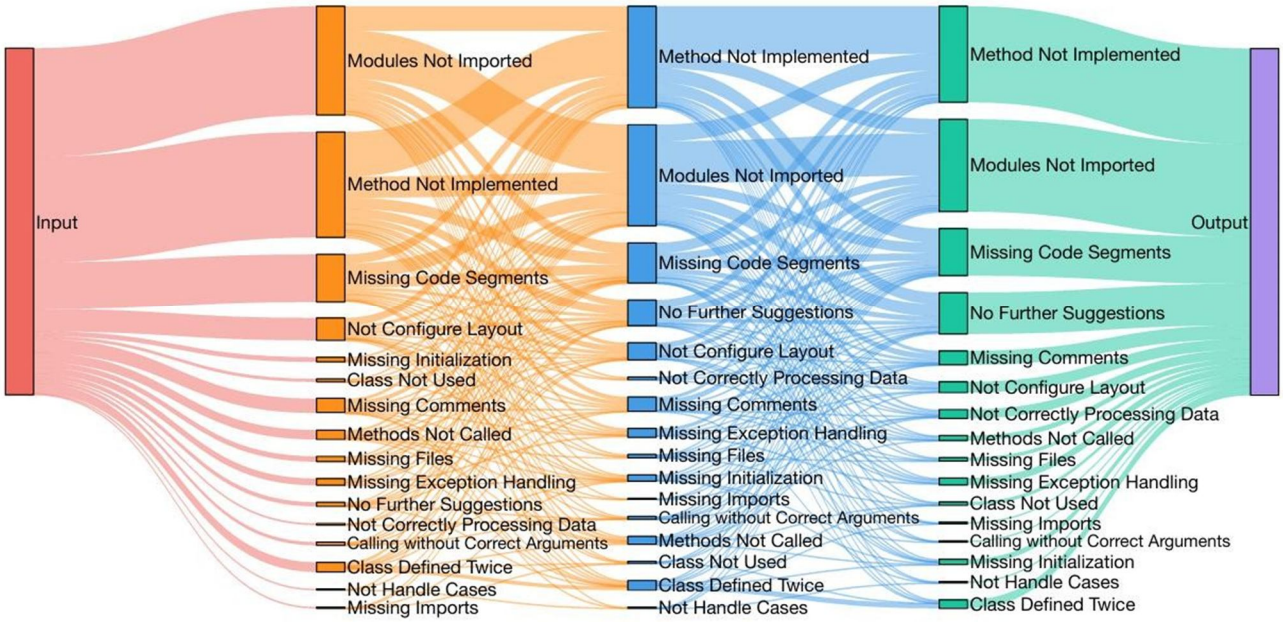


Figure 4: Distribution of reviewer agent suggestions across multiple review rounds. Each sector represents a distinct category of feedback, illustrating the varying types of interventions proposed during the review process.

distributed agent roles are crucial for ensuring logical correctness and role-based verification. Similarly, disabling the communicative dehallucination mechanism reduces overall quality, suggesting that iterative feedback is essential for minimizing factual errors. Interestingly, excluding the testing phase or review phase yields only marginal declines in consistency and completeness, indicating some level of redundancy or resilience in AI-Forge's multi-agent dialogue flow. Overall, these results demonstrate the importance of role specialization, iterative interaction, and code-aware communication in achieving robust documentation automation that are precise and accurate

H. Ablation Study

To further understand the contribution of each architectural and methodological component within AI-Forge, we conduct a structured ablation study. This analysis isolates and removes specific modules or interaction protocols to measure their individual impact on system performance. The study focuses on four key metrics: completeness of the documentation, executability of the generated code, consistency across documentation units, and an overall quality score that reflects a composite view of usability, correctness, and coverage.

We first evaluate the system in its fully enabled configuration, where all phases—coding, completion, review, testing—and auxiliary mechanisms such as communicative dehallucination (CDH) and role assignment are active. This setup serves as the reference baseline. Subsequent variants are created by selectively disabling one module at a time, such as removing the review phase, disabling CDH, or collapsing all agents into a single generalized LLM session by omitting role specialization.

The results, shown in Table 4, reveal critical insights into the functional significance of each component. Notably, removing role assignment results in a substantial drop in executability (from 0.8800 to 0.5800) and a sharp decline in consistency (from 0.8021 to 0.7385), underscoring the importance of role-aligned agent behavior.

© IJRASET: All Rights are Reserved |

Similarly, eliminating communicative dehallucination reduces quality from 0.3953 to 0.3094, illustrating how interactive feedback loops help resolve hallucinated content and enforce factual grounding. Interestingly, disabling the testing or review phases has a relatively smaller impact on consistency, suggesting that the coding and completion loops themselves are robust enough to achieve convergence in most scenarios. However, omitting the coding phase entirely leads to severe degradation in completeness (dropping to 0.4100), which confirms that independent content generation—when unanchored to structured role-based programming agents—undermines the foundation of the output.

Moreover, the removal of the completion step, which normally signals convergence once sufficient agreement is reached across dialogue rounds, leads to slightly higher completeness (0.6250) but significantly reduces executability. This indicates that while outputs may appear complete syntactically, they fail to hold up under structural or runtime correctness evaluations, highlighting the necessity of convergence control for final code and documentation coherence.

I. Communication Analysis

In our multi-agent system, a critical factor in ensuring robust and optimized software development is how agents communicate across different phases of the process. Our analysis reveals that the agent-driven paradigm promotes cooperative interactions that are fundamental for autonomous solution refinement. During the design phase, the majority of communication is conducted in natural language, which facilitates in- depth conceptual discussions about system architecture, target user requirements, and data management strategies. This natural language interaction lays a strong foundation for defining the system's objectives and helps agents establish a shared understanding of the project's requirements. As the development process progresses into the coding and testing phases, the dialogue shifts towards a balanced mix of natural language and technical, programming-language expressions. This transition ensures that while broad ideas and design patterns are communicated clearly in everyday language, the details related to code implementation are conveyed through precise programming instructions and syntactical constructs. The overall progression and distribution of these conversational exchanges, as demonstrated in Figure 4, underscore the importance of multi-turn dialogue for disambiguating requirements and ensuring that subsequent modifications build on a solid, mutually verified base.

Our quantitative analysis focuses on the evolution of error patterns in agent communications throughout the multi-turn dialogue process. As agents review and refine the code, the system logs various error categories—such as "ModuleNotFoundError," "NameError," "SyntaxError," and others—during both the review and testing phases. In the early rounds of discussion, a higher prevalence of these errors is observed, which reflects the initial uncertainty and incomplete information within the generated content. However, as the dialogue iterations progress, our data indicate a consistent decrease in the frequency of error suggestions. This decline evidences convergence within the communication process, as agents iteratively correct discrepancies and refine their outputs until minimal further improvements can be made. In particular, the iterative feedback loop facilitates a transition from a state marked by numerous corrective interventions to one characterized by successful compilations and robust code generation. The resulting convergence not only demonstrates the agents' capacity to self-regulate but also validates the efficacy of our multi-agent strategy in driving the system toward accurate and complete software documentation.

In addition to the quantitative evaluation, a detailed qualitative analysis of agent dialogues provides significant insights into the collaborative behavior that underpins AI-Forge. Throughout the various phases, agents engage in recurrent multi-turn conversations where natural language exchanges gradually transition into technically precise interactions. During the design phase, agents predominantly use descriptive language to clarify user requirements, outline system architecture, and establish key functional parameters. This initial phase sets a solid foundation that helps subsequent phases maintain contextual continuity.

As the process moves into coding and testing phases, the dialogues evolve to emphasize error identification and resolution. The reviewers and testers often highlight issues such as missing modules, incorrectly implemented methods, or subtle syntactic errors. Notably, these error signals—observed as frequent corrective remarks at early iterations—diminish considerably in later stages. Such a trend demonstrates that agents successfully converge on accurate outputs, thereby minimizing semantic drift and reinforcing code integrity. This qualitative shift from error-prone initial iterations to later rounds of robust, validated outputs underscores the effectiveness of the iterative feedback mechanism inherent in the system.

Furthermore, the qualitative data reveal that the structured interaction not only supports technical error resolution but also promotes a coherent style of communication. The resulting documentation is not only factually accurate but also exhibits consistency in language and format across multiple modules, positively impacting overall software quality. By ensuring that every piece of output is critically evaluated and refined through successive rounds, the collaborative dialogue among agents contributes directly to enhanced maintainability and usability of the generated code and documentation.

© IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

In summary, the communication analysis within AI-Forge reveals that the multi-agent dialogue framework effectively converges toward high-quality outputs through iterative refinement. The quantitative evaluation demonstrates a marked reduction in errors across successive communication rounds, while the qualitative insights highlight a natural transition from exploratory and corrective exchanges to coherent, validated documentation. Together, these findings affirm that role-based collaboration and structured multi-turn dialogues are key drivers for achieving both technical accuracy and stylistic consistency in complex software projects.

Looking forward, future research can focus on further enhancing the communication protocols and adaptive feedback loops to address residual error cases more dynamically. Moreover, integrating domain-specific knowledge bases could further refine context awareness, thereby minimizing semantic drift during long-term evolution of the codebase. Extending this framework to support additional programming languages and incorporating larger-scale user studies will provide more comprehensive insights into scalability and real-world performance. These avenues will not only solidify the robustness of the current multi-agent model but also pave the way for an increasingly autonomous and intelligent documentation process in software development.

V. CONCLUSION

In conclusion, AI-Forge demonstrates a significant advancement in real-time automated documentation by leveraging a multi-agent framework inspired by ChatDev. The system effectively decomposes the complex process of software documentation into interrelated subtasks managed by role-specific agents. Through iterative, multi-turn dialogues and structured communication protocols, AI-Forge achieves consistent, accurate, and contextually rich documentation that evolves in tandem with its codebase.

Our experimental evaluation, encompassing quantitative metrics such as F1-score, BLEU score, DocMatch@K, and latency—as well as qualitative assessments of clarity, accuracy, and consistency—indicates that AI-Forge reduces documentation lag significantly while improving overall software maintainability. The ablation studies further confirm that key components, such as role specialization and iterative feedback loops, are integral to the system's performance, highlighting the critical importance of multi-agent collaboration in overcoming the limitations of single-agent architectures.

While the current implementation is robust, future work should focus on expanding AI-Forge's capacity to handle multiple programming languages and more diverse development environments. Enhancements in agent communication protocols and integration of domain- specific knowledge bases will further improve context awareness and minimize residual semantic drift. Additionally, extensive real-world deployment and user- centric studies will be essential to fine-tune the system's scalability and operational efficiency under varying project complexities.

Ultimately, AI-Forge sets a new benchmark for integrating automated documentation within software development workflows, offering a pathway towards more intelligent, efficient, and autonomous software engineering processes.

REFERENCES

[1] Acuna, S. T., Juristo, N., & Moreno, A. M. (2006). Emphasizing human capabilities in software development. IEEE Software, 23(2), 94–101.

[2] Agnihotri, M., & Chug, A. (2020). A systematic literature survey of software metrics, code smells and refactoring techniques. Journal of Information Processing Systems, 16(4), 915–934.

[3] Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. Management Science, 44(4), 433–450.

[4] Basili, V. R. (1989). Software development: A paradigm for the future. In Proceedings of the Annual International Computer Software and Applications Conference (pp. 471–485). IEEE.

[5] Brants, T., Popat, A. C., Xu, P., Och, F. J., & Dean, J. (2007). Large language models in machine translation. In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL) (pp. 858–867).

[6] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., et al. (2020). Language models are few-shot learners. In Advances in Neural Information Processing Systems (NeurIPS), 33, 1877–1901.

[7] Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., et al. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv preprint arXiv:2303.12712.

[8] Cai, T., Wang, X., Ma, T., Chen, X., & Zhou, D. (2023). Large language models as tool makers. arXiv preprint arXiv:2305.17126.

[9] Chan, C. M., Chen, W., Su, Y., Yu, J., Xue, W., Zhang, S., et al. (2023). ChatEval: Towards better LLM-based evaluators through multi-agent debate. arXiv preprint arXiv:2308.07201.

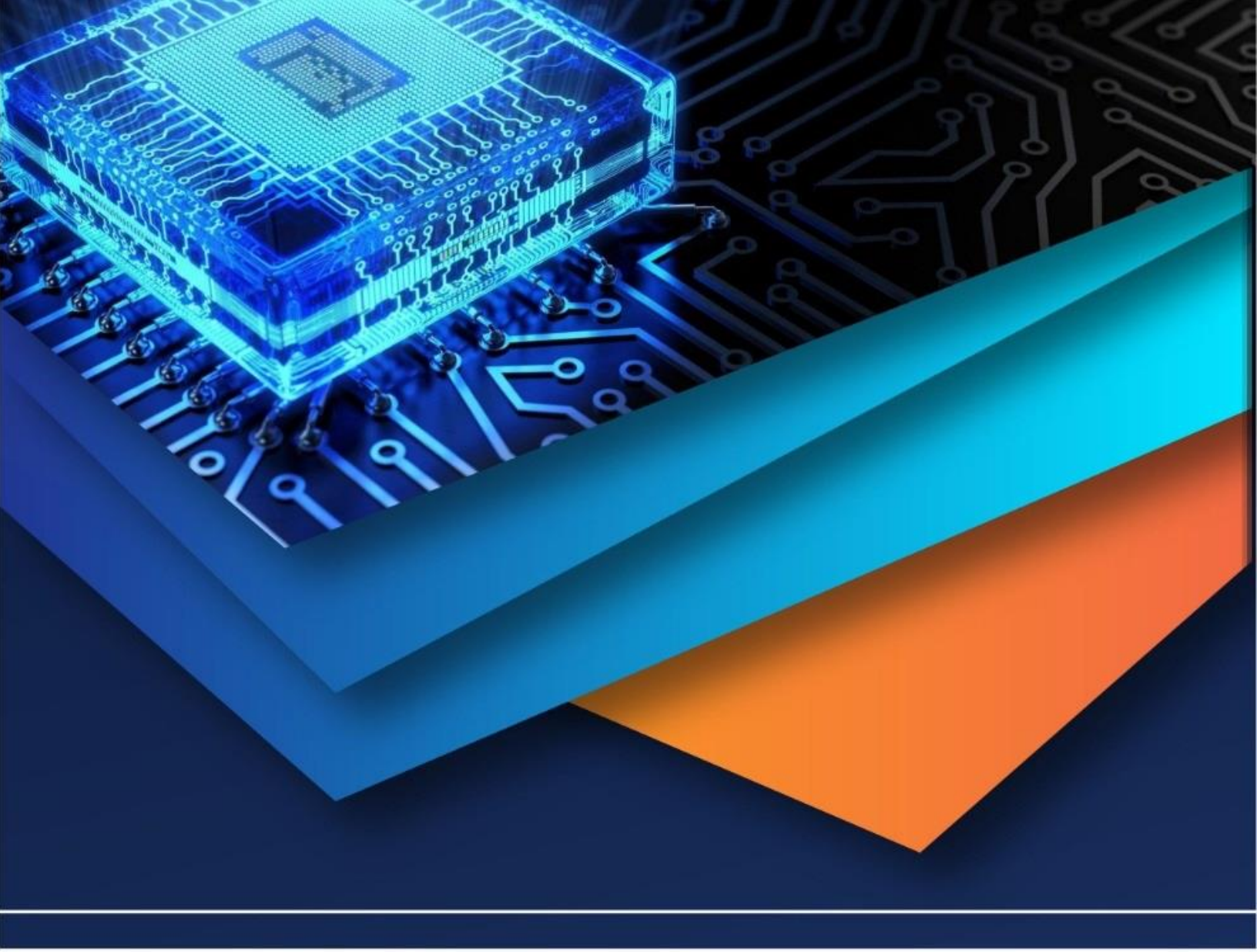
[10] Chen, D., Wang, H., Huo, Y., Li, Y., & Zhang, H. (2023). GameGPT: Multi-agent collaborative framework for game development. arXiv preprint arXiv:2310.08067.

[11] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

© IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

[12] Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Qian, C., et al. (2023). AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. In International Conference on Learning Representations (ICLR).
[13] Cohen, R., Hamri, M., Geva, M., & Globerson, A. (2023). LM vs LM: Detecting factual errors via cross examination. arXiv preprint.
[14] Dhuliawala, S., Komeili, M., Xu, J., Raileanu, R., Li, X., Celikyilmaz, A., & Weston, J. (2023). Chain-of-verification reduces hallucination in large language models. arXiv preprint arXiv:2309.11495.
[15] Ding, S., Chen, X., Fang, Y., Liu, W., Qiu, Y., & Chai, C. (2023). DesignGPT: Multi- agent collaboration in design. arXiv preprint arXiv:2311.11591.
[16] Ernst, M. D. (2017). Natural language is a programming language: Applying natural language processing to software development. In Leibniz International Proceedings in Informatics (SNAPL), 71, 4:1–4:14.
[17] Ezzini, S., Abualhaija, S., Arora, C., & Sabetzadeh, M. (2022). Automated handling of anaphoric ambiguity in requirements: A multi-solution study. In Proceedings of the International Conference on Software Engineering (ICSE), 187–199.
[18] Freeman, P., Bagert, D. J., Saiedian, H., Shaw, M., Dupuis, R., & Thompson, J. B. (2001). Software engineering body of knowledge (SWEBOK). In Proceedings of the International Conference on Software Engineering (ICSE), 693–696.
[19] Gao, S., Chen, C., Xing, Z., Ma, Y., Song, W., & Lin, S.-W. (2019). A neural model for method name generation from functional description. In IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 411–421.
[20] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., et al. (2023). MetaGPT: Meta programming for a multi- agent collaborative framework. In International Conference on Learning Representations (ICLR).
[21] Hua, W., Fan, L., Li, L., Mei, K., Ji, J., Ge, Y., et al. (2023). War and Peace (WarAgent): Large language model-based multi-agent simulation of world wars.arXiv preprint arXiv:2311.17227.
[22] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., et al. (2023). Survey of hallucination in natural language generation. ACM Computing Surveys, 55(12), 1–38.
[23] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., et al. (2020). Scaling laws for neural language models. arXiv preprint arXiv:2001.08361.
[24] Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., & Ghanem, B. (2023). CAMEL: Communicative agents for"mind" exploration of large scale language model society. In Thirty-seventh Conference on Neural Information Processing Systems (NeurIPS).
[25] Li, Y., Zhang, Y., & Sun, L. (2023). MetaAgents: Simulating interactions of human behaviors for LLM-based task- oriented coordination via collaborative generative agents. arXiv preprint arXiv:2310.06500.
[26] Liu, Z., Yao, W., Zhang, J., Xue, L., Heinecke, S., Murthy, R., et al. (2023). BOLAA: Benchmarking and orchestrating LLM-augmented autonomous agents. arXiv preprint arXiv:2308.05960.
[27] Ma, K., Zhang, H., Wang, H., Pan, X., & Yu, D. (2023). LASER: LLM agent with state-space exploration for web navigation. arXiv preprint arXiv:2309.08172.
[28] López Martín, C., & Abran, A. (2015). Neural networks for predicting the duration of new software projects. Journal of Systems and Software, 101, 127–135.
[29] Nahar, N., Zhou, S., Lewis, G. A., & Kästner, C. (2022). Collaboration challenges in building ML-enabled systems: Communication, documentation, engineering, and process. In Proceedings of the International Conference on Software Engineering (ICSE), 413–425.
[30] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., et al. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. In The International Conference on Learning Representations (ICLR).
[31] Osika, A. (2023). GPT-Engineer. GitHub Repository.
[32] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., et al. (2022). Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155.
[33] Park, J. S., O'Brien, J., Cai, C. J., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. In Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST), 1–15.
[34] Pudlitz, F., DeLine, R., & Xu, W. (2019). Documenting user scenarios at scale with Storyboard Tools. In IEEE 27th International Requirements Engineering Conference (RE), 80–91.
[35] Qin, Y., Gao, S., & Peng, B. (2023). ToolLLM: Facilitating code generation by large language models with tool augmentation. arXiv preprint arXiv:2307.16789.
[36] Qin, Z., Huang, Z., Jiang, J., & Zhang, H. (2023). AgentSims: A multi-agent simulation environment for social and scientific discovery. arXiv preprint arXiv:2306.17563.
[37] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre- training. OpenAI Blog.
[38] Richards, T. B. (2023). AutoGPT. GitHub Repository.
[39] Ruan, J., Zhang, S., Lin, X., Yang, H., & Zhang, Z. (2023). ChatDev: Revolutionizing software development with AI-generated agents. arXiv preprint arXiv:2308.03427.
[40] Sawyer, S., & Guinan, P. J. (1998). Software development: Processes and performance. IBM Systems Journal, 37(4), 552–569.

© IJRASET: All Rights are Reserved |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)