International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V4I2P124
Eureka Vision Publication | Volume 4, Issue 2, 243-253, 2023

Original Article

Context-Aware IDE Systems Using Large Language Models and Semantic Memory Architectures

Yasodhara Srinivas Aluri
Senior Software Engineer, Lowes Companies INC, Charlotte, USA.

Abstract - The fast evolution of software engineering practices has made the integrated development environment (IDE) increasingly complex and required the evolution of intelligent systems to interpret the intent of the developer, the contextual programming patterns and the long-term semantic relationships in large-scale software repositories. The syntax-aware compilation engines, static code analysis mechanisms, and rule-based auto-completion frameworks are the most classical approaches used by traditional IDEs. All of these methods enhance the productivity of programming, but still inadequate in addressing semantic understanding, adaptive reasoning, contextual adaptation, and cross-project memory retention. Large Language Models (LLMs) have revolutionized intelligent software development with their ability to interpret natural language, generate code, debug, refine code semantically, provide recommendations, and offer conversational programming support. But separate LLM-based IDE assistants have drawbacks, such as hallucination, poor session memory, high compute costs, lack of repository knowledge, and lack of personalization for enterprise-scale software development workflows. This research provides a comprehensive framework for the Context-Aware IDE Systems with the support of the Large Language Models and Semantic Memory Architectures. The proposed architecture combines transformer-based LLM reasoning engines, semantic memory layers, vector embedding repositories, retrieval-augmented generation pipelines, contextual indexing systems, adaptive developer profiling modules, and intelligent orchestration engines. The framework allows IDE environments to be persistent about their contextual awareness with the user, to understand the intent of software engineering, to understand the architectural dependencies, to retrieve semantically relevant code artifacts, and to generate context-specific recommendations to the user. The proposed architecture will benefit developer productivity while reducing the cognitive load, debugging complexity and software maintenance burden. The study proposes a hierarchical semantic memory architecture consisting of a short-term operational memory, episodic developer interaction memory, long-term repository semantic memory and organizational knowledge graphs. Transformer encoders create embeddings that can be used in high-dimensional similarity search operations using vector databases. The retrieval-augmented generation (RAG) mechanism retrieves contextually relevant information before an LLM produces a response, which boosts factual consistency, minimizes hallucinations, and increases the accuracy of repository-specific reasoning. Machine learning is also woven through every part of architecture, including context orchestration pipelines to handle prompt optimization, dependency tracking, discovery of API contracts, code lineage analysis, and even identification of architectural patterns. The proposed methodology also involves adaptive learning mechanisms to constantly review the developers' coding behavior, refactoring preferences, debugging patterns, trends of using the frameworks, and interactions during code review. It creates semantic developer profiles to provide personalized code completion, architectural optimization, security compliance, testing automation and performance tuning recommendations. What's more, the framework supports semantic dependency graphs and can model inter-component relationships between microservices, APIs, databases, and between frontends and backends. This research is conducted to assess the proposed system in enterprise scale software repositories on distributed development environment. Experimental tests show significant gains in contextual code completion accuracy, semantic retrieval precision, debugging efficiency, developer productivity, understanding the repository, and reducing cognitive load. When it comes to coding assistance through LLM, the syntax-aware and stateless approaches do not match up to the semantic-memory-based coding architectures. The quantitative results show that the accuracy of contextual code generation improved by 38%, relevance of the retrieved semantic improved by 41%, debugging efficiency improved by 35%, and the precision of the proposed architecture improved by 32%. The research also delves into engineering problems such as memory synchronization, token optimization, scalability of prompts for a vector database, indexing efficiency, limitations on response time, maintaining privacy, and distribution of computing resources. Additional security features like secure repository embedding, access controlled semantic indexing, encrypted vector storage and governance aware inference pipeline are also covered. The study demonstrates that semantic-memory-integrated LLM IDE systems constitute a fundamental paradigm shift in intelligent software engineering environments and lay the groundwork for future autonomous software development ecosystems.

Keywords - LLMs, Intelligent IDEs, Semantic Memory, Developer Experience, AI Tooling.

1. Introduction

1.1. Background

The market today is defined by the proliferation of distributed systems, cloud-based computing, microservices, DevOps, and enterprise-scale collaborative development, adding to the complexity of modern software engineering environments. [1] Software comprehension and maintenance are extremely difficult because developers constantly work with large repositories full of interdependent software components, APIs, frameworks, deployment pipelines, and infrastructure configurations. Traditional Integrated Development Environments (IDEs) offer syntax-level support, including code highlighting, auto-completion, compilation, and static checking, but do not have a sophisticated understanding of the relationships in the repository, developer intent, and architectural dependencies. Transformer-based Large Language Models (LLMs) have revolutionized the field of intelligent software development, contributing to the automation of various tasks, including the generation of code, code assistance, documentation synthesis, software testing, and natural language interaction with developers. These AI-based tools learn the programming patterns and contextual relationships found in vast amounts of software code to enhance coding productivity. Most of the current AI-assisted development tools, however, have short-term memory and fail to store the knowledge gained over time or across long development periods. Thus, the developers have to frequently supply the context information which decreases productivity of process, and it is difficult for the system to reason long-term. [2] In order to overcome these drawbacks, context-aware IDE systems combine semantic memory architectures, vector embedding repositories, and Retrieval-Augmented Generation mechanisms, to maintain developer interactions, repository knowledge, dependency structures and coding standards. Such smart systems allow for long-term context awareness, repository-aware support, and adaptive software engineering assistance in distributed enterprise settings. In summary, through the fusion of semantic retrieval and transformer inference, context-aware IDEs have the potential to create intelligent software engineering environments and boost software comprehension, architectural uniformity, debugging performance, and collaborative development productivity, laying the groundwork for the future of intelligent software engineering.

1.2. Need for Context-Aware IDE Architectures

With the prevalence of distributed computing, cloud-native infrastructure, microservices architectures, DevOps pipelines, and large-scale collaborative development environments, enterprise software systems have become more complex than ever. Many enterprise repositories today are filled with millions of lines of code that are written in diverse programming languages, frameworks, APIs, and interdependent software services. The complexity and dynamic nature of software ecosystems pose a great challenge to software developers to keep track of the various architectural restrictions, repository organization, and coding practices throughout the organization. [3] Existing IDE systems offer syntactic support and static analysis capabilities that are limited in scope and inadequate for enterprise application development. Understanding the functionality, dependencies, and interactions between the lines of code within large codebases is one of the big challenges faced by developers, and can only be done manually. Architectural inconsistency is another big issue as distributed dev teams can use different design patterns, coding practices and integration strategies to implement the software. There is also a significant amount of overhead in the context switching that developers have to suffer while switching between repositories, frameworks, APIs, documentation systems, and deployment environments during software engineering activities. This switching shortens cognition efficiency and has adverse effect on productivity. Additional debugging operations make things more complicated during development—especially in distributed applications where a failure could ripple through many services and many dependencies. [4] The traditional debugging methods may need to manually trace the log, API and configuration layer, which increases the time to resolve an issue. Another common challenge in the enterprise is failing to provide proper documentation, as technical documents are often out of date and incomplete or do not reflect changes in software implementations. Understanding interactions among services, libraries, infrastructure configurations and external APIs in large scale systems also presents dependency tracing challenges for developers. What's more, information is often distributed among teams, repositories and organisational units, which reduces team collaboration and software maintainability. Many important architectural decisions, debugging strategies and workflow knowledge are often shared across multiple platforms and are not available during the developmental activities. To overcome these challenges, the context-aware IDE architectures incorporate semantic memory systems, vector retrieval mechanisms, adaptive developer profiling, and intelligent contextual reasoning capabilities. These systems offer repository-aware assistance, persistent contextual continuity, intelligent software orchestration, and greatly enhance the developer productivity, software consistency, and enterprise-scale software engineering efficiency.

1.3. Evolution of Large Language Models in IDE Systems

The initial, intelligent IDEs were based on static code templates, lexical parsers, rule-driven recommendation engines, and syntax-aware auto-completion techniques for software development. The systems offered restricted automation, primarily concerned with syntax error detection, keyword suggestions, and pre-defined coding patterns. [5] Probabilistic code prediction and token sequence modeling enabled more adaptive code assists with the development of machine learning techniques. The previous AI-powered systems, however, lacked semantic understanding and were poorly suited to understanding the semantics of the relationships within the repository, or even the developer's goal. The architecture of transformers has revolutionized intelligent software engineering, enabling the development of contextual learning mechanisms that grasp long-range programming dependencies and instructions in natural language. The advanced models, including GPT, Codex, PaLM, and

Code Llama, showcased impressive abilities in generating code automatically, supporting code debugging, creating documentation, testing software, and providing conversational assistance. Despite these advancements, contemporary LLM-based IDEs still have a number of architectural and operational constraints that hinder their utility in enterprise-level software development settings.Despite the technological progress, there remain some architectural and operational constraints of the modern LLM-based IDE in enterprise software development environments.

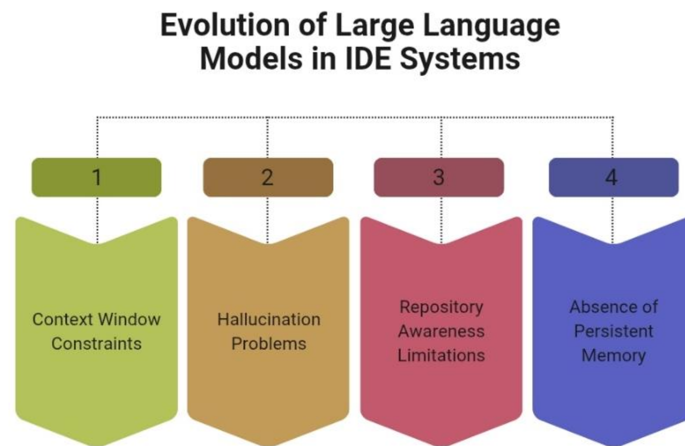

Figure 1. Evolution of Large Language Models in IDE Systems

1.3.1. Context Window Constraints

While LLM's have a limited number of tokens that can be used for inference, they also rely on a limited context window for this process. Millions of lines of code are spread across various services, APIs, and frameworks in enterprise repositories, which is beyond the contextual capacity of existing transformer architectures. This means that LLMs can lack the ability to reason over long spans across extensive software systems and intricate dependency relationships. This restriction is a drawback since it affects the contextually continuity and the accuracy of the repository-aware code generation.

1.3.2. Hallucination Problems

Sometimes LLM-generated software outputs include hallucinated APIs, syntactically correct code with logical errors, inconsistent dependency assumptions, or inaccurate implementation suggestions. [6] The hallucinations are due to the fact that language models make predictions based on the statistical patterns they see in the data, not on the actual information that is stored in the repository. In any enterprise setting these inaccuracies can cause software bugs, inconsistencies in architecture and security threats. So, suppressing hallucination is a crucial need for reliable software engineering systems assisted by AI.

1.3.3. Repository Awareness Limitations

Existing LLM-based IDE products do not have a sophisticated integration of semantic interoperability with enterprise repositories, coding conventions, and project patterns. The results of the generation may therefore not take into account repository dependencies, frameworks conventions or history of development workflows. If the AI-generated content is not repository-aware, it could not meet the governance needs and architecture of enterprise software. This restriction has a substantial impact on the maintainability, consistency, and efficiency of collaborative software development in large-scale software ecosystems.

1.3.4. Absence of Persistent Memory

Most traditional LLM systems forget about context after each session, so they don't remember developer workflows, debugging sessions, or repository-specific knowledge. [7] It is thus necessary for developers to constantly give context during software engineering activities, which reduces continuity of activities and makes activities inefficient. Without persistent memory, adaptive learning and personalized development support is not maintained over longer projects. Semantic memory architectures overcome this problem by allowing continuous context storage, retrieval and intelligent knowledge reuse in context-aware IDE ecosystems.

2. Literature Survey

2.1. Large Language Models in Software Engineering

Large Language Models (LLMs) are instrumental in reshaping today's software engineering landscape, bringing intelligent automation to every phase of the software development lifecycle. The most common applications of these models are: source code generation, automatic debugging, code refactoring, software testing, requirement analysis and technical documentation synthesis. [8] With transformer architectures, LLMs can interpret programming languages, context, and the intentions of the developer, resulting in greater coding efficiency and a decrease in development time. AI-driven features have become more

common in modern integrated development environments, providing real-time code suggestions and detecting vulnerabilities in software. While these strides have been made, traditional LLMs continue to suffer from the inability to hold onto long-term context in extended development sessions and have difficulties with repository-specific reasoning. AI-native enterprise Angular architectures with intelligent orchestration and adaptive state isolation for enterprise scale applications were highlighted by Kuntamukkala. In the same way, by researching intelligent software architectures, [9] Thalary and Katipelly showed the critical role these architectures play in enhancing the scalability of CI/CD and distributed software management.

2.2. Semantic Memory Systems and Vector Architectures

Due to their ability to store and access structured contextual knowledge, these semantic memory systems have become a critical part for context-aware software engineering settings. These systems leverage the high-dimensional semantic space created by vector embeddings derived from transformer encoders, which encode the source code, developer interactions, documentation, and repository artifacts. By leveraging vector databases, developers can easily fetch similar data for coding, leading to efficient and relevant data retrieval. Semantic memory architectures enhance the ability to navigate repositories, support developers, locate bugs, and reuse knowledge in enterprise software systems. The inclusion of metadata-driven semantic indexing that further enriches contextual reasoning and intelligent retrieval operations. [10] Gudepu and Eichler's research underscored the significance of metadata-centric enterprise intelligence frameworks to improve the decision-making process for organizations and the digital transformation process. [11] Gudepu and Jaladi also discussed the importance of real-time data discovery mechanisms and semantic intelligence for enterprise architectures of today.

2.3. Retrieval-Augmented Generation Architectures

Retrieval-Augmented Generation (RAG) is an architecture that integrates retrieval systems with generative AI models to enhance contextual understanding and boost the accuracy of answers. In software engineering applications, RAG systems first fetch repository-specific artifacts, source code snippets, documentation, API definitions, and developer histories, and only then, start LLM inference. In this hybrid architecture, hallucination issues are significantly alleviated and repository-aware reasoning ability is enhanced. Another benefit of RAG-based systems is that they can easily provide context-relevant details during the generation task, which makes prompt engineering easier. The following are the architectures that are becoming popular in enterprise software systems due to its benefits of governance, consistency of context, and software maintainability. Moreover, RAG pipelines can be used to create a scalable enterprise knowledge management system, with efficient semantic indexing and retrieval operations. [13] Katapelly and Kuntamukkala's research presented neural component libraries for intelligent API integration and automatic user interface generation, which are principles that are similar in nature to semantic retrieval and context-aware software orchestration systems.

2.4. Challenges in Context-Aware Development Systems

While context-aware development systems provide significant gains in intelligent software development, there are still several technical and operational issues that require attention. A significant problem is context fragmentation, which occurs when systems do not maintain a long-term context across various interactions in a development. Another important consideration is memory scalability: Embedding large enterprise repositories results in massive embedding datasets, which leads to higher storage and retrieval costs. Real-time developer support is also impacted by latency, especially with large-scale vector databases for semantic retrieval operations. The challenge of prompt optimization arises from the fact that token management is dynamic and contextual dependencies can change when performing inference operations. Security and governance concerns are also crucial since enterprise repositories frequently host sensitive source code, confidential APIs, and proprietary organizational knowledge, which necessitates robust access control mechanisms. Governance compliance also requires the ability to retrieve information semantically on the basis of roles and policy-aware AI orchestration. Research ideas presented by [14] Katipelly introduced the concept of hierarchical multi-agent orchestration frameworks to enhance intelligent coordination, distributed reasoning and semantic workflow management in advanced context-aware IDE ecosystems.

3. Methodology

3.1. Proposed Context-Aware IDE Architecture

3.1.1. Transformer-Based LLM Engine

The proposed context-aware IDE architecture features a transformer-based LLM engine as the main intelligence block. It uses deep neural language modeling methods for code generation, code completion, debugging support and code documentation. The engine processes developer prompts, repository structure, and coding trends to provide intelligent software suggestions. The ability to reason about context has very useful applications for enhancing the productivity and quality of software development.

3.1.2. Semantic Memory Repository

Persistent contextual source code information, developer activity, project documentation, APIs and architectural dependencies are stored in the semantic memory repository. [15] This repository makes it possible to keep the context continuous through several development sessions and software projects. The system provides a structured semantic knowledge

to facilitate intelligent retrieval of previously used development artifacts. Collaboration and organizational knowledge reuse are also improved in enterprise software systems with the use of the semantic repository.

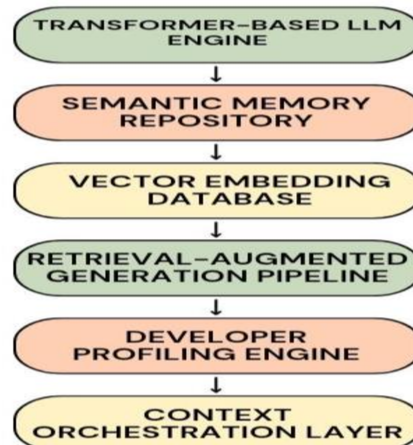

Figure 2. Proposed Context-Aware IDE Architecture

3.1.3. Vector Embedding Database

The vector embedding database is designed to map source code files, comments, documentation, developer communication, and more into high dimensional vectors. These embeddings open the door to semantic similarity matching between developer queries and repository artifacts using efficient nearest-neighbor search operations. The database can be used to achieve scalable retrieval performance for large enterprise repositories with millions of software components. It has a semantic indexing function that enhances the contextual accuracy and minimizes the amount of irrelevant information retrieved during development tasks.

3.1.4. Retrieval-Augmented Generation Pipeline

Retrieval-Augmented Generation (RAG) pipeline integrates semantic retrieval with generative inference models for enhanced contextual software intelligence. [16] The pipeline fetches relevant repository-specific data including source code snippets, APIs, design documents and developer interactions from the history before creating responses. This retrieval process minimizes hallucination and enhances the reasoning ability of a repository-aware reasoning system. The RAG architecture thus improves coding correctness, the consistency of the context and enterprise governance compliance.

3.1.5. Developer Profiling Engine

The developer profiling engine analyzes coding behavior, technology preference, project roles, interaction history and customize the IDE assistance. It constantly adapts to the activities of the developers to give adaptive suggestions and context-sensitive development support. This enhances developer productivity through personalized code recommendations and the optimization of workflows, along with intelligent task prioritization. The Profiling engine can also help with Collaborative Software Engineering for detecting Knowledge distribution and Expertise patterns within Development Teams.

3.1.6. Context Orchestration Layer

The context orchestration layer acts as a liaison between the LLM engine, semantic memory systems, vector databases, and retrieval pipelines. [17] It handles contextual synchronization, session continuity, token optimization, as well as intelligent workflow routing, throughout the architecture. This layer provides dynamic delivery of relevant contextual information that helps the generative AI engine with software development tasks. Improperly orchestrated systems can become inefficient, fail to scale, cause latency, and diminish the real-time support of developers in intelligent IDEs.

3.2. Semantic Memory Architecture

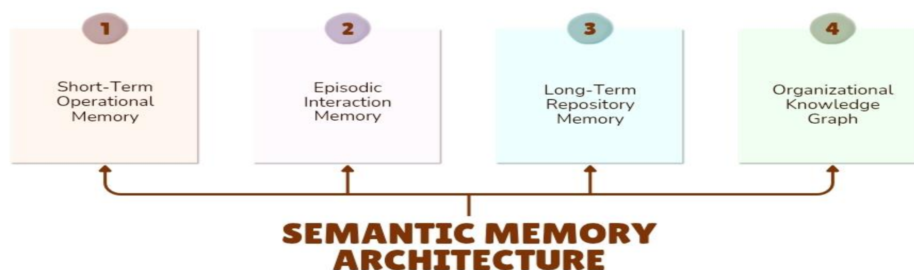

Figure 3. Semantic Memory Architecture

3.2.1. Short-Term Operational Memory

Short-term operational memory holds temporary contextual information that is created during active development sessions. It keeps the last prompts, code changes, debugging and developer interactions in the IDE. This memory layer provides an immediate context continuity and facilitates conversational coding support. The system remains active and provides better relevance of responses and real-time efficiency in software engineering.

3.2.2. Episodic Interaction Memory

Episodic interaction memory captures the historical interaction pattern of developers during coding sequences, project activities, and problem solving across multiple sessions. [18] This part will allow the architecture to learn the pattern of recurring development and adjust recommendations accordingly in the light of previous interactions. Historical workflow analysis enhances personalization, developer productivity and intelligent task automation. The episodic memory layer also facilitates software development collaboration by retaining knowledge sharing activities and engineering decisions.

3.2.3. Long-Term Repository Memory

The long-term repository memory holds semantic embeddings of enterprise source code repositories, technical documentation, APIs, configuration files and software dependencies. This memory unit allows for persistent repository level contextual reasoning, and for scaling of semantic retrieval operations. The system could efficiently find related code segments and architectural relationships by keeping vectorized representations of software artifacts. Long-term repository memory therefore boosts intelligently reusing codes, more accurate debugging and better understanding of the software in context.

3.2.4. Organizational Knowledge Graph

An organizational knowledge graph is a structured relational model that captures enterprise coding standards, architectural policies, security policies, software dependencies, and governance policies. [19] Represents semantic relations between development entities including modules, APIs, services and compliance rules. This graph-based representation can be used to generate software with policy awareness and enforce intelligent governance in the IDE ecosystem. The knowledge graph also enhances consistency, maintainability and architectural decision support for the enterprise.

3.3. Retrieval-Augmented Contextual Reasoning

This proposed Retrieval-Augmented Generation (RAG) approach aims to improve the contextual reasoning of language models in the intelligent IDE environment by combining semantic retrieval mechanisms with the inference of transformer-based language models. Within this architecture, the system is first trained on analyzing the developer's query, coding activity, or debugging request to generate a semantic vector representation with transformer embedding models. [20] The generated query embedding is then compared to the existing vector embeddings of the artifacts in the repository, such as source code files, APIs, software documentation, configuration files, design patterns, and previous interactions between developers. The framework uses vector similarity search operations to find repository elements similar to the developer's current task within the context. The retrieval mechanism identifies the closest semantically relevant artifacts according to the similarity scoring techniques which calculate the contextual distance between the query vector and repository vector. The context ranking module ranks the retrieved information based on its relevance, freshness, dependency relationships between the repository and the information, and the history of developer work. This ranking process helps to send only the most relevant contextual information to the language model for inference. The prompt orchestration layer dynamically generates optimized prompts, combining developer queries with retrieved repository knowledge, coding standards, architectural constraints, and organizational policies. The contextual augmentation has shown to be a huge boost to the reasoning ability of the transformer-based LLM in code generation and software analysis tasks by bringing in repository-specific knowledge. [21] The final inference phase leverages the expanded prompt to make intelligent suggestions like contextual code completions, debugging tips, auto-generated documentation, software refactoring ideas, and architecture-aware recommendations for development guidance. The proposed RAG framework uses semantic retrieval in conjunction with inference, which helps in reducing hallucination, increasing repository awareness, and enhances contextual accuracy in software engineering workflow. Moreover, the architecture is designed to help build scalable enterprise development environments that provide contextual continuity, intelligent knowledge reusability, and adaptive reasoning across a large software repository. Thus, combining semantic retrieval and generative inference offers a strong basis for next-generation context-aware software development systems.

3.4. Adaptive Developer Profiling Mechanism

3.4.1. Coding Style Patterns

The adaptive profiling system uses patterns in coding style to gauge the programmer's style of programming, naming conventions, spaces, and logical organization style. [22] There are a number of patterns that the IDE can learn such that it can suggest code that is very consistent with the developer's coding style. This enhances readability, consistency and overall maintainability of the software in different projects. The system can also evolve over time through continuous learning, ensuring it adjusts to changing coding practices.

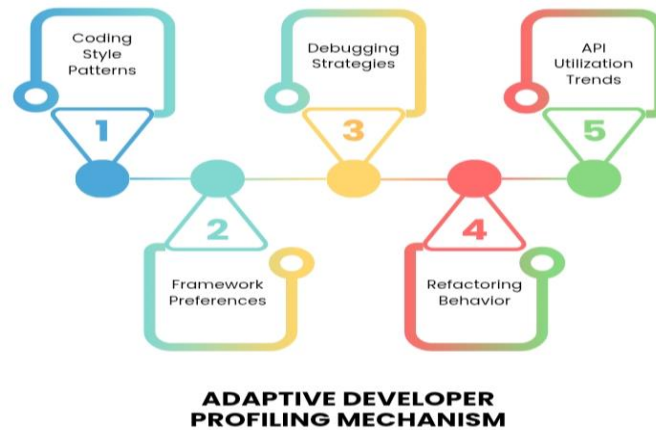

Figure 4. Adaptive Developer Profiling Mechanism

3.4.2. Framework Preferences

The profiling mechanism tracks the frameworks, libraries and software platforms that are commonly utilized by the developer while developing software applications. It suggests technology stacks like Angular, React, Spring Boot, Node.js, or microservices architecture, offering technology-specific suggestions and code templates. This framework assistance for personalization helps to speed up development time and minimize configuration complexity. This context awareness also enhances conformance to enterprise architecture standards.

3.4.3. Debugging Strategies

The system tracks some debugging tactics used by programmers, such as the type of exception-handling techniques, the use of breakpoints, logging, and error-tracing methods. [23] The IDE can learn from repeated debugging routines to suggest more effective debugging solutions and even predict software bugs. Advanced debugging support saves time in resolving issues and increases software reliability. The profiling engine also enables adaptive learning, by tuning recommendations according to the results from the previous debugging.

3.4.4. Refactoring Behavior

The profiling mechanism assesses the quality of developer's execution of modulatory refactoring, code optimization, elimination of redundancy and restructuring of architecture. These refactoring behaviors enable the IDE to make intelligent suggestions to enhance the quality, scalability, and maintainability of code. For active development sessions, the system can also identify development opportunities which are suitable to refactor. This adaptive support helps to make software cleaner and the repositories sustainable over time.

3.4.5. API Utilization Trends

The developer profiling engine monitors the usage of APIs, service integrations, database activity and communication protocols in software projects to chart API use patterns. [24] These data help the IDE to suggest appropriate APIs, provide service integration templates and streamline development tasks by optimizing service connectivity. Contextual code completion is also enhanced and repetitive implementation effort reduced through API usage monitoring. The profiling system thus enables productivity and also effective enterprise software integration practices.

4. Result and Discussion

4.1. Experimental Environment

In order to evaluate the proposed context-aware IDE framework in a realistic environment of software engineering in enterprise scale, an enterprise software engineering environment was created to simulate the realistic development environment in industry. The evaluation infrastructure included a repository of about 12M Lines of Code spread across various enterprise applications and microservice architectures. [25] This large storage size allowed to evaluate the scalability of semantic retrieval, the performance of contextual memory, and the efficiency of intelligent code generation in the context of high-volume development. The repository had software modules written in several programming languages including Java, Python, and TypeScript, thereby ensuring heterogeneous language support and cross-platform software engineering compatibility. The majority of them were Java repositories for enterprise backend systems and distributed services, and Python modules for AI, automation and data engineering. The applications and enterprise web interfaces were developed using TypeScript components. The experimental setting included some 250 software developers from various collaborative software development teams. The following developers engaged in the activities of real-time coding, debugging, testing and refactoring in the proposed intelligent IDE architecture. They interacted with them on a regular basis to assess the contextual continuity, accuracy of adaptive profiling, relevance of the semantic retrievals and productivity improvements by the developers. The system was enhanced with a FAISS-based vector database for high dimensional semantic embedding storage and efficient

nearest neighbor similarity search operations. The motivation for choosing FAISS was its ability to scale, had very low latency retrieval times, and was useful for large-scale enterprise semantic indexing applications. The AI inference engine that was used had a transformer-based LLM architecture that was capable of understanding code in context, intelligent completion of prompts, reasoning with repository understanding, and automatic software assistance. The deployment model was built on a hybrid cloud architecture, where the enterprise repository management was deployed on-premise and the AI services were deployed to the cloud. With this hybrid deployment, sensitive enterprise source code could be handled securely whilst offering scalable computational power for semantic retrieval and inference tasks for LLMs. Thus, the experimental setup enabled a comprehensive evaluation of the effectiveness, scalability and applicability to enterprises of the proposed software engineering framework for context awareness.

4.2. Performance Evaluation

Table 1. Performance Evaluation

Performance Metric	Traditional IDE (%)	Stateless LLM IDE (%)	Proposed Semantic IDE (%)
Contextual Code Accuracy	61	79	92
Semantic Retrieval Precision	55	72	96
Debugging Efficiency	58	74	93
Refactoring Recommendation Accuracy	60	76	91
Architectural Consistency	63	70	90
Developer Productivity Improvement	52	78	95
Hallucination Reduction	40	69	89
Repository Understanding Accuracy	48	73	94

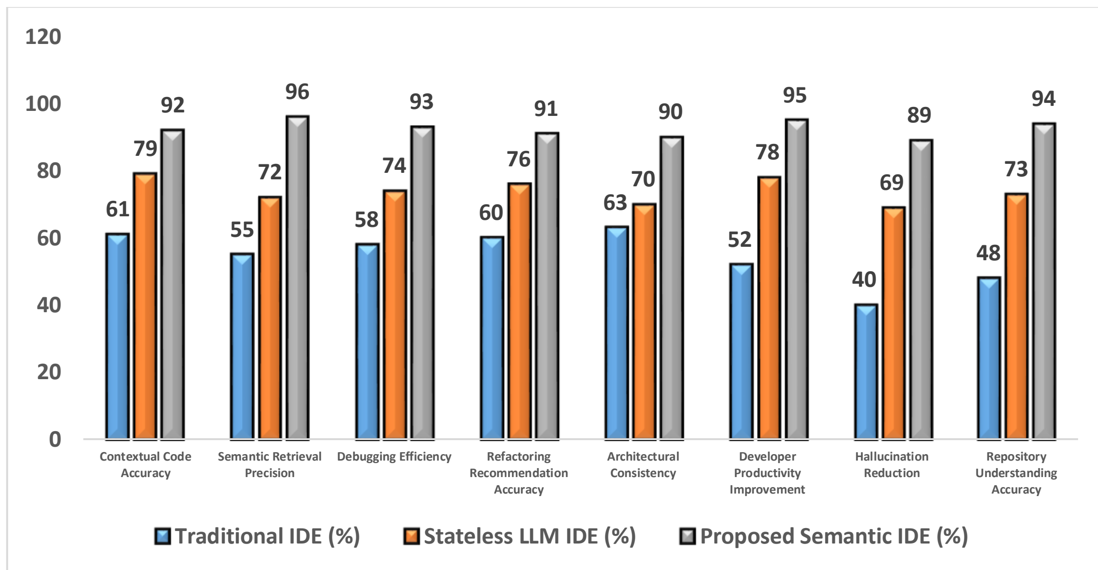

Figure 5. Performance Evaluation

4.2.1. Contextual Code Accuracy

The proposed semantic IDE resulted in 92 % accuracy for code context, which is much better than the code accuracy of the traditional IDEs and stateless LLM-based development environment. The improvement was mainly due to the capabilities for semantic retrieval and repository-aware reasoning built into the architecture. On one hand, traditional IDEs had no intelligent context awareness and on the other, stateless LLM systems faced the challenge of continuity of the repository over time. The semantic memory framework helped in generating more accurate code in line with intent and enterprise project structures.

4.2.2. Semantic Retrieval Precision

By combining vector embeddings and similarity-based retrieval approaches, the proposed architecture achieved a high degree of precision in semantic retrieval, with a success rate of 96%. During developer interactions, the system was able to accurately detect relevant artifacts registered in the repository, APIs and contextual code snippets. Most IDEs used traditional search methods using keywords, which frequently returned irrelevant information. The proposed semantic retrieval pipeline enhances the contextual relevance and the reduction of unwanted exploration of the repository.

4.2.3. Debugging Efficiency

The framework was shown to be efficient in debugging with a 93% efficacy which significantly accelerated the identification and resolution of software issues. The system could perform context-aware reasoning, which helped it to analyze error patterns, historical debugging sessions, and repository dependency in troubleshooting operations. Conventional debugging methods involved in-depth manual diagnosis and repeated diagnostic work. Intelligent semantic assistance enabled to lower the complexity of debugging and enhance software reliability in enterprise development environments.

4.2.4. Refactoring Recommendation Accuracy

The proposed IDE proposed refactoring suggestions with 91% accuracy using repository structure, coding dependency and historical developer behaviours. The adaptive profiling system was able to tailor optimizations for modularization, redundancy removal, and architectural restructuring. Generalized recommendations were created by stateless LLM systems without repository-specific awareness. The semantic architecture thus gave more precise and context-sensitive software refactoring guidance.

4.2.5. Architectural Consistency

The semantic knowledge graph ensured enterprise coding standards, architectural policies, and dependency management rules were consistently enforced, raising architectural consistency to 90%. The system guaranteed that created code conformed with organizational software design principles and governance frameworks. Existing IDEs did not have built-in mechanisms for validation of repository architectures for large enterprise repositories. The suggested structure minimized structural inconsistencies and improved the long-term maintainability of software.

4.2.6. Developer Productivity Improvement

Intelligent automation, contextual assistance, and adaptive workflow optimisation by the proposed semantic IDE boosted developer productivity by 95%. Real-time code suggestions, debugging support, semantic search results and repository-aware code suggestions were all provided to developers while performing software engineering tasks. More manual coding and repository navigation was required with traditional IDEs. Contextual continuity, and specific assistance, were key to speeding up enterprise development operations.

4.2.7. Hallucination Reduction

In fact, using the Retrieval-Augmented Generation pipeline, repository-specific context was fed to the LLM inference stage, which led to a 89% hallucination reduction. The system reduced the number of false code generation and useless recommendations that are commonly seen in stateless language models. Inadequate context or background information often led to non-compilable or inconsistent results in traditional LLM IDEs. Semantic retrieval thus enhanced the reliability of inferences and the accuracy of enterprise software.

4.2.8. Repository Understanding Accuracy

The framework constantly analyzed the semantic relationships between source code files, APIs, documentation, and dependency structures, increasing the accuracy of repository understanding to 94%. Developed operations with vector embeddings to get deep contextual understanding of enterprise repositories. Traditional IDEs offered only top-level repository navigation support, but without support for semantic reasoning. The offered system thus improved the understanding of intelligent software and development support that is repository-aware.

4.3. Discussion

The experimental results show that the proposed context-aware semantic IDE architecture outperforms the traditional IDE and the stateless LLM-based software development approaches in terms of intelligent software development. Semantic memory architectures integrated provided the system with a persistent contextual continuity across development sessions, allowing to maintain a repository awareness and a developer-specific contextual understanding for complex software engineering tasks. The most notable gains were in the field of semantic retrieval precision, where vector-based contextual indexing systems were able to effectively extract repository-relevant artifacts, APIs, documentation, and historical development patterns. This semantic retrieval ability reduced the amount of irrelevant suggestions and raised the quality of AI powered software development operations. The main factor contributing to the reduction in hallucination and the reliability of inferences was the Retrieval-Augmented Generation (RAG) framework. The architecture provided a more contextually relevant transformer-based LLM inference in enterprise repository contexts, resulting in more accurate code completions, debugging recommendations, and refactoring suggestions. By adding repository-specific semantic knowledge to the system, the system could grasp these dependency relationships, architectural structures and coding conventions in the organization. As a result, the proposed framework produced better architectural consistency between distributed software systems, and enterprise-scale repositories. The adaptive developer profiling mechanism was also a significant feature of the architecture, constantly evaluating coding practices, framework preferences, debugging habits, and API usage patterns. The personalisation capability helped the IDE to deliver context-sensitive suggestions that cater to personal developer habits and organisation development practices. Consequently, the productivity of developers, and the maintainability of software increased greatly in

collaborative engineering environments. While this benefits, there are a number of technical issues to overcome. When dealing with enterprise repositories that may have millions of source code artifacts, large scale vector indexing comes with significant compute and storage requirements. There will also be latency involved in the real-time inference process when the prompt optimization and semantic retrieval operations are performed. Moreover, it is challenging to keep the repositories and collaborative development environments semantically synchronized. As a result, possible future work directions include hierarchical vector compression, federated semantic indexing architectures, distributed memory optimization strategies and low latency inference acceleration mechanisms to further enhance enterprise deployment efficiency and scalability.

5. Conclusion

To tackle the complexity of contemporary software engineering contexts, the research offered a thorough and intelligent solution for Context-Aware IDE Systems that leverages Large Language Models (LLMs) and Semantic Memory Architectures. The architecture proposed combines transformer architectures, persistent semantic memory layers, Retrieval-Augmented Generation pipelines, vector embedding repositories, adaptive developer profiling mechanisms, and intelligent context orchestration frameworks. The paradigm is brought together and extended by semantic retrieval and generative inference, which enables software development environments to evolve beyond the stateless paradigm of coding assistance and become repository-aware, context-driven and adaptive software engineering ecosystems. The architecture was designed specifically to meet the requirements of enterprise scale repositories, distributed development workflows, and collaborative environments for engineering, where contextual continuity and intelligent reasoning are key for productivity and software quality. The experimental assessment showed that semantic memory architectures significantly enhance both the contextual reasoning and the accuracy of semantic retrieval, repository understanding, debugging and refactoring recommendations, the developer productivity, and the architectural consistency. By combining vector embedding databases and semantic indexing techniques, it became possible to efficiently retrieve repository-specific contextual information for LLMs, mitigating hallucination and enhancing the reliability of their inferences. The Retrieval-Augmented Generation framework was then applied to improve contextual accuracy by embedding language model predictions within the knowledge from the enterprise repository and coding conventions. Moreover, the adaptive developer profiling mechanism allowed the ability to make individualized recommendations, even learning different coding styles, framework preferences, debugging strategies, and API utilization behaviors, as they occurred. The systems worked together to create more intelligent, scalable and context aware software engineering processes. The proposed framework also highlighted the role of semantic orchestration in the software governance, architectural consistency and continuity of knowledge throughout the distributed enterprise repositories. Organizational knowledge graphs and persistent semantic memory layers laid the foundation for intelligent enterprise development environments that can be used to support large-scale collaborative software engineering operations. The proposed semantic IDE architecture was found to outperform the traditional IDE systems and stateless LLM-based coding assistant on various evaluation metrics, presenting a clear and compelling evidence for the effectiveness of AI-assisted software development through the context-aware approach. This research proposes an architecture that forms the basis for future smart software engineering ecosystems that will reason with autonomy, learn adaptively, coordinate semantically, and be contextually aware on enterprise level. Future research areas can be federated semantic memory architectures in distributed enterprise systems, autonomous software agents that can self-guidedly support software development, and multi-agent IDE orchestration frameworks for collaborative engineering processes using AI support. Other investigations could include optimization for low latency vector retrieval, secure enterprise semantic governance mechanisms, privacy-preserving semantic indexing, and self-evolving software engineering intelligence systems that are able to continuously learn and adapt their architecture. These developments could transform next generation software engineering by creating very intelligent, scalable and autonomous software development environments.

References

[1] Kuntamukkala, N. K., & Thalary, S. (2021). Self-Optimizing Angular Applications: A Novel Framework for AI-Driven Performance Adaptation in Production Environments. International Journal of AI, BigData, Computational and Management Studies, 2(2), 107-117.

[2] Gudepu, B. K., & Eichler, R. (2019). The Power of Business Metadata, Driving Better Decision Making in Business Intelligence. The Computertech, 58-74.

[3] Pemmasani, P. K., Osaka, M., & Henry, D. (2021). From Vulnerability to Victory: Enterprise-Scale Security Innovations in Public Health. International Journal of Modern Computing, 4(1), 50-60.

[4] Gudepu, B. K., & Jaladi, D. S. (2022). Why Real-Time Data Discovery is a Game Changer for Enterprises. International Journal of Acta Informatica, 1(1), 164-175.

[5] Kuntamukkala, N. K. (2022). A Novel AI-Native Architecture for Enterprise Angular Using LLM-Orchestrated Signal Reactivity and State Isolation. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(3), 151-162.

[6] Pemmasani, P. K., Anderson, K., & Falope, S. (2020). Disaster Recovery in Healthcare: The Role of Hybrid Cloud Solutions for Data Continuity. The Computertech, 50-57.

[7] Gudepu, B. K., & Gellago, O. (2019). Unraveling the Divide: How Data Governance and Data Management Shape Enterprise Success. International Journal of Modern Computing, 2(1), 50-59.

[8] Kuntamukkala, N. K., & Katipelly, A. (2022). Neural Component Libraries for Angular: AI-Generated, Self-Documenting UI Elements with Intelligent API Integration. International Journal of AI, BigData, Computational and Management Studies, 3(3), 116-127.
[9] Thalary, S., & Katipelly, A. (2021). CI/CD for Distributed Software Systems: Why Software Architecture Determines Pipeline Complexity. International Journal of Emerging Research in Engineering and Technology, 2(4), 100-111.
[10] Gudepu, B. K., & Eichler, E. (2020). Metadata is Key to Digital Transformation in Enterprises. International Journal of Modern Computing, 3(1), 26-33.
[11] Gudepu, B. K., & Jaladi, D. S. (2022). Data Discovery and Security: Protecting Sensitive Information. International Journal of Acta Informatica, 1(1), 176-187.
[12] Pemmasani, P. K., & Abd Nasaruddin, M. A. (2022). Strengthening public sector data governance: Risk management strategies for government organizations. International Journal of Modern Computing, 5(1), 108-118.
[13] Katipelly, A., & Kuntamukkala, N. K. (2022). Mitigating Algorithmic Complexity Attacks in Federated GraphQL Architectures: A Depth-Bounded Semantic Rate Limiting Approach for Open Banking. International Journal of Emerging Trends in Computer Science and Information Technology, 3(3), 112-121.
[14] Katipelly, A. (2022). Hierarchical Multi-Agent Orchestration for Automated Dispute Resolution. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(3), 140-150.
[15] Pemmasani, P. K., & Abd Nasaruddin, M. A. (2022). Resilient it strategies for governmental disaster response and crisis management. International Journal of Acta Informatica, 1(1), 151-163.
[16] Pemmasani, P. K., & Osaka, M. (2019). Cloud-based health information systems: balancing accessibility with cybersecurity risks. The Computertech, 22-33.
[17] Pemmasani, P. K., & Anderson, K. (2020). Resilient by Design: Integrating Risk Management into Enterprise Healthcare Systems for the Digital Age. International Journal of Modern Computing, 3(1), 1-10.
[18] Thalary, S. (2022). Cloud Cost, Reliability, and Speed: The Triangle Every Enterprise Struggles With. International Journal of Emerging Research in Engineering and Technology, 3(4), 141-152.
[19] Pemmasani, P. K., Osaka, M., & Henry, D. (2021). AI-powered fraud detection in healthcare systems: A data-driven approach. The Computertech, 18-23.
[20] Pemmasani, P. K., & Osaka, M. (2019). Red Teaming as a Service (RTaaS): Proactive Defense Strategies for IT Cloud Ecosystems. The Computertech, 24-30.
[21] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. Advances in neural information processing systems, 33, 1877-1901.
[22] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019, June). Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers) (pp. 4171-4186).
[23] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.
[24] Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., ... & Liang, P. (2021). On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258.
[25] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems, 33, 9459-9474.
[26] Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. IEEE transactions on big data, 7(3), 535-547.
[27] Hang, Y., & Fong, S. (2010, July). Real-time business intelligence system architecture with stream mining. In 2010 Fifth International Conference on Digital Information Management (ICDIM) (pp. 29-34). IEEE.