# THE IMPACT OF AGENTIC CODING ON THE MODERN SOFTWARE DEVELOPMENT LIFECYCLE

Alexandru PÎRJAN[1]
Dana-Mihaela PETROŞANU[2]

**Abstract**

The evolution of Large Language Models (LLMs) has triggered a transition from stateless code completion to "Agentic Coding", where autonomous systems engage in multi-step reasoning, planning, and environment interaction. This article studies the systemic impact of this paradigm shift on the modern Software Development Lifecycle (SDLC), contrasting traditional human-centric workflows with emerging agent-augmented architectures. By conducting an in-depth framework analysis, we examine the technical structure of coding agents, focusing on ReAct patterns, multi-agent role specialization, and self-correcting feedback loops where the compiler acts as a reviewer. The study systematically evaluates implications across the SDLC, detailing how agents accelerate requirements engineering through automated prototyping and how they are revolutionizing quality assurance via self-healing test suites. The integration of autonomous agents is not without friction as the analysis highlights a dichotomy between increased implementation velocity and the degradation of long-term maintainability. We identify extremely important risks, including security vulnerabilities arising from hallucinated dependencies and the cognitive load associated with reviewing complex, machine-generated logic. Economically, the shift suggests a complex trade-off between operational token costs and developer productivity gains. Conclusively, the study demonstrates that Agentic Coding demands a redefinition of the developer's role from syntax author to system coordinator. We consider that realizing the full potential of this technology requires rigorous new standards in formal verification and a restructuring of computer science education to prioritize architectural oversight over routine implementation.

**Keywords:** Agentic Coding, Software Development Lifecycle (SDLC), Large Language Models (LLMs), Autonomous Agents, ReAct Pattern, Human-in-the-Loop, Code Maintainability, DevOps Automation

**JEL Classification:** O3, O33, O34, O35, O36, O38

## 1. Introduction

The software engineering landscape is currently undergoing a transformation of a magnitude comparable to the advent of high-level compilers or to the introduction of object-

[1] PhD Hab. Full Professor, School of Computer Science for Business Management, Romanian-American University, 1B, Expozitiei Blvd., district 1, code 012101, Bucharest, Romania, alexandru.pirjan@rau.ro
[2] PhD Lecturer, Department of Mathematics-Informatics, National University of Science and Technology Politehnica Bucharest, 313, Splaiul Independentei, district 6, code 060042, Bucharest, Romania, dana.petrosanu@upb.ro

oriented programming. At the heart of this transformation is the evolution of LLMs [1,2], which has triggered a decisive transition from simple, stateless code completion to a new operational paradigm known as "Agentic Coding" [3–6]. In order to understand the significance of this shift, one must first distinguish between the capabilities of previous generations of AI assistance and the emerging autonomous systems. Traditional code assistants [7–9], often referred to as "Copilots", operate primarily as advanced stochastic predictors [8–10]. They are stateless entities that predict the next likely token based on a static window of preceding text [4,11]. While effective at reducing repetitive code ("boilerplate"), these systems lack an intrinsic understanding of the broader system state, the result of code execution, or the interdependencies across a complex file tree [7,12].

### 1.1. Defining Agentic Coding Beyond Autocomplete and Copilots

Agentic Coding represents a fundamental departure from this passive model. In this case, autonomous systems are capable of engaging in multi-step reasoning, strategic planning, and active interaction with the development environment [3–6]. Unlike a text-predictor that suggests a function body and ceases operation, a coding agent perceives the codebase as a mutable environment [3,4,11,13–17]. It uses specific architectural frameworks to decompose high-level instructions into discrete tasks, executes terminal commands in order to verify its outputs, and interprets compiler error messages in order to iteratively refine its solution [3,14,15,18]. This cyclical process of action and observation allows the agent to manage the development lifecycle with a degree of autonomy which had been previously reserved for human engineers.

The distinction consists in the capacity for self-correction and goal-oriented behavior [3,4,11]. Where a Copilot requires the human to identify a bug and prompt for a fix, an agentic system can run a test suite, observe the failure, hypothesize a correction, apply the patch, and re-run the test to confirm validity [7–9]. This capability relies heavily on ReAct ("Reasoning and Acting") [13,19–22] patterns and self-correcting feedback loops [23] where the compiler effectively acts as an automated reviewer [3,6,24,25]. Consequently, Agentic Coding represents a broader transformation of the development process, redefining how code is being produced, validated, and incorporated, therefore relocating decision-making from syntax-level micro-management to goal-level coordination [3,4,11].

### 1.2. The Paradigm Shift from Human-Centric to Agent-Augmented Development

The introduction of autonomous agents into the software production line leads to a profound systemic impact on the modern SDLC, requiring a reevaluation of workflows that have traditionally been strictly human-centric. We are witnessing a migration from a workflow where the human developer is the sole author and architect, using tools merely for assistance, to an agent-augmented architecture where the human acts as a supervisor or "human-in-the-loop" [3–5,11,16,26]. This sociological and operational shift redefines the primary bottleneck of software production. In the traditional model, the speed of development has been limited by the human cognitive capacity to synthesize logic into syntax. In the agent-augmented model, the constraint shifts to the human capacity to review, audit, and integrate machine-generated logic.

This integration of autonomous agents is not without friction, presenting a complex trade-off between velocity and stability. While agents can dramatically accelerate the implementation phase, creating a surge in code volume, this creates a new class of challenges regarding the long-term maintainability of the software [3,15,18]. The analysis highlights a critical dichotomy, namely the increase in implementation velocity often correlates with a degradation of maintainability, a phenomenon termed the "drift" of human context [4,13,14,18,27,28]. As agents generate vast quantities of code autonomously, the human developer's mental model of the system, the "context" begins to erode [29]. When developers are no longer the primary authors of the codebase, their intimate understanding of its structure erodes, impairing rapid debugging and long-term architectural changes.

Furthermore, this paradigm shift imposes a heavy cognitive load associated with reviewing complex, machine-generated logic, which may not always align with human readability standards. The developer's role is forcibly redefined from that of a syntax author to a system coordinator, responsible for managing multiple agents rather than writing loops and conditionals. This transition fundamentally alters the economics of development as well, suggesting a complex trade-off between operational token costs, the price of running the models and the gains in developer productivity [4,29,30]. Ultimately, the shift to Agentic Coding extends beyond just a technical upgrade, reshaping the labor structures and cognitive workflows that define software engineering.

### 1.3. Research Objectives and Scope

The primary objective of this study is to systematically evaluate the implications of Agentic Coding across the entire SDLC in order to provide a structured analysis of this emerging technology. We aim to contrast traditional human-centric workflows with emerging agent-augmented architectures in order to isolate exactly where value is being created and where risk is being introduced. An important focus of this research is to conduct an in-depth framework analysis of the technical structure of coding agents, specifically examining how ReAct patterns and multi-agent role specialization enable autonomous problem solving. We seek to understand the mechanics of self-healing test suites and how the compiler serves as a feedback mechanism for iterative refinement.

Furthermore, this article intends to critically assess the risks associated with passing code generation down to autonomous systems. We have identified extremely important risks, including security vulnerabilities arising from hallucinated dependencies where agents invent non-existent software packages up to the previously mentioned cognitive load placed on human reviewers. The scope of our research also extends to the economic viability of this approach, namely analyzing the trade-offs between the financial cost of inference (tokens) and the reduction in human hours required for routine tasks.

We consider that attaining the full potential of this technology requires more than just better models, it demands a restructuring of computer science education and the adoption of rigorous new standards in formal verification. By detailing how agents accelerate requirements engineering through automated prototyping and revolutionize quality assurance, this study aims to provide a comprehensive roadmap for industry practitioners and academic researchers as they transition to autonomous software development.

### 1.4. Structure of the Paper

The remainder of this scientific article is organized to provide a logical progression from historical context to future implications. Section 2 traces the evolution of AI in software engineering, distinguishing the capabilities of modern agents from early static analysis tools and predictive LLMs. Section 3 details the architectural framework of coding agents, exploring core components such as context windows, memory systems, and the mechanisms of tool use and environment interaction. Section 4 constitutes the core contribution of the paper, offering a granular analysis of the impact of agentic coding on each phase of the SDLC, from requirements engineering and design up to implementation, testing, and deployment. Section 5 provides a quantitative and qualitative impact analysis, discussing productivity metrics and the economic shifts inherent in this new model. Section 6 addresses the critical challenges and risks, including security vulnerabilities and the "drift" of human context. Section 7 discusses the transforming role of the human developer, while Section 8 outlines future directions such as the integration of formal verification methods. Finally, Section 9 concludes the study with a summary of key findings and recommendations for industry adoption.

## 2. The Evolution of AI in Software Engineering

The trajectory of software engineering tools has historically followed a path of increasing abstraction, moving from binary machine code to high-level languages, and eventually to complex development environments [31,32]. The software engineering field is undergoing a transformation of a magnitude comparable to the advent of high-level compilers or to the introduction of object-oriented programming. In order to understand the specific impact of "Agentic Coding", it is necessary to contextualize it within the broader history of automated development tools. This evolution encompasses a distinct progression from deterministic, rule-based heuristics to stochastic code completion, and finally, to the emerging era of autonomous agents capable of reasoning and environment interaction [11,31,32].

### 2.1. Historical Context of Static Analysis, IntelliSense, and Early Heuristics

Before the advent of probabilistic models, the automation of software development was grounded in deterministic algorithms and rigid rule sets. Early tools focused primarily on syntax verification and static analysis, operating within strictly defined boundaries [32,33]. These systems used Abstract Syntax Trees (ASTs) to parse code and identify deviations from established language standards or logical errors that could be detected without execution [24,32,33]. While these tools were very important in ensuring code quality, they were fundamentally passive. They could flag a syntax error but possessed no capability to understand the intent behind the code or to generate logic autonomously.

The introduction of "IntelliSense" and similar code-completion technologies marked a significant step forward, but they remained heuristic in nature [34]. These tools operated by indexing the codebase and offering suggestions based on type definitions and function signatures. This significantly reduced the cognitive load required for syntax recall,

effectively positioning the developer as the sole author and architect who used tools merely for assistance. In this traditional model, the speed of development was limited strictly by the human cognitive capacity to synthesize logic into syntax. The tools were unable of "hallucination" [1–4,35,36] because they were constrained by the finite set of symbols available in the project libraries, but they were equally unable of creativity or adaptation. They represented a human-centric workflow where the engineer micro-managed every line of code. This era established the foundational labor structures where the developer was responsible for writing loops and conditionals, an approach that remained largely unchallenged until the integration of statistical learning models [1–4,14,32].

### 2.2. The Rise of LLMs in Code Generation

The integration of LLMs [1,2] into the software domain triggered a decisive transition from simple, stateless code completion to more fluid generative capabilities. This phase, characterized by the proliferation of "Copilots", introduced tools that operate primarily as advanced stochastic predictors [7–9]. Unlike their rule-based predecessors, these models use the "Transformer" architecture [1,2] to analyze vast repositories of open-source code, allowing them to predict the next likely token based on a static window of preceding text [4,11]. This statistical approach allowed for the generation of entire function bodies and the reduction of repetitive "boilerplate" code.

Nevertheless, despite their generative power, these systems function as stateless entities [4,11]. A Copilot suggests a snippet of code based on immediate context but ceases operation the moment the suggestion is accepted. It lacks an intrinsic understanding of the broader system state, the result of code execution, or the interdependencies across a complex file tree. The interaction remains strictly synchronous and human-initiated, namely the human must identify a need, prompt the model, review the output, and manually integrate it. If the generated code introduces a bug, the model is unaware of the failure unless the human explicitly prompts for a fix. Consequently, while LLMs increased the velocity of syntax production, they did not alter the fundamental agency of the developer. The developer remained the active driver, while the AI served as a high-speed, although occasionally an unreliable engine. This limitation needed a shift toward systems that could maintain context and act autonomously, which led to the emergence of Agentic Coding [3,11,13,14,37,38].

### 2.3. Emergence of Autonomous Agents for Planning, Reasoning, and Execution

The current frontier of software engineering is defined by "Agentic Coding" [3–6] which represents a fundamental departure from the passive model of text prediction. In this approach, autonomous systems engage in multi-step reasoning, strategic planning, and active interaction with the development environment. Whereas a token-based completion model typically generates a snippet and terminates execution, a coding agent conceptualizes the repository as an evolving, actionable workspace mutable environment [3,4,11,13–17]. This capability is achieved through specific architectural frameworks, most notably the ReAct [13,19–22] patterns, which enable the system to decompose high-level instructions into discrete tasks.

The defining characteristic of these agents is their capacity for self-correction and goal-oriented behavior [3,4,11]. The agent interacts with the development environment by executing terminal commands for verification and by interpreting compiler errors to improve the implementation through successive iterations. This cyclical process of action and observation allows the agent to manage the development lifecycle with a degree of autonomy previously reserved for human engineers. For instance, where a "Copilot" requires the human to spot a failure, an agent can run a test suite, observe the failure, hypothesize a correction, apply the patch, and re-run the test to confirm validity.

This introduces self-correcting feedback loops [23] where the compiler effectively acts as an automated reviewer. The implication is a systemic shift in the SDLC, detailing how agents accelerate requirements engineering through automated prototyping and revolutionize quality assurance [39–41] via self-healing test suites [3–5,12,14]. Nevertheless, this autonomy introduces new risks, such as the "drift" of human context, where the human developer's mental model of the system begins to erode as agents generate vast quantities of code [4,13,14,18,27,28]. The developer is no longer the author of every line and therefore loses the intimate familiarity required for rapid debugging. Consequently, the emergence of agents demands a redefinition of the developer's role from syntax author to system coordinator.

## 2.4 Comparative analysis between Assisted Coding and Agentic Coding

In order to fully appreciate the implications of this technological shift, it is very important to distinguish between the capabilities of previous generations of AI assistance and emerging autonomous systems (Figure 1).

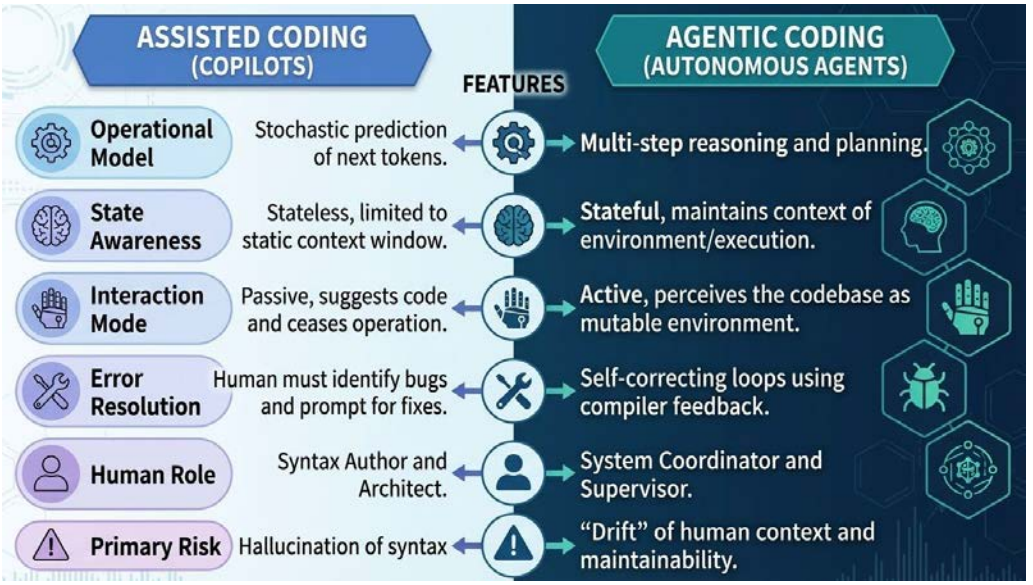


Figure 1: Structural comparison between Assisted Coding and Agentic Coding

This transition reflects an expansion in operational scope and autonomy, alongside improvements in accuracy. While assisted coding focuses on increasing the speed of typing and syntax retrieval, "Agentic Coding" relocates decision-making from syntax-level micro-management to goal-level coordination [17,38]. This distinction fundamentally changes the economics of development, suggesting a complex trade-off between operational token costs and developer productivity gains [4,29,30].

## 3. Architectural Framework of Coding Agents

The transition from stochastic text prediction to autonomous software engineering requires a complex architectural restructuring that extends far beyond the capabilities of the underlying LLM [1,2]. While the LLM serves as the cognitive engine, providing the reasoning capabilities and semantic understanding of code, it is the surrounding architectural framework that enables agentic behavior. This framework transforms a passive request-response system into a stateful, goal-oriented [3,4,11] entity capable of manipulating the software environment. In this section, we analyze the elements of a coding agent, examining the mechanisms that allow for long-term memory retention [3,4,11,29], active environment interaction [3,4,11,13–17], multi-agent collaboration [3,4,11,14,16–18,26], and the extremely important feedback loops [23] that drive iterative self-correction.

### 3.1. Core Components of Coding Agents (LLMs, Context Windows, and Memory Systems)

At the nucleus of the agentic framework lies the LLM [1,2], which functions less as a database of knowledge and more as a reasoning engine capable of syntactic synthesis and logical deduction. Nevertheless, the efficacy of the LLM in a software development context is fundamentally constrained by its context window, namely the finite limit of tokens the model can process at any single instant. In complex software projects, which often span hundreds of files and millions of lines of code, the entirety of the system state cannot fit within this immediate operational memory [29]. Consequently, the architecture of a coding agent must implement robust memory systems that mitigate the limitations of the model's stateless operation while supporting the long-running requirements of the development lifecycle.

In order to overcome the transient nature of the context window, agentic architectures use a dual-memory approach comprising short-term working memory and long-term archival storage [3,4,11,29]. Working memory is strictly managed to contain the immediate problem context, such as the active file being edited, relevant function definitions, and the current set of instructions. Because the model predicts tokens based on a static window of preceding text [4,11], optimizing this window is extremely important for maintaining logical coherence. If irrelevant data saturates the window, the model suffers from "distraction" leading to hallucinations [1–4,35,36] or loss of the original directive. Therefore, agents employ dynamic context pruning, selectively summarizing or discarding information that is no longer strictly relevant to the immediate reasoning step [3,4,11,17,18].

Long-term memory is typically implemented through Vector Databases and Retrieval-Augmented Generation (RAG) [8,42] techniques. Instead of attempting to load the entire codebase into the context window, the agent generates vector embeddings, namely numerical representations of semantic meaning, for every function, class, and documentation file in the project. When tasked with a modification, the agent queries this vector database in order to retrieve only the snippets of code that are semantically relevant to the current objective. This allows the agent to maintain an understanding of system-wide interdependencies without being overwhelmed by raw data. By decoupling the reasoning engine from the storage mechanism, the architecture grants the agent a form of infinite pseudo-memory [3,4,11,29], enabling it to respect architectural patterns established in modules that had been created months prior. This stateful awareness is the primary differentiator between a standard Copilot, which sees only the opened file, and an agent that "understands" the broader repository [3,4,7–9,16–18,26].

### 3.2. Tool Use and Environment Interaction (IDEs, CLIs, Version Control)

While memory systems [3,4,11,29] provide the context required for reasoning, it is the capacity for tool use and environment interaction that transforms that reasoning into tangible software artifacts. Within an agentic coding approach, the codebase functions as a modifiable environment [3,4,11,13–17] accessed via specific interfaces, not merely as a fixed text artifact. The architecture defines a schema of "tools", namely executable functions that the LLM can invoke to perform actions outside its neural network [3–6]. These tools provide an interface between Natural Language Processing [43]and the deterministic operating system, allowing the agent to function as a digital operator.

The most fundamental interaction capability is File I/O [3,4], which grants the agent the permission to read directory structures, open files, and write changes to disk. Nevertheless, advanced agentic frameworks go beyond simple text manipulation by integrating directly with the Command Line Interface (CLI) [3,4,7–9,16–18,26]. This allows the agent to execute terminal commands, a very important capability for verifying the validity of its own work. For instance, an agent acts by creating a new branch in the version control system, installing dependencies via a package manager, or running a build script to check for compilation errors. This interaction is bidirectional, namely the agent issues a command, and the environment returns the standard output (stdout) or standard error (stderr), which the agent then processes as new observation data [3,13].

Furthermore, complex agents integrate with the Language Server Protocol (LSP) [4,5,18,44], the same technology that powers modern Integrated Development Environments (IDEs). By querying the LSP, an agent can request the definition of a symbol, find all references to a function, or view type signatures without needing to "read" every file textually. This mimics the workflow of a human developer who goes through a codebase using "Go to Definition" features rather than scrolling linearly. This structural understanding is extremely important for refactoring tasks, where changing a variable name in one location requires precise updates across multiple dependencies. By leveraging these tools, the agent moves beyond the passive suggestion of code and engages in active construction, managing the environment with a degree of autonomy specific for human

engineers. The environment essentially becomes the "world" in which the agent operates, and the tools are the effectors through which it exerts change.

### 3.3. Multi-Agent Systems' Specialized Roles (Architect, Coder, Reviewer)

As the complexity of software tasks increases, the performance of a single monolithic agent tends to degrade. The cognitive load required to simultaneously maintain the high-level architectural vision, write syntactically correct code, and audit for security vulnerabilities often leads to context drift and hallucination [4,13,14,18,27,28]. In order to mitigate this, modern agentic frameworks employ Multi-Agent Systems (MAS) [3,4,11,14,16–18,26], where the workload is decomposed and distributed across specialized synthetic personas. This architecture mimics the structure of a human software development team, assigning distinct roles and responsibilities to different instances of the model.

In a typical multi-agent configuration [3,4,11,14,16–18,26], a high-level "Architect" agent is responsible for the initial decomposition of the user's prompt. Rather than producing implementation code, it decomposes the feature request into discrete, logical steps and specifies the interfaces between them. This plan is then passed to a "Coder" agent, whose sole focus is the implementation of the specific sub-task within the constraints defined by the Architect. By narrowing the scope of the Coder agent, the architecture reduces the probability of error and ensures that the generated code adheres to the project's established patterns.

Extremely important, the workflow includes a "Reviewer" or "Critic" agent that acts as a quality gate [3,4,11,14,16–18,26]. After the "Coder" submits a solution, the "Reviewer" evaluates it for syntactic correctness, logical validity, security issues, and compliance with the planned design. This "Reviewer" operates with a different system prompt, one primed to be skeptical and analytical rather than generative. If the "Reviewer" detects a potential issue, such as a hallucinated dependency [1–4,35,36] or a violation of the "Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion" (SOLID) [45–47] principles, it rejects the submission and provides feedback to the "Coder", triggering a refinement loop. This separation of concerns prevents the "blind spots" that occur when a single entity reviews its own work. The human developer, in this scenario, shifts from being the primary author to being the "System Coordinator", overseeing the interaction between these agents and intervening only when the consensus between them breaks down. This role specialization ensures that while one agent focuses on the details of syntax, another retains the "context" of the broader system, preventing the erosion of architectural integrity [4,13,14,18,27,28].

### 3.4. Feedback Loops for Self-Correction and Iterative Refinement

The defining characteristic of Agentic Coding, which distinguishes it sharply from the stochastic prediction of Copilots, is the implementation of rigorous feedback loops [3–6,23]. A Copilot operates in an open loop, it suggests code, and if that code is flawed, the system remains unaware unless a human explicitly intervenes. In contrast, autonomous agents operate in a closed loop driven by the ReAct [13,19–22] pattern, where the output

of an action is immediately observed and evaluated in order to inform the next step. This cyclical process is what enables self-correction and iterative refinement.

The most potent feedback mechanism in this framework is the usage of the compiler or interpreter as an objective critic. When an agent generates code it attempts to compile or execute it within a sandboxed environment. If the compiler returns an error, the agent captures the error message, parses it to understand the nature of the failure (e.g., a type mismatch or a syntax error), and uses this data to hypothesize a correction. This "Compiler-as-Critic" loop allows the agent to fix low-level issues autonomously [3,4,11,14,16–18,26], ensuring that the code presented to the human is, at the very least, syntactically valid.

Beyond compilation, agents are increasingly capable of generating and running their own test suites. Upon implementing a feature, the agent writes a corresponding unit test, executes it, and observes the result [3,5,11,24]. If the test fails, the agent engages in a diagnostic process, it determines whether the implementation is incorrect or if the test case itself is flawed. It then applies a patch and re-runs the test to confirm validity. This capability creates a self-healing system [3–5,12,14] where the software is continuously validated against its functional requirements during the generation process. By integrating these feedback loops [23], the architectural framework shifts the burden of error resolution from the human to the machine, redefining the development workflow from manual debugging to supervisory approval. This iterative capacity is the technical foundation that supports the broader systemic impact of agents on the SDLC, enabling the shift from human-centric authorship to agent-augmented coordination.

## 4. Impact on the SDLC

The integration of autonomous agents into the software production line leads to a profound systemic impact on the modern SDLC, calling for a systematic review of workflows that have, until now, been predominantly human-driven. As outlined in the introductory analysis, this technology affects more than a localized efficiency boost by fundamentally changing the labor structures and cognitive workflows that define software engineering. The transition from a workflow where the human developer is the sole author and architect to an agent-augmented architecture affects every phase of production, from the initial elicitation of requirements to the final operational deployment. This section provides a granular analysis of these shifts, contrasting traditional methodologies with emerging agentic behaviors to isolate exactly where value is being created and where risk is being introduced.

### 4.1. Requirements Engineering and Planning

The initial phase of the SDLC, typically characterized by high ambiguity and human-to-human negotiation, is undergoing a significant transformation through the application of Agentic Coding [5,14,16]. Traditionally, requirements engineering has been a manual process of translation, where business logic is slowly converted into technical specifications through iterative meetings and documentation [14,32]. The introduction of agents

accelerates this phase by automating the prototyping of requirements, allowing for a more dynamic interaction between abstract intent and concrete specification.

In the agent-augmented model, the "Architect" agent plays an extremely important role in in translating user prompts into clear, actionable technical requirements. Unlike traditional static analysis tools that require formal syntax [24,32,33], agentic systems possess the semantic understanding necessary to decompose high-level instructions into discrete, logical tasks. When a stakeholder provides a vague functional description, the agent records it and very important, it actively interrogates the request by generating a preliminary requirements document. This process relies on the multi-step reasoning capabilities inherent in the ReAct [13,19–22] pattern, enabling the system to identify missing parameters or logical inconsistencies within the initial prompt.

Consequently, the role of the human operator shifts from writing the specification to reviewing the agent's interpretation of the business need. This rapid feedback loop [3–6,23] ensures that architectural misunderstandings are caught early in the lifecycle, well before implementation begins. The agent leverages its training on vast repositories of open-source code to suggest standard compliance requirements and security protocols that a human author might overlook during the initial drafting. By converting natural language directly into structured technical schemas, agents reduce the cognitive load associated with the "blank page" problem [32,48,49], allowing engineers to refine a generated baseline rather than constructing specifications from scratch.

Beyond sheer text generation, autonomous agents contribute to the planning phase by performing preliminary feasibility analyses. Because these agents perceive the codebase as a mutable environment [3,4,11,13–17] and can access file trees and dependency graphs, they can instantly cross-reference a new feature request against the existing system architecture. If a proposed requirement contradicts an established design pattern or requires a dependency that introduces known security vulnerabilities, the agent can flag this risk immediately.

This capability transforms technical roadmapping from a speculative exercise into a data-driven process. The agent can simulate the implementation path, effectively identifying the "interfaces between them" and planning the necessary modifications to the file structure. This predictive capacity is supported by the agent's ability to retrieve semantically relevant code snippets via Vector Databases [8,42], ensuring that the planning process respects the architectural patterns established in modules created months prior. Therefore, the feasibility analysis becomes a continuous, automated background process rather than a distinct milestone, allowing for a more agile response to changing business requirements.

### 4.2. Design and Architecture

The design phase has historically been the domain of senior human engineers, relying heavily on intuition and experience. The emergence of multi-agent systems [3,4,11,14,16–18,26] with specialized roles, such as the "Architect" and "Reviewer", suggests that even high-level design decisions can be augmented by artificial intelligence.

One of the most immediate impacts of Agentic Coding on the design phase is the collapse of the distinction between design and prototyping [3–6,14]. In a traditional workflow,

creating a prototype is a labor-intensive task that often results in "throwaway" code. In contrast, an agent can generate a functional prototype by decomposing the feature request into sub-tasks and executing them. This allows stakeholders to interact with a working model of the software almost immediately after the requirements have been defined.

This acceleration is facilitated by the agent's ability to handle "boilerplate" code and standard architectural setups, freeing the human architect to focus on novel logic. The agent acts as a force multiplier, rapidly instantiating the initial structure of the application based on best practices retrieved from its training data. This capability allows for "exploratory coding", where multiple architectural approaches can be implemented and compared in parallel [4,14,16,18,21,26], a luxury that time constraints rarely permit in human-only teams. The result is a design process that is empirical rather than theoretical, namely decisions are based on the observed performance of generated prototypes rather than abstract diagrams.

While agents are capable of generating code, the "Architect" agent is specifically designed to maintain the high-level vision and prevent the erosion of architectural integrity [4,13,14,18,27,28]. When a human developer considers a structural change, the agent can serve as a complex decision support system. By analyzing the entire repository through its long-term memory systems [3,4,11,29], the agent can predict the ripple effects of a proposed architectural change across the complex file tree.

This systemic awareness addresses the "drift" of human context [4,13,14,18,27,28], a critical risk where the developer loses the intimate familiarity required for architectural evolution. The agent provides a counter-balance by surfacing relevant dependencies and enforcing "SOLID" [45–47] principles during the design review. It essentially acts as a guardian of the system's structural health, offering recommendations to refactor tightly coupled components or to introduce abstraction layers where necessary. This collaboration allows the human engineer to operate at a higher level of abstraction, coordinating the system's evolution while the agent manages the granular details of component interaction.

### 4.3. Implementation (Coding)

The implementation phase is where the impact of Agentic Coding is most visible and arguably most disruptive. The transition from stateless code completion to stateful, goal-oriented agents fundamentally alters the economics and mechanics of writing code [4,29,30].

The introduction of agents creates a surge in code volume, dramatically accelerating the implementation phase [3,13,14]. Agents can generate vast quantities of code autonomously, effectively removing the human typing speed as a limiting factor in production. Nevertheless, this increase in velocity presents a contrast, namely while the output in terms of "Lines of Code" (LOC) increases, the metric of "throughput" (defined as the delivery of verified, valuable features) becomes constrained by the human's capacity to review the code.

The bottleneck of software production shifts from the cognitive capacity to synthesize logic into syntax to the capacity to audit machine-generated logic. As agents are handling the routine implementation of loops, conditionals, and standard algorithms, the human

developer is being inundated with code that they did not write. This aspect needs a shift in performance metrics from raw code generation to the efficiency of the review and integration process. The sheer volume of produced code can lead to a bloated codebase [50,51] if not rigorously managed, highlighting the trade-off between operational token costs and developer productivity gains.

A very important challenge in the implementation phase consists in the difference between syntactic correctness and semantic validity. Agents, using the "Compiler-as-Critic" loop [3,4,11,14,16–18,26], are highly effective at producing code that compiles and runs without throwing syntax errors. They can parse error messages and hypothesize corrections autonomously. Nevertheless, syntactic perfection does not equate to semantic accuracy.

An agent may generate a function that is mathematically sound and bug-free but completely misaligns with the broader business intent or with the existing domain logic. This is a manifestation of the "drift" of human context, where the machine's localized reasoning diverges from the global system goal [4,13,14,18,27,28]. The "Reviewer" agent attempts to mitigate this by evaluating logic against the planned design, but the ultimate responsibility remains with the human-in-the-loop to ensure that the code "means" what it is supposed to mean [3,4,11,14,16–18,26]. The cognitive load associated with reviewing this complex, machine-generated logic is significant, as understanding code written by another entity (synthetic or human) is often harder than writing it oneself.

One of the most promising applications of Agentic Coding in the implementation phase is the refactoring of legacy systems. Legacy codebases are often characterized by low readability and high complexity, making them difficult for humans to modify without introducing regressions. Agents, equipped with vector embeddings of the entire project [8,42], can analyze these opaque systems to identify redundancy and inefficiency.

By integrating with the Language Server Protocol (LSP) [4,5,18,44], agents can perform precise refactoring operations, such as renaming variables across multiple files or extracting methods to improve modularity. Unlike a human who might miss a reference, the agent uses the deterministic tools of the development environment [3,4,11,13–17] to ensure comprehensive updates. This capability allows organizations to modernize technical debt [32,52] at a scale that would be cost-prohibitive with human labor alone. The agent effectively "reads" the legacy code, constructs a conceptual model of its functionality, and rewrites it to adhere to modern standards, all while verifying the changes through the execution of terminal commands.

### 4.4. Testing and Quality Assurance

The domain of Quality Assurance (QA) [39–41] represents an area where Agentic Coding offers significant technical advancements. The ReAct [13,19–22] pattern integrates testing into the generative process rather than leaving it as a separate verification stage.

In the agent-augmented SDLC, the generation of code is inextricably linked to the generation of tests. When a "Coder" agent implements a feature, it is simultaneously tasked with writing the corresponding unit and integration tests [3,4,11,14,16–18,26]. This ensures that every line of machine-generated code is immediately subjected to verification. The

agent analyzes the control flow of the generated function, identifying edge cases and boundary conditions that require coverage [3,4,28].

This autonomous generation extends to the creation of mock objects and test fixtures, tasks that are typically tedious for human developers. By querying the vector database [8,42] for existing test patterns, the agent ensures that new tests maintain consistency with the project's testing strategy. This anticipatory approach to QA [39–41] shifts the culture from "testing quality in" at the end of the cycle to building it in during the implementation, effectively preventing the accumulation of untested logic.

A defining characteristic of agentic systems is the capability for self-healing test suites [3–5,12,14]. In traditional development, a change in the codebase often breaks existing tests, requiring manual intervention to update the test logic. An autonomous agent, however, can observe a test failure, analyze the assertion error, and determine whether the failure is due to a bug in the application or a mismatch in the test expectation.

If the test is outdated, the agent can rewrite the test script to align with the new functionality. If the application code is at fault, the agent enters a self-correcting loop, namely it hypothesizes a fix, applies the patch, and re-runs the test to confirm validity [3,4,11,14,16–18,26]. This "closed loop" operation creates a robust defense against regression [4,23,28], allowing the test suite to evolve organically alongside the application. It significantly reduces the maintenance burden of QA automation [39–41], which is often a primary cause of test abandonment in legacy projects.

Beyond unit testing, agents are revolutionizing the debugging process through automated reproduction. When a bug is reported, an agent can be tasked with creating a reproduction script that isolates the failure. By interacting with the CLI and analyzing the standard error (stderr) output, the agent can iterate on inputs until the crash or logical error is replicated [3,4,11,14,16–18,26].

This capability is particularly valuable for complex, state-dependent bugs that are difficult for humans to trace. Once the bug is reproduced and captured in a test case, the agent can leverage its reasoning engine to identify the root cause within the file tree. The agent effectively automates the scientific method of debugging through observation, hypothesis, experimentation, and verification, therefore freeing the human developer to focus on high-level system stability rather than transient error resolution.

### 4.5. Deployment and Operations

The final stage of the SDLC, deployment and operations, benefits from the agent's ability to interact directly with the development environment and infrastructure tools.

Agents treat infrastructure configuration with the same rigor as application code [53,54]. By generating and managing "Infrastructure as Code" (IaC) scripts (e.g., Terraform or Kubernetes manifests) [4,5,14,16], agents can provision resources and configure environments autonomously. The "Architect" agent can specify the necessary infrastructure based on the application's requirements, and the "Coder" agent can generate the corresponding configuration files.

This automation reduces the risk of configuration drift, as the agent ensures that the deployed environment matches the defined specifications [4,13,14,18,27,28]. Furthermore, agents can validate these configurations by executing "dry runs" or validation commands via the terminal, ensuring that syntax errors in the infrastructure definitions are caught before deployment. This extends the "Compiler-as-Critic" approach [3,4,11,14,16–18,26] to the domain of operations, creating a safer and more predictable deployment pipeline.

In the context of Continuous Integration and Continuous Deployment (CI/CD) [53–56], agents act as intelligent operators of the pipeline. They can analyze build logs to diagnose failure causes, distinguishing between transient network issues and genuine code defects. Upon detecting a build failure, the agent can automatically revert the respective commit or attempt a "fix-forward" strategy by applying a patch and re-triggering the pipeline.

This capability facilitates a "human-on-the-loop" model for "DevOps", where the agent manages the routine flow of code from commit to production, alerting the human supervisor only when a critical threshold of uncertainty is reached [3,4,11,14,16–18,26]. By optimizing the feedback loop between the build system and the development environment, agents reduce the cycle time of software delivery, achieving the ultimate goal of the modern SDLC, specifically the rapid, reliable, and continuous delivery of value.

## 5. Quantitative and Qualitative Impact Analysis

The structural transformation of the SDLC, as detailed in the preceding sections, needs a rigorous in-depth analysis of how value, efficiency, and quality are measured in software engineering. The migration from a workflow where the human developer is the sole author to an agent-augmented architecture creates a fundamental discontinuity in traditional metrics. While the architectural framework of coding agents, characterized by ReAct [13,19–22] patterns and self-healing mechanisms [3–5,12,14], promises increased velocity, it simultaneously introduces complex variables regarding cognitive load and economic viability.

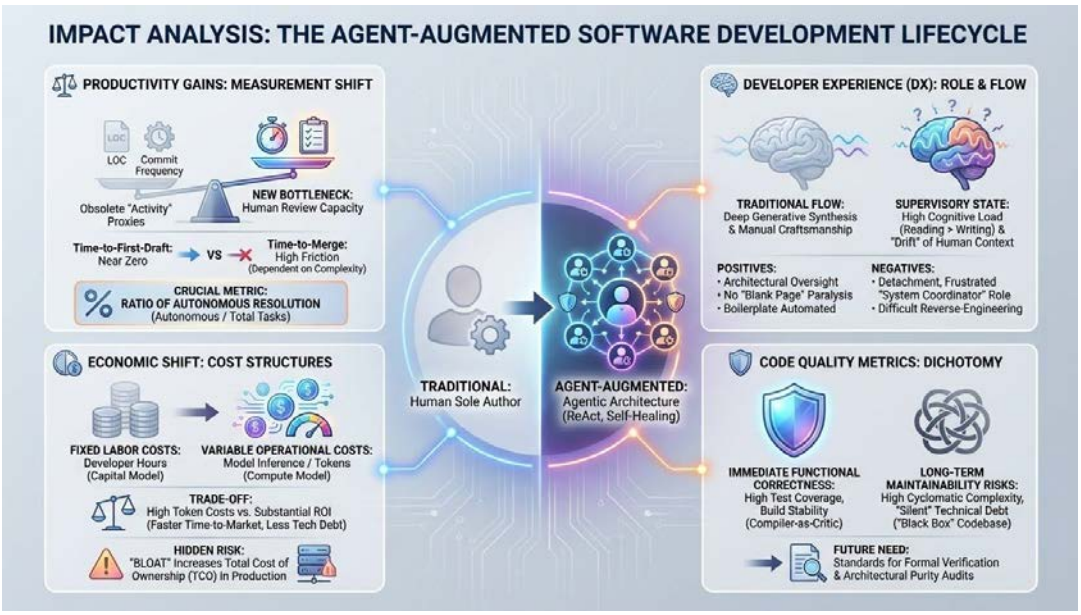


Figure 2: The Impact Analysis of the Agent-Augmented SDLC

This section performs a granular analysis of these impacts, moving beyond the superficial observation of speed in order to evaluate the deeper quantitative and qualitative shifts in productivity, developer experience, economic structures, and long-term code maintainability (Figure 2).

### 5.1. Productivity Gains Measurement Challenges and Empirical Data

The integration of Agentic Coding renders many traditional productivity metrics obsolete, requiring a shift toward multidimensional frameworks such as the "Satisfaction, Performance, Activity, Communication, and Efficiency" ("SPACE") model [3,16,18,26,57,58]. Historically, metrics like "Lines of Code" (LOC) or commit frequency served as proxies for developer activity. Nevertheless, in an agent-augmented model, these metrics become decoupled from actual value delivery. As established in our analysis of the implementation phase, agents can generate vast quantities of code autonomously, effectively removing human typing speed as a limiting factor. Consequently, the "Activity" dimension of the "SPACE" framework [3,16,18,26,57] may show exponential growth without a corollary increase in "Performance" or "Efficiency". If an agent generates two thousand lines of repetitive "boilerplate" code in seconds, the raw volume suggests high productivity, but if that code requires significant human auditing, the actual throughput of verified features may stagnate.

The primary bottleneck of software production has shifted from the cognitive capacity to synthesize logic into syntax to the human capacity to review and audit machine-generated logic [29,59]. Therefore, quantitative analysis must focus on "cycle time" and "change

failure rate" rather than volume [60]. The inclusion of the "Reviewer" agent in the multi-agent architecture [3,4,11,14,16–18,26] aims to mitigate this by filtering syntactic errors before they reach the human supervisor. Nevertheless, empirical observations suggest that while the "time-to-first-draft" is reduced to near zero, the "time-to-merge" remains dependent on the complexity of the generated logic [61]. True productivity gains are achieved only when the agent's output reliability exceeds the human cost of verification. If the friction of reviewing and debugging agent-generated code exceeds the effort of writing it manually, the net productivity gain turns negative. Therefore, the meaningful metric in this new paradigm is the ratio of "autonomous resolution" where the agent successfully plans, codes, and verifies a task without human intervention to the total number of tasks assigned.

### 5.2. Developer Experience (DX) under Cognitive Load and "Flow" State

Qualitatively, the shift to Agentic Coding fundamentally alters the psychological experience of software development, presenting a distinct trade-off between the alleviation of routine tedium and the imposition of high-level cognitive strain. The traditional state of "flow" in programming is often achieved through the deep, uninterrupted synthesis of conceptual models into code. In the agent-augmented workflow, this generative flow is frequently interrupted by the necessity of assuming the role of a "System Coordinator" or of a supervisor. The developer is no longer the author familiar with every line, which leads to a phenomenon called the "drift" of human context [4,13,14,18,27,28]. This drift creates a sense of detachment, where the engineer must constantly expend mental energy to reconstruct the logic behind the code that they did not write in order to approve it or debug it.

This transition imposes a heavy cognitive load associated with reviewing complex, machine-generated logic, which may not always align with human readability standards. Reading code is inherently more difficult than writing code, as it requires reverse-engineering the intent of the author, in this case, a synthetic entity. Nevertheless, the qualitative impact is not entirely negative. Agents significantly reduce the "blank page" situation [32,48,49] and the cognitive burden of syntax recall by handling "boilerplate" code and standard algorithms. By converting natural language directly into structured technical schemas, agents allow engineers to operate at a higher level of abstraction, focusing on architectural oversight rather than implementation details. The Developer Experience (DX) [59,62,63] therefore bifurcates, namely the satisfaction derived from manual craftsmanship diminishes, but the frustration associated with repetitive tasks and syntax errors is virtually eliminated. The challenge for the industry is to redesign the Integrated Development Environment (IDE) to support this new "supervisory" flow, potentially using visualization tools to help the human maintain the conceptual model of the system despite the erosion of the authorship context [4,13,14,18,27,28].

### 5.3. The Economic Shift involving Cost Reduction and Operational Overhead

The economic implications of Agentic Coding represent a complex trade-off between the fixed costs of human labor and the variable operational costs of model inference. The

paradigm shift indicates a move from a labor-intensive capital model to a compute-intensive one [30,31]. In the traditional model, the cost of development is primarily a function of developer hours. In the agent-augmented model, this is supplemented by the operational token costs required to run LLMs. Every iteration of the ReAct [13,19–22] pattern, namely every step of reasoning, planning, tool use, and self-correction consumes tokens. As agents engage in multi-step reasoning and execute self-healing test suites [3–5,12,14], the inference cost per feature scales with the complexity of the task.

Organizations must weigh these token costs against the gains in developer productivity. If an agent can autonomously refactor a legacy module in minutes, specifically a task that might have taken a human team weeks, then the return on investment is substantial, even with high inference costs. Furthermore, the economic model is impacted by the reduction in technical debt [32,52] and the acceleration of time-to-market. The capability of agents to automate requirements engineering and generate rapid prototypes allows for faster validation of business logic, potentially saving significant capital that would otherwise be spent on developing non-viable features. Nevertheless, a hidden economic risk exists in the potential for "bloat" [50,51]. If agents generate inefficient code that passes tests but consumes excessive computational resources in production, the long-term operational costs of the software itself may rise. Therefore, the economic viability of Agentic Coding depends on the cost of code generation and even more on the total cost of ownership (TCO) of the agent-generated systems, requiring a strict accounting of both the inference bill and the maintenance overhead of the resulting software artifacts.

### 5.4. Code Quality Metrics for Cyclomatic Complexity, Maintainability, and Technical Debt

The impact of autonomous agents on code quality metrics presents a constrast between immediate functional correctness and long-term maintainability. Agents, leveraging the "Compiler-as-Critic" feedback loop [3,4,11,14,16–18,26], are highly effective at producing code that is syntactically valid and passes unit tests. Consequently, metrics related to build stability and test coverage often improve in agent-augmented projects. Agents can autonomously generate edge cases and boundary condition tests that humans frequently overlook, leading to a more robust defense against regressions.

Nevertheless, functional validity does not guarantee structural elegance. A significant risk identified in our framework analysis is the degradation of long-term maintainability due to the "drift" of human context [4,13,14,18,27,28]. Agents may solve problems by generating convoluted logic with high cyclomatic complexity [64], provided it satisfies the immediate test constraints. Unlike human developers who often optimize for readability and future extensibility, an agent without specific architectural constraints may prioritize the path of least resistance to a green test. This can lead to an accumulation of "silent" technical debt [32,52], where the codebase becomes a "black box" of machine-generated logic that is difficult for humans to modify or refactor later. The "Reviewer" agent plays an extremely important role in enforcing "SOLID" [45–47] principles and rejecting solutions that violate established design patterns, but the risk remains that the sheer volume of generated code will overwhelm the human capacity to audit for architectural integrity. Ultimately, while defect density may decrease, the complexity density of the codebase risks increasing,

requiring new standards in formal verification in order to ensure that the software remains manageable over its lifecycle.

## 6. Challenges and Risks

The integration of autonomous agents into the software production line leads to a profound systemic impact on the modern SDLC, necessitating a reassessment of workflows that have historically been entirely human-centered. While the preceding analysis demonstrated how Agentic Coding can accelerate requirements engineering and revolutionize quality assurance, this transition is not without any risks. The shift from a labor-intensive capital model to a compute-intensive one introduces a new spectrum of risks that extends beyond simple syntax errors. As organizations migrate to an agent-augmented architecture where the human acts merely as a supervisor, they expose themselves to security vulnerabilities arising from the probabilistic nature of LLMs [1,2], legal ambiguities regarding intellectual property, and a fundamental degradation of the human developer's understanding of their own systems. This section of the scientific paper assesses these challenges, isolating the specific risks introduced when decision-making is relocated from syntax-level micro-management to goal-level coordination [3,4,11].

### 6.1. Security Vulnerabilities Arising from Hallucinations and Dependency Injection

The most immediate and extremely important risk associated with Agentic Coding arises from the inherent operational nature of the underlying models. Because LLMs [1,2] operate as advanced stochastic predictors rather than deterministic databases, they are prone to a phenomenon known as hallucination [1–4,35,36]. In the context of creative writing, a hallucination [1–4,35,36] might be a factual error, but within the strict syntax of software engineering, hallucinations often manifest as the invention of non-existent software packages or libraries. This tendency creates a severe security vector known as software supply chain attacks via dependency confusion [11,28].

When an autonomous agent is tasked with solving a complex problem, it decomposes high-level instructions into discrete tasks and attempts to import the necessary tools to execute them. If the agent's training data contains references to deprecated libraries, or if it logically infers that a library named in a specific format should exist in order to solve the immediate problem, it may hallucinate [1–4,35,36] a dependency name and insert it into the project's configuration files. Attackers can exploit this behavior by predicting common hallucinated package names and registering them on public repositories like npm or PyPI with malicious payloads [28,65,66]. Consequently, when the agent executes terminal commands to install dependencies, it inadvertently pulls malicious code into the secure development environment. Unlike a human developer who verifies the provenance of a library, the agent prioritizes the semantic probability of the package name over its verified existence.

Furthermore, the "Reviewer" agent, while designed to act as a quality gate, often struggles to identify these vulnerabilities because the hallucinated [1–4,35,36] code is syntactically correct. A line of code importing a malicious library looks identical to a legitimate import

to a reviewer focused on checking for logical validity rather than repository integrity. This creates a scenario where the automated prototyping of requirements serves as a rapid injection mechanism for vulnerabilities. The risk is compounded by the fact that agents can generate vast quantities of code autonomously, potentially burying these malicious dependencies deep within the file tree where they escape casual human audit. Therefore, the reliance on agents needs a move beyond standard code review to rigorous, automated dependency verification that is decoupled from the generative model's logic.

### 6.2. Intellectual Property (IP) and Licensing Concerns

The economic shift suggested by Agentic Coding involves a complex trade-off between operational token costs and developer productivity gains, but this economic model is threatened by unresolved legal questions regarding Intellectual Property (IP) [67,68]. The fundamental capability of these agents relies on their training on vast repositories of open-source code [28,65,66], which allows them to predict tokens and generate function bodies. Nevertheless, this training process ingests millions of lines of code governed by diverse licensing structures, ranging from permissive licenses like MIT and Apache to restrictive copyleft licenses like the GNU General Public License (GPL) [67,68].

A significant risk arises from the phenomenon of "regurgitation", where an agent, prompted with a specific context, reproduces snippets of training data near verbatim [28,65,66]. If an autonomous agent generates a solution that includes code derived from a GPL-licensed repository and integrates it into a proprietary commercial codebase, it effectively contaminates the licensing of the entire project. Because the agent perceives the codebase as a mutable environment [3,4,11,13–17] and operates without a legal conscience, it does not distinguish between code that is structurally useful and code that is legally compatible. This stochastic generation makes tracking the provenance of the code nearly impossible. Unlike a human developer who consciously copies code from a specific source, the agent synthesizes logic from a high-dimensional vector space, rendering the origin of any specific snippet opaque.

This opacity creates a "Black Box" liability for organizations [67]. If a company releases software generated by agents, they may inadvertently infringe on third-party IP rights, leading to litigation or the forced disclosure of proprietary source code. Furthermore, the question of ownership regarding agent-generated code remains legally ambiguous [68]. As the developer's role is redefined from syntax author to system coordinator, the threshold of human contribution required to claim copyright becomes blurred. If the human merely prompts the system and the agent decomposes the prompt and implements the logic, it is unclear whether the resulting artifact is protectable intellectual property. This uncertainty necessitates that organizations implementing agentic workflows adopt strict governance frameworks, potentially using the "Reviewer" agent to cross-reference generated code against known open-source databases to ensure license compliance before the code is merged.

### 6.3. The "Drift" Problem of Maintaining Human Context in Agent-Generated Codebases

While security and legal risks are external threats, the "drift" of human context represents an internal, systemic degradation of the engineering process itself [4,13,14,18,27,28] . Our analysis highlights that the increase in implementation velocity often correlates with a degradation of maintainability. As agents generate vast quantities of code autonomously, the human developer's conceptual model of the system, the "context", begins to erode. In the traditional model, the speed of development was limited by the human cognitive capacity to synthesize logic into syntax, which inherently enforced a pace that allowed for the formation of deep understanding. By removing this constraint, agents allow the codebase to grow faster than the human supervisor can comprehend it.

This phenomenon, termed the "drift" of human context [4,13,14,18,27,28], fundamentally alters the nature of ownership. As developers cease to write every line themselves, they gradually lose the deep system familiarity needed for fast debugging and sustained architectural evolution. When a complex system fails, the "Time to Recovery" is heavily dependent on the engineer's intuition regarding the system's inner workings. In an agent-augmented architecture, the code becomes a "black box" of machine-generated logic. The logic may be syntactically valid and pass all self-healing test suites [3–5,12,14], but it may also be convoluted, lacking the readability optimizations that human authors prioritize for their future selves.

The cognitive load associated with reviewing this complex, machine-generated logic is significant. Understanding code written by a synthetic entity is often more difficult than writing it from scratch, as it requires the human to reverse-engineer the agent's intent. Over time, this leads to a state where the human supervisors are afraid to modify the core logic of their applications because they do not fully understand the interdependencies that the agents have created. This results in a fragile ecosystem where the software is ostensibly robust due to automated tests, but architecturally brittle because no human holds the complete system state in their working memory [3,4,11,29]. The "Architect" agent attempts to mitigate this by maintaining high-level vision, but the ultimate responsibility for system evolution cannot be fully delegated without risking the total obsolescence of human control.

### 6.4. Ethical Implications of Training Data Bias and Workforce Displacement

The transformation of the software engineering field extends its impact into the ethical domain, particularly concerning the labor structures and the inherent biases within the technology. The transition to "Agentic Coding" demands a redefinition of the developer's role from syntax author to system coordinator, a shift that disproportionately affects entry-level engineers. Historically, junior developers acquired their skills by writing the repetitive "boilerplate" code that agents now automate with high efficiency. If the industry leverages agents to handle all routine implementation tasks, the apprenticeship model of computer science education is disrupted. There is a risk of creating a "missing middle" [69,70] in the workforce, where there are no pathways for juniors to gain the experience required to become the senior architects capable of supervising these agents.

Furthermore, the "drift" of human context [4,13,14,18,27,28] has ethical implications regarding accountability. If a safety-critical system fails due to code generated by an agent and approved by a human who did not fully comprehend it, the status of moral and legal responsibility is obscured. This is compounded by the bias inherent in the training data.

Agents learn from vast repositories of open-source code, which reflect the historical biases, security flaws, and bad practices of the humans who have written them. An agent does not inherently understand "quality" or "ethics", it understands statistical likelihood [3,4,7–9,16–18,26]. Consequently, it may propagate outdated or discriminatory logic structures found in legacy code, embedding them into modern applications.

The economic trade-off between operational token costs and developer productivity suggests a potential devaluation of human craftsmanship. As the barrier to creating software lowers, the market may be flooded with low-quality, derivative applications, increasing the noise-to-signal ratio in the digital ecosystem. Attaining the full potential of this technology requires a restructuring of computer science education in order to prepare the next generation for the ethical stewardship of autonomous systems besides the coding. Without these structural changes, the efficiency gains of Agentic Coding may be offset by the societal costs of job displacement [71] and the proliferation of unverified, biased algorithmic decision-making systems.

## 7. The Changing Role of the Human Developer

The profound systemic impact of Agentic Coding on the SDLC extends beyond the technical architecture and economic models as it fundamentally necessitates a reconstruction of the human operator's professional identity. As the industry migrates from human-centric workflows to agent-augmented architectures, the definition of "developer" is being rewritten in real-time. This transition needs, besides a change in tooling, a sociological shift that relocates the primary source of value creation from the manual synthesis of syntax to the high-level supervision of autonomous systems. The friction identified between implementation velocity and the degradation of human context suggests that the engineer of the future will function less as a craftsman of code and more as an architect of intent.

### 7.1. Redefining the Developer Role in the SDLC through Prompt Engineering and System Coordination

The most immediate manifestation of this role evolution is the transition from the "syntax author" to the "system coordinator". In the traditional paradigm, the developer's expertise was quantified by their fluency in specific programming languages and their ability to mentally compile complex logic structures. In an environment where autonomous agents engage in multi-step reasoning and execution, the mechanical act of typing code becomes secondary to the skill of precise specification. This elevates "Prompt Engineering" from a colloquial term to a rigorous engineering discipline within the SDLC. Unlike the stochastic interactions typical of creative chatbots, prompt engineering in an agentic context requires the formulation of unambiguous constraints, interface definitions, and functional goals that guide the ReAct [13,19–22] patterns of the model.

The human developer must now act as a coordinator, managing a suite of specialized agents, namely "Architects", "Coders", and "Reviewers", rather than executing the work

personally. This demands a capacity to decompose complex business requirements into discrete logical units that can be processed by synthetic personas without hallucination or deviation [1–4,35,36]. The developer defines the boundaries of the "sandbox" in which the agent operates, establishing the acceptance criteria that serve as the guardrails for autonomous generation. Consequently, the primary operational bottleneck shifts. It is no longer the speed at which a human can type or recall syntax that limits production, but the clarity with which they can express semantic intent to a non-human entity. This coordination role requires a complex understanding of the model's latent space and reasoning limitations, ensuring that the human-in-the-loop remains the strategic driver of a vehicle that is increasingly self-driving. The developer becomes a manager of probability, steering the stochastic nature of LLMs [1,2] toward deterministic software outcomes.

### 7.2. Skill Transformation in Reviewing, Debugging, and System Design

As the generative burden shifts to autonomous agents, the cognitive demands on the human developer undergo a sharp inversion. Historically, developers spent the majority of their time writing code and a minority of their time reading it. Agentic Coding inverts this ratio, placing a massive premium on the skills of reviewing, auditing, and debugging. As identified in our analysis of the "drift" of human context [4,13,14,18,27,28], the rapid generation of code by agents erodes the close connection that the author typically maintains with the codebase. The critical skill for the modern developer, therefore, is the ability to rapidly acquire context and construct a conceptual model of logic that they did not write. This moves the discipline closer to forensic analysis than creative composition.

The "Reviewer" agent may filter syntactic errors, but the human supervisor retains the ultimate responsibility for semantic validity, ensuring the code "means" what it is supposed to mean. This requires an elevated capacity for architectural oversight. The developer must be able to spot subtle logical flaws, security vulnerabilities such as hallucinated dependencies [1–4,35,36], and violations of "SOLID" [45–47] principles within complex, machine-generated logic that may not prioritize human readability. Debugging, too, transforms from correcting one's own errors to diagnosing the reasoning failures of a synthetic entity. The developer must trace the agent's "thought process" through its intermediate reasoning steps in order to understand why a specific implementation path has been chosen. This "black box" debugging requires a deep understanding of system design and integration patterns, as the individual lines of code become less significant than the interactions between modules. The competency profile of a senior engineer is consequently redefined, namely it is no longer about how many lines of code one can produce, but how effectively one can audit the structural integrity of a system built by an army of autonomous interns.

### 7.3. Educational Implications for the Computer Science Curricula

The redefinition of the developer's role leads to new challenges in computer science education, specifically regarding the "missing middle" [69,70] in workforce development. The traditional apprenticeship model relies on junior developers gaining skills and competence by writing the repetitive "boilerplate" code and handling routine bug fixes that

agents now automate with near-instantaneous speed [35,36]. If these foundational tasks are delegated to machines, the industry risks severing the pathway through which novices develop the deep intuition required to become senior architects. Without the struggle of writing loops and conditionals, students may fail to develop the abilities necessary to supervise the agents that write the code.

In order to mitigate this, academic curricula must undergo a structural change, moving away from syntax memorization and towards higher-order systems thinking. Computer Science education [6,7,35,36] must prioritize architectural design, formal verification methods, and the ethics of autonomous systems over routine implementation. Students must be trained in addition to coding, to evaluate the correctness of code through rigorous testing frameworks and logic proofs. The curriculum should treat "Code Review" not as a peripheral activity, but as a core competency equal to algorithm design. Furthermore, as the "drift" problem [4,13,14,18,27,28] creates fragile ecosystems where no human fully understands the system state, education must emphasize documentation standards and maintainability as primary qualities. We must cultivate a generation of engineers who are comfortable operating at high levels of abstraction, capable of managing the "black box" without losing control of the output. The goal of future education is to produce competent coordinators of autonomous intelligence, ensuring that the human element remains the governing force in an increasingly automated lifecycle.

## 8. Future Directions

The transition from human-centric software engineering to agent-augmented architectures marks only the emerging phase of a broader trajectory toward autonomous systems. While the current study has analyzed the immediate systemic impacts of Agentic Coding, ranging from the acceleration of requirements engineering to the risks of context drift, the rapid evolution of LLMs suggests that the underlying paradigms will continue to shift. As we look towards the horizon, three specific areas emerge as extremely important frontiers for research and industrial application, namely the progression towards fully autonomous software development organizations, the necessity of integrating formal verification to counterbalance stochastic generation, and the development of personalized agents that align with individual cognitive styles.

### 8.1. Towards AGI in Software Engineering through Fully Autonomous Software Development Organizations

The logical extrapolation of the current "Multi-Agent Systems" (MAS) [3,4,11,14,16–18,26] architecture is the gradual removal of the human supervisor from the immediate feedback loop, moving from a "human-in-the-loop" model to a "human-on-the-loop" or potentially "human-out-of-the-loop" configuration for defined domains. Currently, agents operate primarily as task-specific executors that require human coordination in order to manage high-level goals and resolve consensus failures. Nevertheless, as the context windows of LLMs expand and memory systems [3,4,11,29] become more robust, we anticipate the emergence of "Fully Autonomous Software Development Organizations". In

this approach, the role of the "Architect" agent will evolve from merely decomposing prompts to autonomously sensing business needs, monitoring production telemetry, and proactively initiating feature development or refactoring cycles without explicit human triggering.

This shift suggests a move towards "Artificial General Intelligence" (AGI) [3,72] within the specific vertical of software engineering. Future research must explore the dynamics of recursive self-improvement, where agents, in addition to writing their application code, also modify their own system prompts and tool definitions in order to optimize their performance. This raises profound questions regarding the "drift" of human context [4,13,14,18,27,28], namely if agents are capable of autonomously conceiving, designing, and deploying software, the resulting systems may become entirely opaque to human understanding, operating as highly efficient but impenetrable "black boxes". Consequently, the focus of software engineering research will likely shift from the mechanics of code production to the governance of autonomous fleets, establishing "Constitutional AI" frameworks that ensure these self-governing development organizations adhere to immutable safety and ethical guidelines even when operating beyond direct human oversight.

### 8.2. Integration with Formal Verification Methods

As observed within the study, the primary risk of Agentic Coding lies in the probabilistic nature of LLMs, which function as stochastic predictors rather than deterministic engines. This stochasticity introduces vulnerabilities such as hallucinations [1–4,35,36] and subtle logic flaws that syntactically correct code may hide. In order to mitigate these aspects, future agentic frameworks must move beyond standard unit testing and integrate directly with "Formal Verification" methods. While traditional testing validates that code works for a finite set of inputs, formal verification provides a mathematical proof that the logic adheres to a specification for all possible inputs. Historically, formal methods were considered too labor-intensive for general software development, but agents are uniquely positioned to address these problems.

Future iterations of the "Coder" and "Reviewer" agents will likely be trained besides standard programming languages on proof-carrying code and specification languages such as "Coq" [73], "Isabelle" [74,75], or "Dafny" [76]. In this workflow, an agent would simultaneously generate the mathematical proof of its correctness along with the function it codes. This integration acts as a definitive countermeasure to the "hallucination" problem [1–4,35,36], as the compiler's feedback loop would be augmented by a theorem prover that rejects any code that cannot be mathematically proven to satisfy the safety constraints. This creates a "correct-by-construction" approach where the security of the system [11,27,28,77] is guaranteed by the deterministic laws of mathematics. This shift is extremely important for the safe deployment of agents in special domains infrastructure, effectively neutralizing the risks of dependency injection and logical drift.

### 8.3. Personalized Agents that Learn from Individual Developer Styles

While current agents rely on vast repositories of open-source code to learn general patterns, the next frontier involves the hyper-personalization of these models to individual developers or specific organizational cultures. Currently, the generic nature of agent-generated code contributes to the "drift" of human context [4,13,14,18,27,28], as the machine produces logic that, while functional, may differ syntactically and structurally from what the human maintainer finds readable. In order to resolve this friction, future research will focus on "Small Language Models" (SLMs) [78] or "Low-Rank Adaptation" (LoRA) [79] techniques that fine-tune base models on a specific team's proprietary repositories.

These personalized agents will function as "digital twins" of the human developer, learning their specific variable naming conventions, commenting styles, and architectural preferences. By mimicking the unique cognitive fingerprint of the human supervisor, these agents can generate code that feels "native" to the author, thereby significantly reducing the cognitive load associated with reviewing machine-generated logic. This alignment preserves the "flow" state of development and helps maintain a cohesive institutional memory [3,4,11,29], as the agent adheres to the tacit knowledge and unwritten rules of the specific codebase. Nevertheless, this personalization introduces a trade-off between local efficiency and global standardization. Research must explore the tension between highly customized agents that maximize individual productivity and the need for standardized coding practices that ensure interoperability and talent mobility across the industry. Ultimately, the goal is to create agents that adapt to the human, rather than forcing the human to adapt to the synthetic output of the machine.

## 9. Conclusions

### 9.1. Summary of Key Findings

This study has systematically established that the emergence of Agentic Coding represents a transformation of the software engineering field comparable in magnitude to the advent of high-level compilers. We have determined that unlike stateless Copilots which function as stochastic predictors, autonomous agents use ReAct [13,19–22] patterns to engage in multi-step reasoning, strategic planning, and active environment interaction. The analysis has confirmed that this new approach impacts the entire SDLC, specifically by accelerating requirements engineering through automated prototyping and revolutionizing quality assurance [39–41] via self-healing test suites [3–5,12,14] where the compiler acts as an objective reviewer. Nevertheless, the integration of these autonomous systems creates a fundamental friction between increased implementation velocity and the degradation of long-term maintainability, a phenomenon identified as the "drift" of human context [4,13,14,18,27,28].

As agents generate vast quantities of code autonomously, the human developer's profound understanding of the system erodes, effectively transforming the codebase into an opaque "black box" of machine-generated logic. Furthermore, the study has highlighted extremely important risks, including security vulnerabilities arising from hallucinated [1–4,35,36] dependencies and the complex economic trade-off between operational token costs and

developer productivity gains. We have observed that while defect density regarding syntax errors may decrease due to iterative self-correction, the complexity density of the logic risks increases. Ultimately, the study has demonstrated that the primary bottleneck of software production has shifted from the cognitive capacity to synthesize logic into syntax to the capacity for architectural oversight, requiring a definitive redefinition of the developer's role from syntax author to system coordinator.

### 9.2. Final Recommendations for Industry Adoption

In order to successfully manage the transition to agent-augmented architectures, organizations must move beyond the passive consumption of generated code and adopt rigorous governance frameworks that address the stochastic nature of LLMs [1,2]. We consider that achieving the full potential of this technology requires the integration of formal verification methods to counterbalance the probabilistic generation of logic, ensuring that agentic outputs adhere to strict mathematical safety constraints rather than merely passing unit tests. Industry practitioners should maintain a "human-in-the-loop" or "human-on-the-loop" operational model where the developer functions as a strategic supervisor responsible for coordinating multi-agent systems [3,4,11,14,16–18,26] rather than executing manual implementation. In order to mitigate the risk of context drift [4,13,14,18,27,28], development teams must enforce comprehensive documentation standards and use specialized "Reviewer" agents to audit machine-generated logic for adherence to "SOLID" [45–47] principles and architectural integrity.

Furthermore, this technological shift demands a structural change in computer science education in order to address the potential "missing middle" [69,70] in workforce development. Curricula must evolve to prioritize systems thinking, architectural design, and ethical stewardship over routine syntax memorization, ensuring that future engineers possess the necessary abilities to supervise autonomous systems. We also recommend that organizations strictly account for the "Total Cost of Ownership", weighing the immediate velocity gains against the long-term inference costs and the maintenance overhead of complex, agent-generated ecosystems. Finally, adoption strategies must account for legal ambiguities by implementing automated dependency verification in order to prevent the accidental incorporation of hallucinated or non-compliant software packages.

## Acknowledgment

"The authors would like to express their gratitude for the logistics support received from the Center of Research, Consultancy and Training in Economic Informatics and Information Technology RAU-INFORTIS of the Romanian-American University and the Center for Computational Science and Machine Intelligence of the Romanian-American University."

## References

[1] Lin, T.; Wang, Y.; Liu, X.; Qiu, X. A Survey of Transformers. *AI Open* 2022, *3*, doi:10.1016/j.aiopen.2022.10.001.

[2] Tay, Y.; Dehghani, M.; Bahri, D.; Metzler, D. Efficient Transformers: A Survey. *ACM Comput Surv* 2023, *55*, doi:10.1145/3530811.

[3] Xi, Z.; Chen, W.; Guo, X.; He, W.; Ding, Y.; Hong, B.; Zhang, M.; Wang, J.; Jin, S.; Zhou, E.; et al. The Rise and Potential of Large Language Model Based Agents: A Survey. *Science China Information Sciences* 2025, *68*.

[4] Li, X.; Wang, S.; Zeng, S.; Wu, Y.; Yang, Y. A Survey on LLM-Based Multi-Agent Systems: Workflow, Infrastructure, and Challenges. *Vicinagearth* 2024, *1*, doi:10.1007/s44336-024-00009-2.

[5] Malamas, N.; Tsardoulias, E.; Panayiotou, K.; Symeonidis, A.L. Toward Efficient Vibe Coding: An LLM-Based Agent for Low-Code Software Development. *J Comput Lang* 2025, *85*, doi:10.1016/j.cola.2025.101367.

[6] Denton, M.; Chasen, A.; Fleming, G.C.; Borrego, M.; Knight, D. Agentic Actions and Agentic Perspectives Among Fellowship-Funded Engineering Doctoral Students. *Educ Sci (Basel)* 2025, *15*, doi:10.3390/educsci15101378.

[7] Addagalla, S.R.D. Engineering in the Age of AI: Leveraging Copilot for Enhanced Software Development. *International Journal of Engineering and Advanced Technology Studies* 2025, *13*, doi:10.37745/ijeats.13/vol13n12943.

[8] Xiao, Z.; He, X.; Wu, H.; Yu, B.; Guo, Y. EDA-Copilot: A RAG-Powered Intelligent Assistant for EDA Tools. *ACM Transact Des Autom Electron Syst* 2025, *30*, doi:10.1145/3715326.

[9] Döderlein, J.B.; Kouadio, N.H.; Acher, M.; Khelladi, D.E.; Combemale, B. Piloting Copilot, Codex, and StarCoder2: Hot Temperature, Cold Prompts, or Black Magic? *Journal of Systems and Software* 2025, *230*, doi:10.1016/j.jss.2025.112562.

[10] Flore, P.; Hussong, M.; Simon, P.M. Bridging the Programming Divide - Democratizing Programming Skills through GPT Assistants in Manufacturing. *ZWF Zeitschrift fuer Wirtschaftlichen Fabrikbetrieb* 2025, *120*, doi:10.1515/zwf-2024-0130.

[11] Yan, B.; Li, K.; Xu, M.; Dong, Y.; Zhang, Y.; Ren, Z.; Cheng, X. On Protecting the Data Privacy of Large Language Models (LLMs) and LLM Agents: A Literature Review. *High-Confidence Computing* 2025, *5*.

[12] Kondus, O.; Tkachenko, O. Intellectualized Support Module of the Software Development. *Technical sciences and technologies* 2025, doi:10.25140/2411-5363-2025-2(40)-312-324.

[13] Moniruzzaman, M.; Alam, A.M. Integration of LLM and ReAct Agents for Enhanced Context Oriented Programming. In Proceedings of the 2024 27th International Conference on Computer and Information Technology, ICCIT 2024 - Proceedings; 2024.

[14] Deva, I.; Sanwal, M. An Autonomous Multi-Agent LLM Framework for Agile Software Development. *International Journal of Trend in Scientific Research and Development* 2024, *Volume-8*.

[15] Ge, S.; Sun, Y.; Cui, Y.; Wei, D. An Innovative Solution to Design Problems: Applying the Chain-of-Thought Technique to Integrate LLM-Based Agents With Concept Generation Methods. *IEEE Access* 2025, *13*, doi:10.1109/ACCESS.2024.3494054.

[16] Pan, B.; Lu, J.; Wang, K.; Zheng, L.; Wen, Z.; Feng, Y.; Zhu, M.; Chen, W. AgentCoord: Visually Exploring Coordination Strategy for LLM-Based Multi-Agent Collaboration. *Comput Graph* 2025, *132*, doi:10.1016/j.cag.2025.104338.

[17] Kalyuzhnaya, A.; Mityagin, S.; Lutsenko, E.; Getmanov, A.; Aksenkin, Y.; Fatkhiev, K.; Fedorin, K.; Nikitin, N.O.; Chichkova, N.; Vorona, V.; et al. LLM Agents for Smart City Management: Enhancing Decision Support Through Multi-Agent AI Systems. *Smart Cities* 2025, *8*, doi:10.3390/smartcities8010019.

[18] Pelluru, K. LangChain & LangGraph in Production: Architectures for Multi-Agent LLM Systems. *Journal of Data and Digital Innovation (JDDI)* 2025, *2*.

[19] Tarek RADAH ReAct-Driven SOC Agent with Integrated Detection Engineering for AI-Enhanced Autonomous Alert Handling. *Journal of Information Systems Engineering and Management* 2025, *10*, doi:10.52783/jisem.v10i53s.10967.

[20] Shaker Mahmoud, A.A.; Shishah, W.; Mistry, N.R. ReACT_OCRS: An AI-Driven Anonymous Online Reporting System Using Synergized Reasoning and Acting in Language Models. *IEEE Access* 2025, doi:10.1109/ACCESS.2025.3571526.

[21] Yuvzhenko, D.; Chymshyr, V.; Shymkovych, V.; Znova, K.; Nowakowski, G.; Telenyk, S. A Multimodal Retrieval-Augmented Generation System with ReAct Agent Logic for Multi-Hop Reasoning. *Information, Computing and Intelligent systems* 2025, doi:10.20535/2786-8729.6.2025.330777.

[22] Yan, X.; Yang, X.; Jin, N.; Chen, Y.; Li, J. A General AI Agent Framework for Smart Buildings Based on Large Language Models and ReAct Strategy. *Smart Construction* 2025, *2*, doi:10.55092/sc20250004.

[23] Yuksekgonul, M.; Bianchi, F.; Boen, J.; Liu, S.; Lu, P.; Huang, Z.; Guestrin, C.; Zou, J. Optimizing Generative AI by Backpropagating Language Model Feedback. *Nature* 2025, *639*, doi:10.1038/s41586-025-08661-4.

[24] Roy, D.; Zhang, X.; Bhave, R.; Bansal, C.; Las-Casas, P.; Fonseca, R.; Rajmohan, S. Exploring LLM-Based Agents for Root Cause Analysis. In Proceedings of the FSE Companion - Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering; 2024.

[25] Hu, Q.; Tu, X.; Guo, C.; Zhang, S. Time-Aware ReAct Agent for Temporal Knowledge Graph Question Answering.; 2025.

[26] Jiang, C.; Yang, X. AgentsBench: A Multi-Agent LLM Simulation Framework for Legal Judgment Prediction. *Systems* 2025, *13*, doi:10.3390/systems13080641.

[27] Tang, X.; Jin, Q.; Zhu, K.; Yuan, T.; Zhang, Y.; Zhou, W.; Qu, M.; Zhao, Y.; Tang, J.; Zhang, Z.; et al. Prioritizing Safeguarding Over Autonomy: Risks of LLM Agents for Science. *Nature Communications 2025 16:1* 2024, *16*.

[28] Tang, Y.; Liu, Y.; Lan, J.; Yan, Z.; Gelenbe, E. Security of LLM-Based Agents Regarding Attacks, Defenses, and Applications: A Comprehensive Survey. *Information Fusion* 2026, *127*, doi:10.1016/j.inffus.2025.103941.

[29] Hatalis, K.; Christou, D.; Myers, J.; Jones, S.; Lambert, K.; Amos-Binks, A.; Dannenhauer, Z.; Dannenhauer, D. Memory Matters: The Need to Improve Long-Term Memory in LLM-Agents. *Proceedings of the AAAI Symposium Series* 2024, *2*, doi:10.1609/aaaiss.v2i1.27688.

[30] Korinek, A. AI Agents for Economic Research: August 2025 Update to 'Generative AI for Economic Research: Use Cases and Implications for Economists. *J Econ Lit* 2025, *61(4)*.

[31] O'Connor, A.J. *Organizing for Generative AI and the Productivity Revolution: Reshaping Organizational Roles in the Age of Artificial Intelligence*; 2024;

[32] Moreschini, S.; Arvanitou, E.M.; Kanidou, E.P.; Nikolaidis, N.; Su, R.; Ampatzoglou, A.; Chatzigeorgiou, A.; Lenarduzzi, V. The Evolution of Technical Debt from DevOps to Generative AI: A Multivocal Literature Review. *Journal of Systems and Software* 2026, *231*, doi:10.1016/j.jss.2025.112599.

[33] Li, H.; Hao, Y.; Zhai, Y.; Qian, Z. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 2024, *8*, doi:10.1145/3649828.

[34] Gao, Z. A Review on Statistical Language and Neural Network Based Code Completion. *Applied and Computational Engineering* 2023, *22*, doi:10.54254/2755-2721/22/20231222.

[35] Pirzado, F.A.; Ahmed, A.; Mendoza-Urdiales, R.A.; Terashima-Marin, H. Navigating the Pitfalls: Analyzing the Behavior of LLMs as a Coding Assistant for Computer Science Students - A Systematic Review of the Literature. *IEEE Access* 2024, *12*, doi:10.1109/ACCESS.2024.3443621.

[36] Jošt, G.; Taneski, V.; Karakatič, S. The Impact of Large Language Models on Programming Education and Student Learning Outcomes. *Applied Sciences (Switzerland)* 2024, *14*, doi:10.3390/app14104115.

[37] Xing, F. Designing Heterogeneous LLM Agents for Financial Sentiment Analysis. *ACM Trans Manag Inf Syst* 2025, *16*, doi:10.1145/3688399.

[38] Kirshner, S.N.; Pan, Y.; Wu, J.X.; Gould, A. Talking Terms: Agent Information in LLM Supply Chain Bargaining. *Decision Sciences* 2025, doi:10.1111/deci.70010.

[39] Pareek, C.S. Accelerating Agile Quality Assurance with AI-Powered Testing Strategies. *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT* 2024, *08*, doi:10.55041/ijsrem15369.

[40] Ahmed, A.; Kerr, E.; O'Malley, A. Quality Assurance and Validity of AI-Generated Single Best Answer Questions. *BMC Med Educ* 2025, *25*, doi:10.1186/s12909-025-06881-w.

[41] Ouyang, T.; MaungMaung, A.P.; Konishi, K.; Seo, Y.; Echizen, I. Stability Analysis of ChatGPT-Based Sentiment Analysis in AI Quality Assurance. *Electronics (Switzerland)* 2024, *13*, doi:10.3390/electronics13245043.

[42] Ben Hajhmida, M.; Lee, E.A. RAG and Agentic Assistant: A Combined Approach. In Proceedings of the Lecture Notes in Computer Science; 2026; Vol. 16220 LNCS.

[43] Pirnau, M.; Botezatu, M.A.; Priescu, I.; Hosszu, A.; Tabusca, A.; Coculescu, C.; Oncioiu, I. Content Analysis Using Specific Natural Language Processing Methods for Big Data. *Electronics (Switzerland)* 2024, *13*, doi:10.3390/electronics13030584.

[44] Aggarwal, V.; Kamal, O.; Japesh, A.; Jin, Z.; Schölkopf, B. DARS: Dynamic Action Re-Sampling to Enhance Coding Agent Performance by Adaptive Tree Traversal. In Proceedings of the Proceedings of the Annual Meeting of the Association for Computational Linguistics; 2025; Vol. 1.

[45] Cabral, R.; Kalinowski, M.; Baldassarre, M.T.; Villamizar, H.; Escovedo, T.; Lopes, H. Investigating the Impact of Solid Design Principles on Machine Learning Code Understanding. In Proceedings of the Proceedings - 2024 IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI, CAIN 2024; 2024.

[46] Ramachandrappa, N.C. SOLID Design Principles in Software Engineering. *International Journal of Computer Trends and Technology* 2024, *72*, doi:10.14445/22312803/ijctt-v72i9p104.

[47] Yanakiev, I.; Lazar, B.M.; Capiluppi, A. Applying SOLID Principles for the Refactoring of Legacy Code: An Experience Report. *Journal of Systems and Software* 2025, *220*, doi:10.1016/j.jss.2024.112254.

[48] Jay, R. *Generative AI Apps with LangChain and Python: A Project-Based Approach to Building Real-World LLM Apps*; 2024;

[49] Auffarth, B. *Generative AI with LangChain: Build Large Language Model (LLM) Apps with Python, ChatGPT, and Other LLMs*; 2023;

[50] Kim, A.G.; Muhn, M.; Nikolaev, V. V. Bloated Disclosures: Can ChatGPT Help Investors Process Information? *SSRN Electronic Journal* 2023, doi:10.2139/ssrn.4425527.

[51] Rekart, J.L.; Baker, R. OI, AI, and Research or Why OI Is the GOAT and AI Is the BLOAT. In *Designing for Human Intelligence in an Artificial Intelligence World*; 2025.

[52] Melo, A.; Fagundes, R.; Lenarduzzi, V.; Santos, W.B. Identification and Measurement of Requirements Technical Debt in Software Development: A Systematic Literature Review. *Journal of Systems and Software* 2022, *194*, doi:10.1016/j.jss.2022.111483.

[53] Vemuri, N.; Thaneeru, N.; Tatikonda, V.M. AI-Optimized DevOps for Streamlined Cloud CI/CD. *Int J Innov Sci Res Technol* 2024, *9*.

[54] Polinati, A. kumar Devops And Ai: Automating Software Delivery Pipelines For Continuous Integration And Deployment. *Nanotechnol Percept* 2024, *20*.

[55] Mohammed, A.S.; Saddi, V.R.; Gopal, S.K.; Dhanasekaran, S.; Naruka, M.S. AI-Driven Continuous Integration and Continuous Deployment in Software Engineering. In Proceedings of the 2024 2nd International Conference on Disruptive Technologies, ICDT 2024; 2024.

[56] SECURITY FOR CONTINUOUS INTEGRATION AND CONTINUOUS DEPLOYMENT PIPELINE. *International Research Journal of Modernization in Engineering Technology and Science* 2024, doi:10.56726/irjmets50676.

[57] Gonçalves, C.A.; Gonçalves, C.T. Assessment on the Effectiveness of GitHub Copilot as a Code Assistance Tool: An Empirical Study. In Proceedings of the Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) ; 2025; Vol. 14969 LNAI.

[58] Forsgren, N.; Storey, M.A.; Maddila, C.; Zimmermann, T.; Houck, B.; Butler, J. The SPACE of Developer Productivity. *Commun ACM* 2021, *64*, doi:10.1145/3453928.

[59] Greiler, M.; Storey, M.A.; Noda, A. An Actionable Framework for Understanding and Improving Developer Experience. *IEEE Transactions on Software Engineering* 2023, *49*, doi:10.1109/TSE.2022.3175660.

[60] Forsgren, N.; Humble, J.; Kim, G.; Press, I.T.R. *Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations*; 2018;

[61] Magnus Chukwuebuka Ahuchogu Evaluating the Impact of Generative AI on Intelligent Programming Assistance and Code Quality. *Power System Technology* 2025, *49*, doi:10.52783/pst.1668.

[62] Zacarias, R.O.; Antunes, L.C.R.; Barros, M. de O.; Santos, R.P. dos; Lago, P. Exploring Developer Experience Factors in Software Ecosystems. *Journal of Systems and Software* 2025, *230*, doi:10.1016/j.jss.2025.112549.

[63] Ghazali, M.; Hidayat, A.N.R. Enhancing the Developer Experience (DX) in Docker Supported Projects. *International Journal of Innovative Computing* 2023, *13*, doi:10.11113/ijic.v13n1.393.

[64] Kafura, D. Reflections on McCabe's Cyclomatic Complexity. *IEEE Transactions on Software Engineering* 2025, *51*, doi:10.1109/TSE.2025.3534580.

[65] Ji, S.L.; Wang, Q.Y.; Chen, A.Y.; Zhao, B. Bin; Ye, T.; Zhang, X.H.; Wu, J.Z.; Li, Y.; Yin, J.W.; Wu, Y.J. Survey on Open-Source Software Supply Chain Security. *Ruan Jian Xue Bao/Journal of Software* 2023, *34*, doi:10.13328/j.cnki.jos.006717.

[66] Ahmed, A.A. A COMPREHENSIVE REVIEW OF ADVERSARIAL ATTACKS IN MACHINE LEARNING. *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT* 2024, *08*, doi:10.55041/ijsrem35239.

[67] Kahveci, Z.Ü. Attribution Problem of Generative AI: A View from US Copyright Law. *Journal of Intellectual Property Law and Practice* 2023, *18*, doi:10.1093/jiplp/jpad076.

[68] Samuelson, P. Legal Challenges to Generative AI, Part I: Questioning the Legality of Using in-Copyright Works for Training Data and Producing Outputs Derived from Copyrighted Training Data. *Commun ACM* 2023, *66*.

[69] Thomas, H.; Anner, M. Dissensus and Deadlock in the Evolution of Labour Governance: Global Supply Chains and the International Labour Organization (ILO). *Journal of Business Ethics* 2023, *184*, doi:10.1007/s10551-022-05177-z.

[70] Silva, V. The ILO and the Future of Work: The Politics of Global Labour Policy. *Glob Soc Policy* 2022, *22*, doi:10.1177/14680181211004853.

[71] Deckker, D.; Sumanasekara, S. Artificial Intelligence and the Future of Job Security: A Narrative Review of Risks, Resilience, and Policy Responses. *International Journal of Research and Innovation in Social Science* 2025, *IX*, doi:10.47772/ijriss.2025.906000333.

[72] Bennett, M.T. What Is Artificial General Intelligence? In Proceedings of the Lecture Notes in Computer Science; 2026; Vol. 16057 LNAI.

[73] Guo, D.; Yu, W. A Comprehensive Formalization of Propositional Logic in Coq: Deduction Systems, Meta-Theorems, and Automation Tactics. *Mathematics* 2023, *11*, doi:10.3390/math11112504.

[74] Lund, S.T.; Villadsen, J. On Verified Automated Reasoning in Propositional Logic: Teaching Sequent Calculus to Computer Science Students. *Vietnam Journal of Computer Science* 2024, *11*, doi:10.1142/S2196888824500064.

[75] From, A.H.; Schlichtkrull, A.; Villadsen, J. A Sequent Calculus for First-Order Logic Formalized in Isabelle/HOL. *Journal of Logic and Computation* 2023, *33*, doi:10.1093/logcom/exad013.

[76] Wang, J.; Chen, S.; Zhu, H. Verification of Scapegoat Trees Using Dafny. In Proceedings of the Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); 2024; Vol. 14627 LNCS.

[77] Nakash, I.; Kour, G.; Uziel, G.; Anaby Tavor, A. Breaking ReAct Agents: Foot-in-the-Door Attack Will Get You In.; 2025.

[78] Fukuda, N.; Nozue, H.; Oishi, H. Small Language Model Agent for the Operations of Continuously Updating ICT Systems. *IEEE Access* 2025, *13*, doi:10.1109/ACCESS.2025.3544637.

[79] Mao, Y.; Ge, Y.; Fan, Y.; Xu, W.; Mi, Y.; Hu, Z.; Gao, Y. A Survey on LoRA of Large Language Models. *Front Comput Sci* 2025, *19*.

**Bibliography**

Addagalla, S.R.D. Engineering in the Age of AI: Leveraging Copilot for Enhanced Software Development. *International Journal of Engineering and Advanced Technology Studies* 2025, *13*, doi:10.37745/ijeats.13/vol13n12943.

Aggarwal, V.; Kamal, O.; Japesh, A.; Jin, Z.; Schölkopf, B. DARS: Dynamic Action Re-Sampling to Enhance Coding Agent Performance by Adaptive Tree Traversal. In Proceedings of the Proceedings of the Annual Meeting of the Association for Computational Linguistics; 2025; Vol. 1.

Ahmed, A.; Kerr, E.; O'Malley, A. Quality Assurance and Validity of AI-Generated Single Best Answer Questions. *BMC Med Educ* 2025, *25*, doi:10.1186/s12909-025-06881-w.

Ahmed, A.A. A COMPREHENSIVE REVIEW OF ADVERSARIAL ATTACKS IN MACHINE LEARNING. *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT* 2024, *08*, doi:10.55041/ijsrem35239.

Auffarth, B. *Generative AI with LangChain: Build Large Language Model (LLM) Apps with Python, ChatGPT, and Other LLMs*; 2023;

Ben Hajhmida, M.; Lee, E.A. RAG and Agentic Assistant: A Combined Approach. In Proceedings of the Lecture Notes in Computer Science; 2026; Vol. 16220 LNCS.

Bennett, M.T. What Is Artificial General Intelligence? In Proceedings of the Lecture Notes in Computer Science; 2026; Vol. 16057 LNAI.

Cabral, R.; Kalinowski, M.; Baldassarre, M.T.; Villamizar, H.; Escovedo, T.; Lopes, H. Investigating the Impact of Solid Design Principles on Machine Learning Code Understanding. In Proceedings of the Proceedings - 2024 IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI, CAIN 2024; 2024.

David Odera; Martin Otieno; Jairus Ekume Ounza Security Risks in the Software Development Lifecycle: A Review. *World Journal of Advanced Engineering Technology and Sciences* 2023, *8*, doi:10.30574/wjaets.2023.8.2.0101.

Deckker, D.; Sumanasekara, S. Artificial Intelligence and the Future of Job Security: A Narrative Review of Risks, Resilience, and Policy Responses. *International Journal of Research and Innovation in Social Science* 2025, *IX*, doi:10.47772/ijriss.2025.906000333.

Denton, M.; Chasen, A.; Fleming, G.C.; Borrego, M.; Knight, D. Agentic Actions and Agentic Perspectives Among Fellowship-Funded Engineering Doctoral Students. *Educ Sci (Basel)* 2025, *15*, doi:10.3390/educsci15101378.

Deva, I.; Sanwal, M. An Autonomous Multi-Agent LLM Framework for Agile Software Development. *International Journal of Trend in Scientific Research and Development* 2024, *Volume-8*.

Diansyah, A.F.; Rahman, M.R.; Handayani, R.; Nur Cahyo, D.D.; Utami, E. Comparative Analysis of Software Development Lifecycle Methods in Software Development: A Systematic Literature Review. *International Journal of Advances in Data and Information Systems* 2023, *4*, doi:10.25008/ijadis.v4i2.1295.

Döderlein, J.B.; Kouadio, N.H.; Acher, M.; Khelladi, D.E.; Combemale, B. Piloting Copilot, Codex, and StarCoder2: Hot Temperature, Cold Prompts, or Black Magic? *Journal of Systems and Software* 2025, *230*, doi:10.1016/j.jss.2025.112562.

Flore, P.; Hussong, M.; Simon, P.M. Bridging the Programming Divide - Democratizing Programming Skills through GPT Assistants in Manufacturing. *ZWF Zeitschrift fuer Wirtschaftlichen Fabrikbetrieb* 2025, *120*, doi:10.1515/zwf-2024-0130.

Forsgren, N.; Humble, J.; Kim, G.; Press, I.T.R. *Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations*; 2018;

Forsgren, N.; Storey, M.A.; Maddila, C.; Zimmermann, T.; Houck, B.; Butler, J. The SPACE of Developer Productivity. *Commun ACM* 2021, *64*, doi:10.1145/3453928.

From, A.H.; Schlichtkrull, A.; Villadsen, J. A Sequent Calculus for First-Order Logic Formalized in Isabelle/HOL. *Journal of Logic and Computation* 2023, *33*, doi:10.1093/logcom/exad013.

Fukuda, N.; Nozue, H.; Oishi, H. Small Language Model Agent for the Operations of Continuously Updating ICT Systems. *IEEE Access* 2025, *13*, doi:10.1109/ACCESS.2025.3544637.

Gao, Z. A Review on Statistical Language and Neural Network Based Code Completion. *Applied and Computational Engineering* 2023, *22*, doi:10.54254/2755-2721/22/20231222.

Ge, S.; Sun, Y.; Cui, Y.; Wei, D. An Innovative Solution to Design Problems: Applying the Chain-of-Thought Technique to Integrate LLM-Based Agents With Concept Generation Methods. *IEEE Access* 2025, *13*, doi:10.1109/ACCESS.2024.3494054.

Ghazali, M.; Hidayat, A.N.R. Enhancing the Developer Experience (DX) in Docker Supported Projects. *International Journal of Innovative Computing* 2023, *13*, doi:10.11113/ijic.v13n1.393.

Gonçalves, C.A.; Gonçalves, C.T. Assessment on the Effectiveness of GitHub Copilot as a Code Assistance Tool: An Empirical Study. In Proceedings of the Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) ; 2025; Vol. 14969 LNAI.

Greiler, M.; Storey, M.A.; Noda, A. An Actionable Framework for Understanding and Improving Developer Experience. *IEEE Transactions on Software Engineering* 2023, *49*, doi:10.1109/TSE.2022.3175660.

Guo, D.; Yu, W. A Comprehensive Formalization of Propositional Logic in Coq: Deduction Systems, Meta-Theorems, and Automation Tactics. *Mathematics* 2023, *11*, doi:10.3390/math11112504.

Hatalis, K.; Christou, D.; Myers, J.; Jones, S.; Lambert, K.; Amos-Binks, A.; Dannenhauer, Z.; Dannenhauer, D. Memory Matters: The Need to Improve Long-Term Memory in LLM-Agents. *Proceedings of the AAAI Symposium Series* 2024, *2*, doi:10.1609/aaaiss.v2i1.27688.

Hu, Q.; Tu, X.; Guo, C.; Zhang, S. Time-Aware ReAct Agent for Temporal Knowledge Graph Question Answering.; 2025.

Jay, R. *Generative AI Apps with LangChain and Python: A Project-Based Approach to Building Real-World LLM Apps*; 2024;

Ji, S.L.; Wang, Q.Y.; Chen, A.Y.; Zhao, B. Bin; Ye, T.; Zhang, X.H.; Wu, J.Z.; Li, Y.; Yin, J.W.; Wu, Y.J. Survey on Open-Source Software Supply Chain Security. *Ruan Jian Xue Bao/Journal of Software* 2023, *34*, doi:10.13328/j.cnki.jos.006717.

Jiang, C.; Yang, X. AgentsBench: A Multi-Agent LLM Simulation Framework for Legal Judgment Prediction. *Systems* 2025, *13*, doi:10.3390/systems13080641.

Jošt, G.; Taneski, V.; Karakatič, S. The Impact of Large Language Models on Programming Education and Student Learning Outcomes. *Applied Sciences (Switzerland)* 2024, *14*, doi:10.3390/app14104115.

Kafura, D. Reflections on McCabe's Cyclomatic Complexity. *IEEE Transactions on Software Engineering* 2025, *51*, doi:10.1109/TSE.2025.3534580.

Kahveci, Z.Ü. Attribution Problem of Generative AI: A View from US Copyright Law. *Journal of Intellectual Property Law and Practice* 2023, *18*, doi:10.1093/jiplp/jpad076.

Kalyuzhnaya, A.; Mityagin, S.; Lutsenko, E.; Getmanov, A.; Aksenkin, Y.; Fatkhiev, K.; Fedorin, K.; Nikitin, N.O.; Chichkova, N.; Vorona, V.; et al. LLM Agents for Smart City

Management: Enhancing Decision Support Through Multi-Agent AI Systems. *Smart Cities* 2025, *8*, doi:10.3390/smartcities8010019.

Kim, A.G.; Muhn, M.; Nikolaev, V. V. Bloated Disclosures: Can ChatGPT Help Investors Process Information? *SSRN Electronic Journal* 2023, doi:10.2139/ssrn.4425527.

Kirshner, S.N.; Pan, Y.; Wu, J.X.; Gould, A. Talking Terms: Agent Information in LLM Supply Chain Bargaining. *Decision Sciences* 2025, doi:10.1111/deci.70010.

Kondus, O.; Tkachenko, O. Intellectualized Support Module of the Software Development. *Technical sciences and technologies* 2025, doi:10.25140/2411-5363-2025-2(40)-312-324.

Korinek, A. AI Agents for Economic Research: August 2025 Update to 'Generative AI for Economic Research: Use Cases and Implications for Economists. *J Econ Lit* 2025, *61(4)*.

Li, H.; Hao, Y.; Zhai, Y.; Qian, Z. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 2024, *8*, doi:10.1145/3649828.

Li, X.; Wang, S.; Zeng, S.; Wu, Y.; Yang, Y. A Survey on LLM-Based Multi-Agent Systems: Workflow, Infrastructure, and Challenges. *Vicinagearth* 2024, *1*, doi:10.1007/s44336-024-00009-2.

Lin, T.; Wang, Y.; Liu, X.; Qiu, X. A Survey of Transformers. *AI Open* 2022, *3*, doi:10.1016/j.aiopen.2022.10.001.

Lund, S.T.; Villadsen, J. On Verified Automated Reasoning in Propositional Logic: Teaching Sequent Calculus to Computer Science Students. *Vietnam Journal of Computer Science* 2024, *11*, doi:10.1142/S2196888824500064.

Magnus Chukwuebuka Ahuchogu Evaluating the Impact of Generative AI on Intelligent Programming Assistance and Code Quality. *Power System Technology* 2025, *49*, doi:10.52783/pst.1668.

Malamas, N.; Tsardoulias, E.; Panayiotou, K.; Symeonidis, A.L. Toward Efficient Vibe Coding: An LLM-Based Agent for Low-Code Software Development. *J Comput Lang* 2025, *85*, doi:10.1016/j.cola.2025.101367.

Mao, Y.; Ge, Y.; Fan, Y.; Xu, W.; Mi, Y.; Hu, Z.; Gao, Y. A Survey on LoRA of Large Language Models. *Front Comput Sci* 2025, *19*.

Melo, A.; Fagundes, R.; Lenarduzzi, V.; Santos, W.B. Identification and Measurement of Requirements Technical Debt in Software Development: A Systematic Literature Review. *Journal of Systems and Software* 2022, *194*, doi:10.1016/j.jss.2022.111483.

Mohammed, A.S.; Saddi, V.R.; Gopal, S.K.; Dhanasekaran, S.; Naruka, M.S. AI-Driven Continuous Integration and Continuous Deployment in Software Engineering. In

Proceedings of the 2024 2nd International Conference on Disruptive Technologies, ICDT 2024; 2024.

Moniruzzaman, M.; Alam, A.M. Integration of LLM and ReAct Agents for Enhanced Context Oriented Programming. In Proceedings of the 2024 27th International Conference on Computer and Information Technology, ICCIT 2024 - Proceedings; 2024.

Moreschini, S.; Arvanitou, E.M.; Kanidou, E.P.; Nikolaidis, N.; Su, R.; Ampatzoglou, A.; Chatzigeorgiou, A.; Lenarduzzi, V. The Evolution of Technical Debt from DevOps to Generative AI: A Multivocal Literature Review. *Journal of Systems and Software* 2026, *231*, doi:10.1016/j.jss.2025.112599.

Mumtaz, M.; Ahmad, N.; Usman Ashraf, M.; Alghamdi, A.M.; Bahaddad, A.A.; Almarhabi, K.A. Iteration Causes, Impact, and Timing in Software Development Lifecycle: An SLR. *IEEE Access* 2022, *10*.

Nakash, I.; Kour, G.; Uziel, G.; Anaby Tavor, A. Breaking ReAct Agents: Foot-in-the-Door Attack Will Get You In.; 2025.

O'Connor, A.J. *Organizing for Generative AI and the Productivity Revolution: Reshaping Organizational Roles in the Age of Artificial Intelligence*; 2024;

Ouyang, T.; MaungMaung, A.P.; Konishi, K.; Seo, Y.; Echizen, I. Stability Analysis of ChatGPT-Based Sentiment Analysis in AI Quality Assurance. *Electronics (Switzerland)* 2024, *13*, doi:10.3390/electronics13245043.

Pan, B.; Lu, J.; Wang, K.; Zheng, L.; Wen, Z.; Feng, Y.; Zhu, M.; Chen, W. AgentCoord: Visually Exploring Coordination Strategy for LLM-Based Multi-Agent Collaboration. *Comput Graph* 2025, *132*, doi:10.1016/j.cag.2025.104338.

Pareek, C.S. Accelerating Agile Quality Assurance with AI-Powered Testing Strategies. *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT* 2024, *08*, doi:10.55041/ijsrem15369.

Pelluru, K. LangChain & LangGraph in Production: Architectures for Multi-Agent LLM Systems. *Journal of Data and Digital Innovation (JDDI)* 2025, *2*.

Pirnau, M.; Botezatu, M.A.; Priescu, I.; Hosszu, A.; Tabusca, A.; Coculescu, C.; Oncioiu, I. Content Analysis Using Specific Natural Language Processing Methods for Big Data. *Electronics (Switzerland)* 2024, *13*, doi:10.3390/electronics13030584.

Pirzado, F.A.; Ahmed, A.; Mendoza-Urdiales, R.A.; Terashima-Marin, H. Navigating the Pitfalls: Analyzing the Behavior of LLMs as a Coding Assistant for Computer Science Students - A Systematic Review of the Literature. *IEEE Access* 2024, *12*, doi:10.1109/ACCESS.2024.3443621.

Polinati, A. kumar Devops And Ai: Automating Software Delivery Pipelines For Continuous Integration And Deployment. *Nanotechnol Percept* 2024, *20*.

Ramachandrappa, N.C. SOLID Design Principles in Software Engineering. *International Journal of Computer Trends and Technology* 2024, *72*, doi:10.14445/22312803/ijctt-v72i9p104.

Rekart, J.L.; Baker, R. OI, AI, and Research or Why OI Is the GOAT and AI Is the BLOAT. In *Designing for Human Intelligence in an Artificial Intelligence World*; 2025.

Roy, D.; Zhang, X.; Bhave, R.; Bansal, C.; Las-Casas, P.; Fonseca, R.; Rajmohan, S. Exploring LLM-Based Agents for Root Cause Analysis. In Proceedings of the FSE Companion - Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering; 2024.

Samuelson, P. Legal Challenges to Generative AI, Part I: Questioning the Legality of Using in-Copyright Works for Training Data and Producing Outputs Derived from Copyrighted Training Data. *Commun ACM* 2023, *66*.

Saravanos, A.; Curinga, M.X. Simulating the Software Development Lifecycle: The Waterfall Model. *Applied System Innovation* 2023, *6*, doi:10.3390/asi6060108.

SECURITY FOR CONTINUOUS INTEGRATION AND CONTINUOUS DEPLOYMENT PIPELINE. *International Research Journal of Modernization in Engineering Technology and Science* 2024, doi:10.56726/irjmets50676.

Shaker Mahmoud, A.A.; Shishah, W.; Mistry, N.R. ReACT_OCRS: An AI-Driven Anonymous Online Reporting System Using Synergized Reasoning and Acting in Language Models. *IEEE Access* 2025, doi:10.1109/ACCESS.2025.3571526.

Silva, V. The ILO and the Future of Work: The Politics of Global Labour Policy. *Glob Soc Policy* 2022, *22*, doi:10.1177/14680181211004853.

Tang, X.; Jin, Q.; Zhu, K.; Yuan, T.; Zhang, Y.; Zhou, W.; Qu, M.; Zhao, Y.; Tang, J.; Zhang, Z.; et al. Prioritizing Safeguarding Over Autonomy: Risks of LLM Agents for Science. *Nature Communications 2025 16:1* 2024, *16*.

Tang, Y.; Liu, Y.; Lan, J.; Yan, Z.; Gelenbe, E. Security of LLM-Based Agents Regarding Attacks, Defenses, and Applications: A Comprehensive Survey. *Information Fusion* 2026, *127*, doi:10.1016/j.inffus.2025.103941.

Tarek RADAH ReAct-Driven SOC Agent with Integrated Detection Engineering for AI-Enhanced Autonomous Alert Handling. *Journal of Information Systems Engineering and Management* 2025, *10*, doi:10.52783/jisem.v10i53s.10967.

Tay, Y.; Dehghani, M.; Bahri, D.; Metzler, D. Efficient Transformers: A Survey. *ACM Comput Surv* 2023, *55*, doi:10.1145/3530811.

Thomas, H.; Anner, M. Dissensus and Deadlock in the Evolution of Labour Governance: Global Supply Chains and the International Labour Organization (ILO). *Journal of Business Ethics* 2023, *184*, doi:10.1007/s10551-022-05177-z.

Vemuri, N.; Thaneeru, N.; Tatikonda, V.M. AI-Optimized DevOps for Streamlined Cloud CI/CD. *Int J Innov Sci Res Technol* 2024, *9*.

Wang, J.; Chen, S.; Zhu, H. Verification of Scapegoat Trees Using Dafny. In Proceedings of the Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); 2024; Vol. 14627 LNCS.

Wu, J.; Zhu, J.; Liu, Y. Agentic Reasoning: Reasoning LLMs with Tools for the Deep Research. *ArXiv* 2025.

Xi, Z.; Chen, W.; Guo, X.; He, W.; Ding, Y.; Hong, B.; Zhang, M.; Wang, J.; Jin, S.; Zhou, E.; et al. The Rise and Potential of Large Language Model Based Agents: A Survey. *Science China Information Sciences* 2025, *68*.

Xiao, Z.; He, X.; Wu, H.; Yu, B.; Guo, Y. EDA-Copilot: A RAG-Powered Intelligent Assistant for EDA Tools. *ACM Transact Des Autom Electron Syst* 2025, *30*, doi:10.1145/3715326.

Xing, F. Designing Heterogeneous LLM Agents for Financial Sentiment Analysis. *ACM Trans Manag Inf Syst* 2025, *16*, doi:10.1145/3688399.

Yan, B.; Li, K.; Xu, M.; Dong, Y.; Zhang, Y.; Ren, Z.; Cheng, X. On Protecting the Data Privacy of Large Language Models (LLMs) and LLM Agents: A Literature Review. *High-Confidence Computing* 2025, *5*.

Yan, X.; Yang, X.; Jin, N.; Chen, Y.; Li, J. A General AI Agent Framework for Smart Buildings Based on Large Language Models and ReAct Strategy. *Smart Construction* 2025, *2*, doi:10.55092/sc20250004.

Yanakiev, I.; Lazar, B.M.; Capiluppi, A. Applying SOLID Principles for the Refactoring of Legacy Code: An Experience Report. *Journal of Systems and Software* 2025, *220*, doi:10.1016/j.jss.2024.112254.

Yuksekgonul, M.; Bianchi, F.; Boen, J.; Liu, S.; Lu, P.; Huang, Z.; Guestrin, C.; Zou, J. Optimizing Generative AI by Backpropagating Language Model Feedback. *Nature* 2025, *639*, doi:10.1038/s41586-025-08661-4.

Yuvzhenko, D.; Chymshyr, V.; Shymkovych, V.; Znova, K.; Nowakowski, G.; Telenyk, S. A Multimodal Retrieval-Augmented Generation System with ReAct Agent Logic for Multi-Hop Reasoning. *Information, Computing and Intelligent systems* 2025, doi:10.20535/2786-8729.6.2025.330777.

Zacarias, R.O.; Antunes, L.C.R.; Barros, M. de O.; Santos, R.P. dos; Lago, P. Exploring Developer Experience Factors in Software Ecosystems. *Journal of Systems and Software* 2025, *230*, doi:10.1016/j.jss.2025.112549.