ESP-JETA
ESP Journal of Engineering & Technology Advancements
ISSN: 2583-2646 / Volume 6 Issue 1 March 2026
Paper Id: JETA-V6I1P115 / Page No: 131-139

Original Article

AI-First Software Development Lifecycle: An Agent-Driven Framework for Autonomous Planning, Coding, Testing, and Deployment

Ambar Nath Saha[1], Debashis Patra[2]

[1]*Engineering Manager, Cognizant Technology Solutions Canada Inc, Nova Scotia, Canada.*
[2]*Computer Science, UT Austin, TX, USA.*

Received Date: 22 February 2026 Revised Date: 28 February 2026 Accepted Date: 29 March 2026

Abstract*: In the era of automation, society has invested significant effort in automating repetitive processes across various sectors to reduce the manufacturing time of many products. However, we have not given similar attention to automating software development, as it involves complex decision-making, contextual understanding, and requires human expertise and coordination.*

Historically, most organizations followed the waterfall methodology for the Software Development Life Cycle (SDLC), and in the early 21st century, they rapidly adopted agile methodologies with the expectation of delivering more robust and scalable products within a shorter timeframe. However, human involvement has remained central in all these methodologies until the emergence of Agentic AI.

Agentic AI has the potential to transform software development in ways that have not been previously explored. In this paper, we propose an agent-driven SDLC framework that adopts an AI-first approach to software development, where human involvement is limited to governance and decision-making.

The framework introduces a Central Orchestrator Agent that coordinates with specialized agents responsible for backlog planning, solution architecture, code generation, automated testing, code review, CI/CD, deployment orchestration, and production monitoring with self-healing capabilities.

This AI-first approach can significantly reduce human effort and software release time, maintain high code quality through automated validation, and enable rapid incident response through autonomous hotfix generation and rollback mechanisms.

Keywords*: AI Agents, Autonomous Software Engineering, CI/CD Automation, Large Language Models, LLM Guardrails, Multi-Agent Orchestration, Self-Healing Systems, Software Development Lifecycle.*

I. INTRODUCTION

Software development has evolved over time and today it has become a mature and significant industry in the modern world. For a long time, organizations had accepted the Waterfall model as the most suitable model for the software development life cycle. After that, agile methodology came into the picture, which addressed many of the existing issues related to the Waterfall methodology and became more flexible to requirement changes and was able to produce robust, scalable software. Recently, a lot of automation has been introduced on the DevOps side, especially in CI/CD pipelines.

Even after all the changes, nowadays, the entire software development cycle is still greatly dependent on humans and every resource's expertise. Starting from understanding requirements, breaking them down into small tasks, writing code, testing, and deployment – every step requires human involvement, and because of that, the quality of the software and the release timeline become inconsistent, and managing a huge team becomes a bottleneck for many companies. Although the project manager role was introduced to manage this type of large resource strength, most of the time this approach has not delivered the desired and predictable results.

Now, with the introduction of Large Language Models (LLMs), things have started changing slowly. Most resources can receive some help from models like GPT, Claude, and Gemini, and they are able to complete software development in a short period of time, but still the entire SDLC is maintained by humans. Also, all those LLMs are still being used as supporting tools,

and the quality of the outcome from LLMs is still dependent on the individual resource's prompt. Although the resource is not directly writing code, they are the ones who feed all the information and provide all the requirements to the LLMs. In a nutshell, generative AI has helped developers a lot, but that has not really helped the entire organization automate the SDLC process to create more robust, scalable, and predictable products.

In this paper, we're approaching the problem from a completely different perspective. We're trying to build a process where AI takes the primary responsibility to build a software product, and human presence will be there at a couple of important checkpoints. The idea is to create multiple agents where each agent will be responsible for a specific task in the process, like requirement gathering, breaking it down into stories or tasks, creating test cases, writing code, reviewing code, running test cases, deploying code to different environments, etc. To maintain all those agents, there will be a central orchestrator agent which will coordinate all the agents, follow the sequence of work, and coordinate the flow and interaction between different agents so that the entire process moves in a structured way.

Please don't think that human interaction will totally be removed from this process. Their involvement is going to be very limited, and they will be present at a couple of important checkpoints like selecting models, validating architecture, approving final code merge, controlling the production rollout, etc. Also, there will be a shared AI knowledge repository from which all the agents will get references of the existing system architecture, data, APIs, and documentation to follow similar patterns in the development process.

II. RELATED WORK

A. AI-Assisted Software Development

Applying artificial intelligence in software engineering has grown rapidly. Github Copilot, which is built on OpenAI code, can easily generate high-functioning and complex code in any programming language. have evaluated developer productivity with AI code assistance and found a significant improvement in task completion speed, documentation, and unit testing. However, those tools are still being used as assistance, and they are totally controlled by a developer.

B. Multi-Agent Systems in Software Engineering

https://github.com/OpenBMB/ChatDev introduced a virtual software company where multiple agents were created to play different roles like CEO, CTO, programmer, and tester to develop software. MetaGPT (https://docs.deepwisdom.ai/main/en/) extended this same concept and included a standardized operating procedure to make the interactions between multiple agents more structured, which reduces cascading errors. SWE-Agent has explained that LLM agents can autonomously resolve real-world GitHub issues. Cognition Labs introduced Devin as an autonomous software engineer. Although this research showed significant advancement in the AI world, they have focused on isolated development tasks. Enterprise-level orchestration, policy enforcement, and knowledge management with AI are still yet to be explored.

C. AI Safety and LLM Guardrails

Studies raise concerns about prompt injection, hallucination, and output safety when LLMs are deployed in a production environment. To handle those scenarios, guardrails frameworks such as NeMo Guardrails and Guardrails AI provide runtime validation against predefined rules and policies. We are integrating all those safety mechanisms within the proposed SDLC framework to ensure that all agent interactions are subject to prompt safety validation and output compliance checking.

D. CI/CD and DevOps Automation

DevOps has already established robust processes for continuous integration and deployment (CICD) with the help of many tools such as GitHub Actions, Jenkins, ArgoCD, etc. However, the intelligence within these CICD pipelines still remains rule-based. This paper has proposed an integration of a CI Gate Agent which will provide AI decision-making capability within CI/CD gates.

III. PROPOSED FRAMEWORK

This section represents the AI-first SDLC framework. It consists of three cross-cutting layers, thirteen phases, multiple human-in-loop checkpoints, and a central orchestrator agent to govern and coordinate the entire software development process. We are going to explain the framework through a concrete scenario: introduce a user notification preference system to allow users to control how they receive alerts (email, SMS, push notifications). Please check Figure 1 below, which demonstrates the complete lifecycle flow with distinct color coding: blue for product design phases, green for agentic control panel, maroon for shared intelligence layer, yellow for build validate and release and gray for runtime learning and recovery.

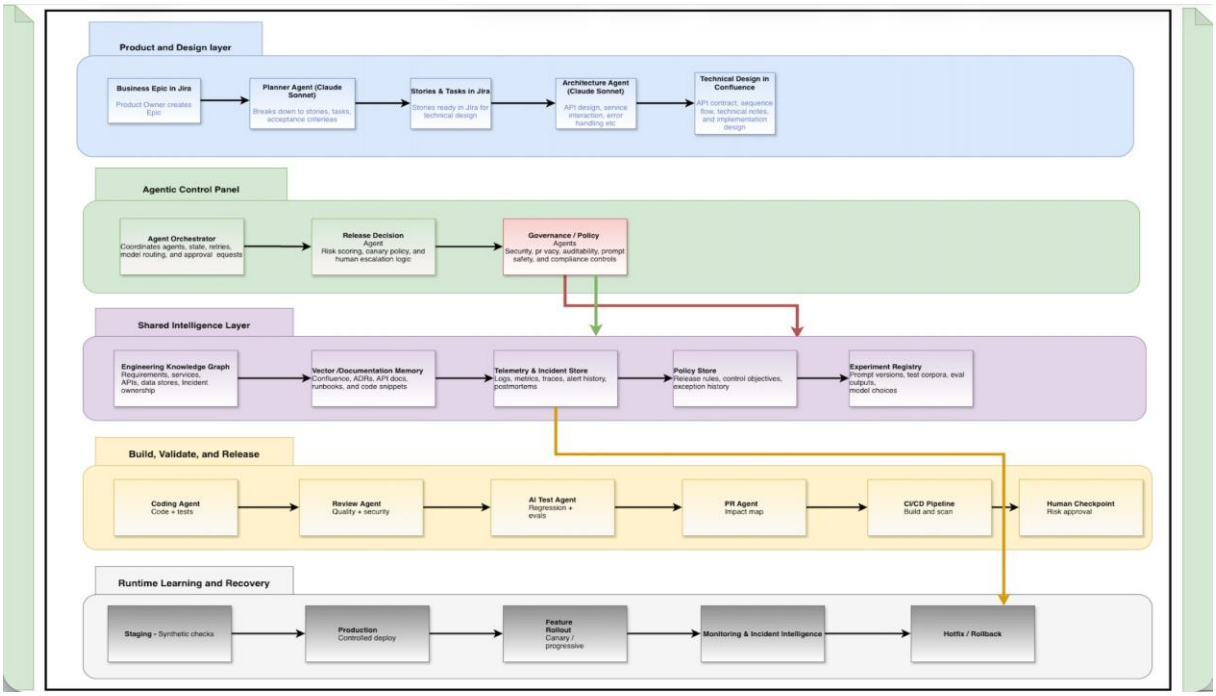

Figure 1: *AI-First Software Development Lifecycle Framework with Central Orchestrator, Knowledge Layer, and Policy Guardrails*

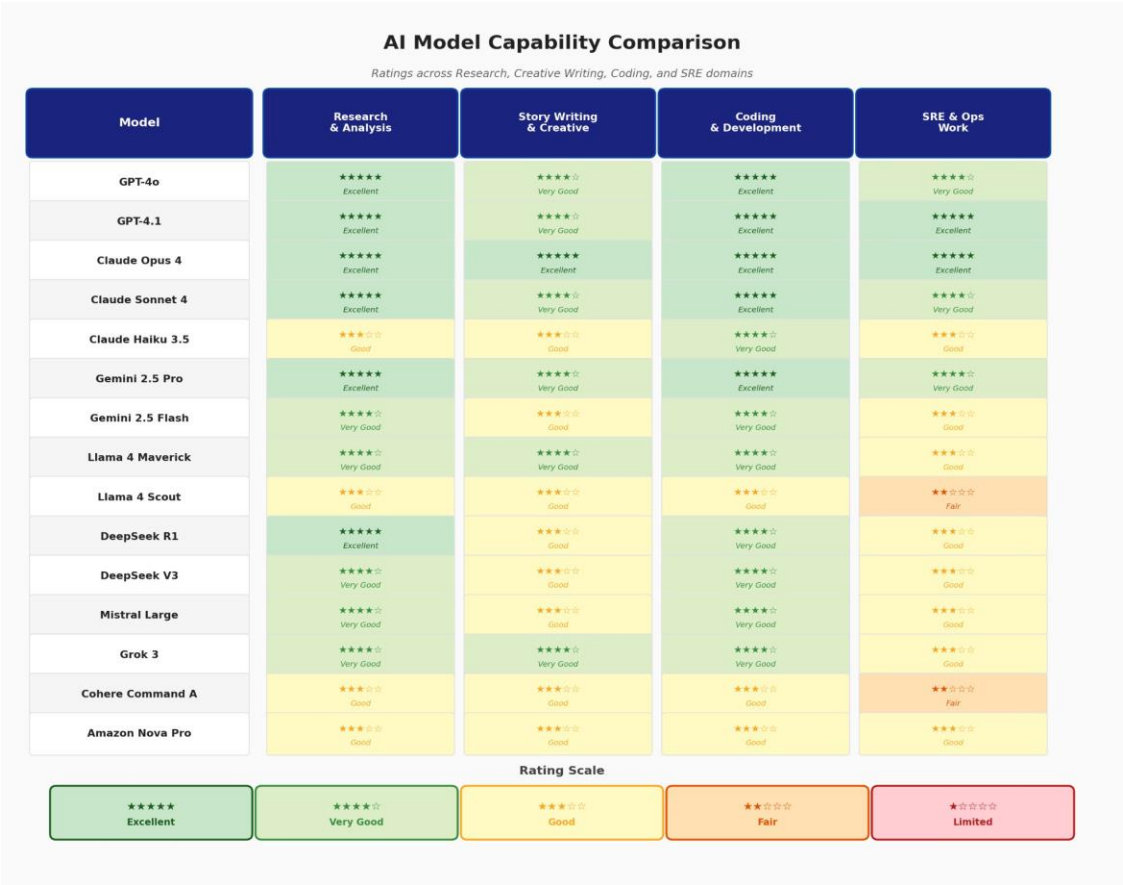

Figure 2: Model compared to create different type of Agent powered by correct model and MCP servers

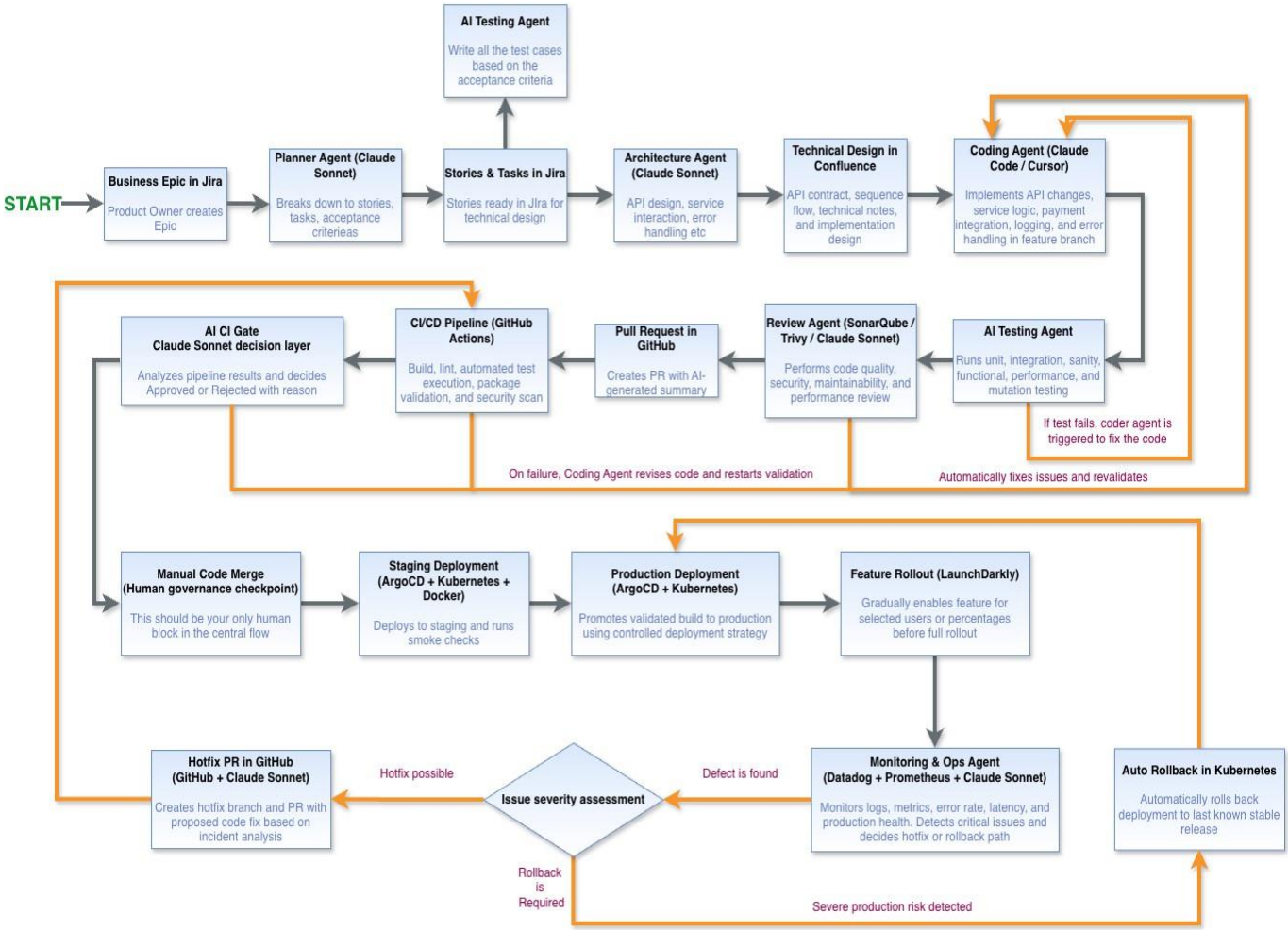

Figure 3: Detail workflow diagram

Execution Tracking & Decision Ledger

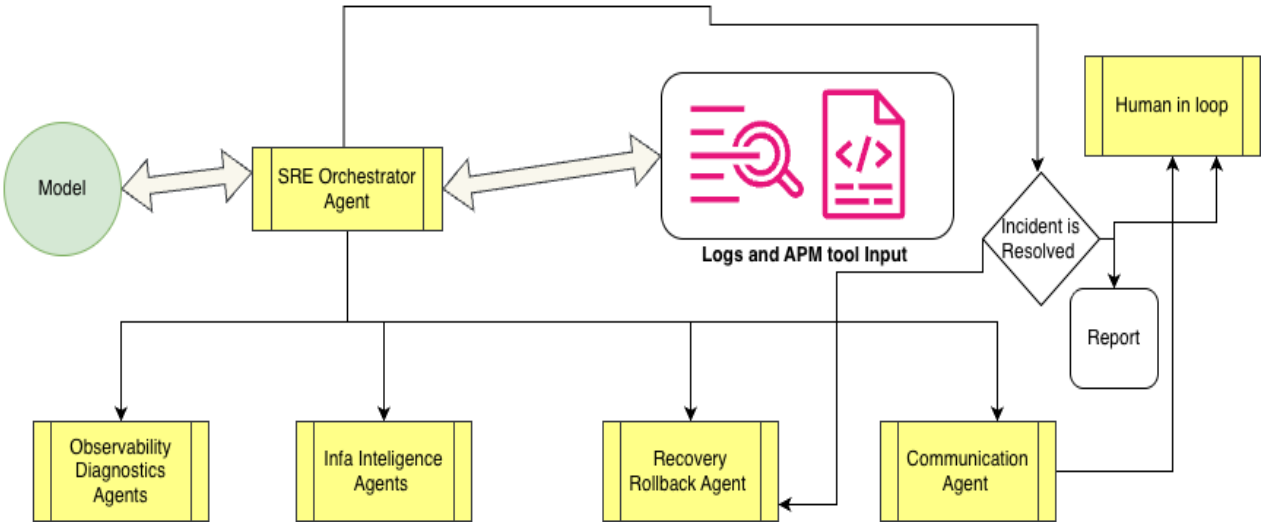

Figure 4: SRE work flow management using Agentic AI

A. Central Orchestrator Agent

The Central Orchestrator Agents plays a project manager role in the entire software development life cycle. As this agent is powered by LLM, it can take many adoptive decisions. The main responsibility of the Central Orchestrator is to monitor every agent's health and task status, and maintain the sequence of flow. It takes care of any agent's failure through the escalation policies. As it possesses the reasoning capabilities, it can decide either to skip human intervention for any low-risk changes or to send for the human review if the agent has low confidence.

B. Knowledge Layer

This layer plays a very important role in SDLC. This layer acts as an AI-powered knowledge repository. It contains design documents, all the existing API specifications, existing architecture information, database information, code patterns, historical data and production incidents, all incident logs, compliance rules, etc. During software development phases, every agent can extract all the necessary information from the knowledge layer and, at the end of the task, write back to the knowledge layer if anything needs to be added or updated based on the agent's experience in solving any issues or developing new features. This knowledge layer addresses a critical gap of stateless LLM interaction by providing organizational memory; otherwise, it would have been very difficult for all the agents to complete tasks that are aligned with the organizational standard.

C. Policy Enforcement, Compliance, and LLM Guardrails

This layer works with every agent and acts as a guardrail. It comprises a couple of components:

1) Compliance Agent: A dedicated agent inside this layer that validates each lifecycle artifact against organizational policies, compliance rules, industry standards, etc. For example, while implementing push notifications, it is going to check the PII compliance as this feature is going to keep the user's email address or phone number in memory.
2) Audit Trail: This agent makes sure to log each and every action, decision, and artifact modification that takes place during the SDLC. It's very helpful when an agent needs to check all the audit logs while critical issues are being investigated.
3) Prompt Safety: It prevents any prompt injection attacks, detects hallucinated code references, and flags potentially unsafe code patterns before the code gets merged into the repository.

D. Phase 1

The lifecycle of autonomous software development starts when the product owner creates an epic using natural language. For example, the product owner will write down an epic like "introduce a user notification preference system which allows users to control how they would like to receive alerts (email, SMS, push notifications)". They will mainly describe the business scenarios, user expectations, and integration with other vendors at a high level, leaving all the technical information. Once the epic is created, an event will be detected by the Agent empowered by Model Context Protocol (MCP) that is connected to Jira, and it is going to invoke the orchestrator agent to initiate the autonomous process.

E. Phase 2

The Orchestrator activates the planner agent, which is powered by an LLM (Claude Sonet). The planner agent reads the epic and understands the business requirement so that it can create all the related stories and tasks in Jira. This agent also connects with the knowledge layer to extract information about similar existing features so that the stories can also be written in similar fashion, it checks compliance policies, security protocols, or any non-functional requirements that help the agent in creating acceptance criteria. Finally, the epic will be converted into stories and tasks with proper acceptance criteria in the Jira backlog.

F. Phase 3

Once all those stories and tasks are created in Jira, an architecture event is triggered by the Orchestrator, which reads the Jira backlog and prepares a detailed technical design, writes it in Confluence, and simultaneously updates the knowledge layer. The design artifact consists of API contracts, changes in the integration layer, database-related changes, sequence diagrams, proposed architecture diagrams, etc.

Human-in-the-loop: In this phase, human review is required if the implementation is complex in nature or a high-risk feature. The Orchestrator decides the complexity or risks, and based on that, it sends a communication to a human architect for review. A senior architect or tech lead reviews the entire technical design and either suggests some changes to the existing design blueprint or scraps the entire AI-generated design and creates it on their own.

G. Phase 4

A Coding agent, which is operated in Claude Code or Cursor, begins the code implementation using least access principal . It wll be controlled with in specific docker environment.. It reads all the stories, tasks, and acceptance criteria from Jira and also retrieves all the knowledge of code patterns, existing frameworks, non-functional requirements, and finally produces preliminary working code in the GitHub repository. Think of a Coding Agent just like your primary developer in legacy. It automatically creates working code, including API endpoints, service layer logic, integration layer-related changes, configuration-related changes, logging, and error handling, etc.

H. Phase 5

A dedicated Testing Agent is going to get triggered, and it first writes all the test cases by reading those stories, tasks, and acceptance criteria. This agent will also connect with the knowledge layer to understand the test case patterns and any historical data about production defects to strengthen its test case variations. Once the test case writing is completed, it keeps all the test cases in Jira and also in the knowledge layer and starts executing all those test cases against the working code. The agent performs various types of testing such as functional testing, sanity testing, integration testing, load testing, mutation testing, security testing, etc. Moreover, this agent follows another process called “The Feedback-Driven Repair Loop,” where if any test case fails, then based on the observation, it is going to fix the code and run those test cases once again. This iteration continues until the working code branch reaches a stable state.

I. Phase 6

Review Agent works with SonarQube and does all sorts of code quality checks, security, scalability, and maintainability assessment. It again connects with the knowledge layer to understand all the organization coding standard, and it autonomously fixes all the review-related issues before it triggers the Pull-Request event.

J. Phase 7

The system creates a Pull-Request in GitHub repository with all the relevant information like the summary of changes, test case results, etc. It's the PR agent's responsibility to associate all the documentation such as design documents, workflows, test case results that were involved during the feature development. Most importantly, the pull request should be linked to the original story or epic.

K. Phase 8

One the PR is created, it automatically triggers the CI/CD pipeline through GitHub Actions. The pipeline performs build validation, security scanning with the help of Trivy and SonarQube. This layer works as an automated verification layer before the code gets merged to the main branch. If somehow the build fails, then Orchestrator triggers Coding Agent to solve the issue and start the process from there.

L. Phase 9

This CL Gate Agent has a reasoning capability, and it can decide whether to send the PR to humans for the final review or to reject the PR and trigger the Coding Agent with the observation. It works as a gatekeeper before it finally passes for the final review and merge. It checks all the test results, analyzes the pipeline reports, observes all the security validation reports and tries to find any hidden pattern and if CL Gate Agent’s confidence is low then it immediately rejects the PR and sends all the detailed analysis to Coding Agent for further work.

M. Phase 10

The final merge remains a human-controlled governance step. A developer, technical lead, or engineering manager reviews the approved PR and decides whether to merge. The reviewer benefits from comprehensive AI-generated context—design rationale, test results, quality reports, compliance status, and risk assessments—enabling informed decisions without manual investigation.

N. Phase 11

Once the PR is merged to the main branch, the staging deployment will immediately be done using ArgoCD, Kubernetes and Docker. System should first do some smoke and sanity testing and make sure the staging environment is healthy and the feature is working as expected. This is important to keep a human in the loop at this stage also. This is where Someone like a QA lead comes into picture who should check the system behavior and the feature workflow to make sure the new feature is working as expected and it doesn’t have any impact on the existing staging system.

O. Phase 12

Now it's time for production deployment. The feature visibility is controlled through LaunchDarkly, which means although the production deployment will be completed, the feature will be visible to the real customer in small numbers initially. In this step also human in the loop is required and the engineering manager is going to decide on what percentages (5%, 10%, 20% etc) the feature will be adapted to the user base.

P. Phase 13

Datadog and Prometheus continuously monitor logs, metrics, response times, and failure rates. If a critical issue is detected, the Orchestrator triggers an Operations Agent that analyzes telemetry, identifies probable root causes, and determines the appropriate response.

IV. RESULTS AND DISCUSSION

A. Reduction in Lifecycle Duration

The AI-First approach will reduce the Software Development timeline to a greater extent. As the entire system doesn't depend on human availability or expertise, it generates deployable code in a very short span, and automated CI/CD will make sure to have a seamless deployment, and the monitoring system will help to monitor anomalies, and self-healing capabilities will help the system to automatically rectify the issue. Generally, the traditional agile SDLC framework spans over 3 weeks to 4 weeks, but using this AI-First approach, one feature can easily be deployed within a week.

B. Code Quality and Consistency

The multi-layered validation—AI-driven testing, automated code review, CI/CD pipeline checks, Compliance Agent validation, and intelligent CI gating—produces code that consistently meets quality thresholds. The feedback-driven repair loop iteratively resolves failures before code reaches review. Studies on LLM-generated code show that iterative refinement through automated feedback significantly improves correctness .

C. Governance and Human Oversight

Although we're proposing a system where AI predominantly takes care of the entire Software Development Cycle, intentionally, we've kept four optional and one mandatory human checkpoint to make sure AI runs the show with human governance. The mandatory checkpoint is the final code merge, where we still think we need to have an engineer to validate based on the person's real-life experience before it gets pushed to the main branch, and a couple of optional human governance checkpoints are kept to make sure that the entire system is always monitored and governed by the engineering team.

D. Safety and Compliance

This is one of the most concerning areas in creating an AI-First framework, but our integrated Policy Enforcement Layer makes sure to work as a guardrail to mitigate such safety and compliance issues. Our prompt safety mechanisms prevent any kind of prompt injection attacks, and on the other hand, LLM guardrails will make sure that any agent's output is in the expected format and adheres to organizational policies. The Compliance Agent keeps checking whether everything is aligned with organizational standards.

E. Knowledge Accumulation and Reuse

The Knowledge Layer creates a compounding advantage: each feature delivery enriches the shared context available to all agents. Over successive deliveries, agents produce higher-quality outputs with fewer iterations because they can reference established patterns, known failure modes, and validated architectural decisions. This organizational learning capability distinguishes the framework from stateless AI coding assistants.

F. Self-Healing and Incident Response

The most astonishing part of this framework is the self-healing capability. In traditional human-driven SDLC, it always becomes a difficult scenario to manage because of resource availability or the lack of expertise to find the root cause of the issue when a critical issue gets reported in the production environment. But in this framework, the autonomous monitoring system keeps on monitoring for any anomaly, and if reported, the Operation Agent immediately gets triggered and starts checking the root cause of the issue without human intervention. If the Operation Agent finds the probable fix, then it automatically creates a PR and sends that for human review as a hotfix.

G. Limitation

So far, we've been discussing all the advantages of the AI-First approach, but we also should highlight a couple of limitations that every organization needs to consider while moving towards implementing this framework. First, LLM output is

always probabilistic. So, there's a chance that LLM may produce incorrect code or may produce a flawed architecture. Although we have implemented many validation layers and guardrails to make sure it always produces organizational standard output, it doesn't eliminate the risk. Second, it is assumed that all the tools (Jira, Confluence, GitHub, CI/CD, monitoring) that are going to be used are integrated properly with each other. Third, using LLMs, Agents, MCP involves costs that organizations need to consider wisely, as without proper budgeting it may surpass the human-driven SDLC framework. Fourth, as the knowledge layer is the backbone of this framework, it needs continuous curation to make sure it doesn't degrade the quality and affect the agent's performance.Engeenerring team has to be matured enough to take care this complex system.

V. CONCLUSION

In this paper, we tried to explore what it would look like if software development is not just assisted by AI but actually driven by it. Nowadays, many developers use LLMs as a helper tool while writing code, and we believe that in this way we are not able to use the core power of Agentic AI. Our idea was to build a system where different agents are going to take ownership of delivering workable software from requirement to deployment, where human involvement will be present at every decision-making checkpoint.

This framework shows that it is possible to reduce a lot of developer-level dependency and manual, hectic team coordination. With the help of a central Orchestrator, a shared knowledge base, multiple validation steps, and self-improvising or self-healing capabilities, the system can develop any feature at lightning speed while keeping the infrastructure and organizational standards intact.

But our aim is not to remove humans from the process; rather, human involvement should be there to govern the entire process, and we have kept humans in the loop when we feel that real-world experience and expertise are needed to make a concrete decision.

There are also some practical challenges that we cannot ignore. The behavior of LLMs is not always predictable, and all the various tool integrations need to be configured very carefully. Managing knowledge requires continuous effort. Finally, cost is a huge factor that every organization needs to consider and plan for properly before moving to an AI-driven framework.

Overall, through our research, we tried to make an attempt to move the conversation from "AI-assisted development" to "AI-driven development." We agree that it's a long way to go, but our AI-first framework will give a starting point for how such a development system can be built—one that is driven completely by AI but governed by humans.

VI. INTEREST CONFLICTS

The author(s) declare(s) that there is no conflict of interest concerning the publishing of this paper.

VII. FUNDING STATEMENT

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

VIII. REFERENCES

[1] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for agile software development*. Agile Alliance.

[2] Kim, G., Humble, J., Debois, P., & Willis, J. (2021). *The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations* (2nd ed.). IT Revolution Press.

[3] Vaithilingam, P., Zhang, T., & Glassman, E. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*.

[4] Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., Liu, Z., & Sun, M. (2023). *Communicative agents for software development*. arXiv. https://arxiv.org/abs/2307.07924

[5] Yang, J., Jimenez, C., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). *SWE-agent: Agent-computer interfaces enable automated software engineering*. arXiv. https://arxiv.org/abs/2405.15793

[6] Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley.

[7] Smart, J. (2011). *Jenkins: The definitive guide*. O'Reilly Media.

[8] Argo Project. (2024). *Argo CD: Declarative GitOps CD for Kubernetes*. https://argo-cd.readthedocs.io Wooldridge, M. (2009). *An introduction to multiagent systems* (2nd ed.). Wiley. Nonaka, I., & Takeuchi, H. (1995). *The knowledge-creating company: How Japanese companies create the dynamics of innovation*. Oxford University Press.

[9] Cohn, M. (2004). *User stories applied: For agile software development*. Addison-Wesley.

[10] Bass, L., Clements, P., & Kazman, R. (2021). *Software architecture in practice* (4th ed.). Addison-Wesley.

[11] Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, 37*(5), 649–678.

[12] Kang, S., Yoon, J., & Yoo, S. (2023). Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*.

[13] Poulton, N. (2023). *Docker deep dive*. Independently Published. Sridharan, C. (2018). *Distributed systems observability*. O'Reilly Media.

[14] LaunchDarkly. (2024). *Feature flag management platform*. https://launchdarkly.com Datadog. (2024). *Datadog cloud monitoring platform*. https://www.datadoghq.com Brazil, B. (2018). *Prometheus: Up & running*. O'Reilly Media.

[15] Beyer, B., Jones, C., Petoff, J., & Murphy, N. (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly Media.

[16] Liu, J., Xia, C., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems, 36*.