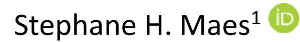
Vibe-Coding and SDLC Constrained And Managed By An Application-Aware AI-Like Agentic Platform

Stephane H. Maes[1]



March 11, 2026

Abstract

The contemporary enterprise software environment is defined by a critical market failure known as the Deployment Paradox. Despite unprecedented capital allocation toward Generative AI infrastructure, a vast majority of enterprise AI pilots fail to graduate to production environments or deliver measurable financial returns. A non-negligible contributor to this failure is the less than stellar outcome from the adoption of Ai assistant and vibe coding, a development paradigm utilizing natural language prompts to generate software autonomously. While vibe coding compresses software development cycles, it introduces new challenges in explainability, security, maintenance and support. Also, it operates at a low granularity of intent. It also increases code volume, with limited to no focus over architectural integrity.

Despite grandiose expectations, developers often spend the same or more time developing and maintaining, and enterprises have to hire new people, to compensate for those who were let go. Indeed, the traditionally recommended mitigation strategy involves applying rigorous Software Development Life Cycle practices, e.g., DevOps, Agile methodologies, to AI generated code snippets. This manual intervention negates the velocity benefits of AI coding and traps organizations in endless integration cycles. This paper proposes a paradigm shift towards using an agentic platform to autonomously perform the AI/vibe coding based on high level intent conversations with a meta-agent and a model the constraints derived on an Application Aware AI utilizing a Real Time Discovery and Coding engine. By deploying a meta agent that interacts with a developer agent within a platform managed lifecycle, enterprises can automate semantic verification and continuous optimization. This architecture leverages a deterministic model of constraints, transactional object memory (for reliability and rewind), and secure sandboxing to neutralize the inherent risks of probabilistic Large Language Models. We detail how this embedded agentic infrastructure addresses the limitations of vibe coding, ensuring secure, maintainable, and self evolving enterprise software systems capable of disrupting traditional enterprise applications.

The application-aware AI agentic platform that we detail is based on Zenera offerings. Others can be considered as long if they follow principle enumerated in this paper of constrained vibe coding.

---

1. Introduction: The Deployment Paradox and the Evolution of Code Generation

[1] shmaes.physics@gmail.com

Cite as: Stephane H. Maes, (2026), "Vibe-Coding and SDLC Constrained And Managed By An Application-Aware AI-Like Agentic Platform", https://zenodo.org/doi/10.5281/zenodo.18972674, https://shmaes.wordpress.com/2026/03/11/vibe-coding-and-sdlc-constrained-and-managed-by-an-application-aware-ai-like-agentic-platform/, March 11, 2026. (https://osf.io/vz6jt/files/yv2xf)

The modern enterprise information technology infrastructure is undergoing a profound structural shift, transitioning from deterministic, rules based operational models toward autonomous, stochastic systems capable of continuous learning and adaptation [8]. Historically, software engineering relied on precise, manual coding practices governed by rigorous architectural standards. The advent of Large Language Models (LLMs) and Generative AI has disrupted this paradigm, introducing new methodologies for rapid application development. By early 2026, the enterprise AI technology market reached a critical inflection point defined by the Deployment Paradox [1]. Despite an estimated annual capital allocation of 30 to 40 billion dollars toward Generative AI infrastructure, a staggering 95 percent of enterprise AI pilots fail to graduate to production environments or deliver measurable financial returns [1].

This failure is not attributable to a lack of model intelligence, or token generation capabilities. It is a consequence of fundamental architectural deficiencies in how probabilistic AI models are integrated into the deterministic, legacy reality of complex enterprise data systems, as well as the gap between what AI coding assistants and vibe coding deliver, versus what it takes to produce, maintain, and support quality, and secure enterprise-grade software (See [3,4] and references therein).

The enterprise landscape is a heterogeneous mix of modern microservices, legacy mainframes, proprietary Enterprise Resource Planning (ERP) systems, and unstructured document repositories [7]. AI teams involved in AI projects, and sometimes everybody else in the enterprise, are often not the owners or familiar with many of the systems [6,9-13]. They do not know all their APIs, or some are missing, they do not know the schemas. Addressing all this can be done but it takes a lot of time and efforts. It takes too long in the ear of AI, and it is a major factor in failure of many enterprise AI projects, simply because challenges pile up one after the others, and approaches like chat bots/copilot-like, Lang chain and Lang graph, or all the RAG variations, or even the typical MCP/A2A agentic approaches with hierarchy of agents, requires access to all the data, without easy way to do so, and have a lot of other problems [9]. In addition, so far, these systems have always operated on strict, deterministic logic that is fundamentally incompatible with the probabilistic, black box nature of GenAI [7].

Amidst this integration crisis, the practice of vibe coding emerged as a popular development trend [2-4]. Coined by AI researchers in early 2025, vibe coding refers to a prompt driven approach where developers, or nonprogrammers, describe a software idea in natural language and allow an AI assistant to generate the implementation [2,3,14]. Vibe coding fully embraces exponential productivity gains, shifting the developer role from writing syntax to guiding an AI through conversational feedback. Functions can be generated in minutes, application programming interfaces (APIs) are wired up quickly, and boilerplate code disappears [4].

However, this democratization of coding has severe repercussions for enterprise software products ([3,4] and references therein). Vibe coding operates at a very low granularity of intent, i.e., a user describes a function, the agent generates a function, and these functions are stitched together to form a workflow [3]. While this produces a massive volume of code, more code does not automatically create enterprise value [17,19]. Instead, it leads to bloated software, where maintenance complexity grows, documentation lags behind, and critical security gaps emerge [3,4,5,15,17,19].

Yes, today AI assistants and Vibe coding tools have improved a lot [18], but we still argue that the problems discussed in [3,4,17,19] remain. Yes productivity with the right seniority of developer can increase dramatically [15], but it does not necessarily mean more speed or lower cost at enterprise scales, where the full SDLC (Software development Lifecycle) needs to be considered and managed. The latest releases of OpenAI and Claude have not overcome the challenges in maintenance and support, they still result in enterprise in problems with junior developers overwhelmed by the “productivity”, and unable to ensure maintainability of the code, only senior

developers can do it consistently [3,15,17,19]. In fact [15,16] do not really consider the rest of the SDLC[2]. And that is despite that fact that some claim that AI now codes better than humans [18].

To overcome the Deployment Paradox and the limitations of vibe coding, enterprises require a fundamentally new category of intelligence. We have shown that Application Aware AI is a approach [6,8-13,20], granted that there may be others.

This paper explores how an embedded meta agent architecture, interacting with a developer agent via Real Time Discovery and Coding (RTDC), solves the maintenance and security crises of vibe coding. By utilizing a model of constraints, transactional object memory, and ephemeral sandboxing, this platform managed approach replaces the brittle manual integration, enabling the safe industrialization of enterprise AI [6-13,20].

We then generalize the lessons learned, and proposed approach to a new pattern for what we call constrained vibe coding, where using application-aware AI agentic is an example.

2. The Support and Maintenance Limitations of Vibe Coding

While vibe coding offers remarkable speed and accessibility, it sacrifices the architectural integrity and maintainability that structured development prioritizes. [3,4] documented these limitations regarding the gotchas of AI[3] coding, pointing out that the long term risks to the software lifecycle (may) often outweigh, for now at least, the initial productivity benefits as in [3,4,5,18] and references therein. The core vulnerabilities of vibe coding span several critical dimensions of software engineering.

2.1 The Crisis of Incomprehensibility and Vibe Debugging

The most significant complication of vibe coding is that the generated codebase frequently becomes completely incomprehensible, even to the developers who originally prompted its creation [3,4]. In traditional software engineering, a developer understands the logical flow, variable assignments, and state transitions of their program. In vibe coding, the AI acts as a black box synthesis engine, outputting verbose, complex, and sometimes copy-

[2] As discussed in [3.4], and clearly demonstrated as still the case now [15,16], it is really a terrible oversight that most publications discussing AI coding, do not discuss the main drawback: a messy and costly SDLC. [15] discusses it, but [16] completely ignores it. It induces people adoption AI coding into exuberant optimism, wrong investment decisions, and terrible after the fact situation. Shame on all these authors pushing AI and vibe coding, without the right caveats (again, yes, [15] does mention it).

[3] i.e., AI coding assistants (think of Cursor), and vibe coding. Albeit the trends may seem to be more and more towards vibe coding, no matter how much many in enterprises still poopoo the latter. [21,22] present a bit of both views.

pasted logic, which satisfies the immediate prompt, but lacks cohesive design. With good practices he documents all these points within the software.

When the software inevitably encounters an error or requires an update, the maintainer faces a massive comprehension gap. Because the underlying logic is opaque, systematic troubleshooting becomes impossible [3,4]. Developers are forced into a cycle of vibe debugging, i.e., an ad hoc process of feeding error messages back into the AI assistant and blindly tweaking prompts until the application seemingly works again [3,4]. This trial and error approach is unsustainable for professional software products that require long term roadmaps, backward compatibility, and guaranteed uptime [4,5]. As the application scales, the inability to trace logic directly correlates to exponentially increasing resolution times for critical incidents.

2.2 The 2026 Landscape of AI Software Engineering

Qwe are seeing recent AI and Vibe Coding Improvements. Yes the challenges discussed in [3-5] and references therein are still alive: SDLC remains a challenge [23]

The rapid sequence of frontier Large Language Model releases in early 2026, i.e., most importantly Claude 4.6, GPT-5.4, DeepSeek-V4, and Mistral Large 3, has undeniably propelled artificial intelligence past historical ceilings in software engineering capability. Recent benchmarks illustrate superhuman capabilities in synthetic, isolated tasks. For example, Claude Opus 4.6 achieves an unprecedented 80.8% on the repository-level SWE-bench, while OpenAI's GPT-5.4 scores between 77.2% and 80%, alongside a 75% score that effectively surpasses human baselines for automated desktop navigation. In parallel, open-weight alternatives like DeepSeek-V4 are demonstrating near-parity on complex benchmarks at fractional inference costs, democratizing enterprise-scale agentic code generation [24].

However, translating these nominal capabilities into sustained organizational productivity presents severe systemic challenges. A 2026 study conducted surveying nearly 3,000 developers reveals a stark dichotomy: while 86% of developers report high satisfaction with AI coding tools, their actual temporal savings are remarkably modest, with only 18% of the workforce saving more than two hours per week [25]. This discrepancy highlights a Great Toil Shift. According to [26], developers continue to spend roughly 25% of their week on toil tasks [26]. Instead of writing repetitive boilerplate syntax, developers now expend their energy managing technical debt, interpreting verbose AI outputs, and correcting subtle algorithmic flaws [26], still aligned with [3,4]. The organizations that are successfully capturing massive, 33-fold reductions in resource consumption are those transitioning from horizontal specialization to Vertical Integration, fundamentally restructuring human workflows around AI orchestration rather than simply substituting human typing speeds [27]. Furthermore, at the highest frontiers of algorithmic heuristic discovery and logic, elite human programmers still edge out leading AI models, demonstrating that human cognitive abstraction and stamina in novel scenarios remain un-replicated [28]. Yet Ai best models still fail miserably at some new AI coding test [29].

Beyond pure productivity, the unrestricted acceleration of code synthesis is creating a profound crisis in software maintainability. AI tools, optimized for rapid text completion over architectural planning, heavily exacerbate technical and cognitive debt. A deep analysis of 4,442 distinct Java assignments demonstrates a disturbing trend: highly capable models frequently produce overly complex code [30]. For example, GPT-5-minimal generated 490,010 lines of code with a staggering cyclomatic complexity of 145,099 for tasks where other models generated

far less, significantly reducing human readability and increasing downstream maintenance costs [30]. This degradation translates to AI-assisted pull requests having 1.7 times more structural issues than human-authored equivalents [31]. Furthermore, developers are encountering cognitive debt, i.e., a loss of their own mental models required to understand and modify the software they oversee [32]. This creates a dangerous verification bottleneck. 82% of developers say AI speeds up coding, but trust in the generated output is simultaneously plummeting. [3,4] still hold splendidly, in term of the problems and our predictions.

Security and explainability present similarly complex trade-offs. The integration of unrestricted AI coding assistants has been correlated with a 23.7% absolute increase in security vulnerabilities within production branches [33]. Dynamic benchmarks like CyberGym [34], which task agents with autonomously navigating and patching historical real-world vulnerabilities, reveal that even state-of-the-art systems like Claude Opus 4.6 (66.6% success rate) and GPT-5 (60.2%) fail to detect or resolve roughly a third of complex security flaws [34].

In response, the industry is trying to save itself by transitioning from Code Generation to Code Cognition [35], something very well supported in the constrained vibe coding with application-aware AI [6,8-13,20]. Modern AI deployment in regulated enterprise environments now often mandates robust Explainable AI (XAI) capabilities, i.e., evaluated through transparency benchmarks like Reveal and SycophancyEval [36], in order to ensure models can describe their reasoning, and manage human-AI mixed-origin errors [37]. At least it means that the industry has listened to us [3,4] and many others, e.g., references in [3,4].

Ultimately, managing the 2026 AI software landscape demands strict architectural governance, such as the Plan-Do-Check-Act (PDCA) framework [38], ensuring that AI acts as a transparent, auditable cognitive collaborator rather than an autonomous proxy. In [4] we proposed other ways. In the rets of the papers we will recommend constrained Application-aware agentic AI vibe coding, and SDLC management performed by the same agentic platform that does the vibe coding.

2.3 Architectural Drift and the Junior Programmer Problem

Vibe coding is inherently feature driven rather than design driven [3,4][4]. Without strict upfront planning, the resulting applications often resemble a patchwork of disconnected solutions rather than a unified, scalable system. Organizations utilizing these tools essentially employ an infinitely fast junior programmer. While this junior programmer can write thousands of lines of code per minute, they lack the systemic awareness to adhere to SOLID principles, Domain Driven Design, or established enterprise architectural patterns [3-5].

Consequently, junior human developers ship vast quantities of AI generated code that senior engineers must later untangle and refactor. This creates an immense accumulation of technical debt. Organizations that initially reduced their development headcount to capitalize on AI efficiencies later find themselves forced to significantly increase their support and maintenance teams to manage the unstable, bloated software that vibe coding produces. There have been reports of companies firing developers to then have to hire support people, e.g., Microsoft, and other

[4] Even in its early instantiations like [39].

companies deploying AI having to rewind their plan after customer complained of the CSM degradation. It is covered in Appendix A [5].

2.4 Testing Blind Spots and Security Vulnerabilities

In the vibe coding paradigm, documentation and testing are often neglected or entirely omitted until after bugs appear in production[6]. Even when tests are implemented, they are frequently generated by the same AI that wrote the flawed code. Because developers do not deeply understand the generated implementation, they cannot verify if the AI generated tests actually cover the necessary logic, edge cases, or security boundaries [3-5]. Yet, as we mention in the footnotes, some executive seems to believe that AI allows us to get rid of QA engineers. Oh my god!

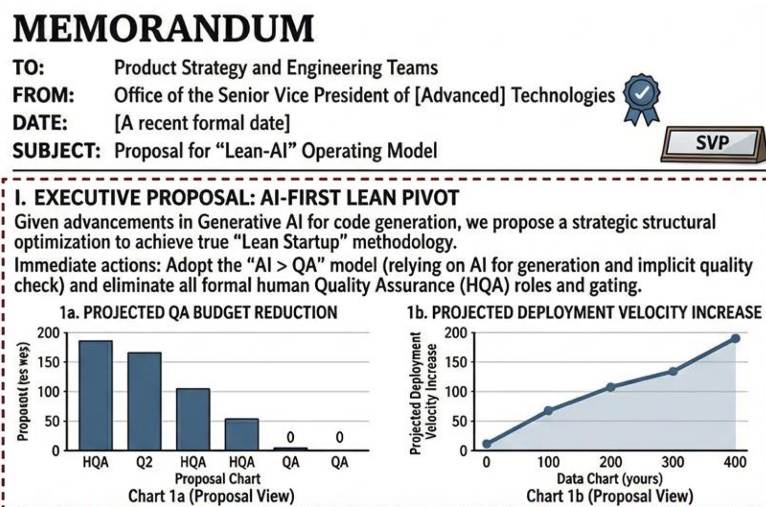


Figure 1: We heard of that story from trusted sources. It shows the misconception in enterprises about AI, especially AI and vibe coding, even in technology companies, by technology savvy executives.

---

[5] And, that is not counting also the countless cases of companies which, explicitly decided to let go the more senior developers, another ridiculous decisions, and those who decide to believe that everything can be done by AI. Let me cite for example the case of an upper R&D manager in ERP companies, SVP level, who believed that, now that they have AI vibe coding, and should be DevOps + forward facing engineers, they do not need QA anymore (everybody should write, with AI their QA testing). He went out to not promote any QA, and started firing them. We heard he as wallowed to do so. Well, we are waiting to see the outcome of that one, especially when that was for a group supposedly maintaining and migrating legacy code (think of the stranger fig pattern discussed in [5,9]). When you have confused management like that who needs competitors, or disrupting new entrants? (and where is HR when they should be involved I guess). Unfortunately, this is not an isolated story. It is typical, and characteristic, of the headless frenzy that seems to have taken over many enterprise software companies. AI attributed firings are just one manifestation of it.

[6] See how this exactly match the story we told in the previous footnote.

Feature Dimension	Traditional Structured Development	Unconstrained Vibe Coding
Development Speed	Generally slower, highly methodical.	Exceptionally fast for prototyping and MVPs.
Architectural Integrity	High by design, utilizing SOLID principles.	Low, prone to patchwork logic and structural drift.
Maintainability	High, easily modified by human teams.	Extremely low, requiring full reverse engineering.
Testing Strategy	Test Driven Development, high coverage.	Tests added late, often AI generated and unverified.
Debugging Process	Systematic tracing and root cause analysis.	Vibe debugging, trial and error prompt adjustment.

Table 1:Traditional development vs. unconstrained vibe coding, i.e., typical use of it today.

Furthermore, vibe coding bypassing rigorous security reviews, e.g., Static Application Security Testing, introduces profound enterprise risks [3-5]. AI models frequently hallucinate[7] API calls, use deprecated libraries, or implement insecure data handling practices. When these vulnerabilities are buried within thousands of lines of incomprehensible machine generated code, the enterprise attack surface expands exponentially [3-5].

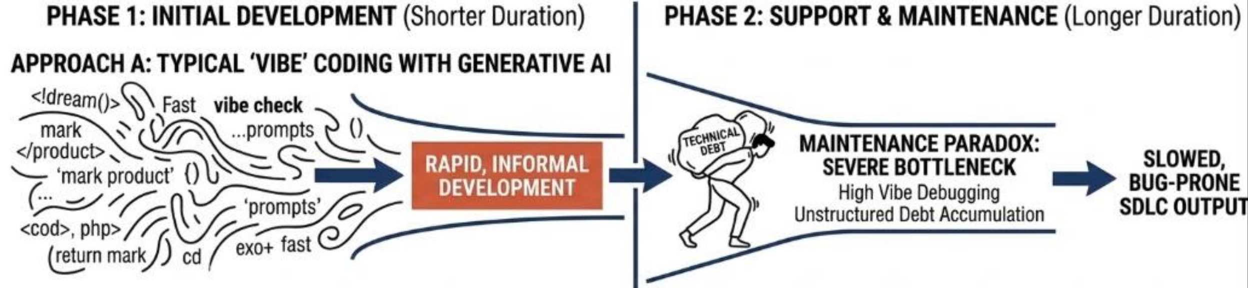


Figure 2: Proposed visual representation illustrating the maintenance paradox of typical vibe coding, and AI coding. The figure sketches a rapid initial development phase collapsing into a severe bottleneck during the support and maintenance lifecycle.

[7] It is inherent to LLMs, see for example [40], and references therein.

3. Mitigations

3.1 The Traditional Mitigation: Applying Good SDLC Practices to AI Snippets

The software industry initial response to the limitations of vibe coding has been to enforce traditional DevOps and Agile methodologies onto the AI generation process. It was also our recommendations [3-5]. This traditional method treats the LLM as an untrusted contributor, requiring human engineers to manually verify, test, and deploy every AI generated code snippet through a rigorous Software Development Life Cycle (SDLC). Since, Anthropic has now released a code review tool [43].

So AI and vibe coding is treated as junior coder, and the code is subject to all the steps and processes of the SDLC. But that typically is bot rigorously followed by junior developers, and non-developers.

Appendix B presents an overview of modern SDLC.

3.2 The VIBE4M Framework

To address the specific maintainability challenges of AI generated code, frameworks such as the Verifiable Inspection Based Enhancement for Maintainability (VIBE4M) were proposed [4]. VIBE4M is a comprehensive, multi layered approach that integrates automated and manual techniques to bridge the gap between intuition and engineering rigor. The framework mandates several strict practices spanning four specific stages of development.

First, Vibe Formalization requires developers to capture and refine the initial intuitive input into a more structured, high level representation before prompting the AI. Second, Constrained Generation dictates that the AI must synthesize code guided by established architectural principles, design patterns, and maintainability heuristics. Third, the Explication Engine stage forces the AI to automatically generate documentation, justifications, and visualizations that link the generated code back to the original intent, preventing the loss of rationale. Finally, Human in the Loop Validation incorporates rigorous human review and feedback to verify the functional and structural integrity of the output.

The implementation strategy of VIBE4M demands mandatory code reviews, automated quality checks for deprecated APIs, and comprehensive regression testing. It insists that developers treat AI generated code with extreme skepticism, demanding the same level of scrutiny applied to untrusted third party contributions.

As we will see, we can continue on this line of thought and go further and have the whole SDLC automated with an agentic platform. It will be discussed in upcoming sections.

[68] shows another example of handling parts of the SDLC rigorously coupled with AI, but we do not know what was manual, and what was automated.

3.3 The Paradox of Manual SDLC

While applying rigorous SDLC discipline (see appendix B), in general, or the VIBE4M framework [4], successfully mitigates the security and maintenance risks of vibe coding, it creates a paradox. The primary value proposition of vibe coding is very high development speed, and the democratization of software creation for nontechnical users [3]. However, executing rigorous code verifications, writing comprehensive tests, doing code reviews, and manually refactoring AI output requires deep expertise from senior software engineers [3-5].

This can significantly negate part of the initial productivity benefits. If a senior developer must spend hours deciphering, testing, and rewriting a black box snippet generated by an AI, the organization has merely shifted the labor from writing code to untangling code. For nonprogrammers, applying these rigorous maintenance policies is virtually impossible, undermining the long term viability of the products they create. This is what is observed widely in practice, why letting go the more senior developers, or firing the QA engineers, as mentioned in an earlier footnote, are such big mistakes, at least at the current state of affairs.

3.4: Vice Coding and SDLC constrained and managed by (application-aware) AI agentic platform

In the rest of the paper, we discuss another approach based on Vice Coding and SDLC constrained and managed by (application-aware) AI agentic platform [6,8-13,20], based on the coding and SDLC success we managed to build on the Zenera platform [6,13].

4. The Paradigm Shift: Application Aware AI and RTDC

In enterprises, the resulting code must often integrate to a large set, and mess, of heterogeneous systems, and data repositories. Some without well know APIs and Schemas. This manual integration approach traps enterprises in the Stitching Trap [6-13]. To connect AI features to legacy environments, engineering teams manually build brittle API wrappers and custom glue code. This process often forces organizations into cycles of 18 to 36 months of R&D, ensuring that by the time the static AI agent is deployed, the underlying business rules or database schemas have already changed. Maintenance of these manually stitched, static agents consumes 35 percent of the initial project investment annually [1,7].

The answer to this Deployment Paradox, i.e. application-aware AI with RTDC, which allow quasi zero effort integration to any well-known or barely known system is not abandoning AI coding, relies on providing pointers and metadata to the APIs, Data repositories, manual, references etc., as well as tribal knowledge, policies, rules, best practices having a meta-agent configured by voice to build a system of constraints, i.e. the guiderails, deploy agentic AI to address the target use cases, connect to these systems, develop new, and update new existing logic/processes/agents/UI, tested and run first in sand boxes, and a transactional memory and model to ensure

reliable / guaranteed agency and the possibly of action rewind [6,8-13,20]. It turns out that this is also a great way to do constrained and reliable vibe coding, where the whole SDLC runs and is managed by the meta-agent. This way we are nor forcing human engineers to manually police every token generated by a model. The solution relies a higher order system that enforces lifecycle discipline natively. This represents a paradigm shift toward Application Aware AI, defined as a new category of artificial intelligence characterized by autonomous Real Time Discovery and Coding (RTDC). It can also be generalized.

4.1 Moving from Design Time to Runtime Execution

Existing tools in the market, e.g., GitHub Copilot or Amazon CodeWhisperer, operate primarily at Design Time. They assist human developers in an Integrated Development Environment, but the output remains a static artifact that humans must compile, test, deploy, and maintain. Design time AI tools lack dynamic enterprise context awareness. They cannot navigate the undocumented dependencies of sprawling microservices meshes or legacy backends where tribal knowledge has been lost to a retiring workforce, a phenomenon referred to as the Silver Tsunami.

Also, the coding, which may be agentic, is a well-defined process in the overall SDLC. No end-to end agentic process extends/links it to manage the rest of the SDLC.

Application Aware AI abandons the design time (only) assistant model [6,8-13]. Powered by the RTDC engine, it functions as a self evolving software organism embedded directly within the enterprise application stack. It acts as the developer at runtime. When a user formulates an intent, or when the enterprise environment undergoes a change, the RTDC engine autonomously synthesizes executable code and dynamic user interfaces immediately. This eliminates the anticipation paralysis where developers must predict every possible view or function a user might need. This way the discovered APIs and schemas, the rules, model of constraints, transactions memory and ACID behavior, explainability (i.e., think also of documentation), code development of application, integration, logic , agents and UI are all managed, including reliably at the transaction level, secure, coachable/refinable, and executable in sandboxes, by the same meta-agent: the whole SDLC is conversationally driven, and autonomously managed by application aware-AI [6,8-13,20]. The support and maintenance, e.g., new features, in implicitly contained in the ability to fix issues, upgrade, add new features, explaining and documenting the code, avoiding security issues, continuously improving etc. In other words, a big part of the pattern, and inherent to the idea of the strangler fig pattern [5-9].

4.2 Total Introspection and the Execution Gap

Before an AI can safely generate code that manipulates enterprise systems, it must deeply understand the environment. The RTDC engine performs Total Introspection, continuously scanning the network to identify active application programming interfaces, including documented golden path APIs and hidden shadow APIs. It connects to SQL databases, NoSQL stores, and legacy mainframes, etc. to infer underlying data schemas [6,7-13,20]. Where direct database access is restricted, the system can employs Robotic Process Automation like introspection to analyze user interface logs and screen buffers, reverse engineering the data model from the outside in. Furthermore, it ingests unstructured documentation, e.g., policy manuals and compliance guidelines, mapping

these semantic documents directly to technical assets within a dynamic Enterprise Knowledge Graph [6,7-13,20]. This total introspection bridges the execution gap, allowing the AI to institutionalize undocumented legacy logic without requiring human engineers to manually build integration wrappers.

5. The Embedded Meta Agent Architecture

Traditional software systems benefit from decades of verification tooling, e.g., compilers that reject malformed syntax, type checkers that catch data mismatches, and linkers that resolve dependencies. When traditional software fails, the failure is localized and diagnosable.

Agentic systems represent a fundamentally different category of software where the logic layer is defined primarily in natural language. Artifacts such as system prompts, tool descriptions, and handoff instructions have no compilation step. A system prompt containing a logical contradiction parses successfully, and an agent instructed to perform an action without the necessary tool will silently improvise, causing semantic drift and non-deterministic hallucinations.

5.1 The Orchestrating Meta Agent

To address this structural vulnerability, the platform utilizes an embedded Meta Agent architecture. The meta agent is an AI system explicitly designed to automate the complete lifecycle of multi agent system development, i.e., design, generation, semantic verification, deployment, runtime monitoring, and autonomous improvement. Instead of humans manually writing prompt configurations, the meta agent receives high level natural language descriptions of desired business objectives and automatically generates all deployment artifacts [6,7-13,20]. This includes engineering system prompts for boundary clarity, defining JSON schema tool bindings, establishing inter agent handoff logic, and configuring context management rules.

Prior to deployment, the meta agent performs rigorous semantic consistency analysis across all generated artifacts. It constructs an internal semantic graph to detect cross prompt contradictions, verify handoff completeness, and ensure tool instruction alignment. Crucially, it employs a Trajectory Prediction Engine to analyze the full graph of possible agent handoffs before execution, identifying infinite loops, dead ends, and context saturation points. If a failure pattern is detected, the meta agent performs self debugging, automatically revising prompts or adding loop breakers before simulating the fix. During runtime, the meta agent continuously classifies trajectory patterns, detecting behavioral drift and autonomously injecting clarifying context to prevent misalignment [6,7-13,20].

As we discussed, the meta—agent with its model of constraints can be 100% correct and / or coachable to reach that level. Any mistake, runtime, or due to reasoning problem, can be reliably corrected, i.e., guaranteed eventual execution of the actions, or roll back in the latter case [6,7-13,20,41].

5.2 The Developer Agent and Automated Integration

Operating beneath the meta agent is the developer agent, executing the runtime synthesis capabilities of the RTDC engine. The developer agent is an agentic AI agent, which is set up by meta agent when it is asked to code / design things[8]. A key failure of standard agent frameworks, e.g., Model Context Protocol (MCP) servers, is tool bloat and context window exhaustion (See [9] and references therein). Generic connectors expose hundreds of operations, causing the model to suffer from selection confusion, and excessive token consumption. Furthermore, legacy enterprise systems rarely support modern RESTful APIs natively, rendering generic MCP approaches useless without massive manual wrapper development.

For example, the developer agent eliminates dependencies on prebuilt generic connectors, which may take months to build when not yet available – a key reason for the 95% failure rate of AI projects [1,6,8-13,20], by generating precision integrations autonomously. It ingests raw API documentation, e.g., OpenAPI specifications, GraphQL schemas, SOAP definitions, or unstructured web documentation, and extracts the exact semantics required. Based on the workflow specific needs, the developer agent synthesizes composed operations that call multiple APIs sequentially, joining and filtering the results into a unified response. These self-coding agents validate the generated integration code within secure sandboxes, persisting successful integrations, and any new agent, process, application etc._) to a standard toolchain for future reuse.

When an external API changes, the developer agent detects the breakage, retrieves updated documentation, regenerates the integration code, and deploys the fix autonomously, enabling self-healing integration maintenance.

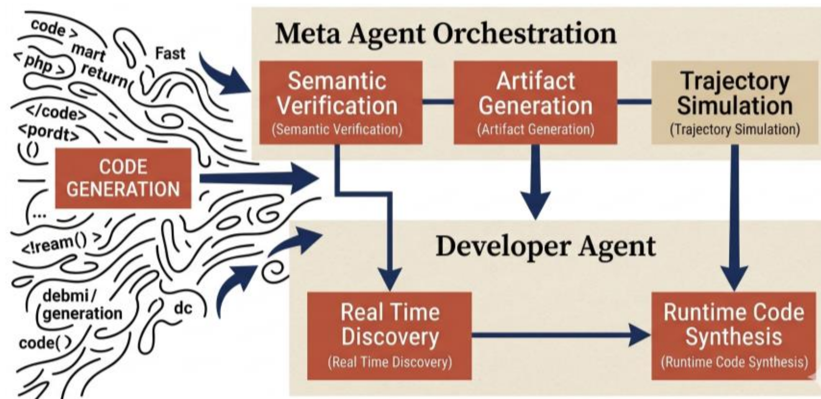


Figure 3: Proposed visual schematic detailing the RTDC (Real Time Discovery and Coding) with Meta Agent. The diagram highlights the semantic verification layer filtering natural language logic before the developer agent synthesizes integration code. From code generation to each of the other steps, the meta agent manages the full SDLC for the code it generates, allowing updates, new features, documentation/explainability, sandboxed testing etc. It all runs autonomously and in very short time compared to manual methods, vibe coding + manual best practices SDLC, and most other agentic and AI coding assistant approaches. This way solves the vibe coding challenges.

---

[8] It can be code, but also design things like CAD design as hinted in [6]. See [6,8-13,20].

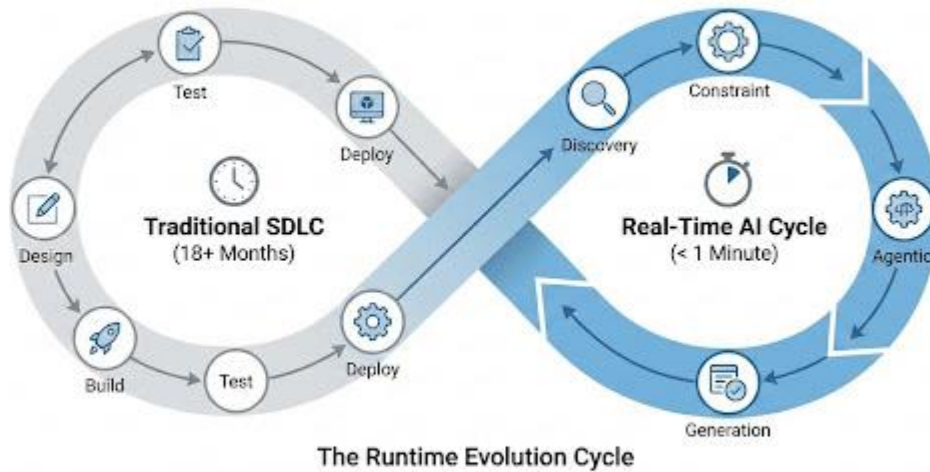


Figure 4: The run time evolution from traditional Software development life cycle to RTDC à la Zenera [6,8-13,20].

With the use of the meta-agent + developer agent, the developer can interact at a way higher level of intent description and delegate the low granularity dialogs to the agents. That is at the difference of typical vibe coding / AI assistants where the dialog ends up being way more granular.

6. Securing the Logic Layer: The Model of Constraints

Vibe coding inherently lacks boundaries, leading to outcomes that are neither reliable nor explainable. To ensure that probabilistic LLMs safely manipulate deterministic enterprise systems, the platform managed lifecycle replaces manual human code review with a deterministic Model of Constraints [6,8-13,2,20,41].

6.1 Enforcing Business Physics

The Model of Constraints represents the laws of physics of the business environment, i.e., immutable rules regarding security, compliance, and financial logic that the AI cannot violate. Operating on strict logic programming principles, this deterministic rule engine intercepts the output of the agentic layer before it reaches the execution environment [6,8-13,20].

For example, if the developer agent synthesizes a SQL query or an API call, the constraint layer simulates its execution against the rule set. If the generated code attempts to post a transaction to a closed fiscal period, or authorize a payment exceeding an established threshold, the action is strictly blocked. This mechanism provides a deterministic safety guarantee, neutralizing hallucinations and preventing the AI from improvising destructive actions.

6.2 Active Knowledge Transforms and Inherited Identity

The constraint layer is populated through Active Knowledge Transforms. Subject Matter Experts can communicate their tribal knowledge to the system in natural language, e.g., explaining why a specific insurance claim was denied due to policy date misalignments. The meta agent semantically parses these explanations and extracts deterministic rules, updating the constraint model to institutionalize the expert knowledge [6,8-13,20].

Furthermore, the constraint layer enforces architectural security through Inherited Identity. The agentic system strictly inherits the Role Based Access Control (RBAC) permissions of the requesting user or the assigned digital persona[9]. This prevents the AI from accessing sensitive data levels that the initiating user is not authorized to view, neutralizing prompt injection attacks at the architectural level rather than the prompt level. Every validation step, applied constraint, and generated action is logged in a Reasoning Graph, providing absolute glass box explainability and auditability for regulatory bodies, e.g., the SEC or HHS.

This is how any rules, policy, best practices etc. can be imposed on the coding process, as well as the whole SDLC.

7. Ensuring Execution Integrity: Transactional Object Memory and Sandboxing

Traditional agentic frameworks treat data storage as an ancillary concern, relying on ephemeral context windows, simple SQL tables, or generic vector databases for semantic recall. If an autonomous agent utilizing these primitive memory structures fails midway through executing a complex supply chain update, the system has no native mechanism to roll back the partial operations [5,6,8-13,20]. This leaves the enterprise databases in corrupted, inconsistent states, rendering traditional frameworks unusable for mission critical deployments.

Guaranteed execution and rewind adds to the integrity capabilities [6,8-13,20,41].

7.1 ACID Guarantees via Transactional Object Memory

To address the fragility of traditional agent memory, the Application Aware AI platform incorporates a Transactional Object Memory system. Built on versioned object storage technologies, e.g., LakeFS layered over MinIO, this architecture provides Git like branch and merge semantics for multi gigabyte datasets [6,8-13,20,41].

Agents operate in completely isolated branches, ensuring their read and write operations do not interfere with the production state or with other concurrently executing agents. The system enforces Atomicity, Consistency, Isolation, and Durability (ACID) guarantees. Changes made by the developer agent are committed in their entirety

[9] To minimize problems best practices should be that all RBAC are managed in one single place, and that databases also enforce these. This is not always the case though (but it could be implemented by adding this in API surrounding the misbehaving legacy systems(.

only upon successful completion and validation of the task; if a constraint violation occurs, the transaction is rolled back entirely.

Every data mutation is versioned with complete lineage and provenance, creating an immutable audit trail. This allows the enterprise environment to rapidly recover from erroneous agent actions by rolling back the memory state to any previous point in time. To manage swarms of agents operating on shared datasets simultaneously, the system employs optimistic concurrency control, and automated conflict resolution mechanisms, achieving high throughput parallel processing without locking the entire enterprise state.

7.2 Durable Workflow Orchestration

Enterprise workflows frequently involve long running processes that span days or weeks, requiring human approval gates and interactions with external systems. The platform utilizes a Durable Execution Engine, powered by technologies such as Temporal, to persist the execution state of every agent workflow to durable storage at every decision point [6,8-13,20,41].

This guarantees that complex agentic operations can survive infrastructure failures, network partitions, and Kubernetes pod migrations without losing state or progress. Workflows seamlessly resume on different compute nodes following a crash, eliminating the need for manual checkpointing logic within the synthesized code. It features automatic retries with exponential backoff and treats human escalation paths as first class citizens.

7.3 Secure Execution via Sandboxing and RASP

The flexibility of arbitrary code execution is the most powerful capability an agent can possess, but it is also the most dangerous. Unconstrained AI generated code might enter infinite resource consuming loops, overwrite critical file paths, or execute unauthorized external network calls.

To contain this risk, the generation layer synthesizes executable code and routes it into secure, ephemeral sandboxes, e.g., WebAssembly containers. These sandboxed environments enforce strict resource limits on CPU and memory utilization, alongside rigid network segmentation policies [6,8-13,20,41]. The sandbox is further instrumented with a Runtime Application Self Protection (RASP) module. The RASP hooks into system calls to provide real time behavioral blocking and anomaly detection, intercepting unsafe operations regardless of prior static validation.

Architectural Layer	Traditional AI Coding Approach	Application Aware Meta Agent Approach

Logic Verification	Manual human code review via VIBE4M.	Automated semantic consistency analysis via Meta Agent.
System Integration	Manual API wrappers; brittle generic connectors.	RTDC developer agent synthesizes precision tools.
Data Integrity	Ephemeral context windows; risk of corruption.	Transactional Object Memory with ACID guarantees.
Security Enforcement	SAST scanning post generation.	Deterministic Model of Constraints and RASP.
Workflow State	Fails upon node restart or network partition.	Durable execution via Temporal persisting state.

Table 2: *Traditional unconstrained vibe coding, i.e., typical use of it today, vs. Vibe coding constrained by the application-aware AI agentic platform*

Additionally, the system utilizes Just in Time realization to construct Generative UI components on the fly, tailoring dashboards, and forms, to the user specific intent. When an application or tool is dynamically provisioned to solve a temporary use case, it is tagged as an ephemeral asset with a defined Time to Live (TTL). Upon TTL expiration, the platform automatically archives the associated data and completely decommissions the application, minimizing the enterprise attack surface and preventing the proliferation of zombie applications.

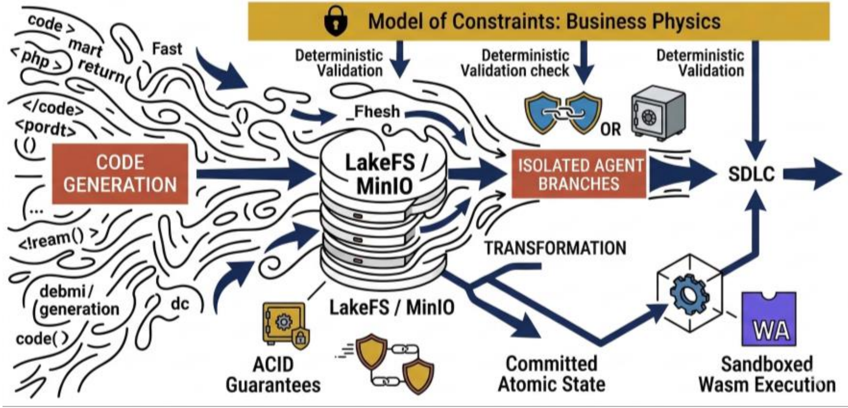


Figure 5: Proposed flowchart illustrating the flow of data through the Transactional Object Memory, highlighting isolated branches and atomic commit procedures.

8. The Disruption of Conventional Enterprise Applications

The successful implementation of Application Aware AI and the meta agent architecture fundamentally reorients the enterprise, serving as the catalyst for the disruption of conventional software suites [9,12,42]. Historically, Information Technology Service Management (ITSM), Enterprise Service Management (ESM), and Enterprise Resource Planning (ERP) (and many other enterprise applications) have relied on monolithic, centralized software platforms. The same applies for composable applications. Platforms like ServiceNow, SAP, SalesForce, IFS or BMC Helix enforce procedural governance through highly complex, user unfriendly graphical interfaces that require extensive personnel training [9,12,42].

By transitioning to an exclusively agentic architecture, enterprises can operationalize complex frameworks, e.g., the 34 management practices of ITIL 4, dynamically and autonomously [9]. Through the Agentic Strangler Fig pattern, an interconnected mesh of autonomous developer agents can envelop the legacy ERP or ITSM core, to first expands it with new features, then provide an agentic implementation of its core processes, till the application can be disregarded and only its system of record remains [9,12,42]. A user no longer logs into a graphical interface to execute a workflow; they instruct a specialized agent via natural language. The meta agent negotiates with domain agents, which query backend systems, confirm logic via the constraint model, and update databases via the transactional memory.

This renders the expensive, seat licensed graphical interfaces of legacy applications entirely obsolete, reducing the traditional monolith to a headless data repository. The shift toward Do It For Me (DIFM) autonomous execution permanently resolves the Deployment Paradox, industrializing enterprise intelligence at zero effort and quasi zero cost [9,12,42].

9. Generalization

The method discussed in this paper, was discovered and used in the context of using an application-aware AI platform [6,8-13,20], as implemented by Zenera [6,13]. However, it can be envisaged with more generic approaches where:

- Vibe coding is constrained by other means but with an AI assistant instructed to respect rules, policies, best practices and tribal knowledge, via an AI model of constraints.
- Vibe coding explains its reasoning and documents the code.
- Code is automatically reviewed by the platform in that context, e.g., think of [43] used in such a context.
- Running the code can be sandboxed, especially for testing.
- Everything is memorized in a transactional repository, so that anything can be guaranteed to be executed, or rewound.
- SDLC, and in particular updates, fixes, are done by the same agent.

If a SDLC with vibe coding is implemented to satisfy these requirements, it will provide the same benefits as discussed so far, and address the limitations of AI assistants and vibe coding. On the other hand [43] does not fit

the definition, but I may help build a rigorous manual SLDC around vibe coding but, on its own, it does not bring the advantages we can claim. Sure a platform as we propose can involve agents, set up by the meta-agent, using the capabilities of [43].

Note that the main difference is solely that RTDC may not be needed in all cases, for the rest, most models following our prescription, amount to application-aware AI patterns, of which Zenera is a particular implementation and commercial case [6,8-13,20].

Interestingly, and as hinted in [13], the approach can also extend beyond code generation to design of anything "learned" by the system, i.e., ingested. In particular it allows vibe design of parts like manufacturing parts, where ethe output are CAD designs [6,8-13].

10. Conclusions

The volatile trajectory of software engineering observed through the enterprise crises of recent years confirms a limit to unconstrained algorithmic automation. Treating probabilistic generative models as unsupervised, wholesale replacements for structured development yields undeniable systemic fragility. Unconstrained vibe coding, i.e., coding assistants and vibe coding performed in isolation in conversation with a developers, but in processes separated from any other SDLC steps, and without a rigorous model of constraints dedicated to the development and use case context, and transactional integrity mechanisms, acts as a massive accelerant for technical debt, and security, support and maintenance challenges [3-5].

Generating vast volumes of syntax via natural language intuition is computationally impressive, and may be of always improving quality as AI improves, yet without architectural foresight, it yields unmaintainable patchworks that alienate maintainers, degrade customer experiences, and exponentially expand attack surfaces [3-5]. When misguided organizations, and there are many of them out there right now, abruptly eliminate senior engineering talent to capitalize on the chimeras of these short-term generation speeds, they inevitably face a humiliating reality: they must rehire expansive support teams to untangle the resulting machine-generated sprawl[10]. The proposed solution to this crisis cannot, however, be a regression to manual SDLC policing. Forcing elite human engineers into a Great Toil Shift, i.e., constantly reviewing, debugging, and refactoring opaque black-box outputs, even if via frameworks like VIBE4M [4], fundamentally defeats the premise of artificial intelligence acceleration. Autonomous solutions mut prevail.

The resolution to the Deployment Paradox demands a permanent architectural evolution from design-time assistants to runtime, autonomous execution. Constrained Vibe Coding, governed by an Application-Aware agentic platform, provides the necessary operational bridge. By subordinating the probabilistic nature of generation to a strict, deterministic Model of Constraints, the orchestrating meta-agent effectively enforces the immutable laws of business physics of an enterprise. When paired with the total introspection capabilities of Real-Time Discovery and Coding (RTDC), the requirement for brittle, manually stitched API wrappers vanishes. More critically, integrating Transactional Object Memory with absolute ACID guarantees ensures that agentic actions are no longer ephemeral risks; they are fully isolated, durably orchestrated, and instantly rewound upon failure. That pairing with RTDC, and the use of RTDC is not mandatory to achieve the constrained vibe coding and SDLC benefits.

---

[10] Our alternately, they can re-hire senior developers! Don't laugh, it happens often these days. FOMO is bigger than common sense it seems.

Platforms demonstrating these capabilities, such as Zenera [6,13], prove that the entirety of the software lifecycle, from initial ideation and semantic verification to sandboxed RASP execution and continuous self-healing, can be natively AI managed by the AI platform itself. The economic implications of this architectural shift are significant. The ability to autonomously execute complex frameworks via conversational interfaces renders the dense, training-heavy graphical interfaces of legacy platforms obsolete, but now it is not just for coding, for throughout the whole SDLC.

The shift toward Do-It-For-Me (DIFM) autonomous execution is not a theoretical roadmap; it is an active market disruption. Enterprises that persist in treating artificial intelligence merely as a high-speed typing substitute will drown in maintenance overhead. True operational dominance belongs strictly to those who deploy platform-managed, constrained agentic meshes to industrialize software creation at zero manual effort. It will also fundamentally disrupt the enterprise application market as discussed in [9].

Appendix A: Many Enterprises Have Lost Their Common Sense. Don't be one of them.

The volatile period spanning 2024 to 2026 serves as the definitive empirical study on the profound limitations of algorithmic automation in enterprise environments. The aggregated market data clearly and unequivocally demonstrates that treating probabilistic large language models as wholesale, unsupervised replacements for human intelligence, whether in the highly deterministic field of software engineering or the highly empathetic field of customer success management, results in systemic infrastructural fragility, the exponential accumulation of unmaintainable technical debt, massive legal liabilities, and the total alienation of the consumer base[11].

The cyclical trajectory of the technology industry during this time, e.g., aggressively firing skilled developers, generating vast quantities of AI slop via unsupervised vibe coding, degrading the customer experience through robotic chatbots, and ultimately being forced to rehire expensive cleanup specialists and thousands of human support staff, illustrates a fundamental truth of the modern digital economy, and the fact that many enterprises are totally on what to do with AI.

Our lesson so far is: Artificial intelligence functions optimally as a cognitive exoskeleton designed to augment human capability, and to constrain its actions, not as a synthetic substitute for human, when all the guardrail and extra functions are neglected, in this case also an autonomous SDLC.

For corporate leadership, boards of directors, and enterprise architects attempting to navigate the integration of generative artificial intelligence into existing workflows without destroying their operational stability, it is highly recommended to anchor all future strategic planning upon two foundational reference points observed during this period. The following two case studies perfectly encapsulate the requested dynamics of firing technical staff only to require massive support rehiring.

---

[11] Consider, for example, the case of Salesforce [43], with the backlash of customers against too much agentic support. One can expect similar pushback from consumers just as we saw for with IVR, think of the phone jail, outsources call center CSRs, chatbots vs. customer support etc. Appendix A2, gives another concrete example.

A.1 The Microsoft Engineering Reallocation (2025)

The calculated elimination of over 15,300 Microsoft employees, heavily concentrated in core software engineering and foundational open-source development (specifically targeting Python and TypeScript architects), serves as the prime indicator of the corporate transition from human coding to AI infrastructure [48]. This case demonstrates the prevailing corporate belief that base-level code generation is fundamentally commoditized by models capable of writing 30% to 95% of standard code [45]. However, it equally highlights the absolute necessity of retaining elite architectural talent to prevent the catastrophic accumulation of technical debt. The fact that this decimation directly resulted into a vibe coding cleanup scramble to rehire proves that human oversight is required to validate machine logic [46].

A.2 The Klarna Customer Support Reversal (2024–2026)

Klarna's highly publicized, data-driven decision to eliminate 700 human customer service agents in favor of an artificial intelligence chatbot, followed by a total collapse in long-term customer satisfaction and the subsequent, humiliating rehiring of 700 human employees, is a cautionary tale of the artificial intelligence era [47]. This case provides incontrovertible proof that, while artificial intelligence automation drives theoretical great short-term operational cost reductions on a balance sheet, the degradation of the customer experience ultimately destroys long-term brand equity, requiring massive and expensive human remediation [47].[4]

A.3 Lessons learned

The organizations that will achieve market dominance in the latter half of the decade will be those that actively reject the false economy of total automation, unless they can actually and credible achieve it, for example with application aware AI total automation and management of a constrained vibe coding and SDLC. True operational resilience will be achieved by Frontier Firms that utilize artificial intelligence to augment the velocity of their elite human capital, deploy rigorous rollback and rewind mechanisms to contain agentic errors, and fiercely protect the human element in all high-stakes client interactions and architectural decisions. Enterprises aiming at this now should seriously consider our proposal, and if looking for such a platform, product and solution, they should explore Zenera [6,13], or similar players.

Appendix B: Overview of Modern Software Development Lifecycle

A well-organized modern software development lifecycle (SDLC) can be described as an iterative, DevOps-enabled flow that spans from ideation to operations and learning [49,54].

Contemporary SDLC practice blends classic phases (requirements, design, implementation, testing, deployment, maintenance) with a continuous DevOps loop (plan, code, build, test, release, deploy, operate, monitor) [52-54]. Rather than a one-way waterfall, most teams execute these phases in short iterations (sprints or continuous delivery cycles) to reduce risk and speed up feedback [50,55].

This sections gives an idea of the phases and tools used in modern SDLC. Of course, these tools are not directly involved in the lifecycle managed by the application aware AI platform proposed in the main body.

B.1 Product Discovery and Planning

Teams first identify stakeholder needs, define scope, assess feasibility, and prioritize a product backlog [49,50,53,56].

Output typically includes a product vision, roadmap, prioritized user stories, and high-level release plans that guide subsequent iterations [52,53].

Representative tools on the market include (discovery/planning):

- Atlassian Jira (work management, backlog, Scrum/Kanban boards) [57-59]
- Azure Boards (Agile planning within Azure DevOps Services)[60]
- Trello (lightweight Kanban boards for smaller teams)[57,58]
- Productboard or Aha! (product discovery and roadmapping) [57]

B.2. Requirements Analysis and Specification

During this phase, teams refine user and system requirements, capture functional and non-functional needs, and document constraints [49,51,61,63].

Formal artifacts include user stories, use cases, acceptance criteria, and sometimes a software requirements specification that is iteratively updated [53,61].

Representative tools (requirements):

- Atlassian Confluence (structured requirements documentation) [57,59]
- Notion (collaborative docs/databases for requirements) [57]
- IBM Engineering Requirements Management DOORS Next (enterprise requirements management) [52,53]
- GitHub Issues/Discussions (lightweight requirements and acceptance criteria close to code) [58]

B.3 Architecture and Detailed Design

Architects and senior engineers derive logical and physical architectures, interfaces, data models, and UI/UX designs from requirements [49,51,53,61].
Typical outputs are architecture diagrams, design specifications, API contracts, and UX mockups that constrain and inform implementation [52,53].

Representative tools (design):

- draw.io / diagrams.net (general architecture and UML diagrams) [57]
- Microsoft Visio (proprietary diagramming widely used in enterprises) [52]
- Lucidchart (web-based collaborative modeling) [57]
- Figma (collaborative UI/UX design and prototyping) [57,59]
- PlantUML (text-based modeling and architecture diagrams) [61]

B.4 Implementation and Version Control

Developers implement features according to design specifications, following coding standards and performing unit testing as they go [49,51,53,61].

Modern teams almost universally use distributed version control and feature branching, integrated with code review and continuous integration [54,58,62].

Representative tools (coding/version control):

- Git (de facto distributed version control standard) [58,62]
- GitHub (cloud Git hosting, code review, Actions integration) [58,59,62]
- GitLab (integrated Git hosting with built-in CI/CD) [58,60,62]
- Bitbucket (Atlassian Git hosting integrated with Jira) [58,59]
- IDEs such as Visual Studio Code, IntelliJ IDEA, and Visual Studio (code editing and refactoring) [62]

B.5 Continuous Integration and Automated Testing

Continuous integration (CI) systems automatically build and test the software whenever changes are committed to shared branches [6,10,11,14].

This phase emphasizes automated unit, integration, and regression tests, as well as static analysis and artifact packaging, to detect defects early [51,54,62].

Representative tools (CI and automated testing):

- Jenkins (widely adopted open source CI server) [57,59,62]
- GitHub Actions (Git-native CI/CD workflows) [58,59,62]
- GitLab CI/CD (pipeline engine integrated with GitLab repositories) [12,14]
- CircleCI (cloud-hosted CI/CD) [57,62]
- Selenium (browser automation for UI testing) [57]
- JUnit / NUnit / pytest (unit testing frameworks across languages) [53]
- SonarQube (code quality and static analysis) [57]

B.6. Deployment, Delivery, and Release Automation

Continuous delivery and deployment pipelines automate packaging, environment provisioning, configuration, and promotion of builds into staging and production [54,58,59,62].

Modern practice relies heavily on containers, infrastructure as code, and release strategies such as blue-green, canary, or rolling deployments [54,62].

Representative tools (delivery/deployment):

- Docker (containerization for consistent runtime environments) [59,62]
- Kubernetes (container orchestration and scaling) [59,62]
- Argo CD (GitOps-based continuous delivery for Kubernetes) [62]
- Helm (Kubernetes packaging and release management) [62]
- Terraform (infrastructure as code for multi-cloud provisioning) [62]
- Ansible, Chef, Puppet (configuration management and deployment automation) [6214]

B.7 Operations, Monitoring, and Incident Management

Once deployed, applications are operated under SLA/SLO constraints, with observability tooling for metrics, logs, traces, and user experience [53,54,62].

Incident management, on-call rotations, and runbooks ensure that failures are detected, triaged, and resolved quickly in production [54,62].

Representative tools (monitoring/operations):

- Prometheus (metrics collection and alerting) [58,62]
- Grafana (visualization and dashboards) [62]
- Datadog, New Relic (commercial APM and observability platforms) [57,58,62]
- ELK Stack (Elasticsearch, Logstash, Kibana for logging) [62]
- PagerDuty (incident alerting and on-call management)[62]
- IFS assyst, including ITOM [64,66,67]

Note that we advocate strict separation of concerns between dev tool chains and ITSM type of tools [64,65,67], and so for examples Jira or Azure DevOps should rather be used by developers, while being integrated with ITSM for interactions with businesspeople.

B.8 Feedback, Analytics, and Continuous Improvement

Product and engineering teams collect user feedback, usage analytics, and operational data to refine requirements and backlog priorities [55,62].

Retrospectives, A/B experiments, and data-driven decision-making close the loop, feeding new work into discovery and planning and making the SDLC a continuous cycle [55,62].

Representative tools (feedback/analytics):

- Google Analytics or similar product analytics (user behavior tracking) [55]
- Amplitude or Mixpanel (product analytics for experiment tracking) [55]
- In-app feedback and survey platforms such as Hotjar or Qualtrics (user feedback capture) [55]
- Jira/Confluence (aggregating feedback into new backlog items) [57,59].

B. 9 Representative SDLC Phases and Tools

SDLC activity	Example tools (non-exhaustive)
Discover & plan	Jira, Azure Boards, Trello, Productboard, Aha! [57,58]
Requirements specification	Confluence, Notion, DOORS Next, GitHub Issues [52,57]
Architecture & design	draw.io, Visio, Lucidchart, Figma, PlantUML [57,61]
Code & version control	Git, GitHub, GitLab, Bitbucket, VS Code [58,62]
CI & automated testing	Jenkins, GitHub Actions, GitLab CI, CircleCI, Selenium, JUnit [57,59,62]
Deployment & IaC	Docker, Kubernetes, Argo CD, Helm, Terraform, Ansible [59,62]
Monitoring & operations	Prometheus, Grafana, Datadog, New Relic, ELK, PagerDuty [57,58,62,64,66,67]
Feedback & analytics	Google Analytics, Amplitude, Mixpanel, Hotjar, Jira [55,57,59]

Table B1: Overview of Representative SDLC Phases and Tools

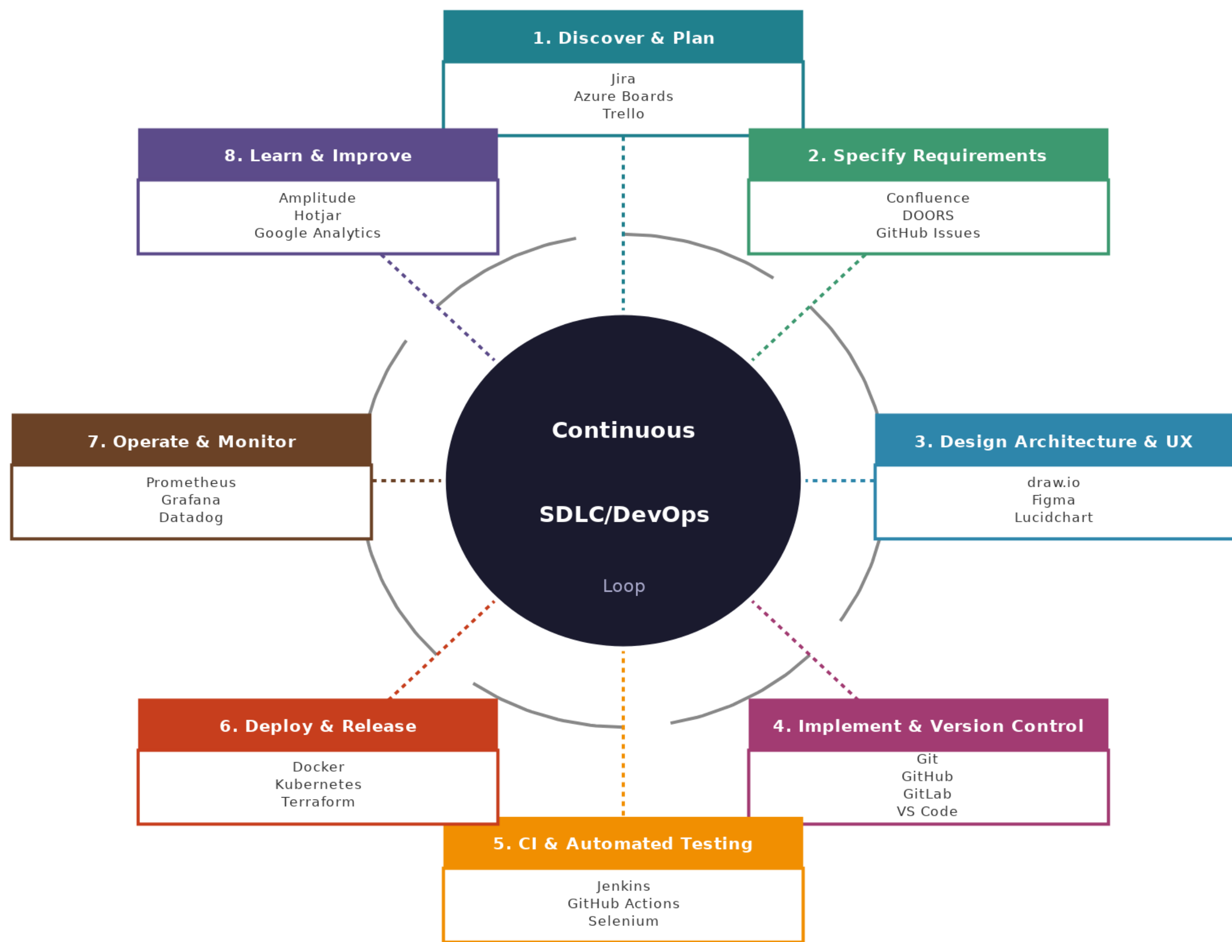


Figure B1: It shows representatives SDLV tools and associated phase, for a typical 8-phases continuous DevOps lifecycle. The figure is organized as a radial hub-and-spoke layout with eight color-coded phase boxes arranged clockwise around a central "Continuous SDLC/DevOps Loop" hub [15-17]. Each colored header names the phase (e.g., 1. Discover & Plan, 5. CI & Automated Testing) [18,19]. Each white body lists the most representative tools for that phase [20,21]. Dotted spoke lines connect the hub to each phase, emphasizing that all phases feed back to the center [16]. Clockwise arc arrows on the inner ring indicate the iterative, forward-moving flow of the lifecycle 17].

Note that while as argued in [x61,x62], DevOps engineers want the freedom to select their own DevOps toolchain, the problem does not occur in the case of the constrained Vibe coding + SDLC managed by application-aware AI, because everything is now done by the platform, though conversational interactions. Developers can still have their preferred tools, when they want to manually inspect, review, correct or code artefact produced by the plication aware process.

——

References

[1]: Aditya Challapally, Chris Pease, Ramesh Raskar, Pradyumna Chari, "The GenAI Divide – State of AI in Business 2025", MIT NANDA, July 2025.

[2]: Wikipedia, "Vibe coding", https://en.wikipedia.org/wiki/Vibe_coding. Retrieved on March 2, 2026.

[3]: Stephane H. Maes, (2025), "The Gotchas of AI Coding and Vibe Coding. It's All About Support And Maintenance", https://doi.org/10.5281/zenodo.15343349, https://shmaes.wordpress.com/2025/04/28/the-gotchas-of-ai-coding-and-vibe-coding-its-all-about-support-and-maintenance/, April 28, 2025, (https://osf.io/kjz9t/download/).

[4]: Stephane H. Maes, (2025), "Ensuring the Maintainability and Supportability of "Vibe-Coded" Software Systems: A Framework for Bridging Intuition and Engineering Rigor", https://doi.org/10.5281/zenodo.15354102, https://shmaes.wordpress.com/2025/05/06/ensuring-the-maintainability-and-supportability-of-vibe-coded-software-systems-a-framework-for-bridging-intuition-and-engineering-rigor/, May 6, 2025, (https://osf.io/2nu8r/download/).

[5]: Stephane H Maes, Ramu Sunkara, (2026), "Vibe Coding = More Code. Is it good, though?", Zenera AI, March 5, 2026 (+ LinkedIn post).

[6]: Zenera, (2026), "The Enterprise AI Agent Factory. Your enterprise is unique. Your AI agents should be too. The Enterprise AI Problem, Solved", https://zenera.ai. Retrieved on March 5, 2026.

[7]: Gartner, "Gartner Predicts Over 40% of Agentic AI Projects Will Be Canceled by End of 2027", June 25, 2025.

[8]: Stephane H. Maes, et al. (2026), US Patent Application, "AI System and Method of Meta-Agent and Application-Aware AI, Including Real-Time Discovery and Coding, for Deterministic Constraint Modeling, and Runtime Evolution of Software Applications", 2026.

[9]: Stephane H. Maes, "Agentic Smart ITIL, And The Disruption Of The Market Of Conventional Enterprise Applications", March 1, 2026.

[10]: Stephane H Maes, (2026), "The Era of Application-Aware AI", Zenera, February 2026.

[11]: Stephane H. Maes, (2026), "Achieving Zero-Effort, Quasi-Zero Cost Integration with Zenera RTDC, and Application-Aware AI", Zenera, February 2026.

[12]: Stephane H. Maes, (2026), "Agentic AI, The Obsolescence of The Enterprise Application & Zenera, The Catalyst", February 2026.

[13]: Zenera, (2026), "Zenera: From Conversation to Production in Minutes", YouTube, https://zenera.ai/introvideo. Retrieved on March 1, 2026.

[14]: Andrej Karpathy, (2025), https://x.com/karpathy/status/1886192184808149383, February 2, 2025.

[15]: Cade Metz, (2026), "A.I. Isn't Coming for Every White-Collar Job. At Least Not Yet. Tech workers are increasingly worried that the artificial intelligence they are building will replace them. But some are optimistic that it is just one more tool to work with.", New York Times, https://www.nytimes.com/2026/02/20/technology/ai-coding-software-jobs.html, February 20, 2026.

[16]: Eivind Kjosbakken, (2026), "How to Create Production-Ready Code with Claude Code. Learn how to write robust code with coding agents.", https://towardsdatascience.com/how-to-create-production-ready-code-with-claude-code/, March 6, 2026.

[17]: r/ClaudeCode, (2026), “Why AI still can't replace developers in 2026”, Reddit, https://www.reddit.com/r/ClaudeCode/comments/1r59hz2/why_ai_still_cant_replace_developers_in_2026/, February 14, 2026.

[18]: Rya Jetha , (2026), “AI writes the code now. What’s left for software engineers? Hiring is down, AI tools are in, and engineers fear they’re sliding toward a “permanent underclass.”, The San Francisco Standard, https://sfstandard.com/2026/02/19/ai-writes-code-now-s-left-software-engineers/, February 19, 2026.

[19]: Aruna Ranganathan, Xingqi Maggie Ye, (2026), “AI Doesn’t Reduce Work—It Intensifies It”, Harvard Business Review, https://hbr.org/2026/02/ai-doesnt-reduce-work-it-intensifies-it, February 9, 2026.

[20]: Zenera, “Enterprise Analytics Reimagined, Build live reports and dashboards in minutes – no pipelines, no coding, https://zenera.ai/zeneraanalytics. Retrieved on March 5, 2026.

[21]: Anton Morgunov, (2026), “Vibe coding killed Cursor”, Ischemist, https://ischemist.com/writings/long-form/how-vibe-coding-killed-cursor, January 1, 2026.

[22]: Leerob, (2026), “About “Vibe coding killed Cursor””, Hacker News, https://news.ycombinator.com/item?id=46465513, January 2026.

[23]: Stephane H. Maes, (2026), “Evaluating the Efficacy of Artificial Intelligence in Software Engineering: A Post-February 2026 Analysis”, in preparation. Will be at https://shmaes.wordpress.com/. It is now published as: Stephane H. Maes, (2026), "Evaluating the Efficacy of Artificial Intelligence in Software Engineering: A Post-February 2026 Analysis", https://shmaes.wordpress.com/2026/03/16/evaluating-the-efficacy-of-artificial-intelligence-in-software-engineering-a-post-february-2026-analysis/, March 13, 2026.

[24]: Greek Ai, (2026), “Stop Everything — DeepSeek V4 Might Be the Smartest Coding AI of 2026”, GoPenAI, Medium, https://blog.gopenai.com/stop-everything-deepseek-v4-might-be-the-smartest-coding-ai-of-2026-c8abda20b7b8, February 10, 2026.

[25]: Valerie Chen, Jasmyn He, Behnjamin Williams, Jason Valentino, Ameet Talwalkar, (2026), “Beyond the Commit: Developer Perspectives on Productivity with AI Coding Assistants”, arXiv:2602.03593v1.

[26]: Prasenjit Sarkar, (2026). “The great toil shift: How AI is redefining technical debt”, Sonar, https://www.sonarsource.com/blog/how-ai-is-redefining-technical-debt/, February 12, 2026.

[27]: Chi Zhang, Zehan Li, Ziqian Zhong, Haibing Ma, Dan Xiao, Chen Lin, Ming Dong, (2026), “From Horizontal Layering to Vertical Integration: A Comparative Study of the AI-Driven Software Development Paradigm”, arXiv:2601.22667v1.

[28]: Jose Enrico, (2025), “Human Just Barely Beats OpenAI's AI in Historic Coding Contest. The next time you use ChatGPT, always be proud that AI can't still replace you.”, Tech Times, https://www.techtimes.com/articles/311429/20250721/human-just-barely-beats-openais-ai-historic-coding-contest.htm, July 21, 2025.

[29]: AIM Network, (2025), “Big Models Fail - Claude Opus 4.6, GPT-5.2 Score Only ~30% on New Coding Text, You Tube, https://www.youtube.com/watch?v=_YRMsYxptkE, March 5, 2026.

[30]: Sonar, “LLM Leaderboard for Code Complexity”, https://www.sonarsource.com/the-coding-personalities-of-leading-llms/leaderboard/complexity, Retrieved on March 5, 2026.

[31]: Mark Levison, (2026), “AI-Generated Code Quality and the Challenges we all face”, Agile Pain Relief, https://agilepainrelief.com/blog/ai-generated-code-quality-problems/, February 2026.

[32]: Margaret-Anne Storey, (2026), “How Generative and Agentic AI Shift Concern from Technical Debt to Cognitive Debt”, https://margaretstorey.com/blog/2026/02/09/cognitive-debt/, February 9, 2026.

[33]: Michael Tridi, (2026), “The Impact of AI Coding in 2026: Developer Productivity Revolution with 90% AI-Generated Code”, Trigi Digital, https://trigidigital.com/blog/ai-coding-impact-2026, ARC Prize 2025 Results and Analysis, January 28, 2026.

[34]: Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, Dawn Song, “Evaluating AI Agents' Real-World Cybersecurity Capabilities at Scale”, CyberGym, https://www.cybergym.io/. Retrieved on March 2, 2026.

[35]: Anil K Shukla, (2025), “From Code Generation to Code Cognition: The Quiet Evolution of AI Reasoning”, Medium, https://medium.com/@shuklaks/from-code-generation-to-code-cognition-the-quiet-evolution-of-ai-reasoning-75c786a65c50, September 30, 2025.

[36]: Xin Guan, (2024), “Navigating the LLM Benchmark Boom: A Comprehensive Catalogue”, Holistic AI, https://www.holisticai.com/blog/navigating-llm-benchmark, July 1, 2024.

[37]: Cole Granger, Dipin Khati, Daniel Rodriguez-Cardenas, Denys Poshyvanyk, (2026), “Tricky²: Towards a Benchmark for Evaluating Human and LLM Error Interactions”, arXiv:2601.18949v1.

[38]: Ken Judy, Ben Linders, (2025), “A Plan-Do-Check-Act Framework for AI Code Generation”, InfoQ, https://www.infoq.com/articles/PDCA-AI-code-generation/, October 20, 2025.

[39]: Stephane H. Maes, Karan Singh Chhina, Guillaume Dubuc, (2021), “Natural language translation-based orchestration workflow generation”, US Patent 11120217.

[40]: Stephane H. Maes, (2024), “Fixing Reference Hallucinations of LLMs”, https://doi.org/10.5281/zenodo.14543939, https://shmaes.wordpress.com/2024/11/29/fixing-reference-hallucinations-of-llms/, November 29, 2024. (osf.io/u38w4/, viXra:2412.0149v1).

[41]: Ramu Sunkara, Stephane H. Maes, (2026), “Agentic AI Platform with Agent Rewind, and Reliable execution”, Zenera Newsletter, March 2026.

[42]: Stephane H. Maes, Ramu Sunkara, (2026), “The death of SAAS and the enterprise apps: an application of the strangler fig pattern”, LinkedIn, March 4, 2026.

[43]: Frederic Lardinois, (2026), “Anthropic launches a multi-agent code review tool for Claude Code. As AI coding tools drive a surge in pull requests, Anthropic's Code Review dispatches parallel code review agents to catch bugs.”, https://thenewstack.io/anthropic-launches-a-multi-agent-code-review-tool-for-claude-code/, March 9, 2026.

[44]: Vud Zjindak, (2025), “Salesforce AI faces backlash from customers”, TheStreet, https://www.thestreet.com/technology/salesforce-ai-faces-backlash-from-customers, October 4, 2025.

[45]: Mike Kaput, (2025), “Microsoft Just Laid Off 6,000 Workers. And AI Might Be to Blame”, Marketing AI Institute, https://www.marketingaiinstitute.com/blog/microsoft-layoffs-ai, May 20, 2025.

[46]: HyperAI, (2025), “AI-Generated Code Backfires: Companies Hire Specialists to Fix “Vibe Coding” Messes”, Trending Stories, HyperAI, https://hyper.ai/en/stories/73e5432b0d69cb3c60b1ee313c4ab9f8, October 2025.

[47]: Michael Szczepanik, (2025), “I have heard about dress code, but what is vibe code?”, Medium, https://medium.com/mobilepeople/i-have-heard-about-dress-code-but-what-is-vibe-code-f17a4d03300f, December 29, 2025.

[48]: Jason Lemkin, “Microsoft: We Need A Lot Less Employees. And a Lot More AI Infrastructure.”, https://www.saastr.com/microsoft-we-need-less-employees-but-more-ai-infrastructure/. Retrieved on March 2, 2026.

[49]: Harness Team, (2023), “The Seven Phases of the Software Development Life Cycle”, Harness, https://www.harness.io/blog/software-development-life-cycle-phases, December 13, 2023.

[50]: Atlassian, “The complete guide to SDLC (Software development life cycle)”, https://www.atlassian.com/agile/software-development/sdlc. Retrieved on March 2, 2026.

[51]: Blessing Onyegbula, (2025), “The Software Development Lifecycle: The Most Common SDLC Models”, Splunk, https://www.splunk.com/en_us/blog/learn/software-development-lifecycle-sdlc.html, May 15, 2025.

[52]: IBM, “What is the software development life cycle (SDLC)?”, https://www.ibm.com/think/topics/sdlc. Retrieved on March 2, 2026.

[53]: Science Direct, (2016), “Software Development Lifecycle”, https://www.sciencedirect.com/topics/computer-science/software-development-lifecycle. Retrieved on March 2, 2026.

[54]: Chrystal R. China, “DevOps lifecycle, defined”, https://www.ibm.com/think/topics/devops-lifecycle. Retrieved on March 2, 2026.

[55]: Maria DiCesare, (2025), “The 5 Stages of the Agile Software Development Lifecycle”, https://www.mendix.com/blog/agile-software-development-lifecycle-stages/, August 16, 2025.

[56]: Charter Global, (2025), “What are the 5 phases in the Software Development Life Cycle (SDLC)?”, https://www.charterglobal.com/what-are-the-5-phases-in-the-software-development-life-cycle-sdlc/ , April 8, 2025.

[57]: BrowserStack, (2025), “Top 15 SDLC tools. Explore the ultimate SDLC tools to optimize your development process, boost team collaboration, and deliver high-quality software faster.”, https://www.browserstack.com/guide/sdlc-tools, January 2, 2025.

[58]: Rajni Rethesh, (2025), “How to Centralize Your SDLC Tools Without Breaking a Sweat?”, Middleware, https://middlewarehq.com/blog/how-to-centralize-your-sdlc-tools-without-breaking-a-sweat, April 10, 2025.

[59]: Febna V M, (2025), “Top 10 SDLC tools”, Beagle Security, https://beaglesecurity.com/blog/article/top-10-sdlc-tools.html, December 9, 2025.

[60]: Atlassian, “DevOps Tools. Choose tools for each phase of the DevOps lifecycle”, https://www.atlassian.com/devops/devops-tools. Retrieved on March 2, 2026.

[61]: GeekforGeek, “Software Development Life Cycle (SDLC)”, https://www.geeksforgeeks.org/software-engineering/software-development-life-cycle-sdlc/. Retrieved on March 2, 2026.

[62]: William Imoh, “What Are the 7 Key Phases of the DevOps Lifecycle?”, https://roadmap.sh/devops/lifecycle. Retrieved on March 2, 2026.

[63]: Somya Gupta, Janvi Banga, Sourabh Dabas, Dr. Manjot Kaur Bhatia, (2022), "A Comprehensive Study of Software Development Life Cycle Models" https://www.ijraset.com/research-paper/software-development-life-cycle-models, December 4, 2022.

[64]: Stephane H. Maes, (2022), "WHAT IS ITOM (IT OPERATIONS MANAGEMENT)? | ITOM EXPLAINED | ITOM VS ITSM VS ITAM", IFS Blog, September 26, 2022.

[65]: Stephane H. Maes, (2022), "ITSM and ESM in the Bigger World. A Modern Approach of ITIL for the Enterprise", Pink Elephant 2022, June 2022, Las Vegas

[66]: IFS, "ESM Software for Streamlined Enterprise Operations", https://www.ifs.com/en/products/esm, Retrieved on March 2, 2026.

[67]: Stephane H. Maes, (2022), "ITSM and ESM in the Bigger World. Separation of concerns: A Modern Approach of ITIL for the Enterprise", viXra:2207.0049v1, https://shmaes.wordpress.com/2022/06/26/itsm-and-esm-in-the-bigger-world-a-modern-approach-of-itil-for-the-enterprise/. Formal paper behind Pink Elephant 2022 Presentation. June 26, 2022.

[68]: Steef-Jan Wiggers, (2026), "Cloudflare Releases Experimental Next.js Alternative Built With AI Assistance", InfoQ, https://www.infoq.com/news/2026/03/cloudflare-vinext-experimental/, March 10, 2026.