Agentic Software Engineering: Foundational Pillars and a Research Roadmap

AHMED E. HASSAN, Queen's University, Canada

HAO LI*, Queen's University, Canada

DAYI LIN, Huawei Canada, Canada

BRAM ADAMS, Queen's University, Canada

TSE-HSUN CHEN, Concordia University, Canada

YUTARO KASHIWA, Nara Institute of Science and Technology, Japan

DONG QIU, Huawei Canada, Canada

Agentic Software Engineering (SE 3.0) represents a new era where intelligent agents are tasked not with simple code generation, but with achieving complex, goal-oriented SE objectives. To harness these new capabilities while ensuring trustworthiness, we must recognize a fundamental duality within the SE field in the Agentic SE era, comprising two symbiotic modalities: SE for Humans and SE for Agents. This duality demands a radical reimagining of the foundational pillars of SE (actors, processes, tools, and artifacts) which manifest differently across each modality. This new vision of SE requires two distinct, purpose-built workbenches (aka tools) for these two collaborative modalities: the Agent Command Environment (ACE), a command center where humans orchestrate, mentor, and oversee agent teams while managing an inbox of agent-generated events like Merge-Readiness Packs (MRPs) and Consultation Request Packs (CRPs); and the Agent Execution Environment (AEE), a digital workbench where agents not only execute tasks but can proactively invoke human expertise when facing complex trade-offs or ambiguity. This bi-directional partnership, which supports agent-initiated human callbacks and handovers, gives rise to new, structured engineering activities (i.e., processes) that redefine human-AI collaboration, elevating the practice from agentic coding to true agentic software engineering. This paper presents the Structured Agentic Software Engineering (SASE) vision, outlining several of the foundational pillars for the future of SE. The paper culminates in a research roadmap that identifies a few key challenges and opportunities while briefly discussing the resulting impact of this future on SE education. Our goal is not to offer a definitive solution, but to provide a conceptual scaffold with structured vocabulary to catalyze a community-wide dialogue, pushing the SE community to think beyond its classic, human-centric tenets toward a disciplined, scalable, and trustworthy agentic future.

CCS Concepts: • **Software and its engineering** → **Software development techniques**; **Collaboration in software development**; **Software creation and management**; • **Computing methodologies** → **Artificial intelligence**.

Additional Key Words and Phrases: Agentic Software Engineering, AI Agent, Agentic AI, Coding Agent

*Corresponding author.

Authors' Contact Information: Ahmed E. Hassan, ahmed@cs.queensu.ca, Queen's University, Kingston, ON, Canada; Hao Li, hao.li@queensu.ca, Queen's University, Kingston, ON, Canada; Dayi Lin, dayi.lin@huawei.com, Huawei Canada, Kingston, ON, Canada; Bram Adams, bram.adams@queensu.ca, Queen's University, Kingston, ON, Canada; Tse-Hsun Chen, peterc@encs.concordia.ca, Concordia University, Montreal, Canada; Yutaro Kashiwa, yutaro.kashiwa@is.naist.jp, Nara Institute of Science and Technology, Ikoma, Japan; Dong Qiu, dong.qiu@huawei.com, Huawei Canada, Kingston, ON, Canada.

1 Introduction

The emergence of powerful autonomous agents (aka AI teammates [14]) capable of writing, testing, and submitting code has moved the Software Engineering (SE) field beyond the era of AI-Augmented development (SE 2.0) into a new, more potent era of Agentic Software Engineering (SE 3.0) [14]. This transition, fueled by impressive demonstrations of frontier LLMs generating entire micro-applications from short one-off prompts, holds the promise of unprecedented productivity. However, building, and more importantly, shipping complex and trustworthy software that satisfies the ever-evolving needs of many stakeholders is an endeavor of a different magnitude, requiring structured, iterative and trustworthy SE practices. There is no doubt that the SE field is facing one of the most significant shifts in its history, which exposes a fundamental tension between the velocity of automation and the required rigor to build trustworthy software.

While autonomous coding agents (e.g., Google's Jules, OpenAI's Codex, Anthropic's Claude Code, and Congition's Devin) are already responsible for hundreds of thousands of merged pull requests (PR) [19], their hyper-productivity is revealing a significant "speed vs. trust" gap. Recent, deeper examinations of agent-generated code and agent-driven PRs (and our own hands-on experiences with leading autonomous agents) reveal that a large percentage of agent efforts fail to meet the quality bar of being truly "merge-ready," often containing subtle regressions, superficial fixes, or a general lack of engineering hygiene (e.g., [38]). This creates a critical bottleneck, as every failed check requires a demanding human-in-the-loop review. When combined with the massive volume of agent-produced code, such essential verification processes overwhelm human developers. Nevertheless, a new class of practitioners is emerging from this chaotic frontier: developers achieving 100x or even 1,000x productivity. By mastering the nascent best practices of this new agentic era, these super developers are demonstrating what is possible. Their success serves as a powerful proof-of-concept, yet it also highlights a familiar challenge.

The core purpose of the SE field has always been to ensure solutions are trustworthy and delivered economically, and much of the SE field exists because we cannot assume that every team is composed of "super developers." The industry has long acknowledged the phenomenon of the "10x developers," a small fraction of developers whose impact far exceeds the median [22]. A significant portion of SE, from structured processes like Agile to sophisticated tools like IDEs, is designed to give non-super developers the scaffolding and opportunity to perform at a 10x level. Agentic SE radically reshapes this landscape, moving the conversation beyond 10x to the realm of 100x and even 1,000x productivity while also redefining the characteristics of such top-tier developers, away from raw coding prowess and toward effective collaboration with fleets of agents (aka AI Teammates).

As the industry forges ahead, a Cambrian explosion of ad-hoc practitioner techniques is emerging. However, these grassroots innovations highlight a vacuum of robust, validated approaches. Current methods, relying heavily on informal, conversational prompting, are inadequate for developing trustworthy large-scale long-lived software. This informality fails to establish the robust processes that are needed for reproducibility, the auditable artifacts required for ensuring trust, or a durable mechanism for human-agent collaboration. It keeps the paradigm locked in the realm of 1-to-1 "agentic coding," rather than unlocking the potential of N-to-N "agentic software engineering" where teams of humans and agents collaborate at scale. Early attempts to impose order, like the Plan-Do-Assess-Review (PDAR) loop, are a crucial shift but do not constitute a complete engineering methodology. This new reality demands more than incremental adjustments; it compels us to fundamentally reconsider the pillars upon which the SE field is built: the **Actors**, the **Processes** they follow, the **Tools** they use, and the **Artifacts** they shape.
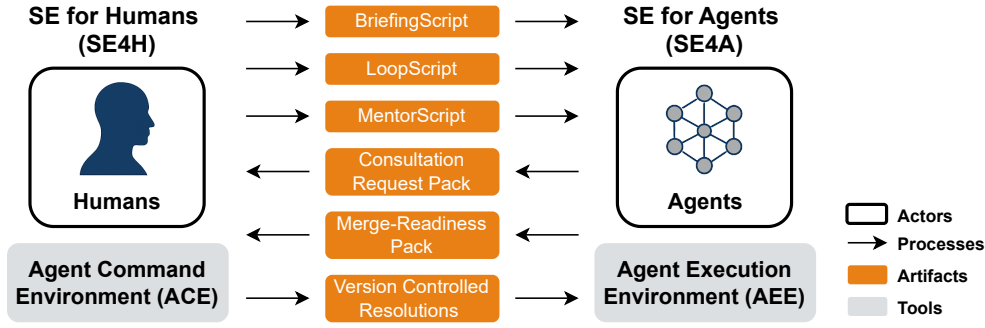
Fig. 1. Structured Agentic Software Engineering (SASE) overview.

This call for structure is not unique to software engineering. Parallel debates are unfolding in education, where frameworks for "human-AI co-thinking" are being explored to transform learning [28]. These frameworks emphasize synergistic partnerships, with humans retaining roles of verification and evaluation while treating AI as an intellectual collaborator. Structured Agentic Software Engineering extends this philosophy to engineering, proposing specific structures and disciplines to make such partnerships succeed at scale.

This paper proposes a vision for this reconsideration: Structured Agentic Software Engineering (SASE), summarized in Fig. 1. SASE acknowledges that SE is a "wicked problem" where rigid, universal processes are futile. It therefore prioritizes adaptable solutions, arguing that an AI teammate that can be quickly onboarded into the context of a specific team, project or organization is more valuable than a brilliant but brittle specialist agent that falters outside its narrow domain. The core thesis of SASE is the introduction of a structured duality, which posits that the field must simultaneously serve two distinct modalities:

- **SE for Humans (SE4H)**, which redefines the human's role to focus on high-level intent, strategy, and mentorship as an Agent Coach.
- **SE for Agents (SE4A)**, which establishes a structured and predictable environment where multiple agents can operate effectively.

This duality requires systematically rethinking the four pillars of SE for an agentic era, as they manifest differently across each modality:

- **Actors:** The cast expands from human developers to a hybrid team of human "Agent Coaches" and specialized software agents.
- **Processes:** Ad-hoc prompting gives way to structured, repeatable engineering activities that govern human-agent collaborations.
- **Artifacts:** Transient, informal prompts are replaced by durable machine-readable structured artifacts that serve as contracts and institutional memory, including not only human-authored briefs (BriefingScript) but also agent-generated Consultation Request Packs (CRPs) for invoking human expertise.
- **Tools:** The traditional all-in-one human-centric Integrated Development Environment (IDE) is replaced by specialized workbenches designed for the distinct needs and strengths of humans and agents.

The traditional IDE is ill-equipped for this new era. As the central tooling pillar, SASE proposes two distinct, purpose-built environments.

- The **Agent Command Environment (ACE)** is the command center for the human "Agent Coach." It is a workbench optimized for human cognition, enabling strategic tasks like specifying intent, orchestrating complex workflows, and reviewing evidence-backed results, while offering full observability into agent activities and associated costs.
- The **Agent Execution Environment (AEE)** is the agents' world, a digital workbench optimized for their unique capabilities, such as high-speed computation, massive parallelism, and tireless, repetitive execution–far exceeding the limitations of human cognition and stamina.

The interaction between these two environments is not a monologue of informal chat, but a structured dialogue managed through a series of explicit, version-controlled, machine-readable, structured living artifacts that enable new processes for our new actors. While the human initiates the conversation with the **BriefingScript** (mission plan), **LoopScript** (workflow playbook), and **MentorScript** (best-practices guide), this is not a static handoff. The agent responds and continues the dialogue by generating its own formal artifacts—the **Consultation Request Pack (GRP)** to consult with humans for expertise and the **Merge-Readiness Pack (MRP)** to present a final, evidence-backed deliverable. The loop is then completed when humans respond with **Version Controlled Resolutions (VCRs)**, auditable artifacts that formally address each GRP or MRP, ensuring that the entire collaboration is a continuous and traceable conversation. This interactive dialogue, and the ongoing loop of structured clarification and feedback is captured as versioned updates to these core artifacts. This ensures that they are living documents that always reflect the complete and current shared understanding of tasks, processes, and a team's collective wisdom and tribal agreements, transforming agentic SE from a craft into a true engineering discipline.

The presented SASE framework is intentionally visionary. Our primary goal is not to offer a definitive solution, but rather to serve as a conceptual scaffold to catalyze an urgent dialogue throughout the SE community. As autonomous agents become first-class actors in the SE lifecycle, the time has come to re-evaluate the foundational tenets of the SE field. We must look beyond the long-held focus on source code as the canonical artifact and the human as the sole actor, and instead build the new processes and tools that are essential for a collaborative, agentic future. This paper is offered as a first step in that collective rethinking, with the express purpose of shaping the discussions that will define the future of the SE field. The paper culminates in a research roadmap that identifies a few key challenges and opportunities, and briefly discusses the resulting impact of this future on SE education.

2   From Agency to Autonomy: A Hierarchical Framework for AI in SE

To situate the SASE vision within the broader evolution of AI in SE, it is crucial to formalize the progression of intelligent SE (aka the integration of AI capabilities in the SE field). Just as the automotive industry relies on standardized autonomy levels to chart progress in self-driving [25], we need a comparable framework in SE. We first must distinguish between **agency**, defined as the capacity of a system to act and execute plans to achieve a given goal, and **autonomy**, which represents the capacity of a system to self-govern and independently formulate those goals. This distinction allows us to formulate a hierarchical framework, analogous to the SAE Levels for autonomous driving, that classifies AI capabilities from simple assistance to full automation. Our presented framework below helps to situate and clarify the transition from AI-Augmented SE (SE 2.0) to the Agentic SE (SE 3.0) era that is the focus of this paper.

Level 0: Manual Coding (No-AI SE) [SE 1.0]

- **Canonical Use Case:** No AI mapping. The human manually translates ideas into tokens by typing.

- **Example Technology Manifestations:** Plain text editors like Notepad, vi and emacs.
- **Car Autonomy Parallel (SAE Level 0):** No Automation. The human performs all driving tasks.

Level 1: Token Assistance (AI-Augmented Coding) [SE 1.5]

- **Canonical Use Case:** Maps a developer's immediate editing intent to predicted tokens.
- **Example Technology Manifestations:** Standard auto-complete features in all modern IDEs.
- **Car Autonomy Parallel (SAE Level 1):** Driver Assistance. The vehicle features a single automated system for driver support, such as cruise control.

Level 2: Task-Agentic (AI-Augmented SE) [SE 2.0]

- **Canonical Use Case:** Maps a planned code change (e.g., a function description) to a complete, generated block of code. Similar levels of automations for other SE tasks like testing and reviewing code exist.
- **Example Technology Manifestations:** GitHub Copilot, Amazon CodeWhisperer.
- **Car Autonomy Parallel (SAE Level 2):** Partial Automation. The vehicle controls both steering and speed, but the human must constantly supervise and remains responsible.

Level 3: Goal-Agentic (Agentic SE) [SE 3.0]

- **Canonical Use Case:** Maps a technical goal (e.g., "add a caching layer") to a detailed plan of code changes.
- **Example Technology Manifestations:** Emerging agents like Cognition's Devin, Anthropic's Claude Code, Google's Jules, and OpenAI's Codex aim for this level. They can take a well-defined goal and execute a multi-step plan (whether self-devised or human-guided) to implement the required changes across code, documentation, and other essential project artifacts.
- **Car Autonomy Parallel (SAE Level 3):** Conditional Automation. The vehicle drives itself under specific conditions, but the driver must be ready to intervene.

Level 4: Specialized Domain Autonomy [SE 4.0]

- **Canonical Use Case:** Maps a broad technical mandate for a specific domain (e.g., "ensure the reliability of the payment service") to a list of concrete technical goals.
- **Example Technology Manifestations:** This level reflects deep, specialized expertise, a frontier where today's most advanced LLMs are starting to aim for. Specialization typically occurs along two primary axes: the **technical stack** and **quality attributes**. For example, Foundation Models like GPT-5 are now being specialized for the frontend web development domain. As highlighted in GPT-5's official prompting guide, this involves fusing technical skills (such as "rigorous implementation abilities" with Next.js and Tailwind CSS) with quality attributes like an "excellent baseline aesthetic taste." Conversely, a Security Agent would specialize along the other axis. It would focus on a single quality attribute (i.e., security) but would be tasked with applying its deep expertise across a diverse range of technology stacks to safeguard all types of software. At level 4, either axis has to be ensured in one domain.
- **Car Autonomy Parallel (SAE Level 4):** High Driving Automation. A Level 4 car is fully self-driving but restricted to a limited operational domain, such as a "geo-fenced" area or, say, specific weather conditions. Similarly, a Level 4 SE has high autonomy but only within a particular "technical domain" be it a specific technology stack or a specific quality attribute domain.

Level 5: General Domain Autonomy [SE 5.0]

- **Core Function:** Maps a general technical mandate (e.g., "ensure all our systems are robust") to a domain-specific technical mandates for any unfamiliar domain it encounters.
- **Example Technology Manifestations:** The overarching challenge of domain autonomy lies in scaling this fused capability: consistently achieving deep, specialized expertise across the full spectrum of technology domains (e.g., server backends, embedded systems) and quality attributes (e.g., performance, reliability, accessibility, security), transitioning from the domain-specific Level 4. As such, general domain autonomy currently is at the conceptual/research stage, i.e., it does not yet exist.
- **Car Autonomy Parallel (SAE Level 5):** Full Driving Automation. A Level 5 car can travel anywhere on any road, in all conditions. Likewise, a Level 5 SE can apply its high-autonomy capabilities to any technical challenge, regardless of its technology stack and domain, becoming a truly generalized expert.

While this framework outlines a complete trajectory toward the ultimate goal of **Autonomous SE** (Levels 4.0 and 5.0), the immediate, industry-defining challenge lies in mastering the **Agentic SE** era (aka SE 3.0). The transition from Level 2.0 to Level 3.0 is not merely an incremental step; it represents a fundamental shift in the human-computer relationship, introducing immense complexity in workflow orchestration, trust, and verification. Before the community can realistically pursue full autonomy, we must first establish the disciplined practices that are required to manage goal-agentic systems. Therefore, the SASE vision presented in this paper is focused squarely on the artifacts, processes, and tools necessary to successfully engineer trustworthy software within the SE 3.0 (aka Agentic SE) era.

3 The Emergence of Agentic Software Engineering

3.1 Industrial Relevance

The SE field has emerged as a primary proving ground for demonstrating the Return On Investment (ROI) on large-scale generative AI models like Large Language Models (LLMs). This strategic focus by frontier AI labs and the broader industry is motivated by a unique convergence of factors:

a **High-Cost Workforce:** Software engineers command premium salaries, meaning even modest productivity gains among this workforce can translate into substantial financial returns.

b **Rich Training Data:** Code repositories, issue tickets, and commit histories constitute one of the most extensive and well-structured datasets of any knowledge-work domain (a fact the Mining Software Repositories (MSR) community has recognized for decades [13, 15]).

c **Measurable Outcomes:** The field offers clear, quantifiable success metrics (e.g., compiler errors, test outcomes, defect rates) that are critically valuable for creating effective and robust reward functions for reinforcement learning (RL) [21, 32].

d **Robust Safety Nets:** The presence of automated testing and CI pipelines mitigates the risk of failures of such models when deployed in practice.

e **Transferable Benefits:** Foundation Models honed on SE workflows [18] generalize well to other business tasks that lack equivalent guardrails and data. This powerful combination of factors has, in turn, ignited a cut-throat competition among several frontier companies, all racing to define and dominate the nascent market for Agentic SE.

This concerted industrial focus has rapidly transitioned **Agentic Software Engineering**, corresponding to the **Goal-Agentic (Level 3)** stage of our framework, from a theoretical concept to a boardroom-level strategic imperative. This is evidenced by the release of specialized coding agents by all leading players (e.g., **Google's Jules**, **OpenAI's**

Codex, and **Anthropic's Claude Code**) and a flurry of strategic M&A activities aimed at securing the invaluable stream of developer feedback data generated from AI-native development environments. Recent events highlight this trend, from OpenAI's reported attempt to acquire the AI-native editor **Cursor**, to the complex acquisition saga of **WindSurf**, which saw the company enter an agreement with OpenAI, only for Google to perform a talent acquisition of its key engineers, leading OpenAI to withdraw and Devin to acquire the remaining company assets (only to lay off all of its staff), highlighting that the ultimate prize in these acquisitions is often the user base and its associated stream of feedback data. These maneuvers underscore the industry's intense focus on securing the "SE data flywheel" needed to refine the next generation of SE-focused LLMs [16, 23, 30].

3.2 What is an Agent and Key Recent Observations

Despite the high stakes of Agentic Software Engineering, the fundamental concept of an "agent" remains loosely defined. To bring clarity, we situate different agent implementations on a spectrum defined by **agency** (executing a given plan) and **autonomy** (formulating the plan itself). This mapping helps clarify the distinction made by frameworks like Anthropic's[1]:

- **Workflow Agents (High Agency):** These are predefined orchestrations of LLMs and tool invocations. Because their high-level logic and flow are hardcoded by a human developer, they primarily exhibit **agency**—executing a given plan to achieve a goal. While low-level tasks within the plan might be handled with some autonomy, the overall system is not self-governing.
- **Autonomous Agents (High Autonomy):** These are systems where agents are given a high-level goal and exhibit **autonomy** by planning, reasoning, and invoking tools to formulate their own path to completion.

This distinction is critical. Workflow agents, as systems of agency, require ongoing manual updates to their core orchestration logic to adapt. In contrast, autonomous agents can be iteratively guided and improved by human developers through natural language. This latter approach eliminates low-level code rewrites and enables flexible adaptation, representing a form of "FMware" that is coded and rewired using English prose [7]. This marks a fundamental shift in how we build software: from explicitly coding logic to declaratively describing behavior.

3.3 Brief Survey of Today's Agentic Solutions on Benchmarks like SWE-Bench

A critical insight emerging in the Agentic SE space is that passing tests alone is no longer enough. Recent deeper examination of prior results of SWE-Bench [17] (the de facto benchmark for evaluating Foundation Model capabilities in SE) highlight a key limitation: the code generated by today's Foundation Models is still far from being merge-ready for professional codebases:

- 29.6% of "plausible" fixes introduced behavioral regressions or were incorrect upon rigorous retesting [31].
- True solve rates for GPT-4 patches dropped from 12.47% to 3.97% after detailed manual audits, revealing widespread weak or cosmetic solutions [1].
- AI agents frequently produced superficial patches limited to single files, unlike human developers [3].
- Many patches passing unit tests failed broader CI checks due to style or hidden regressions [36].

---

[1]https://www.anthropic.com/engineering/building-effective-agents

Even in mature, production-grade ecosystems like the .NET runtime developers are observing the same pattern: **passing tests is far from sufficient**.[2] Achieving merge-ready status requires a deeper understanding of context, intent, and the broader system—qualities today's agents still struggle to demonstrate reliably.

3.4 Brief Survey of Today's Agentic Solutions in the Wild using GitHub Data

Projections from industry leaders, including Google's Chief Scientist Jeff Dean,[3] suggest that AI agents will soon perform at the level of junior developers, a shift substantiated by large-scale analyses of open-source activity [19]. The nature of agentic contributions exhibits distinct, measurable development patterns. In terms of productivity, the median time to complete a pull request authored by GitHub Copilot is just 13.2 minutes, enabling a development velocity where individuals can accomplish years' worth of work in days [19]. Different agents demonstrate high acceptance rates for different types of contributions; for instance, 49.5% of Claude Code's accepted pull requests focus on new features, whereas 42.2% of Copilot's target bug fixes. Notably, agent-authored code trends towards simplicity, with one study showing only a 9.1% increase in cyclomatic complexity compared to 23.3% for human-authored changes. However, this hyper-productivity introduces severe challenges, with code review emerging as a primary bottleneck. Over 68% of agent-generated pull requests reportedly face long delays or remain unreviewed, creating an urgent need for scalable review automation [19].

Despite this review challenge, the quality of contributions is demonstrably high in specific contexts, with Autonomous Coding Agents like OpenAI's Codex achieving near-human acceptance rates for tasks such as documentation and bug fixes [19]. The forward-looking implications are substantial, as major technology firms like Google are reportedly preparing for a tenfold increase in code volume flowing into production. This body of evidence underscores that the Agentic SE era is a current phenomenon actively reshaping development workflows worldwide, which motivates new research into it and its downstream effects.

4 Motivational Example: The Anatomy of an Agentic SE Workflow

Let's ground the Agentic SE era in a concrete example: a developer resolving seven distinct pull requests in a production codebase. This workflow (as shown in Fig. 2) provides a lens through which we can observe both the impressive new capabilities offered by this era and the underlying process and tooling challenges that SASE is designed to solve.

4.1 The New Workflow: A Glimpse into Agentic SE in Practice

In this scenario, the developer's role shifts from coder to specifier. Instead of writing code for each ticket, they spend approximately 1.5 hours authoring detailed, natural-language specifications and guidance for each of the seven tickets. These specifications then trigger a team of autonomous agents to work asynchronously, generating 28 distinct pull requests in parallel (4 pull requests per ticket). This highlights the re-emergence of N-version programming [4, 5, 20], a powerful practice that not only serves as a form of inference-time compute to increase the probability of a successful outcome through trial and error but also enables creative exploration.

The developer then evaluates the different solutions, selecting the most promising one if pull requests satisfy the initial natural-language specification or refining the latter if none of the four pull requests is acceptable, followed by re-triggering the agents. Once all tickets have yielded an acceptable solution, the latter are submitted for review and (eventually) are approved and merged into the code base.
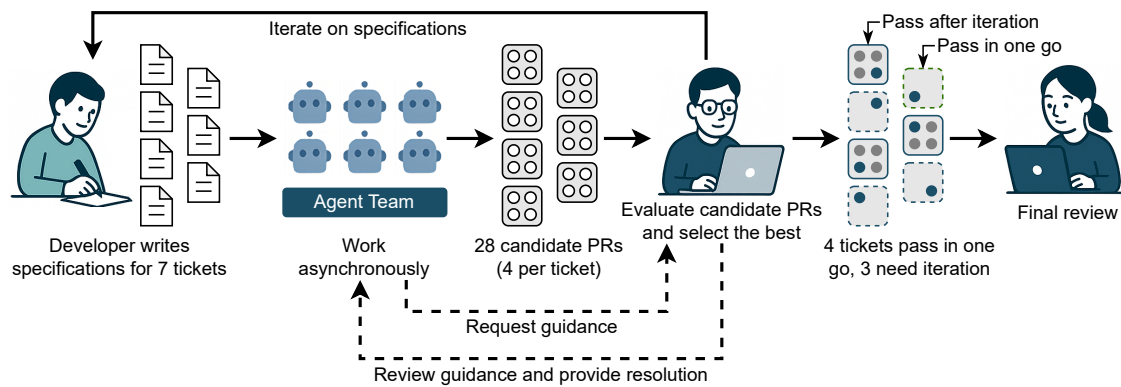

Fig. 2. Overview of an agentic SE workflow

4.2 The Process and Artifact Gaps

This new workflow exposes critical gaps in current SE processes and the artifacts that are used to support them:

4.2.1 The Art of the Briefing – From Vague Tickets to Actionable Briefing Packs. A common failure pattern when using AI agents is to simply paste a raw ticket and expect magic. The informal, natural-language tickets are a source of ambiguity. A more rigorous process would treat this specification as a first-class artifact, moving towards structured specifications where the degree of formality can be adapted to the task at hand. The 100x/1000x developers treat an agent not as a magical tool, but as a junior team member or an outsourced partner. They excel at providing a comprehensive initial briefing that includes not only the specification but also the bigger picture, relevant context, and strategic advice on how to break down the task and how to go about doing it.

To formalize this crucial skill, we advocate for moving beyond ad-hoc prompts toward a structured artifact we call a **Briefing Pack**. This is much more than a specification of intent; it's the detailed work order that a senior developer would give a junior one to ensure success, complete with:

- **What & Success Criteria:** Defines the scope with a verifiable checklist, similar to Scrum's "Definition of Done," but enriched with formal, testable properties like pre-conditions and invariants.
- **Architectural Context:** Clarifies where the work fits in the system, identifying key modules, data models, or APIs to interact with.
- **Strategic Advice:** Recommends specific implementation approaches, such as libraries to use or patterns to avoid, guiding the agent's problem-solving strategy.
- **Potential 'Gotchas':** Highlights known pitfalls or tricky areas to watch out for, like subtle business logic, performance constraints, or dependency issues.

Crucially, a Briefing Pack is not a rigid, one-shot specification. Like pair programming or collaborative design, it evolves through iterative dialogue between the human coach and the agent. Early drafts may be lightweight, progressively enriched with clarifications and refinements based on agent feedback. This iterative style avoids the brittleness of upfront, waterfall-style specifications and reflects how elite engineers already operate in practice. This approach aligns with broader work on "human-AI co-thinking" [28], where humans act as verifiers and evaluators while AI provides generative power. By codifying guidance into Briefing Packs, we ensure that agents operate with

both clarity and accountability. In addition, the Briefing Pack must be a living document, not a static one. Subsequent feedback and clarifications between humans and agents must be incorporated back into the Briefing Pack as versioned updates. This approach directly mirrors the principles of managing institutional knowledge in large-scale software engineering. As detailed by Hyrum Wright and others in "Software Engineering at Google" [34], elite engineering organizations rely on shared, version-controlled knowledge bases as a durable "source of truth" for human developers. By applying this same proven principle to human-agent collaboration, the Briefing Pack transforms from an initial set of instructions into an evolving, auditable record that always reflects the complete and current shared understanding of the task.

Emerging practices, such as the Product/Feature Requirement Prompt (PRP),[4] move in this direction. We can make this process more robust by creating a dedicated language like **BriefingScript** to create version-controlled Briefing Packs. This approach can be viewed as a modern, agent-oriented evolution of Donald Knuth's concept of "literate programming." Instead of treating code as the primary artifact, the focus shifts to the human-readable Briefing Pack: a document that explains the logic and intent from which the agent's work is derived. Looking ahead, creating high-quality Briefing Packs is a significant skill of the elite software engineer, and AI-powered tools could greatly assist developers in drafting them. As an agent becomes more attuned to a codebase and its human collaborator(s), the explicit, human-drafted portion of these packs should shrink, making the entire SE process more efficient.

4.2.2   The Multidimensional Nature of Agentic Feedback and Mentorship. In the agentic SE era, the feedback and mentorship loop becomes significantly more complex than traditional code review. Guidance is not limited to the final code but extends to the entire SE process, encompassing both explicit instructions and implicit principles that the agent must infer. Key dimensions of this new feedback model include:

- **Explicit & Durable Mentorship:** Direct, generalizable guidance from a human coach (e.g., "Avoid obvious comments; instead, comment on the design rationale") must be captured durably. This prevents the agent from repeating mistakes and is the motivation for **MentorScript**, a version-controlled rulebook that codifies best practices.
- **Inferred Mentorship:** Not all guidance is articulated as a general rule. An agent must be able to infer a broader principle from a specific contextual correction by a human, enabling it to learn instead of being spoon-fed. For example, after a human refactors a piece of code for better readability, the agent should propose a new, general rule about that pattern for the coach to approve and add to MentorScript.
- **Holistic Process Feedback:** Mentorship extends beyond the code to encompass the entire SE lifecycle. A human coach may provide feedback on the agent's problem-solving approach, its test planning strategy, the way it debugs or fixes build issues, or its choice and use of tools.
- **Feedback on Multiple Solutions:** The ability of agents to generate multiple potential solutions (*N-versions*) introduces novel feedback patterns. A coach's guidance might involve synthesizing a final solution from different drafts, such as instructing an agent to "combine the UI from solution 1 with the backend logic from solution 2."

4.2.3   From Ambiguous Control to Explicit Orchestration. As observed by industry leaders like Andrej Karpathy,[5] agents cannot infer the expectations or "stakes" of a task; they may "overthink" a simple request or under-deliver on a critical one. For some tickets, a coach may want to grant the agent full autonomy, while for others, they might wish to enforce a strict process. This process can be defined at an organizational level by process engineers (e.g., for regulatory purposes)

---

[4]https://github.com/Wirasm/PRPs-agentic-eng
[5]https://x.com/karpathy/status/1954224651443544436?s=46&t=ayaQZ2-uUhbu4rg402jW5w

and optionally overridden by the coach for a specific ticket, with the override being recorded. Today, this is handled via ad-hoc prompt hacking in the master prompt for these agents.

This motivates the need for **LoopScript**, a declarative language for defining the agent's workflow, allowing a coach to explicitly communicate the required level of rigor and enforce a precise Standard Operating Procedure (SOP) when needed. We are already seeing this happen in the frontier Autonomous Coding Agents: while some of them, like Google Jules, had a planning step from their inception, others, like Claude Code, only recently added a planning mode activated on-demand by the human developer where the agent will generate a plan and await human review before proceeding.

4.2.4 From Code Review to Evidence-Based Oversight. The ultimate goal is to produce a merge-ready contribution. Instead of forcing a human to review dozens of raw pull requests, their cognitive load should be focused on auditing a structured **Merge-Readiness Pack**. This is a bundle of evidence designed to bridge the critical gaps between current agent outputs and the standards of a truly merge-ready contribution. The pack proves the agent's work is trustworthy by providing clear evidence for five key criteria:

(1) **Functional Completeness:**
   - **The Gap:** Agents often produce superficial or partial fixes that pass a narrow set of tests but fail to address the holistic user need.
   - **The Evidence:** The pack must provide proof (e.g., end-to-end test results) that the feature is complete and behaves as specified in realistic scenarios.

(2) **Sound Verification:**
   - **The Gap:** Agents may generate code that passes an existing, weak test suite, or they may fail to create new, robust tests for their own logic.
   - **The Evidence:** The pack includes not just passing test logs, but the agent's test *plan* and the new test cases it generated, proving the verification strategy itself is sound.

(3) **Exemplary SE Hygiene:**
   - **The Gap:** Agent-generated code can be functional but difficult to maintain, often violating project style guides or fundamental engineering principles (e.g., DRY, SOLID).
   - **The Evidence:** The pack includes reports from static analysis, linting, and complexity checkers to demonstrate that the code is clean, readable, and does not introduce technical debt.

(4) **Clear Rationale and Communication:**
   - **The Gap:** An agent's reasoning is often buried in low-level, verbose **trajectory files** or chat logs that are impractical for a human to audit for high-level intent.
   - **The Evidence:** The pack synthesizes this into a clear, human-readable summary (analogous to a well-written pull request description) that explains the approach taken and the trade-offs considered.

(5) **Full Auditability:**
   - **The Gap:** While the initial prompt provides some traceability, true **reproducibility is a major challenge**. Due to agent non-determinism or environment changes, running the same prompt twice may not yield the same result.
   - **The Evidence:** The pack provides a "frozen" audit trail, including versioned links to the exact BriefingScript/-MentorScript, tools, and agent trajectory used, ensuring the result can be reliably audited and reproduced.

To manage this density of information, the pack must support **"progressive disclosure,"** allowing a reviewer to see a high-level summary and then drill down into specific evidence like test logs or execution traces as needed.

4.3 The Tooling Gaps

Finally, this motivational example highlights a fundamental mismatch between the new agentic workflow and the tools that are available for both humans and agents:

4.3.1 A New Workbench for the Human Developer – The ACE. The traditional Integrated Development Environment (IDE) is ill-equipped for the new era of agent-assisted SE. Today's AI-IDE tools like Cursor remain too code-centric and have yet to treat mentorship (a practice that involves a rich set of engineering artifacts beyond just code) as a central engineering activity. The human developer, acting as a coach, requires a new kind of command center to manage and orchestrate this parallelized workflow. We call this the **Agent Command Environment (ACE)**, designed to support not only the 1-to-N interaction of a single developer collaborating with many agents but also the N-to-N collaboration where a human team can collectively collaborate with a shared fleet of AI teammates. These activities differentiate team-level software engineering, which involves multi-role governance, evidence-based review, and cross-functional consults, from single-developer coding. Consultation Request Packs (CRPs), in particular, are the core artifact that operationalizes this team-level collaboration. Crucially, the ACE treats humans as callable endpoints ('humans-as-MCP-tools') that an agent can invoke through CRPs, with the ACE orchestrating the notification and presentation of the request to the appropriate human.

The ACE must integrate capabilities that are currently missing from standard development tools. This includes dedicated capabilities for performing N-version programming in a disciplined and efficient manner, allowing a developer to easily visualize, compare, and mix-and-match components from multiple solutions generated by agents. It must also provide advanced program comprehension and visualization tools to help human developers grasp quickly the architectural impact and scope of generated changes, moving beyond simple textual diffs. Furthermore, the ACE must provide first-class authoring support for the new artifacts that guide agent behavior. This means offering intelligent auto-completion, versioning, archival, and analysis for scripts like BriefingScript, MentorScript and LoopScript. ACE also needs to provide specialized tools for curating the complex context that is required by agents, which often differs significantly from what a human developer might need. This support extends to the strategic management of the agents themselves. An ideal ACE would provide tools for a coach to compose the right team of agents for the job, being aware of their capabilities and cost (much like a manager hires contractors). This implies an "evaluation process" for agents, where underperforming teammates could be "fired," retrained, or demoted. Crucially, the ACE must allow the coach to seamlessly "jump in," when direct implementation is more efficient than specification (such as writing a complex mathematical formula rather than describing it) the developer must be able to instantly transition to a traditional IDE view for surgical code changes, then seamlessly return to a coaching role.[6]

Looking ahead, we expect ACE to adopt voice as a primary interaction modality, as it is faster than typing [10] and reduces context switching in software development workflows [2]. Modern foundation-model ASR systems (e.g., OpenAI's Whisper) already produce accurate transcripts across accents, background noise, and technical vocabulary, making voice-driven ACE workflows practical. A growing developer community codes by speaking with tools like Talon Voice,[7] often paired with Cursorless for VS Code,[8] suggesting a straightforward adoption path for ACE. Within SASE, these capabilities enable a human to conduct fluid, high-bandwidth dialogues to issue commands, dictate specifications, and provide mentorship.

---

[6]The demo provides an example of performing a surgical code change when developing mobile apps: https://www.linkedin.com/posts/ahmed-e-hassan_se3-harmonyos-ainativese-activity-7274452304278265856-7F7L
[7]https://talonvoice.com
[8]https://www.cursorless.org

4.3.2 An Optimized Workbench for the Agent – The AEE. Just as humans need a new command center, agents require their own specialized environment designed for their unique capabilities. The tools that excel for humans are often suboptimal for agents. Much of modern SE has focused on creating high-level tools that reduce cognitive load for humans. This optimization, however, is often at odds with the needs of an agent. Agents, unburdened by human cognitive limits, thrive on raw, low-overhead tools that are optimized for computational efficiency and provide structured, machine-readable feedback. The fact that many of today's autonomous coding agents still rely on basic utilities like grep exposes this fundamental mismatch.

This necessitates the creation of an **Agent Execution Environment (AEE)**: a workbench built for agents, not humans. Instead of human-centric interfaces, the AEE must be equipped with agent-native tools that leverage their strengths. These might include hyper-debuggers capable of analyzing vast state spaces, powerful semantic search utilities, and structural editors that manipulate code as an abstract syntax tree rather than as simple text. Beyond these task-oriented tools, the AEE must also include a robust monitoring infrastructure to manage the agents' operational health. This internal system would autonomously handle low-level issues such as spotting security vulnerabilities, flagging agents that are incurring unexpectedly high computational costs, or repairing and replacing broken virtual environments. The goal of this self-monitoring is to ensure that only significant problems requiring strategic human intervention are surfaced to the human in the ACE.

5 The Engineering Activities of SASE

The SASE vision is operationalized through a set of structured engineering activities. The activities outlined in this section should not be considered a definitive or exhaustive list. Instead, they represent an initial conceptual scaffold intended to catalyze a community-wide dialogue. These activities provide a foundational framework for effective N-to-N collaboration, structuring the interactions between human coaches, between AI teammates, and across the hybrid team of human and AI counterparts. Crucially, these activities differentiate team-level software engineering, which involves multi-role governance, evidence-based review, and cross-functional consults, from solo coding. Consultation Request Packs (CRPs), in particular, are one of the core artifacts that operationalize this team-level collaboration. By enforcing the discipline required for merge-ready contributions, we can establish a systematic process that leads to the development of robust and trustworthy software. We strongly encourage the community to challenge, refine, and extend this initial set by defining these activities in greater detail and proposing entirely new ones as the Agentic SE field matures.

5.1 Briefing Engineering (BriefingEng): The Art of the Mission Briefing

In the SE 3.0 era, the primary creative output of an engineer evolves from implementation logic to the articulation of unambiguous intent and guidance. Briefing Engineering (BriefingEng) is the activity that codifies this crucial skill. This activity does not seek to reinvent the wheel; instead, it builds upon the decades of foundational work from the Requirements Engineering (RE) and Agile/Scrum communities, adapting their principles for an agentic context. It is a hybrid discipline that fuses requirements specification with architectural design, strategic implementation advice, and test planning into a single, cohesive artifact. As the "brief" becomes as, if not more, critical than the code itself, we see a vital opportunity for researchers and practitioners from these communities to lead the charge in co-designing the next generation of specification practices for a future where humans guide and agents build.

- **Purpose**: To move beyond the common failure pattern of pasting a raw, vague ticket and expecting magic. This activity treats the mission brief as a first-class artifact, ensuring an autonomous agent receives a comprehensive and

actionable work order. Unlike a traditional Software Requirements Specification (SRS) which is often implementation-agnostic, a BriefingScript is a specification for action: a version-controlled, testable, and machine-readable document that is as central to the engineering process as the source code itself.

- **Actor**: The human Agent Coach.
- **Workbenches**: All BriefingEng activities are centered in the Agent Command Environment (ACE). AI assistance can be used to help the coach author high-quality briefs by flagging ambiguity, surfacing edge cases, ensuring logical consistency, and generating property-based acceptance tests for the final output from the briefs.
- **Artifacts**: The **BriefingScripts**. A BriefingScript is a structured, version-controlled document that serves as the work order. To facilitate this, a purpose-built language becomes essential, making the specification itself traceable, reviewable, and testable. While its structure provides discipline, it is not intended to enforce a rigid, up-front specification. Instead, BriefingScripts are best authored through an interactive, iterative process, where the initial draft may be lightweight and refined over multiple cycles of agent interaction. This flexibility ensures briefs evolve naturally alongside the task, rather than locking humans and agents into an unrealistic "waterfall" style process. While the RE field has long pursued this goal, it becomes more feasible in the agentic context for three reasons:

(1) The primary consumer is a machine, which necessitates and benefits from a formal structure;

(2) Briefs are often for more granular tasks than a monolithic SRS, making formalization tractable;

(3) Modern AI assistants can help humans write these structured briefs, lowering the barrier to entry that hindered past formal methods.

- **Example of Emerging Industry Efforts–The Product Requirement Prompt (PRP)**: The Product Requirement Prompt (PRP) is an emerging industry pattern that exemplifies a well-formed BriefingScript. For instance, Amazon's Kiro[9] demonstrates the shift toward "spec-driven development," where the specification drives the process. However, a complete brief goes beyond a static specification and is composed of several distinct sections:
  - **Goal & Why**: This sets the stage with a high-level objective and the business value behind it. This gives the agent not just a task, but a purpose, which helps it make better micro-decisions during implementation.
  - **What & Success Criteria**: This section defines the scope with a verifiable checklist for the "definition of done." More than just business outcomes, this is where the coach can express pre- and post-conditions, invariants (e.g., "the sorting operation must be idempotent"), and other semantic requirements, transforming abstract goals into testable properties.
  - **All Needed Context**: This provides the agent with curated information. This is arguably the most critical section for preventing agent hallucination, but it is a careful balancing act; the goal is to provide concise, relevant context, as overwhelming an agent with an entire "kitchen sink" of information can increase cost and cause performance degradation due to context overload. This includes links to documentation, relevant existing files, and "Known Gotchas"–explicit warnings about common pitfalls.
  - **Implementation Blueprint**: This section provides high-level strategic guidance and constraints, not a low-level implementation plan. It guides the agent's approach (e.g., "Use the Strategy pattern here," "Do not modify this legacy API") while leaving the detailed implementation choices to the agent's autonomy.
  - **Validation Loop**: This codifies the acceptance testing plan directly into the brief. It often specifies a multi-level validation strategy, from linting and unit tests to integration tests. When an agent's output diverges, traceability

---

[9]https://kiro.dev/blog/introducing-kiro/

assistants can pinpoint the exact briefing clause that failed, providing actionable feedback for refining either the brief or the agent's training.

5.2 Agentic Loop Engineering (ALE): Disciplined Orchestration

With a clear brief in place, Agentic Loop Engineering (ALE) governs how agents execute tasks. Just as BriefingEng builds on the work of RE and Agile, ALE is deeply rooted in the principles pioneered by the DevOps community. It transforms the agent's work from an opaque, black-box process into a disciplined, auditable, and reproducible workflow, moving beyond simple iterative cycles like the Plan-Do-Assess-Review (PDAR) loop. The declarative pipelines, infrastructure-as-code, and focus on observability that are central to modern DevOps are the direct precursors to the automated, agent-driven workflows defined by a LoopScript. We therefore see a crucial role for the DevOps community in designing the next evolution of CI/CD.

- **Purpose:** This involves defining how agents work together (or alone), the patterns of their collaboration, and how they engage with their toolset. Such explicit direction is essential because agents cannot infer the "stakes" of a task; they may "overthink" a simple request or under-deliver on a critical one.
- **Actors:** The workflow is orchestrated by the **human coach**, but the execution is performed autonomously by **agents**.
- **Workbenches:** The workflow is defined in the **ACE**, but the agents execute it within the **AEE**.
- **Artifacts:** The **LoopScripts**. Instead of relying on ad-hoc prompt hacking, the coach uses a declarative language like LoopScript to define the Standard Operating Procedure (SOP). Like all SASE artifacts, the LoopScript is a living document. A coach might dynamically adjust the workflow (for instance, by allocating more agents to a promising path or by adding a new review checkpoint if early results look uncertain) ensuring that the orchestration strategy is customized for a task when needed. A LoopScript can specify:
  - **Task Decomposition and Parallelization:** A BriefingScript can be assigned to multiple agents, which could involve multiple instances of the same model or, more powerfully, a heterogeneous team composed of specialized agents. This reflects the emerging practice of using different models for their distinct strengths, for example, using a model like Gemini 2.5 Pro for high-level planning while leveraging Claude Opus or Sonnet for the detailed code generation. This makes powerful practices like N-version programming routine. For instance, a developer resolving seven tickets can trigger such a team to generate 28 distinct pull requests in parallel (4 per ticket), enabling creative exploration and increasing the probability of a successful outcome. The key metric shifts from single-task latency to overall system throughput.
  - **Workflow Strategy:** The coach can define the required level of rigor, granting full autonomy for a simple bug fix while enforcing a strict, multi-stage review process for a critical security patch.
  - **Evidence-Based Acceptance Criteria:** The LoopScript defines the structure of the final deliverable: a "Merge-Readiness Pack." This isn't just a pull request; it's a bundled collection of evidence that proves the agent's work meets all five criteria for being merged:
    * **Functional Completeness:** All acceptance criteria are met, and the feature behaves as specified in realistic scenarios with no follow-up work required.
    * **Sound Verification:** A well-reasoned test plan provides high confidence in the code's correctness, covering happy paths, edge cases, and failure modes.
    * **Exemplary SE Hygiene:** The change set is small, focused, logically structured, and adheres to key engineering principles (e.g., SOLID, DRY) and project style guides.

  * **Clear Rationale and Communication:** Comments and pull request summaries effectively convey intent–the "what" and the "why"–not just mechanics.
  * **Full Auditability and Evidence:** The change is accompanied by an organized evidence bundle (e.g., test results, static analysis logs) that supports both quick review and in-depth inspection.
- **Example of Emerging Industry Efforts–SuperClaude:** SuperClaude[10] is an example of a new ecosystem of tools that are being built to support these structured workflows, moving beyond simple chat interfaces to provide powerful orchestration and validation capabilities. SuperClaude is an open-source toolkit that enhances Anthropic's Claude Code CLI with a collection of predefined commands and templates for common development tasks like generating PRPs, reviewing code, or running debugging workflows. By encapsulating best-practice prompts into simple slash-commands, it eliminates redundancy and ensures a consistent, high-quality interaction with the Autonomous Coding Agents, boosting the productivity of a solo developer.

5.3 AI Teammate Mentorship Engineering (ATME): Codifying Team Norms and Best Practices

To ensure agent-generated code is not just functional but also maintainable and aligned with team culture, agent guidance must be treated as first-class code.

- **Purpose:** To transform mentorship from an implicit, ephemeral activity (e.g., comments in a code review) into an explicit, evolving, and codified discipline. This is "mentorship-as-code."
- **Actors:** Guidance is provided by the **human coach** and durably consumed by **agents**.
- **Workbenches:** Mentorship rules are authored in the **ACE** and directly influence agent behavior in the **AEE**.
- **Artifacts:**
  - **MentorScripts:** These are structured, machine-readable rulebooks that codify project norms (aka tribak knowledge and aligned understandings). A MentorScript allows teams to define rules ranging from granular checks ("all new functions must have deterministic tests") to high-level principles. This makes mentorship, once an implicit and ad-hoc activity, an explicit, reviewable, and continuously evolving discipline. MentorScript rules would be subject to their own quality gates, including linting, unit testing, and conflict detection, ensuring they are atomic and deterministic. Crucially, every action an agent takes is traced back to the MentorScript rules that were considered (through prompt interpretation techniques such as PromptExp [8] and reasoning observability techniques such as Watson [24]), enabling rapid root-cause analysis when the behavior of an agent deviates from expectations. This explicit guidance reduces the burden on the human coach to infer complex rule interactions, making the behavior of agents more predictable and reliable.
  - **Structured Mentorship:** The review process generates a critical, multidimensional artifact. When a coach provides feedback, it is captured and structured. This feedback is not limited to the final code but extends to the entire process.
    * **Explicit Mentorship:** A direct instruction like, "Avoid obvious comments; instead, comment on the design rationale," is captured.
    * **Inferred Mentorship:** An agent should be able to infer a broader principle from a specific correction. After a coach refactors code for readability, the agent should propose a new, general rule for the coach to approve and add to the MentorScript.

---

* **Feedback on Multiple Solutions:** N-version programming introduces novel feedback patterns, such as instructing an agent to "combine the UI from solution 1 with the backend logic from solution 2."
- **Example of Emerging Industry Efforts–Meta-Prompt Files (AGENT.md):** A grassroots example of this principle is the use of meta-prompt files. To ensure consistency across an entire project, practitioners use files often named CLAUDE.md or .clinerules or AGENT.md. Before starting any task, the agent is instructed to read these files, effectively loading the project's "institutional/tribal knowledge." The goal is to reduce redundant instructions in individual prompts, ensure stylistic and architectural uniformity, and allow teams to codify lessons learned, creating a continuously improving "employee handbook" for their AI teammates. However, this practice remains highly experimental and highlights a critical area of open exploration. Currently, the community has no consensus on what these files should contain, nor the appropriate level of detail–whether they should be sparse, high-level principles or exhaustive, fine-grained rules. This uncertainty underscores that future work is not just about defining a language like MentorScript, but also about discovering and codifying the best practices for its effective use.

5.4 Agentic Guidance Engineering (AGE): Leveraging the Human in the Loop

While BriefingEng initiates work, **Agentic Guidance Engineering (AGE)** governs the structured role of the human in reviewing and responding to agent-generated artifacts and clarification requests (e.g., during the BriefingEng and Mentoring activities). AGE elevates the human from a passive approver of outputs to an active, on-demand consultant who intervenes precisely where their expertise adds the greatest value.

- **Purpose:** To formalize and optimize human participation in the agentic loop, ensuring that when agents escalate issues through a Consultation Request Pack (CRP) or submit a Merge-Readiness Pack (MRP), human input is efficient, targeted, and becomes a durable part of the project record.
- **Actors:** The human engineer (who may be the task initiator or a domain specialist).
- **Workbenches:** All AGE activities are performed within the **ACE**, which provides an inbox-like interface for triaging CRPs, auditing MRPs, and issuing structured resolutions.
- **Artifacts:**
  - **Consumed Artifacts – Consultation Request Packs (CRPs):** Generated when an agent requires human input to proceed. A CRP is contextualized by the active BriefingScript, potentially triggered by LoopScript or MentorScript rules, and documents the specific uncertainty or decision point.
  - **Consumed Artifacts – Merge-Readiness Packs (MRPs):** A structured evidence bundle submitted for human approval, designed to demonstrate functional completeness, sound verification, engineering hygiene, rationale, and auditability.
  - **Produced Artifacts – Version Controlled Resolutions (VCRs):** The outcome of AGE activities is a version-controlled Resolution. Each Resolution is explicitly linked to the artifact that it addresses (CRP or MRP), preserving traceability and enabling downstream auditing and learning.

5.5 AI Teammate Lifecycle & Infrastructure Engineering (ATLE & ATIE): Building the SE for Agents Foundation

To fully unlock the agentic SE (SE 3.0), we must simultaneously support the SE activities of agents (i.e., SE for Agents). This involves fundamentally rethinking our tools, practices, and artifacts when the primary actor is an agent, not a human. The best practices that have served us for decades, designed around the constraints of human cognition and patience, must now be re-examined and, in many cases, inverted. Engineering this new, agent-centric foundation is a

monumental task that falls squarely within the expertise of the Platform Engineering community. As agents begin to operate at scale, the need for the robust, secure, and scalable platforms that are the core mission of Platform Engineering becomes paramount. Their work is essential in building the next generation of "internal developer platforms," not for humans, but for the fleets of autonomous agents that will inhabit them, ensuring these systems are reliable, efficient, and secure.

- **Purpose:** To engineer the agent's environment (the AEE), enable agents to retain memory and learn over time (ATLE), and build the agent-native toolchains they need to operate effectively (ATIE).
- **Actors:** The fundamental shift is in the actor itself. For decades, SE has been optimized for the human developer. In an agent-centric world, the primary actor is **computational**. The human engineer's role evolves into that of a strategist, mentor, and conductor, serving as the ultimate arbiter of value and the indispensable conduit for tacit, "tribal" knowledge. The ultimate goal is not agents that are perfect out of the box but ones that can learn and ramp up first.
- **Workbenches:** This work is fundamentally about architecting the **Agent Execution Environment (AEE)**.
- **Core Concepts:**
  - **Persistent Memory (ATLE):** Agents are embedded with long-term memory of project history and decision logs, allowing them to maintain continuity across tasks without the coach repeatedly supplying the same guidance.
  - **Agent-First Code Practices (ATLE):** With the agent as the primary actor, long-standing process principles must be re-evaluated. For instance, the **"Don't Repeat Yourself" (DRY)** principle is often reversed. Code cloning, a source of maintenance debt for humans, can become a viable strategy for an agent, as it simplifies its reasoning process while the downside (updating all instances) is trivial. This theoretical shift is supported by industry observations, such as the GitClear report noting sharp increases in code duplication on GitHub since the emergence of co-pilot.

    Furthermore, the **ROI of Clean Code** (high cohesion, low coupling, comprehensive documentation) becomes crystal clear, as these practices make a codebase a more fertile environment for agents to inhabit. Such efforts are analogous to making an open-source project accessible to novices through a well-defined plugin architecture, in turn attracting and growing a large community around that project. Previously, this meant redirecting resources away from immediate business needs like feature development. Now, a small human team, aided by agents, can perform this cleanup, which in turn unlocks massive productivity gains for its fleet of feature-developing agents. This new model also favors programming languages with strong, compile-time safety guarantees, such as **Rust and TypeScript**. The up-front effort to satisfy a strict compiler is less of a barrier for an agent, and the payoff is immense, as the strong type system prevents entire classes of bugs by construction.
  - **Agent-Native Toolchain (ATIE):** The tools built for human developers are often ill-suited for agents. We have spent decades building tools like IDEs and visual debuggers to reduce cognitive overload, but an agent has no such limitations. This shifts the entire optimization landscape. SE for Humans has historically focused on precision@K for a small K, because human time is precious. In an agent-first world, precision@100 is perfectly acceptable if a subordinate agent can post-process the results, opening up entirely new avenues for automated analysis where the human is completely out of the loop.

    Expressive feedback is paramount. Rust's toolchain exemplifies this, as its rich, constructive compiler messages enable agents to learn quickly from failures, providing a blueprint for agent-friendly environments. The path forward involves creating Agent-Native Model Context Protocol (MCP) servers [12] that return deep, interpretable feedback and support agent-driven refinement of tool usage descriptions. This kind of self-improving tooling loop,

already being adopted manually by teams like Anthropic by optimizing their MCP descriptions to suit Agents instead of Humans, is proving essential for making agentic SE robust, scalable, and fault-tolerant.

– **Engineered Multi-Agent Teams:** Paradoxically, while some human-centric rules are inverted, foundational principles for managing system complexity, like Modularity and Separation of Concerns, become even more critical. This is driving a key trend away from monolithic, general-purpose agents and toward engineered multi-agent teams. These systems are structured workflows where agents are assigned specialized roles like "planner," "coder," "tester," and even a "critic" to create an internal feedback loop.

While this specialization is a practical solution to the core limitations of today's large models (protecting each agent's context from the "pollution" of side-tasks and an overwhelming number of tool choices), it also points to a more robust, long-term architectural principle. A modular design improves the interpretability of the system's behavior, as the trajectory of a specialized agent is far easier to audit than the interleaved reasoning of a monolithic one. This also paves the way for an ecosystem with highly specialized agents. This, in turn, could lead to the emergence of "agent stores": digital marketplaces analogous to today's app stores. In such a future, a bespoke team could be dynamically composed, with a coordinating agent (whether human or AI) selecting best-in-class agents from various vendors for specific roles, such as a "React Refactoring Agent" or a "Python Security Audit Agent." Moreover, Multi-Agent Teams provide critical security and control benefits. By assigning specific tools as well as security and resource-usage policies to specialized agents (e.g., a "Test-Agent" that can run tests but not commit code), the system limits the potential "blast radius" of a misbehaving or compromised agent.

These observations are already manifesting in different ways. For instance, the evolution of commercial Autonomous Agents like Anthropic's Claude Code have shifted from a monolithic agent to a multi-agent architecture where specialized sub-agents are spawned for specific tasks. A more comprehensive example is the BMAD (Breakthrough Method for Agile AI-Driven Development) framework, which takes the "team" metaphor literally, organizing agents into a full-fledged agile structure to tackle complex projects.

– **Lifetime Teammates:** Perhaps the most significant shift is moving from stateless agents to persistent teammates that learn and grow over time. This is the focus of **AI Teammate Lifecycle Engineering (ATLE)**, a discipline aimed at giving agents memory and long-term context.

  * **From Contractors to Partners:** The goal is to evolve agents from "one-off contractors" who start every new task from scratch to "life-long partners" who retain institutional knowledge. Early examples of this concept, like the **DeepWiki** used by the Devin agent, allow an agent to build and refer to its own documentation and decision logs across multiple tasks, creating continuity and preventing it from repeating mistakes.

  * **Proactive Maintenance:** An agent with persistent memory and access to the codebase can become a proactive partner. During idle compute cycles, the ACE can schedule agents to perform valuable maintenance tasks, such as scanning for technical debt, identifying documentation gaps, or proposing code refactorings. These proposals would be filed as new BriefingScripts, entering the standard SASE workflow for human review and prioritization, thus transforming maintenance from a reactive chore into a continuous, autonomous improvement process and eventually moving SE 3.0 into SE 4.0, where we remove the requirements to get human approval on such optimization activities.

• **Example of Emerging Industry Efforts–BMAD (Breakthrough Method for Agile AI-Driven Development):** BMAD[11] is a comprehensive framework that organizes AI agents into roles like Product Owner, Architect, Developer,

---

[11]https://github.com/bmad-code-org/BMAD-METHOD

and Tester. It begins with an "Agentic Planning" phase where agents collaborate to produce detailed PRDs and architectural designs. A Scrum Master agent then "shards" this work into discrete "story files," each containing the specific context needed for a Developer agent to implement a piece of the feature. This task decomposition and role specialization is BMAD's key strength, enabling it to tackle complex projects with a high degree of structure and parallelism.

6 Discussion

6.1 The Critical Gap: Observability, Archival, and Revision Control

Despite the rapid, practitioner-driven innovations, the current tooling landscape reveals critical gaps in the foundational pillars of SE. The current agentic offerings are unprepared for the fundamental SE needs of traceability, observability, and revision control.

On one extreme, powerful command-line interfaces like Claude Code and other CLI platforms grant developers immense control and flexibility. However, these platforms often result in ephemeral interactions. The rich conversational context (the back-and-forth dialogue of planning, clarification, and refinement between the human and the agent) is lost today, existing only in a terminal's scroll-back buffer. The lack of systematic archival of the agent's reasoning or the human's guidance makes it nearly impossible to reconstruct the evolution of design decisions or reproduce specific outcomes – a shortcoming that we previously underscored in our SE 3.0 call for a new paradigm of conversational development [14].

On the other extreme, more integrated platforms like GitHub Copilot are much further ahead in addressing this by anchoring human-agent interactions to pull requests, thereby creating a persistent historical record. The agent's suggestions and the resulting code changes are tracked. However, these systems treat the agent mentoring and the code as separate, unlinked artifacts. One can roll back a code change, but this does not roll back the state of the agent or the conversational thread that produced the code. The causal link between a specific piece of mentorship and its materialization in code is not explicitly maintained. This creates a fragmented history, where the "why" behind a change is decoupled from the "what."

Moreover, no mainstream system today provides adequate observability into the agent's internal state or a unified, interlinked revision control system for the combination of code, prompts, and conversational context. The artifacts of this new process are not being managed with the same rigor as traditional code. For agentic SE to mature from a craft into a true engineering discipline, we must develop new foundational building blocks that systematically support this new way of working, ensuring that the entire human-agent collaboration is observable, versionable, and trustworthy.

6.2 Embracing the Bitter Lesson in the Agentic SE Era

At first glance, our emphasis on structured processes (from Briefing Packs to structured orchestration via LoopScripts) might seem to run counter to the core message of Rich Sutton's **"Bitter Lesson"** [27]. That lesson powerfully argues that general methods that scale with computation and data ultimately triumph over approaches that rely on baking in specific human knowledge and processes. One might think that our attempts to add structure into Agentic SE is a futile effort to impose human-centric designs on this new era.

However, the lesson's power is most potent where data is abundant like at lower levels of abstraction or for common problems like building a web application. Its application becomes far more complex for novel tasks or in niche domains where training data is scarce. For these settings, relying solely on large-scale data is inefficient; a human is still needed

to provide the overarching structure and connect the dots. This need arises from fundamental constraints inherent in both current and future AI: they lack embodied human experience necessary for physical-world verification, contextual judgment, and deep ethical reasoning [28]. The structures proposed by SASE are therefore not mere attempts to encode human knowledge, but rather recognitions that the human-AI partnership must be designed around complementary roles. In this partnership, the human provides strategic and ethical guardrails (the "why") with precision for AI agents. Even within that unique software system, there may be repetitive sub-problems where an agent can and should be granted full autonomy. We also do note that in any system, a baseline of structured interactions, a Standard Operating Procedure (SOP), is essential for coordination and governance, especially in regulated settings.

Therefore, our vision does not reject the power of scale; it seeks to create the conditions for it to succeed reliably (aka reproducibility) within a complex engineering context. Furthermore, our focus extends beyond mere process control to address the critical SE needs of traceability, robust human-agent communication [35], and fostering an agent-first mindset in how we structure codebases and design tools. This reinforces the idea that SE, especially in this new era, is as much about the journey (the collaboration, the evolution of intent, the trail of decisions) as it is about delivering the final outcome.

Ultimately, we believe software engineers must embrace the Bitter Lesson as a guiding principle. The core skill of the modern super software engineer is mastering the **duality of control**: strategically deciding when to impose a structured workflow (for novel, high-level tasks) and when to "let the agent loose" on a well-defined problem where it can leverage scaled learning. This is analogous to managing a team of brilliant experts; one must know when to provide a detailed work order and when to trust a team member with full autonomy.

6.3 Agentic Software Engineering, Not Just Agentic Coding: The Centrality of N-to-N Collaboration

A foundational premise of this paper is the distinction between gentic coding and agentic software engineering. Agentic coding, which characterizes the current state of most available tools, focuses primarily on the 1-to-1 interaction between a developer and an AI assistant to accelerate implementation tasks. It is fundamentally an augmentation of a solo activity, aimed at boosting individual productivity.

Software Engineering (SE), by contrast, has always been a **team sport**. It is not only about producing code but also about managing complexity, coordinating across diverse roles, reconciling competing stakeholder needs, and ensuring the long-term sustainability of shared artifacts. These inherently collective challenges demand the acceleration of structured collaboration, not just individual acceleration.

Structured Agentic Software Engineering (SASE) is explicitly designed for this broader scope. It provides the artifacts, processes, and workbenches necessary to support **N-to-N collaboration**, where many humans and many agents interact as a coordinated team. This model manifests at several levels:

- **1-to-N Human-Agent Collaboration:** A single human orchestrates and mentors a fleet of agents, directing parallelized workstreams.
- **1-to-N Agent-Human Collaboration:** An agent escalates a Consultation Request Pack (CRP) to the appropriate human specialist, enabling targeted, domain-specific feedback.
- **N-to-N Hybrid Collaboration:** Multiple humans collectively oversee and mentor a shared pool of agents, while agents collaborate with one another or even with specialized sub-agents.

For instance, an agent may route a database schema issue to the designated database architect, who provides feedback through the ACE. In more advanced settings, that "architect" role may itself be filled by another specialized agent. Without explicit, durable artifacts such as the CRP, these complex multi-actor workflows would be ephemeral,

untraceable, and ultimately unmanageable. SASE ensures that these interactions leave behind structured, auditable records, transforming ad-hoc agentic coding into a disciplined engineering practice.

7   From Vision to Reality: A Research Roadmap for Structured Agentic SE and Its Implications

While the traditional SE era celebrates the "10x engineer," the structured, collaborative Human+AI paradigm of SASE introduces the potential for 100x or even 1000x productivity. Achieving this requires building a new foundational infrastructure to support the agentic era. The research directions outlined in this section are not intended to be comprehensive; rather, they are presented to catalyze a shift in the community's focus: away from the purely code-centric problems of the pre-SE 3.0 world and toward the novel challenges of engineering with and for intelligent agents. We encourage the community to explore these and other related avenues to build the future of our field. Finally, we conclude by discussing the profound implications of this agentic shift on the human actor, specifically the urgent need to reimagine software engineering education.

The efficacy of the Structured Agentic Software Engineering (SASE) framework is predicated on the alignment of six mutually reinforcing engineering activities. This framework begins with **Briefing Engineering (BriefingEng)**, which establishes clear intent, operational guidelines, and verifiable acceptance criteria for an agent's task. The execution of this brief is governed by **Agentic Loop Engineering (ALE)**, which focuses on orchestrating how agents explore, iterate, and converge on solutions. To ensure that these solutions align with established team norms and quality standards, **AI Teammate Mentorship Engineering (ATME)** provides explicit, testable, and codified guidance. The structured involvement of humans throughout the agentic loop is captured by **Agentic Guidance Engineering (AGE)**, which formalizes the way human expertise is injected into Consultation Request Packs (CRPs). The long-term viability and growth of the AI teammate are managed through **AI Teammate Lifecycle Engineering (ATLE)**, which endows agents with persistent memory and mechanisms for proactive codebase maintenance. Finally, all activities are supported by **AI Teammate Infrastructure Engineering (ATIE)**, which provides the essential interfaces, compute fabrics, and agent-native tools required for effective operation. In essence, these activities form a cohesive system: BriefingEng tells an agent **what** to achieve, ALE governs **how** it executes, ATME defines **why** its decisions must follow team norms, ATLE ensures it **remembers and improves** beyond a single task, and ATIE provides the environment where it can **thrive at scale**. Fig. 3 outlines the relations between the tools, activities/processes, actors, and artifacts across the two dualities.

7.1   Briefing Engineering (BriefingEng)

BriefingEng is the discipline of creating clear, unambiguous, and machine-executable mission briefs for agents. This activity is more than just requirement specification; it is about transforming the articulation of high-level intent, domain context, strategic guidance, and acceptance criteria into a rigorous engineering artifact.

- **Formalizing BriefingScript:** A primary research direction is the design of a **BriefingScript language**. How can we create a language that allows engineers to express high-level properties (e.g., "the user authentication flow must be stateless," "the sorting operation must be idempotent") and domain constraints without over-specifying implementation details? Research is needed to define the right formalisms and primitives that balance expressiveness with ease of use.
- **AI-Powered Authoring and Review:** How can we build AI assistants to help engineers author high-quality briefs? This involves research into ambiguity detection in natural language specifications, automated generation of edge cases, and ensuring logical consistency between different clauses. Critically, such an assistant should guide the coach
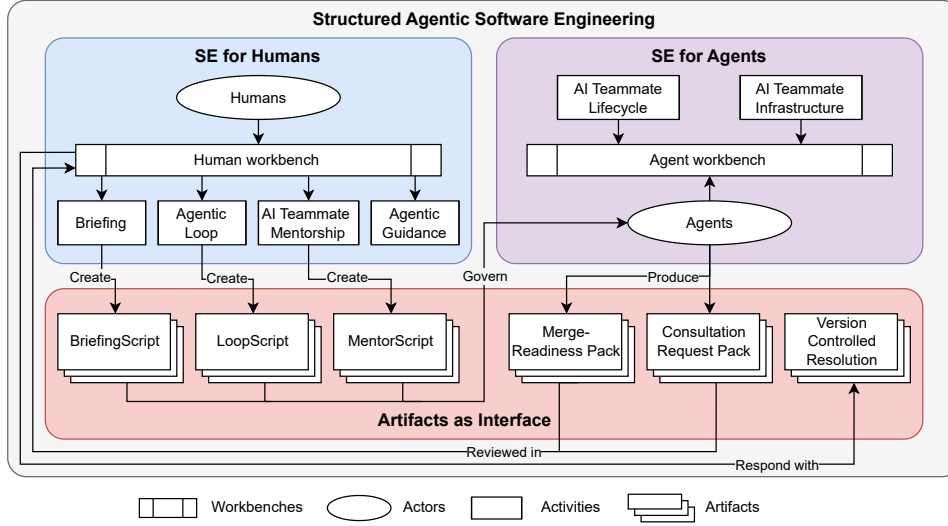
Fig. 3. The Structured Agentic Software Engineering (SASE) framework: dual domains for humans and agents, engineering activities, and artifacts.

toward defining high-level, property-based acceptance criteria rather than brittle, example-based test cases, leading to more robust and generalizable solutions.

- **Interactive Feedback and Synthesis:** With agents capable of N-version programming, a key challenge emerges: how do we design interfaces for efficiently reviewing and synthesizing multiple solutions? Research should explore novel interaction models that allow a coach to visually compare different outputs, select the best components from each (e.g., "take the UI from solution A and the data model from solution B"), and merge them into a final, superior result.

- **Traceability for Debugging:** To build trust, every piece of agent-generated code must be traceable back to the intent that created it. A critical research area is the development of tooling that provides a verifiable link from a line of code back to the specific BriefingScript clause that prompted it. Recent work in cognitive observability, such as the Watson framework, which provides reasoning observability into the implicit thought processes of agents, represents a foundational step in this direction. A key research challenge is to extend such concepts to the more complex, multi-artifact and multi-agent setup of SASE.

- **Pre-declarative Consultation Policies:** How can a BriefingScript be used to define predictable escalation paths? Research is needed into syntax for pre-declaring consult policies (e.g., "For any architectural choice, generate a CRP and escalate to the 'Architect' role"), making the consultation process auditable and aligned with project governance.

7.2 Agentic Loop Engineering (ALE)

ALE governs *how* an agent executes its mission. It transforms the agent's internal problem-solving process (a form of agentic search) into a disciplined, auditable, and reproducible workflow. The research here focuses on making this loop transparent, controllable, and efficient.

- **Designing LoopScript:** A core challenge is defining a declarative LoopScript language for orchestrating agentic workflows. What are the necessary abstractions to define task decomposition, specify parallel execution strategies, and set checkpoints for human review? Future work should investigate how languages from business process automation can be adapted and extended for this new context. Early platforms like FMArts, designed for general FMware orchestration, provide a conceptual starting point, but adapting these ideas to the unique demands of the SE domain remains an open challenge.
- **Human-in-the-Loop Control:** Agents will inevitably get stuck or explore suboptimal paths. Research is needed on mechanisms for seamless human intervention in a running agentic loop. How can a coach pause a workflow, provide a quick hint to redirect a stuck agent, or terminate unpromising branches without needing to restart the entire process from scratch?
- **Defining and Automating Evidence Packs:** The concept of a "Merge-Readiness Pack" requires formalization. What constitutes sufficient evidence of correctness, security, and quality? Research should focus on automatically generating these packs, which would bundle test results, static analysis reports, performance benchmarks, and formal verification proofs into a single, auditable artifact that can be drilled up and down.
- **Quantifying and Optimizing Feedback Signals:** Agents learn and iterate based on the feedback from their tools. A fascinating research direction is to quantify the "informational richness" of tool feedback. For example, a Rust compiler error is far more instructive than a simple segmentation fault. Research can explore how to systematically design and adapt development tools to provide maximally informative, structured feedback that accelerates an agent's search and reduces trial-and-error cycles.

7.3 AI Teammate Mentorship Engineering (ATME)

ATME treats the guidance and norms provided to an AI agent as first-class code. "Mentorship-as-code" ensures that team-specific best practices, architectural principles, and coding styles are codified, versioned, and applied consistently.

- **Developing the MentorScript Language:** A key research area is the design of **MentorScript**, a structured language for defining mentorship rules. What is the right level of abstraction? The language must be expressive enough to capture nuanced guidance (e.g., "prefer composition over inheritance in the services layer") yet simple enough for an entire team to contribute to and review. Research into rule engines and policy-as-code languages can inform its design.
- **Quality Assurance for Mentorship Rules:** If mentorship is code, it needs its own quality assurance. This opens a research avenue into building tools to lint, test, and formally verify MentorScript rules. How can we detect conflicting rules or ensure that a new rule does not cause unintended regressions in agent behavior?
- **Automated Inference of Mentorship Rules:** Can an agent learn to be a better teammate over time? A powerful research direction is developing techniques for agents to automatically infer new MentorScript rules from human feedback. For example, if a human repeatedly refactors an agent's code for better readability, the system could propose a new, generalizable rule for the human coach to approve and add to the MentorScript.
- **Efficient and Observable Rule Application:** As MentorScript grows, its application could become a performance bottleneck. Research is needed on optimizing rule engines for speed and providing full traceability for their decisions. A developer must be able to ask, "Why did the agent make this specific choice?" and receive an answer that points directly to the MentorScript rule(s) that were applied.
- **Peer Mentorship and Autonomous Rule Evolution:** While the current vision of ATME focuses on a human coach mentoring agents, a frontier research question is whether agents can mentor each other. This opens up inquiries into

agent-to-agent mentorship loops: Could a specialized "critic" agent observe a "coder" agent's work and automatically generate a new MentorScript rule to correct a recurring error or enforce a newly discovered best practice? Research in this area would need to explore mechanisms for inter-agent feedback, automated rule negotiation, and safeguards to prevent the propagation of incorrect mentorship, creating a truly self-improving multi-agent system.

- **Codifying Consultation Heuristics:** MentorScript can be extended to define heuristics for when an agent should seek guidance. Research can explore rules like, "If a function's cyclomatic complexity exceeds 20 during refactoring, generate a CRP for the 'Tech Lead' role." Furthermore, the resolutions provided by humans to CRPs create a rich dataset that can be mined to automatically infer and propose new, durable MentorScript rules.

7.4 Agentic Guidance Engineering (AGE)

AGE research focuses on optimizing the human side of the collaborative loop by augmenting the human's ability to provide guidance efficiently.

- **Rapid Contextual Onboarding for Deciders:** A key challenge is designing ACE interfaces that summarize the context within a CRP so a human who is not the task initiator can make an effective decision quickly. Research should explore progressive disclosure ("zoom-levels") and "what-if" toggles in the CRP interface.
- **Low-Cost Reversible Inputs:** The CRP and ACE must be designed to treat human guidance as a branch in a decision tree, not an irreversible command. Research is needed on interfaces that log guidance as a reversible event and on extending LoopScript to support low-cost counterfactual re-runs based on alternate guidance.
- **Cross-Human Handover and Routing:** Open questions remain on the policies for routing CRPs. This involves research into role-based, workload-aware, and expertise-based routing algorithms and defining protocols for stitching together decisions when the task initiator and decider are different people.

7.5 AI Teammate Lifecycle Engineering (ATLE)

ATLE focuses on transforming agents from stateless, single-task executors into persistent, long-term teammates. This requires giving them memory, enabling them to be proactive, and adapting coding best practices to their unique strengths.

- **Models for Persistent Agent Memory:** How do we build agents that remember? Research is needed on both internalizing SE experience into model weights (with methods from the continual learning field [11, 26]) and externally storing long-term memory in the ideal data structures, such as hybrid systems combining graph databases, vector stores, and decision logs. However, a core challenge is that simply accumulating history is not feasible. This ongoing memory growth will either overflow the model's finite context window or, more subtly, degrade performance by introducing irrelevant information that causes hallucinations. This necessitates research into memory compression and summarization [6, 9, 33, 37], but while this is a general problem for all long-lived agents, off-the-shelf solutions are likely insufficient for the SE domain. The structured, syntactically sensitive, and highly relational nature of software artifacts, from source code and build logs to commit histories, means that generic text summarization could discard critical details. Therefore, research must focus on specialized, structure-aware techniques, such as abstracting code into semantic representations or creating summaries of dependency changes, to ensure the agent's memory is not only compact but also preserves the high-fidelity semantic detail required for complex engineering tasks like debugging, refactoring, and maintaining architectural consistency.
- **Algorithms for Proactive Maintenance:** An agent with persistent context can become a proactive custodian of the codebase. This requires research into scheduling and prioritization algorithms for autonomous maintenance. During

idle compute cycles, an agent could scan for technical debt, identify documentation gaps, or propose code refactorings. The challenge is to ensure these proactive tasks provide high value without disrupting active development.

- **Formalizing the Economics of Agent-First Code:** The rise of agents forces a re-evaluation of long-held software engineering principles. Research should formally model the economic trade-offs of practices like Don't Repeat Yourself (DRY) in an agentic context. When is code duplication acceptable or even preferable if an agent can reliably update all instances? How does the compile-time safety of languages like Rust or TypeScript change the cost-benefit analysis of testing?

- **Designing for Agent-Oriented Operations:** As agents generate more production code, a new frontier emerges: agent-driven observability and maintenance. This requires research into designing runtime environments that are built for machine analysis. How do we create machine-parsable logs, structured traces, and self-describing metrics that enable an agent to autonomously diagnose issues, propose hotfixes, and perform safe rollbacks?

7.6 AI Teammate Infrastructure Engineering (ATIE)

ATIE is concerned with building the underlying infrastructure, the interfaces, compute fabrics, and tools that agents need to operate effectively and collaborate with humans.

- **The Post-IDE Human-Agent Interface:** As the traditional IDE becomes less central, a major research question arises: what is the "command center" for the human to coach, orchestrator and mentor teams of AI agents? Research must explore novel interfaces for specifying intent, visualizing, comparing, and splicing N-versioned solutions, orchestrating complex agent workflows, and providing structured mentorship, moving far beyond today's text-centric environments.

- **Distributed Compute Fabrics for Agents:** Multi-agent systems require a robust runtime. Research is needed on designing and extending distributed compute fabrics like Ray to specifically support agentic SE workloads. Crucially, beyond performance, these fabrics must provide isolated, hermetic, and secure execution environments to manage the blast radius of faulty agents while ensuring predictable, reproducible outcomes in a cost-effective manner. The declarative nature of LoopScript is the key enabler for advanced optimizations, as it transforms a series of unpredictable LLM calls into a structured, analyzable workflow. This allows for sophisticated scheduling and resource optimizations, as demonstrated by related work on SLA-aware and context-aware CodeLLM serving [29], providing a blueprint for the cost-effective, high-performance execution of agentic systems.

- **Creating an Agent-Native Toolchain:** Today's agents often resort to basic tools like grep because the human-centric toolchain is suboptimal for them. A significant research effort is needed to reimagine the entire SE toolchain for an agent-first world. This includes creating agent-native protocols for tool interaction, developing new tools like hyper-debuggers and structural editors, and exploring self-improving systems where agents learn to optimize their own tool usage over time.

7.7 The Human Differentiator: A Call to Reimagine SE Education

The SE field is defined by its four pillars: actors, processes, tools, and artifacts. While the SASE vision provides a structured approach for the latter three, we must not forget that the **actor** (the human engineer) remains the most critical variable in the equation. The 100x and 1,000x productivity gains emerging today are not the result of magical tools, but of skilled individuals who have mastered the art of working with agents. They are the proof that even with today's nascent technology, the human's ability to specify intent, curate context, and provide strategic oversight is the ultimate differentiator.

Even if the entire SASE framework were realized tomorrow, it would not automatically create a generation of 1000x engineers. It would provide the scaffolding, but the ability to use that scaffolding effectively (to excel at Briefing Engineering, to design elegant LoopScripts, to codify insightful mentorship) will still separate the exceptional from the average. The human is not being automated away; their role is being elevated from a crafter of code to a conductor of agents.

This educational imperative aligns directly with forward-thinking national strategies, such as the "Human-AI Co-Thinking" vision proposed for Swiss education [28]. That framework advocates making co-thinking a core competency taught from elementary school onward, where students learn to amplify their intellect while maintaining their critical role as verifiers. The goal is to cultivate a "sixth sense" for recognizing the cognitive dissonances of AI, preparing the next generation not just to use AI, but to think with it in a structured, responsible, and effective way. The SASE vision for SE education can thus be seen as a domain-specific application of this broader pedagogical shift.

This elevates a critical challenge that extends beyond research: we must fundamentally rethink SE education. Current curricula are largely designed to train students to *be* the agent (to write code, create tests, and perform low-level implementation). In the agentic era, we must instead train them to *manage* fleets of agents. This requires a profound pedagogical shift away from pure implementation and toward strategic skills: system-level thinking, architectural reasoning, rigorous specification, and the art of mentorship-as-code. This is not a call to simply add a "prompt engineering" module to an existing course; it is a call for a deep and holistic reimagining of what it means to educate the next generation of software engineers. This challenge is further compounded by the concurrent shift in the nature of software itself towards "FMware" (FM-powered applications), a separate but equally transformative trend that also demands its own educational rethinking, though we do not address that here.

8 Related Efforts to Agentic Software Engineering

The SASE vision builds on a growing body of methods and tools that structure human–AI software work. Broadly, prior art clusters into two streams: (1) iterative, single-agent workflows that boost an individual developer's throughput, and (2) multi-agent frameworks that mirror human agile teams. Understanding these lines clarifies both SASE's alignments and its distinct contribution.

8.1 Iterative and Prompt-Driven Workflows

PDAR with Product Requirement Prompts (PRPs): The Plan–Do–Assess–Review loop formalizes a single task's lifecycle: a human and AI plan, a dev-agent implements, an agent self-assesses, and a human reviews. PRPs act as the "minimum viable packet" capturing goals, justification, acceptance criteria, and curated context. This aligns with SASE's insistence on structured, testable intent. However, PDAR is scoped to one-off execution; it does not by itself establish durable mentorship, agent lifecycle learning, or cross-task traceability that SASE treats as first-class.

SuperClaude and CLI toolkits: Command-line workflows that template PRPs and common loops raise consistency and convenience for solo developers. They embody SASE's artifact-centric stance but stop short of a team-level methodology. They neither codify mentorship as versioned rules nor address agent memory, observability, or merge-readiness evidence as explicit deliverables.

8.2 Multi-Agent and Agile-Inspired Frameworks

BMAD (Breakthrough Method for Agile AI-Driven Development): BMAD organizes agents into agile roles (e.g., Product Owner, Architect, Developer, Tester). Up-front agentic planning yields PRDs and designs; a Scrum-like

shard step creates "story files" with focused context; specialized agents execute in parallel. BMAD's strengths (role specialization, task sharding, and high parallelism) map well to SASE's N-version programming and orchestration. SASE goes further by (i) converting review feedback into persistent MentorScript rules (mentorship-as-code), and (ii) specifying the environments and disciplines (ACE for human coaching and orchestrating, while AEE for agent execution) plus ATLE/ATIE for memory, lifecycle, and agent-native tooling.

8.3 How SASE Aligns and Differentiates

Alignment: SASE adopts the best of these efforts like PRP-like briefs for intent formalization, PDAR-style iterative loops, and BMAD-like multi-agent parallelism.

Differentiation: SASE elevates these pieces into a holistic engineering methodology with four distinguishing features:

(1) **Mentorship-as-Code (ATME).** Review guidance becomes a version-controlled, testable MentorScript, enabling cumulative, auditable improvement across tasks and teams.

(2) **Dual Workbenches.** The Agent Command Environment (ACE) optimizes human cognition for specification, orchestration, and evidence-based review; the Agent Execution Environment (AEE) optimizes for agent strengths (e.g., massive parallelism).

(3) **Merge-Readiness as the Target Artifact.** The loop's output is a Merge-Readiness Pack which is a progressive-disclosure bundle proving functional completeness, sound verification, SE hygiene, rationale, and full auditability.

(4) **Consultability as a First-Class Artifact.** SASE introduces the Consultation Request Pack (CRP) as a structured and auditable artifact for agent-initiated human consultation. This elevates humans to callable experts and enables traceable cross-role handovers, shifting the paradigm from solo agentic coding to team-based Agentic Software Engineering.

(5) **Lifecycle & Infrastructure (ATLE & ATIE).** Agents become persistent teammates with memory, observability, and secure, hermetic execution, shifting from stateless contractors to evolving collaborators.

9 Conclusion

The transition to the Agentic Software Engineering (SE 3.0) era represents a fundamental inflection point for the SE field, demanding more than incremental adjustments to existing practices. In this paper, we presented the Structured Agentic Software Engineering (SASE) vision, a conceptual framework designed to impose structure, predictability, and trustworthiness onto the emergent practice of agentic SE. Our core contribution is the introduction of a structured duality (SE for Humans and SE for Agents) that reimagines the actors, processes, and tools of our field through the lens of humans, agents and their collaborations. This vision is operationalized through dedicated environments (ACE and AEE) and a suite of version-controlled artifacts (BriefingScript, LoopScript, MentorScript, CRP and MRP) that transform the human role from a direct implementer to a strategic "Agent Coach and Orchestrator," a transformation with profound implications for SE education. SASE is offered not as a definitive, final solution, but as a conceptual scaffold intended to catalyze a necessary and urgent dialogue within the SE community. By embracing this structured, dualistic approach, we can move beyond impressive but brittle demonstrations and begin the collective work of building the disciplined, scalable, and robust engineering foundations required to realize the full potential of agentic SE. The future of the SE field will be defined not by the speed of our agents alone, but by our ability to mentor, orchestrate, and trust them as true engineering partners.

References

[1] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. *SWE-Bench+: Enhanced Coding Benchmark for LLMs.* arXiv:2410.06992 [cs.SE] https://arxiv.org/abs/2410.06992

[2] Sayed Mahbub Hasan Amiri, Md. Mainul Islam, Mohammad Shakhawat Hossen, Sayed Majhab Hasan Amiri, Mohammad Shawkat Ali Mamun, Sk. Humaun Kabir, and Naznin Akter. 2025. *Hear Your Code Fail, Voice-Assisted Debugging for Python.* arXiv:2507.15007 [cs.PL] https://arxiv.org/abs/2507.15007

[3] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. *SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents.* arXiv:2505.20411 [cs.SE] https://arxiv.org/abs/2505.20411

[4] S.S. Brilliant, J.C. Knight, and N.G. Leveson. 1989. The consistent comparison problem in N-version software. *IEEE Transactions on Software Engineering* 15, 11 (1989), 1481–1485. doi:10.1109/32.41339

[5] Liming Chen and A. Avizienis. 1995. N-version programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.* IEEE Computer Society, Los Alamitos, CA, USA, 113. doi:10.1109/FTCSH.1995.532621

[6] Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. 2023. Adapting Language Models to Compress Contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing.* 3829–3846.

[7] Filipe Roseiro Cogo, Gopi Krishnan Rajbahadur, Dayi Lin, and Ahmed E. Hassan. 2024. A Tutorial on Software Engineering for FMware. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024).* Association for Computing Machinery, New York, NY, USA, 710–712. doi:10.1145/3663529.3663820

[8] Ximing Dong, Shaowei Wang, Dayi Lin, Gopi Krishnan Rajbahadur, Boquan Zhou, Shichao Liu, and Ahmed E Hassan. 2024. Promptexp: Multi-granularity prompt explanation of large language models. *arXiv preprint arXiv:2410.13073* (2024).

[9] Weizhi Fei, Xueyan Niu, Pingyi Zhou, Lu Hou, Bo Bai, Lei Deng, and Wei Han. 2024. Extending Context Window of Large Language Models via Semantic Compression. In *Findings of the Association for Computational Linguistics ACL 2024.* 5169–5181.

[10] Kajsa Gullberg, Victoria Johansson, and Roger Johansson. 2024. In Scriptura Veritas? Exploring Measures for Identifying Increased Cognitive Load in Speaking and Writing. *Languages* 9, 3 (2024). doi:10.3390/languages9030085

[11] Haiyang Guo, Fanhu Zeng, Fei Zhu, Jiayi Wang, Xukai Wang, Jingang Zhou, Hongbo Zhao, Wenzhuo Liu, Shijie Ma, Da-Han Wang, et al. 2025. A comprehensive survey on continual learning in generative models. *arXiv preprint arXiv:2506.13045* (2025).

[12] Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan. 2025. *Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers.* arXiv:2506.13538 [cs.SE] https://arxiv.org/abs/2506.13538

[13] Ahmed E. Hassan. 2008. The road ahead for Mining Software Repositories. In *2008 Frontiers of Software Maintenance.* 48–57. doi:10.1109/FOSM.2008.4659248

[14] Ahmed E. Hassan, Gustavo A. Oliva, Dayi Lin, Boyuan Chen, and Zhen Ming Jiang. 2024. *Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap.* arXiv:2410.06107 [cs.SE] https://arxiv.org/abs/2410.06107

[15] Ahmed E. Hassan and Tao Xie. 2010. Mining software engineering data. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (Cape Town, South Africa) *(ICSE '10).* Association for Computing Machinery, New York, NY, USA, 503–504. doi:10.1145/1810295.1810451

[16] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[17] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations.* https://openreview.net/forum?id=VTF8yNQM66

[18] Hao Li, Cor-Paul Bezemer, and Ahmed E. Hassan. 2025. Software Engineering and Foundation Models: Insights from Industry Blogs Using a Jury of Foundation Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* 307–318. doi:10.1109/ICSE-SEIP66354.2025.00033

[19] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. *The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering.* arXiv:2507.15003 [cs.SE] https://arxiv.org/abs/2507.15003

[20] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zahl, and Klaus Wehrle. 2017. Floating-point symbolic execution: A case study in N-version programming. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 601–612. doi:10.1109/ASE.2017.8115670

[21] Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. 2025. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. *arXiv preprint arXiv:2502.20127* (2025).

[22] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.* 11, 3 (July 2002), 309–346. doi:10.1145/567793.567795

[23] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting*

of the Association for Computational Linguistics (Volume 1: Long Papers), Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15174–15186. doi:10.18653/v1/2024.acl-long.810

[24] Benjamin Rombaut, Sogol Masoumzadeh, Kirill Vasilevski, Dayi Lin, and Ahmed E. Hassan. 2025. *Watson: A Cognitive Observability Framework for the Reasoning of LLM-Powered Agents.* arXiv:2411.03455 [cs.AI] https://arxiv.org/abs/2411.03455

[25] Alex Serban, Erik Poll, and Joost Visser. 2020. A standard driven software architecture for fully autonomous vehicles. *Journal of Automotive Software Engineering* 1, 1 (2020), 20–33.

[26] Haizhou Shi, Zihao Xu, Hengyi Wang, Weiyi Qin, Wenyuan Wang, Yibin Wang, Zifeng Wang, Sayna Ebrahimi, and Hao Wang. 2024. Continual learning of large language models: A comprehensive survey. *Comput. Surveys* (2024).

[27] Richard Sutton. 2019. The bitter lesson. *Incomplete Ideas (blog)* 13, 1 (2019), 38.

[28] AI Swiss. 2025. *For a Swiss education transformed by AI.* White paper. https://a-i.swiss/resources

[29] Kishanthan Thangarajah, Boyuan Chen, Shi Chang, and Ahmed E. Hassan. 2025. *Context-Aware CodeLLM Eviction for AI-assisted Coding.* arXiv:2506.18796 [cs.SE] https://arxiv.org/abs/2506.18796

[30] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations*.

[31] You Wang, Michael Pradel, and Zhongxin Liu. 2025. *Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study.* arXiv:2503.15223 [cs.SE] https://arxiv.org/abs/2503.15223

[32] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449* (2025).

[33] David Wingate, Mohammad Shoeybi, and Taylor Sorensen. 2022. Prompt Compression and Contrastive Conditioning for Controllability and Toxicity Reduction in Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2022*. 5621–5634.

[34] T. Winters, T. Manshreck, and H. Wright. 2020. *Software Engineering at Google: Lessons Learned from Programming Over Time.* O'Reilly Media.

[35] Jie JW Wu and Fatemeh H. Fard. 2025. HumanEvalComm: Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agents. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 189 (Aug. 2025), 42 pages. doi:10.1145/3715109

[36] Wendong Xu, Jing Xiong, Chenyang Zhao, Qiujiang Chen, Haoran Wang, Hui Shen, Zhongwei Wan, Jianbo Dai, Taiqiang Wu, He Xiao, Chaofan Tao, Z. Morley Mao, Ying Sheng, Zhijiang Guo, Hongxia Yang, Bei Yu, Lingpeng Kong, Quanquan Gu, and Ngai Wong. 2025. *SwingArena: Competitive Programming Arena for Long-context GitHub Issue Solving.* arXiv:2505.23932 [cs.CL] https://arxiv.org/abs/2505.23932

[37] Zhaozhuo Xu, Zirui Liu, Beidi Chen, Shaochen Zhong, Yuxin Tang, Jue Wang, Kaixiong Zhou, Xia Hu, and Anshumali Shrivastava. 2024. Soft Prompt Recovers Compressed LLMs, Transferably. In *International Conference on Machine Learning*. PMLR, 55186–55203.

[38] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025. Which Agent Causes Task Failures and When? On Automated Failure Attribution of LLM Multi-Agent Systems. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=GazlTYxZss