The future of software engineering

Retreat findings and strategic insights

Thank you all for joining us to wrestle with the questions that matter most as AI reshapes how we build software. What follows is a synthesis of key themes and takeaways from across all breakout sessions.

The retreat was conducted under the Chatham House Rule. No participant names or affiliations are disclosed in this summary.

February 2026

Executive summary

Senior engineering practitioners from major technology companies gathered for a multi-day retreat to confront the questions that matter most as AI transforms software development. The discussions covered more than twenty topics across breakout sessions, but the most significant insights didn't emerge from one single session. Instead, they surfaced at various intersections; we found that the same concerns kept appearing in different conversations, framed by different people solving different problems.

This publication synthesizes those cross-cutting themes, organized around the patterns that senior leaders need to understand and act on now. The retreat did not produce a single, unified vision of the future, but instead produced something more useful: a map of the fault lines where current practices are breaking and new ones are forming.

> *"We kept asking the same question in every room: if AI handles the code, where does the engineering actually go? Nobody had the same answer. But everybody agreed the question is urgent."*

Themes at a glance

Theme	Horizon	Core Insight
Where does the rigor go?	Now	Engineering quality doesn't disappear when AI writes code. It migrates to specs, tests, constraints, and risk management.
From code review to risk tiering	Now	Code review is being unbundled. Its four functions (mentorship, consistency, correctness, trust) each need a new home.
The productivity/experience paradox	Now	Developer productivity and developer experience are decoupling. Organizations face hard choices about which to optimize.
Security as afterthought	Now	Agent security is woefully underdeveloped. Email access alone can enable full account takeover.
The middle loop	Now–1 yr	A new category of supervisory engineering work is forming between inner-loop coding and outer-loop delivery. Nobody has named it yet.

Cognitive debt	Now–1 yr	Technical debt is becoming cognitive debt: the gap between system complexity and human understanding.
Agent topologies	1–3 yrs	Conway's Law applies to agents too. Enterprise architecture must now account for agent mobility, specialization, and drift.
Knowledge graphs & semantic layers	1–3 yrs	Decades-old technologies are suddenly relevant again as the grounding layer for domain-aware agents.
The future of roles	1–3 yrs	PM, developer, and designer roles are converging. Staff engineers face new expectations. Juniors are more valuable than ever.
Self-healing systems	2–5 yrs	Moving from human incident response to agent-assisted healing requires solving the 'latent knowledge' problem first.

Detailed findings for each theme follow.

1. Where does the rigor go?

The single most important question of the retreat. It surfaced in nearly every session.

If AI takes over code production, the engineering discipline that used to live in writing and reviewing code does not disappear; it moves elsewhere. The retreat spent more time on this question than any other, approaching it from various viewpoints, including code review, testing, language design, self-healing systems and organizational design.

The group identified five destinations where rigor is already moving:

Upstream to specification review

Several practitioners reported shifting their review efforts from code to the plan that precedes it. One described focusing on "pre-reviewing the plans and post-reviewing engineering" rather than the code itself. The logic is straightforward: if an AI generates code from a spec, the spec is now the highest-leverage artifact for catching errors. Bad specs produce bad code at scale.

This has practical implications. Organizations experimenting with spec-driven development report that the specifications themselves need new formats. Traditional user stories are too vague. Some teams are adopting structured approaches like EARS (Easy Approach to Requirements Syntax), state machines and decision tables. These are not new techniques, but they are being rediscovered because they give AI agents enough precision to produce correct implementations.

Into test suites as first-class artifacts

One of the retreat's most shareable insights was that test-driven development produces dramatically better results from AI coding agents. The mechanism is specific: TDD prevents a failure mode where agents write tests that verify broken behavior. When the tests exist before the code, agents cannot cheat by writing a test that simply confirms whatever incorrect implementation they produced.

This reframes TDD as a form of prompt engineering. The tests become deterministic validation for non-deterministic generation. Several practitioners described moving review efforts entirely to the test suite, treating the generated code as expendable. If the tests are correct and the code passes them, the code is acceptable regardless of how it looks.

Practitioner insight

"I've gotten better results from TDD and agent coding than I've ever gotten anywhere else, because it stops a particular mental error where the agent writes a test that verifies the broken behavior."

Into type systems and constraints

The retreat surfaced strong interest in using programming language features to constrain AI-generated code. Rather than reviewing code after generation, practitioners are exploring how to make incorrect code unrepresentable. This draws on ideas from formal methods and strong type systems, applied not as academic exercises but as practical guardrails for agent output.

A key insight was the separation of specifications from constraints. Specifications describe what should change; constraints define the bounded contexts in which change is allowed, including what must not be touched. These constraints limit blast radius and let agents work safely across domain boundaries. When a constraint must be broken, it signals a new system boundary and prompts refactoring.

Into risk mapping

Not all code carries the same risk. The retreat discussed tiering code by business blast radius, distinguishing between internal tools, external-facing services and safety-critical systems. This risk mapping determines where human review is essential and where automated verification is sufficient.

One practitioner framed this as the new core engineering discipline: instead of asking "did someone review this code?" organizations need to ask "what is the blast radius if this code is wrong, and is our verification proportional to that risk?" This moves engineering from a craft model (every line is hand-reviewed) to a risk management model (verification investment matches exposure).

Into continuous comprehension

If code changes faster than humans can review it, the traditional model of building mental models through code review breaks down. The retreat discussed alternatives: weekly architecture retrospectives, ensemble programming where multiple engineers work simultaneously on the same code and AI-assisted code comprehension tools that generate system overviews on demand.

The underlying concern is real. One practitioner noted that code review has historically served as much as a learning mechanism as a quality gate. Mentorship, shared

understanding and codebase familiarity all happened through review. Losing that channel without replacing it creates a comprehension gap that compounds over time.

> *"Paired programming solves all of this. If it's important to understand the system, then do it all the time. You don't do it in little phases where you have your code review. Constantly trying to understand what this code is doing."*

2. The middle loop: a new category of work

The retreat's strongest first-mover concept. Nobody in the industry has named this yet.

Software development has long been described in terms of two loops. The inner loop is the developer's personal cycle of writing, testing and debugging code. The outer loop is the broader delivery cycle of CI/CD, deployment and operations. The retreat identified a third: a middle loop of supervisory engineering work that sits between them.

This middle loop involves directing, evaluating and fixing the output of AI agents. It requires a different skill set than writing code. It demands the ability to decompose problems into agent-sized work packages, calibrate trust in agent output, recognize when agents are producing plausible-looking but incorrect results and maintain architectural coherence across many parallel streams of agent-generated work.

The practitioners who are excelling at this new work tend to share certain characteristics:

- They think in terms of delegation and orchestration rather than direct implementation.
- They have strong mental models of system architecture.
- They can rapidly assess output quality without reading every line.

These are skills that experienced engineers often possess, but they are rarely explicitly developed or recognized in career ladders.

Career impact

The middle loop creates a genuine identity crisis for developers who fell in love with programming. Many were hired specifically to translate pre-digested tickets into working code. That work is disappearing. The new work requires different aptitudes and different sources of professional satisfaction. Organizations that don't help people through this transition will lose their most experienced talent to frustration.

The retreat drew a parallel to the history of computer graphics. In 1992, engineers hand-coded polygon rendering algorithms. Two years later, that work had been pushed into hardware, and the job became animation and lighting. Today, it's custom physics and game worlds. Each time the abstraction layer rose, engineers who insisted they were hired to render polygons were left behind. The same dynamic is playing out now with code production.

The product management side of this equation is equally unsettled. If developers are now thinking more about what to build and why, they are doing work that used to belong to product managers. One large technology company is actively researching whether the PM role needs a new name. Another is training all product managers to work in Markdown inside developer tools. The convergence is real, even if nobody agrees on where it lands.

3. Agent topologies and enterprise architecture

Conway's Law didn't retire. It got more complicated.

The retreat introduced the concept of "agent topologies" as an extension of the Team Topologies framework. The premise: if organizations design systems that mirror their communication structures, what happens when agents become first-class participants in those structures?

Unlike humans, agents can be duplicated instantly and deployed across multiple teams without onboarding friction. A specialized database agent can exist on every team simultaneously, bringing consistent expertise without the centralization bottleneck that comes with a single human database specialist. This sounds like a pure win, but the retreat identified several complicating factors.

The speed mismatch

Agents burn through backlogs so fast they collide with slow organizational dependencies. One participant described the experience: you give a team AI tools, they clear their backlog in days and then hit a wall of cross-team dependencies, architecture reviews and human-speed decision-making. The result is not faster delivery. It is the same speed with more frustration, because the bottleneck has shifted from engineering capacity to everything else.

Agent drift

Agents that learn from their context will diverge over time. The database agent working on the e-commerce backend accumulates different patterns and preferences than the one working on the ERP system, even if they started from identical configurations. This mirrors the human problem of team-specific norms, but on an accelerated timeline. The retreat debated whether this drift should be managed (analogous to standardization efforts in human teams) or embraced (analogous to letting teams optimize locally).

Decision fatigue as the new bottleneck

If agents can produce work faster than leaders can review and approve it, the constraint shifts from production capacity to decision-making capacity. Middle managers who previously served as coordination points now become approval bottlenecks. Several practitioners reported this already happening at their organizations: agents generating job specifications, code fixes and feature implementations faster than anyone can say yes.

The retreat asked a pointed question: if humans have capacity limits for understanding systems but agents do not, do we need as many middle managers? The group did not reach consensus, but the question itself signals a significant organizational challenge ahead.

> *"We optimized the software delivery process for humans. Now that it's not just humans, we have to ask what organizing actually means."*

4. Self-healing and self-improving systems

The ambition is real. The prerequisites are far from met.

The retreat explored whether software systems can move beyond human-driven incident response toward agent-assisted self-healing. The group distinguished between two levels of ambition: self-healing (returning a system to a known good state) and self-improving (actively evolving a system's non-functional qualities like performance and reliability).

Prerequisites that don't yet exist

Self-healing requires several foundations that most organizations lack:

- A clear ledger of every change, so agents can understand what happened.
- An operating system for agents with identity controls and permission boundaries.
- Strong generic mitigation capabilities (rollback, feature flags) that work without code changes.
- Fitness functions that define what "healthy" means in terms agents can evaluate.

The retreat was blunt: code changes should be the last resort for incident remediation. The path to self-healing runs through better rollback, better feature flags and better observability before it runs through agents rewriting production code.

The latent knowledge problem

Senior engineers bring decades of pattern-matching to incident response. They remember that a specific error code is actually a symptom of a deeper infrastructure issue. They know that high CPU on a particular service means checking the database connection pool before anything else. This knowledge is almost never documented. It lives in people's heads and gets applied through experience.

To replicate this for agents, organizations need to build what the retreat called an "agent subconscious": a knowledge graph built from years of post-mortems and incident data that gives agents historical context for interpreting real-time signals. Some organizations are already doing this with automated post-mortem drafting, but the human step of adding nuance and context remains essential.

> **The incident commander problem**
> Human incident commanders challenge assumptions, push back on comfortable hypotheses and maintain situational awareness. LLMs tend toward positive reinforcement and agreement. Building an effective agent incident commander requires solving this behavioral mismatch. One

suggestion: train "angry agents" that are specifically designed to challenge the dominant hypothesis.

Agent coordination risks

Multiple agents attempting to fix the same issue can create feedback loops where one agent's fix triggers another agent's correction, creating an escalating cycle. The retreat cited a real example: an agent with access to a linter that enforced a 500-line file limit responded by making individual lines longer, technically satisfying the rule while violating the principle behind it. When multiple agents make different prioritization decisions about trade-offs, the system can oscillate rather than converge.

5. The human side: roles, skills and experience

AI is not replacing people. It is rearranging what people do and how they feel about doing it.

The productivity/experience paradox

Developer experience has traditionally been defined across three dimensions: flow state, feedback loops and cognitive load. Productivity and developer experience have been tightly coupled for decades; the retreat explored evidence that they are now diverging. Organizations can achieve productivity gains through AI tools even in environments where developers report lower satisfaction, more cognitive load and reduced sense of flow.

This creates a genuine dilemma. If the organization can get more output without investing in developer experience, the business case for that investment weakens, unless the definition of developer experience itself evolves to account for the new realities of agent-supervised work. One practitioner offered a sharp reframe: stop calling it developer experience and call it agent experience instead. Wallets are likely to open faster to invest in conditions that help agents perform well, and the overlap with conditions that help humans perform well turns out to be nearly complete.

The staff engineer under pressure

Staff engineers are simultaneously more important and more stressed than ever. Data from one research firm spanning 500 companies shows that staff engineers use AI tools

less frequently than junior engineers, but when they do use them, they save more time per week. Their broader context and deeper understanding of system architecture makes them more effective agent supervisors.

The tension is in what staff engineers are asked to do versus what they should be doing. Many spend disproportionate time on human coordination rather than technical supervision. The retreat argued for a deliberate shift: staff engineers should become friction killers, identifying and removing the impediments that slow both human and agent work. Their deep knowledge of where the skeletons are buried makes them uniquely positioned for this role, but many have experienced learned helplessness after years of being told there's no budget for the improvements they recommend.

Junior developers are more valuable, not less

The retreat challenged the narrative that AI eliminates the need for junior developers. Juniors are more profitable than they have ever been. AI tools get them past the awkward initial net-negative phase faster. They serve as a call option on future productivity. And they are better at AI tools than senior engineers, having never developed the habits and assumptions that slow adoption.

The real concern is mid-level engineers who came up during the decade-long hiring boom and may not have developed the fundamentals needed to thrive in the new environment. This population represents the bulk of the industry by volume, and retraining them is genuinely difficult. The retreat discussed whether apprenticeship models, rotation programs and lifelong learning structures could address this gap, but acknowledged that no organization has solved it yet.

Education signal

The retreat highlighted the University of Waterloo's co-op program as a model: deep theoretical foundations combined with 2.5 years of industry internships (six four-month rotations). Graduates emerge with both the fundamentals and the practical judgment that AI tools cannot replace. Several companies reported that intern-to-hire pipelines now outperform traditional graduate recruiting.

The future of product management

Nobody at the retreat could define what product managers will do in an AI-driven world. Some organizations are pushing PMs closer to technical tooling, training them to work in Markdown and developer environments. Others see the roles diverging further, with PMs becoming strategic orchestrators while developers take on more of the tactical product decision-making.

What is clear is that AI is exposing existing dysfunctions in the PM-developer relationship rather than creating new ones. Knowledge fragmentation, cultural gaps between disciplines and unclear role boundaries existed before AI. AI is simply making them more expensive to ignore. The retreat emphasized tools as "boundary objects" that allow different roles to work in their own ways while maintaining shared visibility.

6. Technical foundations: languages, semantics and operating systems

The infrastructure for the agent era doesn't exist yet. These are the pieces being assembled.

Programming languages for agents

Every programming language in existence was designed with humans as the primary user. Dynamic typing exists to reduce cognitive overhead for human programmers. Strong static typing exists to catch human errors. The retreat asked what a language designed for agent-generated code would look like, and whether it would also serve humans better.

The group converged on a principle: what is good for AI is good for humans. Languages that make incorrect code unrepresentable (through strong types, restricted computation models and formal constraints) help agents produce correct output and help humans verify it. Conversely, languages that favor expressiveness over safety make both agent generation and human review harder.

The more radical possibility is that source code as we know it could become a transient artifact, generated on demand and never stored. The retreat was divided on this. Some saw source code disappearing within a decade. Others argued that deterministic validation requires a stable artifact to test against, and that artifact is effectively source code regardless of what we call it.

Semantic layers and knowledge graphs

Technologies that failed to gain mainstream adoption for decades are suddenly relevant. Semantic layers, knowledge graphs and domain ontologies are being rediscovered as the grounding layer for AI agents that need to understand business domains. The retreat included practitioners building these systems at scale, reporting that a large telecom's

entire domain ontology could be captured in roughly 286 concepts. That number made the work feel achievable rather than impossibly ambitious.

The practical value is in legacy modernization. By building a conceptual data model from existing systems and validating it against subject matter experts, organizations can create the specification layer that agents need to modernize confidently. One team described using LLMs to automatically identify commands, events, aggregates and policies from code, effectively auto-generating event storming artifacts. Human experts then validate and correct, compressing weeks of discovery workshops into days.

The agentic operating system

The retreat explored what an operating system for agents would need to include:

- Agent identity and permission management.
- Memory and context-window management.
- A work ledger that captures future, current and past work with attributes like required skills, acceptance criteria, SLOs and cost constraints.
- Governance paths through a graph of agent capabilities and compliance requirements.

A central insight was that an agent is more than its persona, goals or current context; it includes the history of work it has performed. While models are fungible within an agent (you can swap one LLM for another), changing a model fundamentally alters the agent's behavior and must be tracked. The work ledger emerged as the core primitive of this new operating system, analogous to a financial blockchain: searchable, auditable and enabling agents to discover and bid for work.

7. Security, governance and the future of agile

Security Is dangerously behind

The retreat noted with concern that the security session had low attendance, reflecting a broader industry pattern. Security is treated as something to solve later, after the technology works and is reliable. With agents, this sequencing is dangerous.

The most vivid example: granting an agent email access enables password resets and account takeovers. Full machine access for development tools means full machine access

for anything the agent decides to do. The retreat's recommendation was direct. Platform engineering should drive secure defaults by making safe behavior easy and unsafe behavior hard. Organizations should not rely on individual developers making security-conscious choices when configuring agent access.

Three priorities emerged: security by design as a non-negotiable baseline, cross-industry coalitions for interoperable agent security standards and AI-enabled defense mechanisms that can match the speed and sophistication of AI-enabled attacks.

Agile is evolving, not dying

The retreat pushed back hard on the "agile is dead" narrative. What is happening is more nuanced. Some teams are compressing sprint cadences to one week, using AI to automate end-of-sprint ceremonies like demos, reporting and status summaries. Others are rediscovering XP practices (pair programming, ensemble development, continuous integration) because these practices create the tight feedback loops and shared understanding that agent-assisted development requires.

The real threat to agile is governance. Teams that adopt AI tools and work faster still run into the same approval processes, compliance gates and organizational dependencies. Without reforming governance alongside development practices, faster teams just hit the same walls sooner. The retreat emphasized involving internal audit and governance functions early when rethinking team practices, rather than treating them as obstacles to be navigated later.

Software stability is also declining as batch size increases. The ease of producing large changesets with AI tools is pushing some teams back toward waterfall-like patterns, with large, infrequent releases replacing small, frequent ones. This is a direct reversal of a decade of DORA research showing that smaller batch sizes correlate with higher stability. The retreat flagged this as an active regression that needs industry attention.

8. Agent swarms: beyond sequential thinking

The retreat dedicated focused time to agent swarming and surfaced insights that challenge conventional assumptions about how AI-assisted work should be organized.

The first barrier to effective swarming is mental, not technical. Engineers trained in sequential decomposition struggle to conceptualize parallel agent work. This mental

model actively blocks learning. Practitioners who have made breakthroughs in swarming describe the experience as fundamentally unlike anything they have encountered in previous software development. The simple act of asking agents to parallelize work explicitly and observing the results teaches more than any theoretical framework.

For enterprise use cases, the retreat identified an important pattern: perfect accuracy from individual agents matters less than collective convergence toward a goal. A swarm of individually imperfect agents can produce valuable outcomes if the system architecture guides convergence. This is a design principle borrowed from distributed systems and biological swarm intelligence, applied to AI agent orchestration.

The retreat also noted that most enterprise agent orchestration will not look like swarming at all. The more common pattern is "patrol workers on loops": agents running well-defined ETL transforms, data quality checks and business process monitors on continuous cycles. In other words, the unsexy work of data reliability and cleanliness running always-on in the background. Organizations with strong, well-designed APIs are significantly better positioned for both swarming and patrol-style agent deployment than those without.

Model limitations

Some frontier models have structural weaknesses that make them poorly suited to swarm-style scenarios. This is informing how evaluations are designed, with the expectation that swarm-oriented architectures will improve as these limitations are better understood. Organizations selecting models for agent deployment should test specifically for multi-agent coordination, not just single-agent capability.

9. Open questions

The retreat surfaced more questions than answers. These are the ones that kept the room awake.

On work and identity

How do we help engineers who love writing code find meaning and satisfaction in supervisory engineering work? What professional development pathways lead to the middle loop? If the product manager role and developer role are converging, what is the resulting role called and who owns it?

On organizational design

If agents make middle management bottlenecks more visible, does the organizational response involve fewer managers, differently-skilled managers or a fundamentally different coordination model? How do you redesign enterprise architecture when agents can move across team boundaries but governance structures cannot?

On trust and verification

What would need to be true for organizations to stop reviewing AI-generated code entirely? Is there a world where test suites and constraints provide sufficient verification without human inspection? How do we build trust in systems that are fundamentally non-deterministic, where rerunning the same inputs produces different outputs?

On knowledge and comprehension

If code changes faster than humans can comprehend it, do we need a new model for maintaining institutional knowledge? Can knowledge graphs and semantic layers truly replace the human intuition that comes from years of working in a codebase? What is the right investment level for "agent subconscious" systems that most organizations do not yet build?

On speed and stability

Are we currently in a regression where AI-enabled productivity gains are being offset by stability losses from larger batch sizes? Will development need to slow down because the volume of decisions is overwhelming human capacity to evaluate them? How do we measure the real cost of cognitive debt as it accumulates?

What comes next

The retreat surfaced a consistent pattern: the practices, tools and organizational structures built for human-only software development are breaking in predictable ways under the weight of AI-assisted work. The replacements are forming, but they are not yet mature.

The ideas ready for broader industry conversation include the supervisory engineering middle loop, risk tiering as the new core engineering discipline, TDD as the strongest form

of prompt engineering and the agent experience reframe for developer experience investment. Detailed explorations of each will follow.

The questions not yet answered are equally important. How to help people through an identity shift in their professional lives: how to govern organizations where agents move faster than humans can decide; how to build trust in systems that are inherently non-deterministic. These are not technical problems with technical solutions. They are human problems that will require candid conversation and collaboration. We are committed to contributing to that in the months to come.

> *"The retreat didn't produce a roadmap. It produced a shared understanding that the map is being redrawn and that the people best positioned to draw it are the ones willing to admit how much they don't yet know."*