

How Developers Collaborate with Agentic AI.

A Field Study at Techflouu

Seven developers, interviewed over two weeks in August 2026, on how engineers **steer agents through context and intent**, **calibrate trust around reversibility**, and **keep ownership of code they no longer write by hand**.

AUTHOR

Anggi Maulinda

PUBLISHED

14 October 2026

KEY TAKEAWAYS

Point, don't prescribe: Developers supply the relevant context and withhold instruction on how to implement. Over-specifying narrows what the model will consider, and over-supplying dilutes what it attends to.

Delegation relocates effort: Execution is largely automatic now, so effort concentrates in specification, control, and review.

Review is the new bottleneck: Generation capacity has risen far faster than the capacity to review its output. Productivity can no longer be measured by generated code, as code generation is no longer scarce.

Developers control agents on irreversibility: The hard limits (production database, deployment) are drawn at what cannot be undone.

A rule an agent has loaded is not a rule an agent will follow: Project rules are visible in context, but nothing tells a developer whether a rule was applied to a given decision or quietly dropped.

Ownership relocates from text generation: Developers re-anchoring the decisions on how code implements rather than in the text. Responsibility stays with the developer regardless of how the code was produced.

KEYWORDS *Agentic AI, Human-Computer Interaction (HCI), Human-AI Collaboration (HAIC), Developer Experience (DevEx)*

Introduction

The software engineering landscape is currently undergoing a significant conceptual shift as generative Artificial Intelligence (AI) tools transition from passive code completers into active collaborative partners integrated directly into development workflows [1–3]. Beyond using simple prompting, modern engineering environments increasingly leverage the use of AI agents. These agentic systems are capable of reasoning, multi-step planning, tool execution, and reflective self-correction across multiple iterations with minimal human intervention [1, 4]. This workflow requires more high-level human judgement and decision making strategy. It also raises the stakes of error because agents execute chains of dependent actions, and even a small per-step error rate compounds across those steps, so an early mistake distorts later reasoning and can become difficult to recover from [5].

This shift in the role of generative AI redefines the programmers' professional identity and core responsibilities from keystroke-level direct manipulation toward collaborative planning and curation of AI behaviour [1, 2, 6]. Historically, developers have spent approximately 70% of their workday reading and understanding existing source code instead of writing new lines [7, 8].

In the agentic paradigm, this cognitive demand persists through high-level decision-making. According to Hitzig et al., empirical evidence from industrial usage of autonomous coding agents shows a stark division of labor: human users make roughly 70% of the planning decisions (what to do) while agents make about 80% of the execution decisions (how to do) [9]. This transition from direct code authorship to high-level agent supervision, from commanding to delegating, represents a profound cognitive shift in the daily developer experience [1, 5, 6].

At Techflouu, the implementation of agentic AI follows a decentralized, project-fit approach rather than top-down standardisation. Development teams retain autonomy to experiment with and adopt agentic tools based on project-specific needs and context. This flexibility allows the organisation to maintain small-team efficiency and contextual decision-making at scale. This autonomy is bounded by structural safeguards as Techflouu maintains rigorous technical controls: robust credential management, strict testing pipelines, and governance protocols defining permitted AI actions and prohibited decision spaces.

Based on internal repository data, in the seven months after adopting agentic AI the team integrated 71% more commits per month and deployed to production 46% more often, while the share of failed production deployments stayed the same. Yet a critical gap in developer cognitive and human-AI collaboration remains unexamined. The organisation has optimised for technical integration and output validation, but has not systematically studied the shifts in cognition, collaboration, and professional identity required when developers transition from direct code manipulation to AI supervision.

To address this gap, we formulated the following central research questions:

1. How do developers adapt their workflows, mental models, and collaborative practices when transitioning from direct code manipulation to high-level supervision of autonomous AI agents?
2. What mental models of trust and collaboration emerge when delegating autonomy to non-human agents?
3. How does this shift in agency affect developer perceived authorship?

To investigate these questions, we applied an informed grounded theory approach focused on collaborative sensemaking and human-AI teaming [10]. Prior to empirical studies conducted by Google [1], Atlassian [8] and Anthropic [11], we establish three initial socio-technical areas below as foundational sensitizing anchors.

Interaction Shift: How developers adapt to new modes of use with AI systems. Developers have historically worked through interfaces with an exact vocabulary, where each action has a precise expression and repeats identically. Agentic AI substitutes free-form natural language, in which intent is stated approximately and the same instruction need not produce the same result [12]. This shift in interaction modality reshapes cognitive load, mental models, and the nature of control itself [3, 13].

Agentic AI as Collaborator: How developers construct collaborative workflows with autonomous agents. This includes understanding AI reasoning through technical representations like code diffs

and logs; navigating the paradox that agents must act autonomously yet remain fundamentally bounded [1]; building confidence through planning, validation practices and inspection; and synchronizing AI-generated work with existing tooling, compilers, and validation pipelines [2, 4]. Together, these dimensions capture how developers actually work with agents, how they reason about agent behaviour, calibrate expectations, establish trust, and integrate AI-generated work into team workflows.

Identity and Socio-Technical Impact: How developers experience shifts toward AI-augmented workflows and the resulting effects on perceived agency and identity [11, 14]. This dimension captures whether developers feel like authors or supervisors, whether collaboration with AI enhances or diminishes their sense of authorship and craft [2, 3].

Findings

While our initial study design was informed by three sensitizing anchors (Interaction Shift, Agentic AI as Collaborator, and Identity and Socio-Technical Impact), our analysis uncovered a fourth, unanticipated area: Tooling and Platform Adoption Dynamics. Table 1 illustrates how this emergent platform layer serves as the operational foundation connecting developer interactions, trust boundaries, and professional identity shifts.

AREA	TYPE	CORE SOCIO-TECHNICAL FOCUS
Interaction Shift (INT)	Anchored	Modality transitions, cognitive load, prompt mechanics
Agentic AI as Collaborator (AGE)	Anchored	Task delegation, context building, reasoning inspection, workflow synchronization
Identity & Socio-Technical Impact (IDE)	Anchored	Authorship, skill continuity, professional agency
Tooling & Platform Adoption (TPA)	Emergent	Sets the agent’s visibility, conditioning inspection and the grounds for claiming ownership

Table 1: Overview of anchored and emergent socio-technical domains after coding analysis

Across the four domains, a consistent pattern emerged: agentic AI did not simply reduce developer effort but redistributed it. As execution became increasingly delegated, developers’ effort shifted toward specifying intent, managing context, making decisions, and reviewing outputs. The resulting constraint was not code-generation capacity but the human capacity to evaluate and govern what agents produced.

Interaction Shift

We anticipated that the primary friction in this transition would be the interface itself, specifically the move from precise, GUI-based manipulation to an open-ended chat box. The interviews instead locate the primary friction elsewhere. No participant described the conversational modality as difficult to adopt. What participants report instead is a friction of explanation: the

work of translating precise technical intent into natural language. This translation appears to constrain how effectively participants can interact with the agent.

On instructing the agent: We found that these problems span from natural language being insufficient for tasks that are rule-heavy or demand precise computation to limitations within participants themselves, such as knowing a concept while losing the word for it. What is notable across these limitations is that no developer responds to the difficulty of expression by specifying more precisely: participant (D1) removes reliance on natural language, (D6) reshapes the problem, (D4) hands the prompt writing to another model, (D3) talks around the word until the model supplies it, and (D5) relies on what is self-described as simple, broken english.

What works, against expectation, is keeping instructions brief. The distinction matters, because it is easily confused with the context-building practices described in the next section, and the two pull in opposite directions by design. Grounding material such as specifications, requirements documents, and design is supplied extensively. Prescriptive instruction about how to implement is deliberately held back. In short, background information goes up and specific instructions go down.

The reason is that over-specifying the implementation narrows what the model will consider. Developer (D3) reports diminishing returns when moving from suggestive guidance into explicit technical instruction, finding that detailed prescription prevents the model from proposing an approach better than the developer's own.

"...I just know that this AI knows way more about the detailed technical stuff, like how memory works... so I don't think it's necessary for me to explain too detailed, because then I might mess up the best practices that they already know."

— D3

The position is not uniformly evidence-based: one developer abandoned role-based prompt engineering as too much of a hassle and has never compared the results, resting the judgement on a gut feeling that the outcome is the same. Taken together, these findings suggest that the challenge of agentic interaction is not primarily learning to prompt, but learning how much intent to specify while leaving implementation space for the agent.

A redistribution of labour: At the upper end of our sample, one participant reported delegating nearly all code execution and estimated that output had increased almost fivefold. That figure does not resolve into a straightforward productivity multiplier. The gain lands on output generation, not on the workflow as a whole. In (D7)'s case the recovered capacity was absorbed by a tighter schedule, so the time saved returned as pressure not just as headroom. Participant (D4) describe how the code generation capacity has risen sharply while human review capacity has not:

"...the main thing is that AI can generate 10 times the code that we are able to generate, but we as the reviewer cannot review 10 times more, right? And I think that's the point that makes me personally feel more tired."

— D4

This shifts the bottleneck in the development workflow from code generation to human verification.

Our cases illustrate effort shifting across the workflow. Because execution is now largely automatic, effort concentrates in planning, review and testing. Figure 1 summarises this redistribution of developer effort on traditional workflow compared with agentic AI workflow.

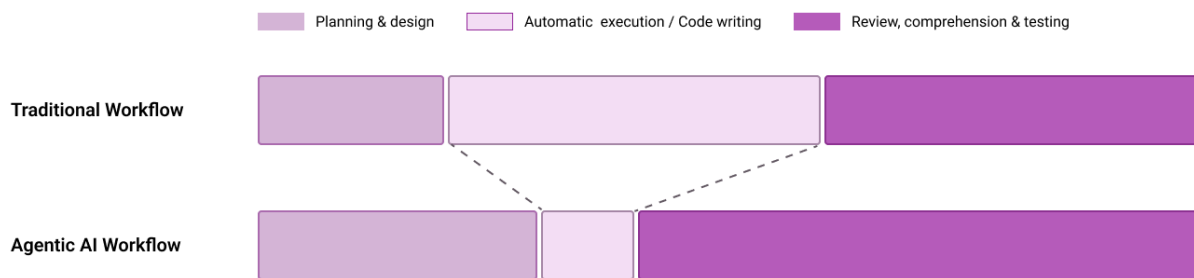


Figure 1: Conceptual model of developer effort redistribution following agentic AI adoption. Segment sizes represent directional and relative changes described across participant accounts.

The relocation was evident across all cases. What does not follow automatically is fatigue. Where it does appear it takes a different form in each case: review volume for (D4), decision volume for (D5), schedule compression for (D7), who calls it “a different headache” instead of a heavier one. In summary, the code writing craft has been exchanged for the continuous work of supervision and intent evaluation.

“...Much more decisions to make... that’s the tiring part. But of course, when you write code, you also get tired with gathering context... so yeah, it’s a different kind of tired. I’ve never had to make this many decisions in one day.”

— D5

The reported fatigue therefore reflects not simply increased workload, but a change in the type of cognitive work required: developers must make more decisions about whether generated work is acceptable rather than spend that time producing the work themselves.

Agentic AI as Collaborator

Across all participants, collaboration with an agent resolves into deciding what to hand over and inspecting agent’s output. These are the two most heavily coded practices in our data. The scope of what is delegated ranges from a bounded function to the entire coding task. The means of inspection ranges from reading the diff line by line to running the software and judging it by its behaviour. Control in this workflow is applied before the agent’s run, not during it.

Context engineering and where it stops: Our participants describe context engineering as the work of turning developer knowledge into artifacts that will guide tasks they have not yet defined. What gets written down is what the agent needs to know about the project, what it needs to be able to reach outside of it, and what a colleague will later need to read in order to pick the work up. The common move is to establish this before any particular task:

“The most important thing for context management is to have the context on the project level, before we actually do the detailed tasks, because then when we need it to do some task it already knows the necessary context.”

— D3

What makes this work non-trivial is that the constraint is not a shortage of material. The codebase, the tickets, the specifications and the design documents are all retrievable; what is scarce is the model’s attention budget. This moves participants to supply which artefact most want the agent to understand:

“I don’t think that context window and the token limit is enough for like the entire code... that’s too large for the entire model”

— D4

The first response to that ceiling is selective grounding, in which the developer manages that budget by selecting the parts of the repository worth spending it on. Feeding the model a curated subset buys a tractable task and forfeits system-level comprehension. Participant (D7) describes directing the agent to particular folders and files rather than letting it locate them:

“I sometimes referring to exact files. Refer to what folders, what files to make the agents do the things much faster.”

— D7

Participant (D7) recalled connecting a repository to an early agentic AI system and exhausting the available context before being able to ask anything of it. Although context windows have expanded considerably since then, the underlying constraint remains. The codebases our participants work with continue to grow, now alongside AI-driven code generation that is no longer limited by human typing speed. The practical constraint is therefore the relationship between how much context exists and how much of it the model can meaningfully attend to.

The mechanical ceiling is not the only limit these artifacts run into. The second is social, and it does not move with the size of the window. Scaffolding scales for what has been written down, and what has not been written down is exactly the part that people hold:

“AI doesn’t have the context. They only know the code base, they only know the tasks that have been written. But the human holds the context. The human holds the business logic, holds the business problem, holds the client’s needs and the AI doesn’t really know any of those.”

— D4

Participant (D6) draws the same line from the other side, retaining business and feature context gathering as his own work while letting the agent propose options on the technical questions. No amount of budget solves this, because the material was never in the repository to be pointed at.

Configuring the collaborator: When participants configure an agent’s behaviour, they design it to offer resistance. No participant in our sample wants an agreeable collaborator, and one

explicitly rejected a tool for being too compliant. What participants debate is not whether the agent should push back, but how strongly, in what tone, and at what stage of the process.

The most demanding setup in our sample was chosen by a developer who configured strict agents to constantly interrogate their plan. This participant credits this interrogation with noticeably improving code quality by surfacing hidden edge cases. When asked if this setting is ever disabled on an exhausting day, the participant answered no, citing trust in the process. Here, agent's behaviour configuration deliberately introduces friction to counter the developer's own tendency to settle for the first solution.

The standard for this resistance is defined most directly by participant (D1), who benchmarks the agent against a human peer rather than a setting. The participant specifically expects the model to replicate reviews that a senior engineer would provide:

"Having AI to challenge my work, maybe it's like a senior engineer when review. I think it's a good assistant for right now."

— D1

Even the participant seeking the least confrontation does not want blind agreement, restricting the persona to strict neutrality:

"I just want them to be neutral, just give me something, not critic, not support, because that's too much. I think being neutral is good."

— D2

What these configurations share is their reference point. In each case the setting is chosen against something about the developer, not just something about the task: a tendency to settle, a limited budget for reading, a wish not to be either flattered or overruled. Notably, the participant who delegates the most tasks configures no persona at all, choosing to review the agent's plan before execution without altering how the agent communicates with them.

Rationing trust and boundaries: Participants across all levels of experience ration what the agent may do. Figure 2 illustrates how task difficulty only governs the soft end of the spectrum, where routine, reversible and low-stakes work (such as unit tests, boilerplate CRUD, minor merge conflicts, reading project history) is handed over whole. The hard limits are drawn at what cannot be undone. Production data, database migrations and deployment are declared off-limits and handled personally.

This suggests that delegation is not governed primarily by whether participants believe the agent is capable, but by whether the consequences of failure remain recoverable. Participants therefore retain control precisely where human judgment has the highest cost of being wrong.

Participant (D5) issues the restriction to the agent as a standing rule:

"I just ask AI: never deploy, never push, never migrate. Just that treatment. So all the changes it makes, it's local."

— D5

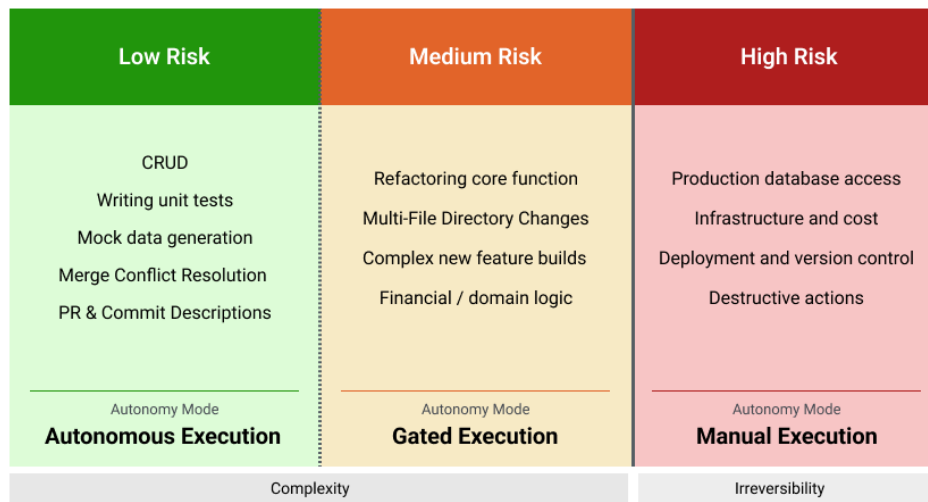


Figure 2: Risk-weighted delegation and autonomy mode. The boundary between autonomous and gated execution is drawn dashed because it is set by task complexity, which varies between developers. Meanwhile, the boundary before manual execution is drawn solid because it is set by irreversibility, is shared across participants as a criterion, and does not respond to evidence of the agent’s accuracy.

Another states the same limit as a principle about infrastructure, and gives the reason directly:

“I still stand by the principle that AI shouldn’t be delegated into infra works. Infra works still need to be done by human, because it’s related to cost, it’s related to billing. So if AI messed up then there’s no going back.”

— D7

Two features separate these limits from ordinary caution. The first is that they are indexed to the cost of failure, not its likelihood, which means a good track record is not the kind of evidence that could move them:

“I saw AI generating the SQL migration file, quite good, nothing wrong... I just prefer to review it first, then run it myself.”

— D5

The second is that what makes an action safe is environmental containment. Confidence is argued from locality: work that stays in a local checkout or a sandbox environment is treated as safe more or less regardless of content, while the same action against production is refused. Participant (D7) permits bypassed permissions and a fully unattended loop on an experimental project confined to a platform whose failure participant judges containable, and refuses the identical arrangement on anything touching the production environment, including for client work:

“The blast radius is much smaller because it’s an experimental project. It doesn’t touch the live services that are in production cloud. So anything production related I still need to do manually...”

— D7

Almost all of these boundaries are precautionary. They are installed against hazards participants have heard about (agents deleting production databases, supply-chain compromises reaching a team through a dependency) and the harm anticipated is destructive. Our data contains one first-hand failure, and it is neither destructive nor a breach. A shared feature was worked on by two developers, each with their own agent. The rule was written for one situation and applied in another, and neither developer caught the edge case before the agents acted on it. Nothing in the correctness of either agent's output would have revealed it, because both were doing exactly what the rules told them to. The participant diagnoses this as a coordination problem rather than an agent error:

"One key learning here is that every time we do changes, or we do something with AI, it needs to be fully documented so that the person, and also their agentic tools, their coding tools, can have a full idea of what is happening step by step, and can adjust based on their needs."

— D7

It is a single case, but it is the only realised harm we have, and it points somewhere the anticipatory reasoning does not.

However, not every retained task is about risk. Participant (D3) delegates nearly all execution but researches a technological stack's strengths and weaknesses himself, through his own reading, before consulting with AI. The commitment that constrains everything built afterwards stays human. Another uses AI to write commit messages but modifies them so they have enough context for teammates to easily understand the change. To sum up, participants avoid delegating tasks that are high-risk, permanent, foundational to future work, or meant for human interaction.

Identity and Socio-Technical Impact

We found that adopting AI agents alters developer identity far beyond workflow efficiency, surfacing primarily through debates over skill friction and agency. Experiences of technical skill range from localised losses to claims of skill preservation driven by high-level code review and comprehension. Regardless of whether they report skill loss, participants proactively assert control over their expertise through self-imposed boundaries like AI-free coding periods. Ultimately, this shift forces a broader renegotiation across two key areas: code ownership and career trajectory.

On who owns the code: Participants defined code ownership along a spectrum moving from line-by-line authoring to high-level decision-making. At one end, participants view code as theirs only if they manually reviewed or edited the majority of it, while others struggle to separate human intent from AI generation. Shifting away from the text entirely, many define ownership through deep system comprehension, such as the ability to debug quickly and address client needs. Participants also define full ownership simply because they made the final decisions. While the code ownership definition varies, none of them proposes that responsibility moves with the authorship. A tool can produce the artefact but it cannot answer for it.

“...let me give you an analogy, cars, right? We humans can walk, but why do we use cars?... It’s basically a tool. But you still have to take driving courses, you still have to get your driving licence to drive a car.”

— D4

“I didn’t fully immerse line by line, but I acknowledge the outline of the changes. If anything goes wrong, it’s me that needs to be pointed out as the mistake. We cannot point the mistake into [the tool].”

— D7

Taken together, these positions describe ownership migrating from the artefact to the product. What is owned is no longer a body of text a developer produced but a working system that someone will one day have to answer for, and the answering is the part that cannot be delegated. The agent can hold the authorship but it cannot hold the responsibility.

Software engineering in the future: Our participants describe the work now moving toward decision making and problem solving. Because execution is delegated to the agent, attention is redirected toward product understanding and the client relationship. The upstream work has always been there and has always been the harder part. Participant (D5) heard that before the AI era that coding was the easiest part of building and that designing the architecture was the hard part, and treats the present as that long-standing claim finally becoming operative. What has changed is that developers now have time to act on it:

“After we adopt this AI thing, we can learn more things rather than the old way where we are fixing code every day... we have more time to do other stuff, like thinking.”

— D2

Additionally, entry barriers have fallen. The threshold for producing working code has fallen far enough that people without engineering backgrounds can now build. Participants hypothesised that the learning curve for engineering is judged to have steepened, since newcomers must now demonstrate value against generated code that is freely available. The harder problem is that the practice through which the underlying competence used to be acquired has been removed. Participant (D6) states it as the central learning difficulty of the moment, and applies it to himself as much as to anyone junior:

“The biggest learning curve is being able to understand code, or understand algorithms, without actually writing. We rarely write... I don’t write as much, but I still need to understand without writing.”

— D6

Participant (D7) also observed that comprehension used to come from nuances of a technology which was learned by working in it. Newcomers who have never written code will not have that distinction available, however good the code they can produce. Participant adds a determinant no one else raises, which is what a newcomer can afford, and the difference is not about competence but about what would survive the tools:

“They will need to pay for the models. If the models are gone, then GG [gaming slang: ‘good game’, used to mark an unrecoverable loss]. That’s the difference with us right now. At least for us, if the models are gone, we still have a recollection of what coding feels like.”

— D7

The participant’s own answer is not to reverse the reliance but to add the layer that catches what reading no longer catches. Engineers, participant argues, must learn infrastructure, because a slow or badly built system reveals itself there when nobody is inspecting the code and it acts as the last line of defence on whether an agent is shipping good work. The account our data supports is not that experienced developers have adapted and newcomers must catch up, but that everyone is re-learning where their value sits, from different starting points and with different things at stake.

Tooling and Platform Adoption

Where the other three domains take the agent as given, this one takes the market that supplies it. This revolves around where the work will happen, what a tool must be able to reach, and how long the choice can be expected to hold. Tool choice determines the visibility, controllability, and reach of the agent, which in turn shapes review and ownership.

Where the work happens: Selection criteria shifted in several cases over the period participants described. During early Large Language Model (LLM) methods were completely detached from the codebase. This requires participants to manually copy-paste the code generated into the codebase. Then it evolved to an inline assistant limited to a single line of a single file with no project context, restricting its use to basic refinement and optimization. When agentic tools arrived, most participants prioritized IDE-integrated solutions simply because they preferred working within familiar environments before even integrating AI.

“If using the terminal, it’s like I’m in hacker mode. I don’t really know what is going on. But from the chat panel of VS Code I can read things much clearer. The UI is better.”

— D2

The second move traded that legibility for reach. Participants who left IDE-integrated solutions did so because it was scoped to the repository that happened to be open, while the command line could move anywhere across the machine and hold several repositories in parallel; one judged the IDE agent limited precisely by its lack of plugins and outside integrations. Participants also describe terminal reduces code observability, but the loss is absorbed because the editor now becomes a review surface instead of for writing.

The third move is the current one, and it follows from the models themselves becoming stable enough to become differentiators. Where capability between the leading models is judged to be on par, the criteria become which harness carries the most capability around them:

“What I prioritise right now is to see which harness supports the most capability... the capability of the models is also on par between them.”

— D6

How the judgement is made from outside: Because AI tools change too fast for anyone to test them all, participants judge them using secondary clues. Participant (D3) decides whether to pay for a tool based on its community, checking if users discuss it for professional work or just casual use. His concern is long-term survival of the tools:

“If it has a big community, then probably it will last a long time. Because if not, then we’ll have to find another one and relearn the whole thing, the whole caveats of the whole environment.”

— D3

The community here functions as insurance as participant reports never having needed to ask it for help, because his workflow is basic and repeats the same steps without major breakdowns. What the participant is buying is the likelihood that the environment that is currently in use will still exist later. Where that evidence is collective for one participant, it is personal for another. Recommendations from trusted peers and senior colleagues play a major role in tool adoption. Watching a peer quickly build a functional service and hearing continuous backing did not instantly sell the developer on the tool, but the accumulated proof convinced him to give it a test run.

Discussion

Complexity and consequence in task delegation: A critical component of established human–AI collaboration frameworks, such as the one proposed by Fragiadakis et al., is the “task allocation” phase, which dictates how authority, execution, and oversight are distributed between the human and the machine [3]. In our study this allocation resolves into a multi-tiered risk-weighted autonomy model and its two boundaries are governed by two different criteria.

The boundary between low and medium-risk delegation is fluid and subjective. No shared definition of a low-risk task emerged across the interviews; what is rationed at this level is task complexity not the consequence of failure, and where each participant places the line varies with their own confidence and working habits. Meanwhile, the boundary between medium and high-risk delegation behaves differently. It is anchored in a criterion every participant who draws a hard line draws in the same place: the irreversibility of the action, independent of how simple or complex the underlying task is. It aligns with the warnings of Ji et al., who observe that in agentic architectures early mistakes can cascade into more severe failure in the end since the agentic action chain acts as amplifier [5].

Guarded and realised hazards: The gating described above rests on anticipation. Our case study also reveals an anticipated-versus-realised hazard mismatch. The threats participants guard against are existential and destructive – security vulnerabilities, supply-chain compromise, deleted production data – and the organisation’s controls are calibrated to the same class. The one realised harm documented in our cohort belonged to none of these categories. A project

rule written for one developer's case was applied by another developer's agent to a case it did not fit, producing repeated work that surfaced only through its cost. The participant cannot determine whether the written rule was insufficient to convey the scope of the work, or whether the agent had it available and did not act on it. From the developer's point of view the two are indistinguishable until output appears. Nothing in the guarded class would have caught it, and no correctness signal would have either.

This qualifies the scaffolding practices described above. Participants treat written artifacts as the mechanism by which context reaches a colleague's agent, but in the one case where that mechanism was tested at a handoff, it did not hold. Two mechanisms could produce this outcome, and our data cannot separate them. In the first, the rule was in context and was applied as written to a case it did not fit. This is the over-adherence DiSorbo et al. document in agentic decision-making, where models follow policies strictly even where adherence is impractical or counterproductive, because rules are incomplete and cannot anticipate every case they will be applied to [15]. In the second, the rule was never loaded, or was loaded and passed over, and the agent proceeded without it. These are opposite failures: a rule followed too literally, or a rule not followed at all.

What our case adds is a gap between loading and adherence. Tooling makes it possible to verify which instruction files an agent has in context, but not whether a rule was applied to a particular decision and no signal is emitted when it is not. The first evidence available is the output. By then the work has already propagated as the conditions under which a small early error compounds across a chain of dependent actions [5].

The asymmetry also explains where developer effort settles. Every practice described in our context engineering findings operates on the loading side. That side is inspectable, and the adherence side is not, so the available work is concentrated where the feedback is. This makes the trajectory noted above worth restating as a risk rather than only as a convenience. Where agentic retrieval takes over the allocation, the developer loses visibility into what was loaded, having never had visibility into what was followed. Our data cannot show this occurring, since the participant who has moved to autonomous retrieval is not the participant who experienced the handoff failure, but the two observations sit close enough together that the combination is worth stating as something to watch.

The Platform-Review-Ownership triad: Across our cases, the grounds on which participants claim ownership of agent-produced code vary with their distance from direct code production and inspection. We categorise these grounds into Review, Comprehension, Decision, and Accountability as positions along a continuum of ownership. These positions capture the primary basis on which participants justify ownership as their direct involvement with the generated code decreases.

- **Review.** Ownership rests on the act of inspection itself, and is claimed once enough of the code has been personally scrutinised.
- **Comprehension.** The artefact is still read but no longer gated line by line. Ownership rests on being able to answer for the system.

- **Decision.** Review is decoupled from the editing surface. Ownership defined as authoring the underlying decisions, not writing the code.
- **Accountability.** The code is not read directly at all. Ownership rests on the name attached to the commit and the obligation to answer for whatever ships under it.

In our data the grounds sort with the inspection of the working environment. Participants working in IDE ground ownership in what they did to the code – inspecting it or understanding it. While participants working on CLI ground it in authorship of decision and responsibility after the code shipped. This co-variation is descriptive and holds across cases. However, we cannot establish direction from it. We therefore retain the relationship as a proposition: the works for inspection agent-produced code change, the ground of code ownership may also shift. Testing it across tooling migration, or comparatively across teams whose inspection environments differ while their work remains similar, would resolve what a single round of interview cannot.

Reading maintains what writing built: Several participants disagree that their ability is declining. They believe it remains important because people need to understand what is being proposed before making a decision. Another argues that reading sustains the skill directly, and that syntax remains available as long as code continues to be read, even if far less of it is written. Both developers view manual typing as a practice they can safely drop, believing that reading and evaluating code now keeps their skills sharp. Rather than dismissing the value of learning to code manually, they simply argue that maintaining an established skill requires active review instead of constant writing.

This interpretation aligns with empirical findings by Shen and Tamkin [11]. In a randomized controlled trial on learning an unfamiliar library, AI-assisted participants showed reduced gains in conceptual understanding, code reading, and debugging, demonstrating that aggressive AI integration in the workplace can impair professional development when cognitive engagement drops [11]. Our participants' position is consistent with that conclusion, and it identifies engagement as the variable an organisation would have to protect. What neither our data nor theirs establishes is whether reviewing agent output in production work, under the schedule pressure described in our cases, constitutes engagement of the kind their high scorers displayed.

All participants in this study learned to program before agentic AI existed, so their accounts describe maintenance of an existing competence, not its formation. Their perspectives on junior developer training are inferences about a cohort they have not yet supervised. Studying post-agentic junior developers, those who entered the profession after agentic tools became standard, remains an essential direction for future inquiry.

Conclusion

This study investigated how developers at Techflouu adapt their workflows, mental models, and sense of authorship as they transition from direct code writing to supervising autonomous AI agents. Delegating execution does not simply multiply productivity; it redistributes developer effort. As code generation becomes increasingly automated, effort shifts toward specification,

control, and review, while generation capacity grows faster than participants' capacity to evaluate its output.

Much of the work that remains is allocative. Participants ration what the model attends to by directing it toward relevant grounding material while deliberately withholding prescriptive implementation instructions. They also ration what the agent is permitted to do. The resulting trust model is not calibrated solely to perceived reliability: participants allocate autonomy according to the consequences of failure and treat irreversibility as a boundary that even a strong track record does not readily move. A similar relocation occurs in authorship. As participants move away from producing the text of the code themselves, ownership re-anchors in the decisions that shape how the code is implemented, while responsibility remains with the developer.

Agentic AI therefore changes the economics of developer effort: when code generation becomes abundant, the scarce resources become human specification, verification, contextual judgment, and accountability. Two implications follow for organisations adopting these tools. First, controls designed around catastrophic failure did not catch the realised failure observed in this study, which produced no correctness signal and became visible only through its cost. Second, when review becomes the constraint, measuring productivity at the point of code generation measures the stage of the workflow that is no longer scarce. Sustainable adoption therefore depends not only on expanding what agents can execute, but on preserving human judgment where consequences are difficult to reverse and strengthening the capacity to verify what increasingly autonomous systems produce.

Methods

Participants (n=7) comprised the active engineering staff across the organisation's product teams at the time of collection. Experience ranged from under three years to over ten, with the cohort distributed across early-career, mid-level, and senior bands. All were engaged in day-to-day development work and had used at least one agentic coding tool in production work.

Data collection: We conducted semi-structured interviews of 45 to 60 minutes each, held remotely over video conferencing. Interview questions covered the three sensitizing areas, with follow-up questions varying by participant. Video was recorded with consent and transcribed using a generative AI transcription tool, with all transcripts checked against the recordings by researcher before analysis.

Analysis: Transcripts were analysed using an informed grounded theory approach [10], in which the three domains established in the Introduction functioned as sensitizing concepts [16]. Analysis followed four stages: (1) transcription; (2) initial coding using gerunds to capture developer actions and processes [17, 18]; (3) focused coding to sort and synthesise codes into categories [17, 18]; and (4) domain mapping, in which categories were located across our anchored and emergent socio-technical domains [16, 18]. Initial coding was carried out with LLM assistance under a human-anchored few-shot procedure, using Claude Opus 5.

Limitations

This is an exploratory internal study conducted within a single small-to-medium enterprise, and no claim to statistical generalisation is made or intended. Data were collected between 11 and 27 August 2026, and participants named specific tools and model versions whose capabilities changed materially within the collection window. Because the cohort is a near-census of a fixed team, theoretical sampling was not possible and no claim to theoretical saturation is made.

This study was conducted by an insider to the organisation, known professionally to all participants. This afforded access and contextual understanding but introduces the possibility of social desirability effects, particularly in disclosing personal skill decline or practices a colleague might judge. Interviews were conducted in English, which is not the first language of the participants. Quotations are reproduced with disfluencies removed but no wording substituted; where a passage remained ambiguous it was not used as evidence. Prior to release, every participant reviewed the direct quotations attributed to them and confirmed both their accuracy and their willingness to have them published.

Acknowledgements

The author thanks Hafizha Fauzani Fitri, for guidance on the literature and methodology throughout this research, and for reviewing the manuscript at every stage. Thanks also to, Nuriyatul Ma'shumah and Andrew Batiduan, for reviewing the final draft. The author is grateful to the developers who gave their time to be interviewed and reviewed the quotations attributed to them prior to publication.

Finally, the author thanks Techflouu for granting permission to conduct this study within the organisation and to publish its findings.

Generative AI Disclosure

This research integrated Generative AI tools to support interview transcription and initial coding, as well as the structural drafting and editing of the final text. All AI-generated output, analytical summaries, and draft revisions were fully reviewed, verified, and curated by humans prior to publication.

References

- [1] T. Dong, H. Sampath, J. Y. Lee, S. Y. Shi, and A. Macvean, "From correctness to collaboration: Toward a human-centered framework for evaluating AI agent behavior in software engineering." arXiv:2512.23844, Dec. 2025.

- [2] E. D. L. Cruz, H. Le, K. Meduri, G. S. Nadella, and H. Gonaygunta, "Redefining the programmer: Human-AI collaboration, LLMs, and security in modern software engineering," *Computers, Materials & Continua*, vol. 85, no. 2, pp. 3569–3582, 2025.
- [3] G. Fragiadakis, C. Diou, G. Kousiouris, and M. Nikolaidou, "Evaluating human-AI collaboration: A review and methodological framework." arXiv:2407.19098, Mar. 2025.
- [4] W. Takerngsaksiri, J. Pasuksmit, P. Thongtanunam, C. Tantithamthavorn, R. Zhang, F. Jiang, J. Li, E. Cook, K. Chen, and M. Wu, "Human-in-the-loop software development agents." arXiv:2411.12924, Jan. 2025.
- [5] Z. Ji, D. Wu, P. Ma, Z. Li, and S. Wang, "Testing and understanding erroneous planning in LLM agents through synthesized user inputs." arXiv:2404.17833, 2024.
- [6] N. S. Shashidhara, "AI in software engineering – how intelligent systems are changing the software development process," *European Journal of Computer Science and Information Technology*, vol. 13, pp. 28–39, Apr. 2025.
- [7] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer – an investigation of how developers spend their time," in *2015 IEEE 23rd International Conference on Program Comprehension*, (Florence, Italy), pp. 25–35, IEEE, May 2015.
- [8] W. Takerngsaksiri, C. Tantithamthavorn, M. Fu, J. Pasuksmit, K. Chen, and M. Wu, "Code readability in the age of large language models: An industrial case study from Atlassian." arXiv:2501.11264, July 2025.
- [9] Z. Hitzig, M. Massenkoff, E. Lyubich, R. Heller, and P. McCrory, "Agentic coding and persistent returns to expertise." <https://www.anthropic.com/research/claude-code-expertise>, June 2026. Accessed: Aug. 06, 2026.
- [10] R. Thornberg, "Informed grounded theory," *Scandinavian Journal of Educational Research*, 2012.
- [11] J. H. Shen and A. Tamkin, "How AI impacts skill formation." arXiv:2601.20245, Feb. 2026.
- [12] Z. Ding, "Towards intent-based user interfaces: Charting the design space of intent-AI interactions across task types." arXiv:2404.18196, 2024.
- [13] H. Subramonyam, J. Im, C. Seifert, and E. Adar, "Human-AI guidelines in practice: Leaky abstractions as an enabler in collaborative software teams." arXiv:2207.01749, July 2022.
- [14] M. Zhang, Y. Xu, and X. Wei, "AI-centered vs. human-centered: Exploring users' attitude toward AIGC in varying forms of human-AI collaboration," in *PACIS 2024 Proceedings*, 2024. Paper 13.
- [15] M. DosSantos DiSorbo, H. Ju, and S. Aral, "Teaching AI to handle exceptions: Supervised fine-tuning with human-aligned judgment." SSRN, 2025.
- [16] G. A. Bowen, "Grounded theory and sensitizing concepts," *International Journal of Qualitative Methods*, vol. 5, pp. 12–23, Sept. 2006.
- [17] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE, 4th ed., 2021.
- [18] K. Charmaz, *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SAGE, 2nd ed., 2014.