

A Preliminary Analysis of the Impact of AI Assisted/Generated Code on Quality, Centrality and Review Time at Scale

Audris Mockus  

The University of Tennessee, Knoxville, TN, USA
Meta Platforms, New York, NY, USA

Peter C. Rigby  

Concordia University, Montreal, QC, Canada
Meta Platforms, Toronto, ON, Canada

Don Stewart  

Meta Platforms, Sydney, Australia

Chandra Maddila  

Meta Platforms, Bellevue, WA, USA

Nachiappan Nagappan  

Meta Platforms, Bellevue, WA, USA

Abstract

AI code-generation tools are widely deployed, yet AI is not applied uniformly: its use varies systematically with the context of the task, the code being modified, and the engineer. Studies that do not account for these contextual differences risk attributing to AI what is actually driven by context. Using data from over 10K developers and 400K diffs at Meta, with character-level provenance tracing of AI-suggested code through editing, review, and landing, we model (a) which context factors predict AI code landing, (b) how landed AI code relates to review time, and (c) its association with production outages (SEVs) – while controlling for contextual confounds. We find that AI code lands more often in less central code, larger changes, test files, and for authors with higher tenure. Review time decreases and SEV rates are lower for diffs with AI code, but both associations are confounded by AI code appearing in less central contexts. These findings demonstrate that naive comparisons of AI vs. non-AI work may reach misleading conclusions, and we provide actionable guidance for controlling these confounds.

2012 ACM Subject Classification Software and its engineering → Software creation and management; General and reference → Empirical studies

Keywords and phrases AI code generation, code review, software quality, context-dependent AI, landing rate

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.93

Category Software Engineering in Practice Track Paper

1 Introduction

Generative Artificial Intelligence (GenAI or AI) code-generation tools are already being deployed widely despite limited and nuanced evidence of their software engineering impact. Many studies find increased speed and productivity [50, 51, 59, 9, 33] but also reductions in code quality [58, 21, 19].



© Audris Mockus, Peter C. Rigby, Don Stewart, Chandra Maddila, and Nachiappan Nagappan; licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 93; pp. 93:1–93:23



Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Studying AI impact is complicated because AI suggestions are made in a code editor, but the code may be modified before landing.¹ The probability that AI-suggested code lands depends on numerous context factors ranging from quality of the suggestion to properties of the code modified by the diff, properties of the diff itself, author, reviewers, etc. Indeed, AI may not be equally applicable or effective in different contexts and for different engineers. The commonly used and easily attained measure of acceptance rate has serious flaws [32, 25], while studies that compare actions of engineers with and without access to AI tools lack the essential link between the context and landing rate. Furthermore, landing rates are also an important way to gauge how AI performance on benchmarks transfer to actual industry software.

Our primary contribution is to demonstrate that AI is applied non-uniformly – its use depends strongly on context – and to identify which context factors drive this variation. This finding has a critical methodological implication: because AI-assisted and non-AI-assisted work differ systematically in their context (e.g., AI is more likely to be used for larger, less central changes), any analysis that simply compares outcomes of AI vs. non-AI work without controlling for these contextual differences will confound the effect of AI with the effect of context. Without such controls, software engineering outcomes attributable to AI tools may be misleading. For example, if AI tends to be applied predominantly to less central code, a study that does not take centrality into account may show increased output or other characteristics of less central code associated with higher use of AI even if such an increase is confounded by centrality. Similarly, if AI is more likely to be applied to larger changes, a naive comparison would attribute the properties of large changes to AI itself. Once we establish how the context affects AI landing rates, we address the remaining questions: how does AI impact code reviews and service outages (SEVs)?

Our contributions are: (1) character-level tracing of AI-suggested code to landed code via Chattributes provenance tracking; (2) demonstrating that AI landing rates are strongly context-dependent (varying with centrality, change size, complexity, language, and tenure), implying that diffs with and without AI code are not exchangeable; (3) modeling review time and SEV probability while controlling for the contextual confounds identified in the landing model; and (4) offering an alternative explanation for the literature finding that AI produces more code – larger changes may simply offer more opportunities to apply AI.

Our study focuses on pre-agentic, completion-based AI assistance – the dominant paradigm from 2022 to 2025 – for several reasons. First, the transition to agentic AI systems that autonomously plan and execute multi-step tasks is well underway, making this the last opportunity to establish rigorous empirical baselines for the pre-agentic era before the technology mix shifts irreversibly. Second, completion tools operate at the character and line level with the engineer in full control of each suggestion, whereas agentic tools make autonomous multi-file decisions; the confounding patterns we identify (centrality, change size, tenure) will manifest differently under agentic workflows, and disentangling the two requires understanding each independently. Third, nearly all published empirical studies of AI coding impact measured pre-agentic tools [51, 65, 66, 9]; understanding the confounds in that setting is necessary to correctly interpret the existing body of evidence. Finally, the methodological template we develop – first modeling where AI is applied, then controlling for those factors in outcome models – transfers directly to evaluating agentic tools, even as the specific coefficients change.

We study the impact of AI in three contexts: authoring, reviewing, and outages (*i.e.* field faults or SEVs). We next provide background, hypotheses, and a results summary for each.

¹ By landing we mean that the code ends up in the primary branch after all reviews and after (successfully) running through a CI pipeline, essentially making AI code a permanent part of the project’s codebase.

1.1 AI impact on code authoring

Existing evidence and common sense suggest that AI may not be equally useful or used in different contexts. For example, the degree to which AI is successfully used² may depend on the particular part of the codebase, engineers' experience and habits, the degree of trust in AI generated code (for example, in risky diffs), and on coding tasks. We, therefore, expect that AI code landing will vary with these and other context factors. For example, a responsible engineer would likely be more wary of AI code that touches central files. Likewise, the literature has shown that junior engineers are more likely to use AI (e.g., [65, 66, 9, 30, 12]). We use logistic regression with presence of AI-generated code as the response variable and include file centrality, author tenure and other common measures such as the size of the change, its cyclomatic complexity, and the number of tests modified.

Our hypothesis is *H1 : The landing of AI code will be affected by change, author, and file. E.g., file centrality, change size or churn.*

Results summary. We find the likelihood that AI code is landed varies with the codebase: AI diffs tend to modify lower-centrality files and test files. Contrary to existing literature, we find that junior engineers are not more likely to land AI code.

1.2 AI impact on code reviewing

With the knowledge of what diffs are more likely to land AI-generated code, we can properly measure how landed AI-code may, in turn, affect various software engineering outcomes. First, there is a perception that AI will transform developers into reviewers [2] and that AI will create more, and harder, work for reviewers [53]. Our next goal is to provide a preliminary understanding of the impact of AI code on manual code reviews, namely code review time. We control for confounds that are associated with landed AI code, such as file centrality. We use a multiple regression to understand the impact of AI on how long it takes to review code.

Our hypothesis is *H2 : The landing of AI suggestions increases diff review time*

Results summary. We find that AI code is reviewed more quickly, but when the centrality of AI code is high, it appropriately increases code review time.

1.3 AI impact on production faults, i.e. SEVs

Finally we focus on the impact of AI-generated code on production faults and outages (SEVs), arguably the most important and the least investigated software engineering outcome. Some previous work has found improved code quality [65, 66, 30], while other work observed a deterioration in code quality [23] and a much higher defect rate [58]. Since the diffs that cause outages (referred to as Site Events or SEVs at Meta) are determined and recorded for each outage in our organization, we use it instead of defect rate. We use a logistic regression to model the probability of a SEV that includes any AI generated code and also include key variables that are used in the models deployed to predict diff fault risk [1, 44]. Our hypothesis is *H3 : The landing of AI suggestions increases the probability of a SEV.*

Results summary. We find that, after controlling for context, a higher fraction of AI code is associated with slightly fewer SEVs; because AI-heavy diffs are larger and occupy higher-risk contexts, their raw SEV rate is not lower. Thus, once context is accounted for, AI use has not led to an increase in SEVs at Meta.

² We define “successfully used” as presence of AI generated code landed in the trunk.

2 Background and Data

Empirical software engineering has its origins in the need to evaluate the impact of software engineering tools and practices as they are expensive to acquire, deploy, and maintain, especially in large organizations. In this study we use fine-grained measurement of the code editing AI tools so that we can associate the part of AI code suggestions that ended up in the code. To our knowledge, none of the existing studies are based on measurements that track AI suggestions to the individual diffs, likely due to extreme complexity of tracking the provenance of each character through a variety of editing surfaces, revising, committing the code and then through diff revisions, code review, and eventual deployment.

2.1 AI use at Meta

Meta is a large software company with a complex code base that covers a wide range of applications from social networking and virtual reality to software engineering infrastructure, such as continuous integration tooling and workplace coordination. Tens of thousands of developers work on billions of lines of code (LOCs) in a monolithic repository that hosts source code written in more than nine programming languages. The development organization spans multiple continents and time zones, with engineers organized into product and infrastructure teams that range from small specialist groups to large platform teams of hundreds. The codebase includes user-facing products serving billions of users, backend infrastructure, data pipelines, mobile applications, and developer tooling – providing a diversity of software types and risk profiles that few single-company studies can match. The state of the source code and the developers associated with the source code keeps evolving constantly - internal libraries and tools get added and deprecated, while developers move across teams and change roles. At Meta’s scale, keeping up with the knowledge required to accomplish coding tasks is challenging. Notably, Meta has no policy restricting AI use on critical code; the organization actively encourages AI adoption across all contexts.

At Meta, an AI-assisted code authoring system named CODECOMPOSE has been deployed companywide [47, 14]. CODECOMPOSE is a pre-agentic, completion-based tool: it provides inline suggestions as a software engineer types code and is designed to predict tokens that would complete the current line (single-line suggestion) or multiple lines (multi-line suggestion) across programming languages. Unlike agentic AI systems that can autonomously plan, execute multi-step tasks, or modify multiple files, CODECOMPOSE operates reactively within the editor, suggesting completions only at the cursor position. Single-line suggestions should be quick, highly accurate, and help with the immediate context. In contrast, multi-line suggestions need to be deeply informed and intelligent. These code blocks can help users discover APIs, best practices, and implementation details. CODECOMPOSE is multi-lingual and has been trained on more than nine programming languages at Meta. As such, it inherits the property of LLMs to be multi-lingual. It is also customized for the organization: CODECOMPOSE is fine-tuned on Meta’s internal code repository, and is thus able to handle languages such as Hack and Flow. CODECOMPOSE is also deeply integrated with Meta’s version of VSCode, and other editors including online Jupyter-like data science environments. CODECOMPOSE can understand inline comments in natural language and generate code that adheres to the comment.

The CODECOMPOSE model exactly recreates the masked line between 40% and 58% of the time and has a high BLEU score between 56% and 73%, depending on the programming language [47] in back-testing. When rolled out into production, the acceptance rate was 22% which is comparable to those at Google and GitHub. A survey and a thematic analysis

of the comments from 70 engineers showed overwhelmingly positive feedback, with only 8.5% of comments containing negative feedback and less than 1% of engineers turning off CODECOMPOSE. The top two feedback comment themes related to discovering APIs and dealing with boilerplate (23%), and accelerating coding (20%).

2.2 Code Provenance via Chattributes

Chattributes is a system at Meta that tracks information about code authoring. Edit events (insert, delete, replacement of characters) in IDEs are observed via extensions available for all major editing surfaces (VS Code, Android Studio, NeoVim, web-based editors, and terminal editors), and the provenance of those changes are tracked. Metadata is associated with each changed span, including which tool or human author made the change, and if the tool was AI assisted. Formatters, copy and paste events and AI authoring tools are tracked. Edit tracking is implemented via a character-based form of the *LineLog* interleaved deltas system. When a commit is made to source control, the provenance of all changed characters since the previous commit is stored as commit metadata. This metadata is then *propagated* over source control operations (rebase, amend). This is done by deriving edit scripts (using Myers diff algorithm) from the underlying source control content, and applying that edit script to the metadata. This allows us to trace provenance of characters through source control actions, allowing us to determine which characters survive review to be landed in the repository. We thus reason precisely about the provenance of landed code, down to character granularity.

Chattributes is deployed in production across Meta and has been validated against alternative approaches that infer AI authorship by correlating editing actions with AI service invocations that occur close in time. The character-level provenance tracking in Chattributes consistently produces more accurate attribution than such temporal-correlation heuristics, which suffer from false positives when developers type simultaneously with AI suggestions and false negatives when network latency separates the invocation from the edit event. Chattributes remains the primary system used at Meta for measuring AI code contribution.

With chattribute data we are able to calculate the number of bytes written by CODECOMPOSE that ended up in the final landed diff. Approximately two months of data includes diffs in spring of 2025 and contains several hundred thousand of diffs excluding automation-generated diffs. Our analysis concerns roughly half of these diffs where chattributes were available (none were bot-generated). The uncovered diffs result from editors or workflows not yet instrumented rather than from the risk profile of the diff: covered and uncovered diffs are well balanced on file centrality, complexity, and prior-fault history. Coverage does, however, over-represent diffs in the earliest-instrumented ecosystems (Hack, Python, and JavaScript relative to C++), so it reflects the order of IDE/tooling rollout rather than risk-based selection. We cannot fully rule out residual selection on unobserved factors, and we discuss this potential selection bias as a threat to validity (Section 5). For each diff, we calculated measures, such as the number of files, line churn, cyclomatic complexity of the modified files, coverage, and so on. Approximately half of the analyzed diffs had at least some CODECOMPOSE-generated code landed.

When modelling AI code, we consider two measures: (1) the fraction of diff chunks or contiguous spans of a changed code file where any part of the AI authoring suggestion was landed, and (2) the fraction of modified bytes originating from that AI suggestion. We also record the AI tool and LLM model that was used as part of the metadata. Multi-line suggestions were substantially less common than single-line completions; in exploratory analysis, distinguishing multi-line from single-line suggestions did not change the direction of the landing associations we report, so for the models below we do not separate them. A dedicated analysis of how suggestion granularity interacts with language (e.g., the lower landing rate for C++) is left to future work.

2.3 Data Sources for diff properties

We collect data from code measurement systems used at Meta, such as lines of code, cyclomatic complexity, and line test coverage. These measures were calculated at the time of diff landing, *i.e.* merging into the trunk.

A separate company-wide system was used to track **every** SEV. The root cause of every outage is manually investigated and the diff or diffs that caused it are recorded. Due to a strong focus on customer experience, the accuracy of SEV data is high. We used Phabricator³ to obtain the remaining measures, such as diff creation and landing times, affected files, author, reviewers, comments, review interval, etc. Again, each measure is calculated at the time diff is closed. So, for example, measures of author experience change after each diff.

3 Statistical Modelling

To achieve our goal of better understanding the relationships between the use of AI-assistants and critical software engineering outcomes, such as code reviews and system outages, we use statistical models. Statistical models help quantify relationships between outcomes and variables that measure various properties of a diff. Two of our models have binary responses, namely if AI code landed and if a SEV was triggered. As is commonly done in such cases, we use Logistic regression. The remaining model has review time as a response and we use a multiple regression.

To satisfy modeling assumptions, all continuous variables are transformed using the $\ln(x + 1)$ function because they have highly skewed distributions. We add one to ensure that there are no zero values in the log transformation. We use $\alpha = 0.05$ as our significance threshold throughout.

We are not aware of validated theories that would help explain the probability of landing AI-generated code or how such inclusion may affect review times or the probability of a SEV. We, therefore, start from known models of reviews and SEVs that happen to be extensively used at Meta. Furthermore, it is essential to include them as control variables in the models that estimate the impact of AI to avoid confounding interpretations that ascribe the impact of the confounding variable to the impact of AI.

Software engineering measures tend to be highly correlated [40, 41], but models with highly correlated measures are hard to interpret, have unstable estimated coefficients, and poor predictive power [13]. To this end, if the correlation between two variables exceeded 0.7 – the threshold recommended by Dormann et al. [13] as the point beyond which collinearity begins to severely distort model estimation – one of them was removed from the model. We complement this pairwise screening with a multivariate collinearity check (GVIF, Section 5), which does not rely on a single arbitrary cutoff. We ensure that remaining variables are diverse (represent distinct aspects of the diff, author, and modified code) and rely on simple but validated measures already in wide use at Meta.

We subdivide variables into classes: prior activity and defects [26, 46], interdependencies [8], coordination [8], author expertise [26, 6], complexity [46], churn [48], and testing [44].

Finally, we exclude diff properties that are determined after the phase we investigate. For example, the authoring model cannot know information about reviewers. As a result, only the SEV model has the full set of variables. As with all statistical models, we want to establish a relationship between the response and variables (independent variables), as

³ <http://phabricator.org>

statistical models do not demonstrate causal relationships. Although we do not fit a formal causal model, our two-stage design is motivated by a causal-confounding logic that can be read as an informal directed acyclic graph (DAG): context factors such as centrality, churn, and test files jointly influence both whether AI code is applied and the downstream outcomes (review time, SEVs). These factors are therefore common causes (confounders) that must be adjusted for before any association between AI code and an outcome can be interpreted, and we treat them as such by first modeling AI landing (H1) and then including its predictors as controls in the outcome models (H2, H3).

We selected our variables based on three criteria: (1) extensive prior use in defect prediction and code review literature, ensuring comparability with existing work; (2) diversity across distinct aspects of the diff (prior history, interdependencies, coordination, expertise, complexity, and testing); and (3) availability at the appropriate temporal stage of each model. We verified linearity assumptions by inspecting residual plots and testing for non-linear terms. We also investigated interaction effects but found none that were statistically significant while also improving prediction and remaining interpretable, including interactions of AI fraction with centrality quartiles and language-specific slopes.

We initially considered a substantially larger set of predictors, including organizational unit, type of software (e.g., infrastructure vs. user-facing product), and additional measures of code and author properties. The variables presented here survived a sequence of selection steps: first, among strongly correlated predictors (correlation exceeding 0.7), we retained the one easier to interpret and more widely used in prior literature; second, we excluded predictors that were not statistically significant in any of the three models, for parsimony and due to space constraints.

3.1 Model Variables

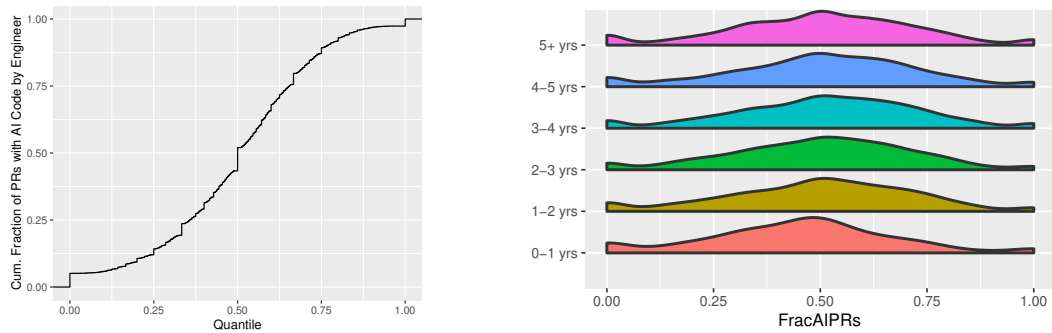
Table 1 lists all variables, their definitions, and rationale. As noted above, the models may include a subset of variables, as some were collinear or determined at later stages. All variables were calculated using approaches described in Section 2.3. We briefly summarize the categories here. *Prior history* (past SEVs, review comments, post-landing changes) captures the principle that past problems predict future ones. *Interdependencies* are represented via Katz centrality [27] computed over combined call-graph (extracted via Glean [39]), co-change, and authorship edges [64, 8] accumulated over a 12-month window. Because risk is driven by the single most critical file a diff touches rather than by its average file, we aggregate file-level centrality over a diff using the maximum (as we do for other file-level risk measures such as past SEVs); using the mean or median instead did not change the direction or significance of the centrality coefficients. *Coordination* needs are proxied by the number of reviewers, author centrality, and maximum past authors on modified files [22, 8]. *Author expertise* is measured via tenure and past modifications to the affected files [46, 26]. *Code complexity* includes churn and cyclomatic complexity [35]. *Testing* is captured by whether test files are modified [44].

3.2 Model Definitions

In the preceding sections, we discussed data sources, data collection, and the measures used in the models below. We evaluate the AI impact in three contexts: authoring code, reviewing code, and SEVs or faults. For each context we introduce dependent variables and describe each model below.

■ **Table 1** Measures. The measures are computed for each diff separately as they were at the time diff landed. $cent(f, t)$: centrality of file f at time t , $CCN(f, t)$, $LOC(f, t)$: cyclomatic complexity and lines of code of file f at time t , d and D , are diffs, a author, r : reviewer, d_a , d_r : indicators that a authored and r reviewed d ; d_f is true if diff d modifies file f ; d_t is the time diff landed; $d_{t_{created}}$ is the time diff was created. $Tests$ is the set of all test files. D_{sev} (or d_{sev}) is true for diffs that triggered a SEV and false otherwise.

Category	Measure	Description	Calculation
Past and current issues	$sev(d)$	True if any of the files in the diff have been modified by a SEV-causing diff in the past. “past faults predict future faults” [20, 6]	$\max_{\{f:d_f\}} \{1 \iff \exists D : D_f \wedge D_{sev} \wedge D_t < d_t; \text{ else } 0\}$
	$sugchg(d)$	Code review suggested making changes. A small discovered issue may hide yet-to-be-discovered issues [37, 3]	
	$ncmts$	Number of comments received during review. Discussion may imply complexity or risk of the diff [37, 3]	
Inter-dependencies	$chgpost(d)$	Indicator of whether the diff had changes after being landed. Issues that surface after landing may indicate higher risk [44]	
	$fcent(f, t)$	The maximum Katz call-graph centrality of the modified files. More central [64] files may have critical functionality [8, 67]	$\max_{\{f:d_f\}} cent(f, d_t)$
	$nfiles(d)$	The number of files modified. Need to modify multiple files in a single diff implies “logical” dependencies and risk [11, 46]	$ \{f : d_f\} $
Coordination	$nrev(d)$	The number of reviewers.	$ \{r : d_r\} $
	$acent(d)$	Katz centrality of the diff author based on co-change, authorship, and call-graph links. More central authors work with more central files (see above) and have more experience [64, 38]	$cent(a, d_t)$
	$maxauth(d)$	Maximum number of distinct previous authors over files modified in a diff. “Too many cooks spoil the broth” [46, 61, 6]	$\max_{\{f:d_f\}} \{a : \exists D : D_f \wedge D_a \wedge D_t < d_t\} $
Expertise	$pmod(d)$	Author expertise measured by the number of changes to the files modified by the diff in the past. Expertise is associated with defect rate [46, 61, 6]	$ \bigcup_{\{f:d_f\}} \{D : D_t < t \wedge D_f \wedge D_a\} $
	$tenure(a, t)$	Author expertise measured via tenure at Meta at the time of diff landing. Expertise is associated with defect rate [46, 61, 6]	six categories: 0–1, 1–2, 2–3, 3–4, 4–5, and 5+ years
File complexity	$churn(d)$	Sum of the lines added and deleted over all files modified by diff: More change more risk [46, 48]	$\sum_{\{f:d_f\}} addedLOC(f, t) + deletedLOC(f, t)$
	$ccn(d)$	Total cyclomatic complexity number (ccn) added over modified files. One of the oldest indicators of risk [35, 34].	$\sum_{\{f:d_f\}} CCN(f, t)$
Testing	$tfiles(d)$	True if at least one test file is modified by the diff. Empirical findings on test coverage vary: from reducing defects, [45], none [24, 29], or that high-risk files tend to have more coverage [44].	$ \{f : d_f \wedge f \in Tests\} > 0$
Programming language	cpp, php, py	Diff modifies at least one file containing source code from the corresponding language. Risk rates and review time may vary with programming language.	
Outcomes	$revtime(d)$	The time spent in code review (interval while the author was waiting for review(s))	
	$AIbytes(d)$	The fraction of bytes that are attributable to AI suggestions	$\ln \frac{ai_bytes+1}{ai_bytes+non_ai_bytes+1}$
	$AIsimps(d)$	The fraction of change chunks/spans (contiguous non-overlapping parts of modified text) that contain at least some AI-suggested code	$\ln \frac{ai_spans+1}{change_spans+1}$



(a) Fraction of diffs with AI code landed over the population of over 10K engineers.

(b) Distribution of the fraction of diffs with AI code landed by engineer's tenure.

■ **Figure 1** AI code landing rates across engineers and tenure categories.

Authoring Model

Our first model concerns the code authoring stage of AI suggestions by modeling the probability of landing any of the AI-suggested code in a diff.

In this model, we focus on variables that predate authoring actions as our goal is to measure the context of a diff that affects the landing rate of AI code. The response is binary (any AI code landed or not) and we use logistic regression. The predictors – listed as column (a) in Table 2 – include file centrality, author centrality and tenure, past author modifications, change size (files, churn, ccn), test files, and programming language.

Reviewing Model

Code review efficiency is commonly measured as the time code spends in review [15, 52, 54]. We use a linear regression with log-transformed review time as the response. All variables from the authoring model are included as controls to avoid confounding, plus the number of reviewers. Our key independent variable is *AIspans*, the fraction of change spans containing AI-suggested code (preferred over the fraction of bytes due to better fit; the two AI measures are closely related – their underlying log AI-span and AI-byte counts correlate above 0.9). Column (b) in Table 2 shows the full set of predictors.

SEV model

Statistical software risk models have been in use for 40 years [46, 26], incorporating previous faults [31], network complexity [8], coordination [22], and code complexity [55]. We use logistic regression with SEV occurrence as the response. The SEV model includes the full set of variables (column (c) in Table 2), adding post-authoring variables unavailable to earlier models: review comments, suggested changes, and post-landing modifications.

Model Fit

Our modeling goal is to explain the variance in the response variables and model fit is the most suitable way to accomplish that. If the model has poor fit, it means that most of the variance is explained by unmeasured factors. Generally, measures of model fit are most useful for comparing the same models with extra variables. From a practical perspective, what constitutes a reasonable fit is usually dependent on prior published work or based on expert opinions.

In logistic regression (our AI landing and Sev models) regular R^2 is not available because, instead of sum of squares (which are not additive), we have only the likelihood values. A commonly used alternative is McFadden’s pseudo R^2 . It measures the ratio of information gain when adding variables to the whole information contained in the data. It has a slightly different interpretation from the regular R^2 . McFadden states [36] on page 306 “... its values tend to be considerably lower than those of the R^2 index...For example, values of 0.2 to 0.4 represent EXCELLENT fit.”

For the review time model we use adjusted R^2 as it is a multiple regression and compare it to models published in academic literature.

Unlike the author and review models, the SEV model represents extremely rare events, which can bias standard maximum-likelihood logistic regression toward underestimating event probabilities [28]. Established remedies include Firth’s penalized-likelihood logistic regression [16] and King and Zeng’s rare-events correction [28]; because our sample contains several hundred SEV-causing diffs—keeping the number of events per predictor above the widely used rule of thumb of ten—and because the rarity is in the base rate rather than the event count and we report relative rather than absolute probabilities, these corrections are not expected to materially change the reported associations. For goodness of fit, because the base rate is so low, McFadden’s pseudo R^2 may be too optimistic and Tjur’s Coefficient of Determination, which is another analog of R^2 , is preferred [57]. It represents the difference between the average predicted probabilities for each outcome category: SEV-causing (p_{SEV}) vs not SEV-causing ($p_{no\ SEV}$): or $p_{SEV} - p_{no\ SEV}$. As the probability of a SEV is proprietary information, we share the relative value of improvement: $\frac{p_{SEV} - p_{no\ SEV}}{p_{all}}$, where p_{all} is the fraction of SEV-causing diffs.

4 Results

4.1 H1: AI impact on code authoring

Before suggesting a positive mechanism between relating landing rate and context, we may start from the negation or null hypothesis that diffs where AI suggestions are landed are exchangeable with diffs where no AI suggestions are landed (conditional on AI availability): $H1_0$: *The landing of AI code is not influenced by file centrality, author tenure, size/churn, and other properties of the changed code.* If that were the case, we could directly compare outcomes between the two classes of diffs. The results shown in Table 2 demonstrate that the assumptions of the null hypothesis are clearly violated.

Model Fit has McFadden’s pseudo $R^2 = 0.12$. As discussed in Section 5, strengthening the AI landing criteria from having at least one to at least 100 AI bytes landed monotonically improves fit: McFadden’s pseudo R^2 rises to about 0.17 (Nagelkerke $R^2 \approx 0.26$), and it continues to increase with the threshold. In other words, with a larger AI footprint threshold the model fits better and does not require additional variables to explain landing rates. The modest fit at the one-byte threshold is expected: our goal is to characterize the *context* in which AI code lands, not to fully predict an individual engineer’s moment-to-moment acceptance decisions. Several determinants of whether a specific AI suggestion is accepted are not observable in our data—individual developer preferences and trust in AI, team norms, the specific task type, the editor and IDE configuration, and the quality of the particular suggestion or prompt—and these unmeasured factors account for much of the residual variance. Importantly, the confounds we *can* measure are exactly the ones that also

■ **Table 2** Three models: (a) probability of landing AI code (logistic, McFadden's pseudo $R^2 = 12\text{--}17\%$), (b) review time (linear, Adj- $R^2 = 14\%$), and (c) probability of SEV (logistic, relative Tjur's $R^2 = 80\%$). Tenure coefficients are contrasts with "tenure 0–1 years."

	(a) AI Landing		(b) Review Time		(c) SEV	
	Est.	p	Est.	p	Est.	p
(Intercept)	-2.65	.00	7.58	.00	-6.66	.00
sev	-0.08	.00	-0.04	.00	0.05	.01
chgpost					0.15	.07
lncmts					0.34	.00
sugchg					-0.37	.01
lfcent	-0.01	.00	0.02	.00	0.07	.00
lacent	0.01	.40	-0.08	.00	0.10	.01
lnrev			-1.12	.00	1.09	.00
lmaxauth	-0.04	.00	0.00	.53	0.06	.03
tenure1-2 yrs	0.03	.45	-0.06	.16	-0.10	.54
tenure2-3 yrs	0.04	.34	-0.12	.01	-0.34	.05
tenure3-4 yrs	0.10	.02	-0.10	.03	-0.16	.34
tenure4-5 yrs	0.05	.25	-0.09	.04	-0.07	.68
tenure5+ yrs	0.05	.23	-0.03	.49	-0.12	.53
lpmod	-0.00	.74	-0.11	.00	-0.04	.13
lnfiles	1.10	.00			-0.03	.67
lchurn	0.29	.00	0.53	.00	-0.01	.82
lccn	0.03	.00	0.07	.00	0.10	.00
tfiles	0.16	.00	0.17	.00	0.16	.08
cpp	0.03	.15	0.35	.00	0.50	.00
php	0.50	.00	-0.09	.00	0.54	.00
py	0.40	.00	-0.33	.00	-0.26	.02
AIspans			-0.08	.00	-0.06	.03

drive the outcomes in H2 and H3, so an incomplete landing model is sufficient for its role as a confounder-identification step. From the model coefficients in Table 2 we can see the following:

- Landed AI code is associated with larger churn, more files, and higher cyclomatic complexity (ccn) of a diff. Larger diffs modify more files and have higher churn which may provide more opportunity for AI code suggestions (and subsequent landing).
- The chance of landing AI code is higher for diffs involving test files.
- Code that is more central (indicating higher interdependencies) is less likely to have AI code landed. This effect is small in magnitude – the `lfcent` coefficient of -0.01 corresponds to roughly a 1% change in odds – and modest relative to the dominant predictors of change size (`lnfiles`, `lchurn`) and programming language. It is also sensitive to the choice of centrality measure, so it should be read as a modest tendency rather than a strong regularity.
- Code with more past authors (indicating higher coordination needs) is less likely to have AI code landed.

For H1, the main findings are that landing AI code is less likely for central code requiring coordination. Larger changes, and higher author tenure are associated with increased chances of landing AI. The programming language strongly impacts the probability of landing AI code.

- There are notable differences among programming languages with Hack and Python having much greater AI landing rates than C++.⁴ C++ is primarily used for performance-critical tasks and, because of the optimizations associated with performance needs, code may be more complex.
- We did not observe increased landing rates for junior or less central engineers in contrast to findings by, *e.g.*, Ziegler et al. [66]. In fact, AI landing rates are highest for authors with 3–4 years of tenure, the only tenure category with a statistically significant positive coefficient. Figure 1a shows the distribution of AI landing rates across the population of engineers, and Figure 1b breaks this down by tenure category.

To illustrate practical significance: the coefficient for `lchurn` (0.29) implies that roughly tripling churn is associated with a 34% increase in the odds of landing AI code ($e^{0.29} \approx 1.34$); `php` (0.50) implies 65% higher odds for diffs modifying Hack files. For the SEV model (Table 2), the `AIspans` coefficient of -0.06 ($p=0.03$) corresponds to a 6% decrease in odds of a SEV per unit increase in AI fraction.

4.2 AI impact on code reviewing

H2₀ : The landing of AI code does not increase diff review time.

Table 2 shows that, after adjusting for a number of factors, the landing of AI-suggested code is associated with reduced review time.

Model Fit. Adjusted $R^2 = 14\%$ is consistent with prior work modeling review times, *e.g.*, [52, 54]. Review time is strongly affected by factors that are difficult or impossible to measure: reviewer workload and how busy the reviewer is, whether they are taking a sick day, time-zone differences between author and reviewer, the urgency of or deadline attached to the change, and team culture around review turnaround. Even the best models of code review time used in Meta do not have much better fit. Furthermore, our focus here is to establish the impact, not to do review time prediction where higher values of R^2 would be necessary. Because most variance remains unexplained, we phrase the review-time conclusions as associations rather than effects and caution against over-interpreting the magnitude of any single coefficient.

Table 2 shows that code with higher fraction of AI spans is reviewed more quickly. To illustrate practical significance: the coefficient for `AIspans` (-0.08) implies that a one-unit increase in log AI span fraction is associated with approximately 8% shorter review time ($e^{-0.08} \approx 0.92$). As for the control variables, our findings match intuition. Higher tenure leads to reduced code review time (compared to 0-1 years), as do other measures of experience: author centrality and past changes to modified files.

However, the more central the code is, the longer the review takes, indicating that engineers use appropriate caution when touching central code. Larger diffs measured via code churn, cyclomatic complexity, and more test files take longer to review. Diffs with C++ code take longest to review and with Python code the shortest amount of time.

⁴ The variable `php` represents Hack, which is syntactically derived from PHP.

Diffs with more reviewers take less time (since code review ends after the first accept). Diffs involving files with many authors take the same amount of time to review, perhaps because it may be easier to find a reviewer who has modified the file and is available. Surprisingly, diffs modifying files with past SEVs take less time to review, perhaps because thorough post-SEV investigation leaves reviewers more familiar with the affected files.

For H2, the main findings are that code with higher fraction of AI spans is reviewed more quickly, with controls showing reduced time with author experience and increased with centrality, size, and cyclomatic complexity.

At first glance the faster review of AI diffs is surprising because AI diffs are also larger (more churn, more files), and larger diffs normally take longer to review. The two effects are separated in the model: churn, number of files, and complexity each carry their own positive coefficients on review time, so the negative *AI spans* coefficient is the association that *remains after* holding change size fixed. In other words, among diffs of comparable size and centrality, those containing more AI-landed code are reviewed somewhat faster – consistent with AI code being concentrated in more routine or boilerplate spans (e.g., test files) that reviewers can scan quickly.

We note that these associations should be interpreted cautiously. Because our study is observational and AI-landed code tends to appear in less central, lower-risk contexts, the reported associations between AI code and faster review times may partly reflect *where* AI is used rather than a direct beneficial effect of AI itself.

4.3 AI impact on production faults, *i.e.* SEVs

H3₀ : *The landing of AI suggestions does not increase the probability of a SEV.* Some prior work has found deterioration of code quality associated with the use of AI tools [58, 21, 19, 60]. We investigate if such deterioration can be observed in our study.

Model Fit. As discussed in the methods section, the rarity of SEV events suggests the use of Tjur’s coefficient of determination (an analog to R^2) which is the difference of model-predicted probabilities for the two outcomes. We report relative Tjur’s analog to $R^2 = 80\%$, or, in other words, that difference represents 80% of the overall probability of a SEV.

Control variables show that, as expected, diffs modifying files with past SEVs, of higher cyclomatic complexity, coordination needs and dependencies (measured via past authors and file centrality), and more reviewers and review comments are associated with higher SEV rates. SEV rate does not depend on engineers’ tenure or past changes to modified files.

Having suggested changes is associated with lower SEV rates, perhaps reflecting that more thorough code reviews prevent some SEVs. Size of the diff (number of files or churn) does not help predict the risk of a SEV, nor having changes after accept, or having test files. Diffs with C++ and PHP files have elevated SEV rate while diffs with Python files have lower rates.

5 Threats to Validity

Generalizability. The findings are specific to Meta’s unique environment, monolithic repository, and internal tooling, which may limit their direct applicability to other organizations or open-source projects. However, our study does cover a very wide swath of software

We find that, *after adjusting for context*, a higher fraction of AI-generated code within a diff is associated with a slightly lower probability of causing a SEV. This protective association is context-adjusted: AI-heavy diffs are also larger and touch higher-risk code, so their *raw* SEV rate is not lower, and the negative association emerges only once size and centrality are controlled. The observed association therefore does not by itself demonstrate that AI *reduces* SEV risk. A cautious conclusion is that, once context is accounted for, the fraction of AI-generated code is not associated with an increase in SEVs at Meta.

engineering. The software system covers millions of lines of code and ten thousand developers who are both colocated and working at multiple locations across the world. We also cover a wide range of domains from user facing social network products and virtual and augmented reality projects to software engineering infrastructure, such as calendar, task, and release engineering tooling. LLMs used to implement AI tools are widely available and predate agentic solutions.

The particular infrastructure used to track the provenance of AI code may not be broadly available, but it can be implemented in other companies or for open source projects. The open-sourcing of popular AI authoring tool CoPilot [56] may enable such efforts.

The rapid evolution of GenAI tools means that the “true estimate” of AI’s impact is a moving target, making this study a snapshot of its effect at a particular point in time. The observation window is roughly two months, so seasonal effects, tool-rollout and novelty effects, and mid-window model updates could in principle influence the estimates. Two factors limit this risk: CODECOMPOSE had been rolled out companywide well before our window (so it is not a novelty period), and our prior-year replication over the same calendar span with the same developers yields nearly identical coefficients, indicating the relationships are stable rather than artifacts of a particular short interval. Still, a longer, multi-season window would strengthen the temporal generalizability of the results. In particular, our findings reflect pre-agentic, completion-based AI assistance (CODECOMPOSE); the impact patterns of newer agentic AI systems that autonomously plan and execute multi-step coding tasks may differ substantially.

Scope of Analysis. Our analysis differentiates developers primarily by experience (tenure) and does not examine differences across departments or product areas (except on the smaller subsamples as noted below), as cross-team comparisons require additional organizational metadata and raise privacy considerations. Importantly, tenure here measures time *at Meta*, not total industry experience or general programming skill. It is best read as a proxy for familiarity with Meta’s codebase, tooling, and norms rather than for seniority; the non-monotonic peak at 3–4 years may therefore reflect a point at which engineers have enough codebase familiarity to apply AI confidently while still working on code amenable to completion, and it may partly stand in for correlated factors such as role, team, or prior experience that we do not observe.

Selection Bias. Chattributes coverage of roughly half of diffs is driven by IDE and workflow instrumentation, but if coverage correlated with unobserved diff or developer properties, the analyzed sample could differ from the full population in ways our controls do not capture. Similarly, engineers self-select where to invoke and accept AI, so the landing model describes AI use *as it occurs* rather than under random assignment. Both forms of selection limit causal interpretation, which is why we frame all outcomes as associations and rely on the two-stage confound-control design rather than a direct AI-vs-non-AI comparison.

Construct Validity. The potential set of constructs used in the mining software repositories community is enormous. We have selected a straightforward set of measures widely used in bug prediction literature [63, 3], *e.g.*, prior defects and churn. We also use collaboration measures relying on software supply chains and centrality [5, 67, 62]. We select a diverse set of variables that represents distinct aspects of the authors, diffs, codebase, and historic events to ensure that we do not miss aspects that might affect outcomes of interest.

A more fundamental construct concern is that Chattributes measures the *provenance of landed characters*, not the cognitive contribution of AI. The measure can overstate AI's role when AI-attributed bytes are code the developer would have typed anyway (imports, common idioms), and can understate it when a rejected suggestion still shapes what the developer types next, or when AI helps via API discovery or pattern hints that leave no attributed bytes. It thus captures "AI code that survived to landing" rather than "value added by AI," and our findings should be read accordingly. Even so, landed provenance is a substantial improvement over acceptance rates or tool-availability proxies, and it is precisely the quantity relevant to what ends up in the codebase and can cause SEVs.

Internal Validity. As is commonly necessary for statistical models in software engineering [7, 41, 40, 42, 43], we log-transformed variables with highly skewed distributions and inspected residuals to check for nonlinear or non-monotone relationships. We also investigated interactions among variables, but none satisfied our requirements of improving the prediction and being easily interpretable.

While some of the variables are correlated, we found that we could pick variables from within the highly correlated clusters without negatively affecting model performance. Since some of the variables are categorical, we check for collinearity using Generalized Variance Inflation Factor (GVIF) [17] implementation in R [18], which is particularly suited for categorical variables (as correlation is generally not the most informative approach for categorical variables). In all three models, $GVIF^{\frac{1}{2 \times df}} < 1.8$: under the suggested threshold of 2 [49]. This threshold corresponds to the regular VIF being less than 4 for numerical variables.

The observational nature of the data makes it difficult to argue that observed relationships are causal even though we selected variables that have values measured before (in time) the outcomes. While we capture a number of important context variables, it is challenging to explain counter-intuitive coefficients, *e.g.*, why Sev rate increases with author centrality? A standard statistical explanation of latent variable, *i.e.*, an unobserved property of a diff that brings on more experienced engineers also makes it more risky, is very difficult to communicate.

To ensure that models are representative of the code development at Meta we have done various robustness analyses. First, we checked if the models used to estimate the impact of AI accurately describe review time and chances of a SEV by fitting them over the same period during the prior year with the same developers; the code-complexity, churn, and author-experience controls reproduce with nearly identical direction and magnitude, confirming that our model structure captures stable relationships. Because development activity is seasonal (with evaluation periods and year-end holidays being substantially different), the prior-year comparison provides a better baseline than an immediately preceding period and confirms that our model structure captures stable relationships. Additionally, over 80% of changes in our sample are authored from North America (about half from the US Pacific zone alone), whose working hours largely overlap, so unobserved reviewer availability and timezone confounds are likely to have limited impact on the review time model.

Thresholds and the binary definition. It is reasonable to object to our definition of landed AI code whereby landing even a single byte traced to AI puts the entire diff into AI class, treating a one-line and a large AI contribution alike. Two aspects of our design mitigate this. First, the binary definition is used only as the response of the authoring model (H1); the outcome models (H2, H3) use the *continuous AIs spans* (and, equivalently, *AIbytes*) fraction, so the magnitude of AI contribution, not merely its presence, drives those estimates. Second, for H1 we ran experiments trying thresholds up to 100 bytes and found that the same variables were statistically significant and pointing in the same direction (in fact, increasing the threshold improved model fit). This shows that the context factors affecting the chances of landing AI code stay the same even when we consider a diff AI-supported only when at least 100 bytes of AI-generated code lands.

Modeling assumptions. The sample of diffs is authored by over 10K engineers and outcomes may vary among them even after adjusting for author centrality, tenure, and past changes. To account for this, we fitted mixed-effects models (random effects for authors) on several random subsamples of authors – ranging from a few hundred to a few thousand – using *lme4* [10] in *R*, as the full logistic mixed-effects model (*glmer*) did not converge on the complete dataset. We also explored author and organization fixed effects, but the very large number of parameters prevented convergence on the full data. In all subsample analyses – across different random author samples and different organizations – the fixed effects were similar to the reported results with no noticeable changes in the direction of coefficients. For clarity we present models without author random effects.

Ethics and Privacy. The study uses telemetry that reflects individual developer activity. Meta handled the data under applicable privacy and data-governance policies; authorized employees analyzed them on internal systems, and we report only aggregate results that do not identify individual developers. We report no per-developer results and share only relative effect sizes for sensitive quantities such as SEV probability. We caution that our practitioner recommendation to build provenance infrastructure (P1) carries responsibilities: in jurisdictions with GDPR-like requirements, developer-level provenance data constitutes personal data, and organizations should pair such infrastructure with purpose limitation, aggregation, access controls, and transparency to developers, and should avoid using it for individual performance evaluation.

6 Literature and Discussion

Productivity and speed. A growing consensus is that GenAI tools increase the speed and volume of code generation [50, 51, 59, 9, 33], though the magnitude varies (21–65% speed improvements depending on the study). However, this consensus is contradicted by an RCT with experienced OSS developers where AI actually increased task completion times despite developers perceiving themselves as faster [4]. GitClear notes that AI assistants “do beget more lines” of code [21], and Weisz et al. [60] found only small perceived productivity improvements.

Our findings relate to this prior work in several ways. Landed AI code is associated with larger churn, more files, and higher cyclomatic complexity as in *e.g.*, [60, 59, 4, 9]. We conjecture that the causality might go in the opposite direction: larger diffs may provide more opportunities for suggestions, so we would observe positive association of more code with AI even though the nature of the diff may drive this difference, not the use of AI.

This is a concrete example of how context-dependence of AI application can mislead: if studies compare output volume without accounting for the fact that AI is disproportionately applied to larger, more complex tasks, they will attribute to AI what is actually a property of the task itself. We recommend future work conducting specially designed experiments to establish the direction of causality. Results may also be reversed depending on what outcome is measured, as when perceived improvements contrasted with measured times [4]. All studies suggest that the use of AI varies with context, further supporting our central finding that contextual controls are essential for isolating AI's true effects.

Code quality. The consensus on code quality is primarily negative. Uplevel [58] found a 41% higher bug rate with Copilot access. GitClear [21] documented eroding quality: increased copy/pasted and duplicated code blocks, and 20–25% higher churn. DORA [19] noted a 7.2% decrease in delivery stability per 25% increase in AI adoption.

Weisz et al. [60] reported “acceptable” but imperfect quality requiring user effort to correct, and Liang Yu et al. [33] found code generation tasks required the most adjustment. In contrast, Cui et al. [9] found no quality decrease at Microsoft, and Weber et al. [59] found no significant differences in objective code quality.

GenAI excels at routine or boilerplate tasks but struggles with complex, domain-specific code [33, 60]. Our increased landing rates for test code and less central files support this. We did not find increased SEV rates (in contrast to [58]), and lower AI landing rates for highly central files with past SEVs suggests limited trust in AI code [60].

Self-regulation vs. policy compliance. A natural alternative explanation for our H1 finding is that the organization restricts AI use on critical code, and our model merely captures compliance with such a policy. This is not the case: Meta actively encourages AI adoption across all development contexts with no restrictions based on code criticality. The observed pattern – less AI in central, high-risk code – is therefore emergent self-regulation by engineers who exercise professional judgment about where AI suggestions are appropriate. This distinction matters: it suggests that the context-dependence of AI use is a fundamental property of how engineers interact with AI tools, not an artifact of organizational governance, and is likely to manifest in other organizations as well. We caution, however, that the centrality signal underpinning this reading is small in magnitude and sensitive to how centrality is measured; the dominant context factors are change size and programming language. The self-regulation interpretation should therefore be treated as a modest, completion-era tendency that may weaken as AI tooling shifts from inline completion toward more autonomous, agentic workflows.

Developer choice vs. model capability. A second alternative reading of H1 is that the lower AI landing rate in central, complex code (and the lower rate for C++ relative to Hack and Python) reflects not developer caution but limits of the model: CODECOMPOSE may simply produce weaker suggestions on complex or performance-critical code, so fewer of them survive to landing. Our data are consistent with both mechanisms and we cannot fully separate them, because both “the engineer declined the suggestion” and “the suggestion was poor” manifest as unlanded AI code. Two observations suggest developer choice plays a real role: landing rates drop with file centrality and past-SEV history even after controlling for size and complexity, and the drop appears across languages rather than only where model quality is weakest. Nonetheless, disentangling suggestion quality from developer selection – for example by pairing landing data with suggestion-level accept/reject and edit-distance signals – is important future work, and applies equally to the language differences we observe.

The “tail wags the dog” problem. Our findings highlight a methodological risk that extends beyond this study. When AI is applied non-uniformly, factors that predict AI use (such as change size, centrality, and programming language) also predict outcomes of interest

(such as review time and fault rates). A study that regresses an outcome on AI use without controlling for these shared predictors will attribute the predictors' effects to AI. For example, if larger changes both attract more AI use and take longer to review, a model omitting change size will incorrectly conclude that AI increases review time. We address this by first modeling AI landing probability (H1) to identify its predictors, then including those predictors as controls in outcome models (H2, H3). We urge future studies to adopt this two-stage approach: identify what drives AI application before estimating what AI drives.

What transfers to agentic tools. Because our study measures pre-agentic, completion-based assistance, it is worth separating findings likely to be specific to that paradigm from those likely to generalize. Two results are tied to the completion setting and may weaken under agentic workflows: the specific magnitude of landing coefficients, and the review-time association, since agentic tools generate larger, multi-file changes whose review dynamics differ from line-level completions. The most transferable finding is the *methodological* one – that AI application is context-dependent and must be modeled before its effects are estimated; if anything, agentic tools that act more autonomously make confound control more, not less, important. We also expect the qualitative direction of the context effects (less AI in central, high-risk code; more in test and boilerplate code) to persist, though the mechanism may shift from per-suggestion human acceptance toward human approval of agent-proposed plans. Establishing whether these patterns hold under agentic assistance requires the same provenance infrastructure applied to agent-authored changes, which is a direct extension of this work.

7 Conclusion and Contribution

The study investigates the landing of code from pre-agentic, completion-based AI tools. AI-generated code is present in 52% of diffs at Meta and nearly 8% of all modified and landed bytes originate from AI suggestions.

Our central finding is that AI is not applied uniformly: its use is strongly context-dependent. AI code is less likely to land in central or risky code areas and more likely in larger diffs, test files, and for engineers with 3–4 years of tenure (contradicting prior literature on junior engineers). Diffs with landed AI code were reviewed more quickly and were associated with fewer SEVs, but both associations are confounded by AI code appearing in less central contexts. These findings underscore that naive comparisons of AI vs. non-AI work risk misleading conclusions, and that character-level provenance tracing (Chattributes) is essential infrastructure for rigorous evaluation.

Lessons Learned. First, *landing rates should replace acceptance rates* as the primary measure of AI tool effectiveness, since accepted code is frequently edited or removed before landing. Second, *context must be controlled*: naive before/after or with/without comparisons confound AI effects with task properties. Third, *AI use is non-uniform and mildly self-regulated*: at Meta, developers use somewhat less AI in central, high-risk code, and after adjusting for context we find no associated increase in SEVs. This tendency is emergent (Meta has no policy discouraging AI use on critical code), but the centrality signal is small and measure-sensitive, so we frame it as a modest, completion-era pattern rather than strong evidence of deliberate risk avoidance.

Actionable Guidance for Researchers. (R1) *Include covariates of AI use.* Studies must control for code centrality, change size, author experience, and programming language – all systematically associated with AI application (H1). Without these controls, outcome models (H2, H3) confound context effects with AI effects.

(R2) *Guard against “tail wags the dog” inference.* Model AI application probability *first* (as in H1), then include its predictors as controls in outcome models (H2, H3). Factors related to the outcome – such as change duration, size, and type – must be controlled rather than attributed to AI.

(R3) *Distinguish tool availability from tool application.* Comparing groups with and without AI access measures a bundled effect; our diff-level measurement of actual AI code presence (H1) reveals context-dependent variation invisible to group-level designs.

(R4) *Do not assume junior engineers benefit most.* Our finding (H1) that engineers with 3–4 years of tenure have the highest AI landing rates contradicts the common assumption that AI primarily helps inexperienced developers. Studies should measure tenure effects directly rather than assuming a monotonic relationship.

Actionable Guidance for Practitioners. (P1) *Invest in provenance infrastructure* that tracks AI-suggested code from editor to landing (Section 2.2) – without it, productivity claims rest on acceptance rates that overstate AI contribution. (P2) *Interpret AI productivity reports with caution:* our H1 results show that increased output may reflect AI being preferentially applied to larger or less complex tasks, not AI itself increasing output. (P3) *Monitor rather than pre-emptively restrict:* H1 and H3 suggest engineers already apply somewhat less AI in higher-risk code and that, after adjusting for context, AI use is not associated with more SEVs; monitoring landing patterns by centrality and change size can track this directly rather than imposing blanket restrictions. (P4) *Do not target AI tooling solely at junior engineers:* H1 shows mid-tenure engineers (3–4 years) have the highest AI landing rates, suggesting that sufficient codebase familiarity helps engineers apply AI effectively. Training and tooling strategies should account for this non-monotonic relationship.

Our findings would benefit from triangulation with qualitative evidence such as developer interviews. Open questions include why acceptance rates diverge from landing rates for junior developers and how AI assistance could be improved across more contexts and languages.

Data Availability Statement

Applicable privacy and confidentiality requirements prevent public release of the underlying telemetry. We provide detailed descriptions of all variables, model specifications, and statistical procedures to support transparency and reproducibility of the methodology.

References

- 1 Rui Abreu, Vijayaraghavan Murali, Peter C. Rigby, Chandra Maddila, Weiyan Sun, Jun Ge, Kaavya Chinniah, Audris Mockus, Megh Mehta, and Nachiappan Nagappan. Moving faster and reducing risk: Using llms in release deployment. *arXiv preprint*, abs/2410.06351, 2024. doi:10.48550/arXiv.2410.06351.
- 2 Amazon. At amazon, some coders say their jobs have begun to resemble warehouse work, 2025. URL: <https://www.msn.com/en-in/technology/artificial-intelligence/at-amazon-some-coders-say-their-jobs-have-begun-to-resemble-warehouse-work/ar-AA1FrEb4>.
- 3 Peter C. Rigby Andrey Krutauz, Tapajit Dey and Audris Mockus. Do code review measures explain the incidence of post-release defects? *International Journal of Empirical Software Engineering*, 25:3323–3356, 2020. doi:10.1007/S10664-020-09837-4.
- 4 Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity, 2025. URL: https://metr.org/Early_2025_AI_Experienced_OS_Devs_Study.pdf.

- 5 Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009. doi:10.1109/ISSRE.2009.17.
- 6 Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011. doi:10.1145/2025113.2025119.
- 7 B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- 8 Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, the structure of work dependencies and their impact on failures. *IEEE Transactions on Software Engineering*, 2009.
- 9 Zheyuan Kevin Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The effects of generative ai on high skilled work: Evidence from three field experiments with software developers. *Available at SSRN 4945566*, 2024.
- 10 Bates D and Maechler M. Package lme4, 2009. URL: <http://lme4.r-forge.r-project.org>.
- 11 Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *2009 16th Working Conference on Reverse Engineering*, pages 135–144. IEEE, 2009. doi:10.1109/WCRE.2009.19.
- 12 Thomas Dohmke, Marco Iansiti, and Greg Richards. Sea change in software development: Economic and productivity analysis of the ai-powered developer lifecycle, 2023. arXiv: 2306.15033.
- 13 Carsten F Dormann, Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gabriel Carré, Jaime R García Marquéz, Bernd Gruber, Bruno Lafourcade, Pedro J Leitão, et al. Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):27–46, 2013.
- 14 Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Maddila, Vijayaraghavan Murali, et al. Multi-line ai-assisted code authoring. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 150–160, 2024.
- 15 M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976. doi:10.1147/SJ.153.0182.
- 16 David Firth. Bias reduction of maximum likelihood estimates. *Biometrika*, 80(1):27–38, 1993.
- 17 John Fox and Georges Monette. Generalized collinearity diagnostics. *Journal of the American Statistical Association*, 87(417):178–183, 1992.
- 18 John Fox and Sanford Weisberg. *An R companion to applied regression*. Sage publications, 2018.
- 19 Google Cloud. Devops research and assessment (DORA) report, 2024. URL: https://services.google.com/fh/files/misc/2024_final_dora_report.pdf.
- 20 T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.
- 21 William Harding and Matthew Kloster. Coding on copilot: 2023 data suggests downward pressure on code quality. *GitClear*. Accessed: Mar, 1:58–99, 2024.
- 22 James Herbsleb and Audris Mockus. Formulation and preliminary test of an empirical theory of coordination in software engineering. In *2003 International Conference on Foundations of Software Engineering*, Helsinki, Finland, October 2003. ACM Press. URL: <http://dl.acm.org/authorize?787510>.
- 23 Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 319–321, 2022. doi:10.1145/3510454.3522684.
- 24 Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014. doi:10.1145/2568225.2568271.
- 25 Nick Jiang, Anshul Ramachandran, Mehul Raheja, and Michael Li. The golden metrics: Characters per opportunity and percentage code written, 2024. URL: <https://windsurf.com/blog/golden-metrics-characters-per-opportunity-percentage-code-written>.

- 26 Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 2013. doi:10.1109/TSE.2012.70.
- 27 Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.
- 28 Gary King and Langche Zeng. Logistic regression in rare events data. *Political Analysis*, 9(2):137–163, 2001.
- 29 Pavneet Singh Kochhar, D. Lo, Julia L. Lawall, and Nachiappan Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66:1213–1228, 2017. doi:10.1109/TR.2017.2727062.
- 30 David Kreitmeir and Paul A. Raschky. The heterogeneous productivity effects of generative ai, 2024. doi:10.48550/arXiv.2403.01964.
- 31 Andrey Krutauz, Tapajit Dey, Peter Rigby, and Audris Mockus. Do code review measures explain the incidence of post-release defects? case study replications and bayesian networks. In *ESEC/FSE’20 Journal First Track*, November 2020.
- 32 Matthew Li, Mehul Raheja, and Douglas Chen. Why you should not trust all the numbers you see, 2023. URL: <https://windsurf.com/blog/code-llm-eval-why-you-should-not-trust-all-the-numbers-you-see>.
- 33 Jenny T Liang, Chenyang Yang, and Brad A Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13, 2024. doi:10.1145/3597503.3608128.
- 34 Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11–19. IEEE, 2017. doi:10.1109/ESEM.2017.8.
- 35 T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976. doi:10.1109/TSE.1976.233837.
- 36 Daniel McFadden. Quantitative methods for analysing travel behaviour of individuals: some recent developments. In *Behavioural travel modelling*, pages 279–318. Routledge, 2021.
- 37 Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189, 2016. doi:10.1007/S10664-015-9381-9.
- 38 Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, 2008. doi:10.1145/1453101.1453106.
- 39 Meta. Glean: Code indexing and search. <https://glean.software/>, 2025.
- 40 Audris Mockus. Software support tools and experimental work. In V Basili and et al, editors, *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, volume LNCS 4336, pages 91–99. Springer, 2007. doi:10.1007/978-3-540-71301-2_25.
- 41 Audris Mockus. Missing data in software engineering. In J. Singer et al., editor, *Guide to Advanced Empirical Software Engineering*, pages 185–200. Springer-Verlag, 2008. doi:10.1007/978-1-84800-044-5_7.
- 42 Audris Mockus. Organizational volatility and its effects on software defects. In *ACM SIGSOFT / FSE*, pages 117–126, Santa Fe, New Mexico, november 7–11 2010. doi:10.1145/1882291.1882311.
- 43 Audris Mockus. Engineering big data solutions. In *ICSE’14 FOSE*, 2014. URL: <https://dl.acm.org/authorize?N14216>.
- 44 Audris Mockus, Rui Abreu, Peter C. Rigby, David Amsallem, Parveen Bansal, Kaavya Chinniah, Brian Ellis, Peng Fan, Jun Ge, Bingjie He, Kelly Hirano, Sahil Kumar, Ajay Lingapuram, Andrew Loe, Megh Mehta, Venus Montes, Maher Saba, Gursharan Singh, Matt Steiner, Weiyan Sun, Siri Uppalapati, and Nachiappan Nagappan. Leveraging risk models to improve productivity for effective code un-freeze at scale. *ACM Transactions on Software Engineering and Methodology*, 2025. URL: <https://api.semanticscholar.org/CorpusID:276950332>.

- 45 Audris Mockus, Nachiappan Nagappan, and T Dinh-Trong, Trung. Test coverage and post-verification defects: A multiple case study. In *International Conference on Empirical Software Engineering and Measurement*, Lake Buena Vista, Florida USA, October 2009. ACM.
- 46 Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, april–june 2000. doi:10.1002/BLTJ.2229.
- 47 Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, Nachiappan Nagappan, and Peter C Rigby. Ai-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1066–1085, 2024. doi:10.1145/3643774.
- 48 Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005. doi:10.1145/1062455.1062514.
- 49 Ramzi W Nahhas. *Introduction to Regression Methods for Public Health Using R*. CRC Press, 2024.
- 50 Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. How much does ai impact development speed? an enterprise-based randomized controlled trial. *arXiv preprint*, 2024. arXiv:2410.12944.
- 51 Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint*, 2023. doi:10.48550/arXiv.2302.06590.
- 52 Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998. doi:10.1145/268411.268421.
- 53 Liam Proven. Ai-generated bug reports are seriously annoying for developers, 2024. URL: https://www.theregister.com/2024/01/04/aiassisted_bug_reports_make_developers/.
- 54 Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35, 2014.
- 55 Dag I.K. Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: a comparative case study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, ESEM'12*, pages 107–110, New York, NY, USA, 2012. ACM. doi:10.1145/2372251.2372269.
- 56 Microsoft: VS Code team. Vs code: Open source ai editor, 2025. URL: <https://code.visualstudio.com/blogs/2025/05/19/openSourceAIEditor>.
- 57 Tue Tjur. Coefficients of determination in logistic regression models—a new proposal: The coefficient of discrimination. *The American Statistician*, 63(4):366–372, 2009. doi:10.1198/tast.2009.08210.
- 58 Uplevel. Can generative ai improve developer productivity?, 2025. URL: <https://resources.uplevelteam.com/gen-ai-for-coding>.
- 59 Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. Significant productivity gains through programming with large language models. *Proceedings of the ACM on Human-Computer Interaction*, 8:1–29, 2024. doi:10.1145/3661145.
- 60 Justin D. Weisz, Shraddha Kumar, Michael J. Muller, Karen-Ellen Browne, Arielle Goldberg, Katrin Ellice Heintze, and Shagun Bajpai. Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2025.
- 61 Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13:539–559, 2008. doi:10.1007/S10664-008-9082-8.
- 62 Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. Categorizing bugs with social networks: a case study on four open source software communities. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1032–1041. IEEE, 2013. doi:10.1109/ICSE.2013.6606653.
- 63 Yunhua Zhao, Kostadin Damevski, and Hui Chen. A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35, 2023. doi:10.1145/3567550.

- 64 Minghui Zhou and Audris Mockus. Developer fluency: Achieving true mastery in software projects. In *ACM SIGSOFT / FSE*, pages 137–146, Santa Fe, New Mexico, november 7–11 2010. doi:10.1145/1882291.1882313.
- 65 Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29, 2022. doi:10.1145/3520312.3534864.
- 66 Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring github copilot’s impact on productivity. *Communications of the ACM*, 67(3):54–63, 2024. doi:10.1145/3633453.
- 67 Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540, 2008. doi:10.1145/1368088.1368161.