Autonomous Refactoring for Perfective Maintenance at Scale: The Agentic RACER System and its Empirical Evaluations

Audris Mockus*
Arjun Singh Banga
Payal Bhuptani
Sky Jazayeri
Jun Ge
audris@meta.com
arjunsingh@meta.com
pbhuptani@meta.com
jazi@meta.com
jakege@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

Gurinder Grewal
Emma Lai
Jianmin Li
Feng Liang
Rishab Mangla
Sergey Parshin
gurinderg@meta.com
jiaolai@meta.com
jianminli@meta.com
liangfeng@meta.com
rishabmangla@meta.com
parshin@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

Vishal Parekh
Peter C Rigby†
Gursharan Singh
Matt Steiner
Nachiappan Nagappan
vparekh@meta.com
peter.rigby@concordia.ca
gurshi@meta.com
mattsteiner@meta.com
nnachi@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

Abstract

Automating software development is an important aspect of software engineering. Generative Artificial Intelligence (GenAI) Agents that control multiple tools and use feedback with some autonomy to iteratively solve problems are emerging as promising tools for increased software production. To achieve autonomous software maintenance with minimal human involvement, such agents must not only solve human-posed problems but also decide which problems to address. We report our experiences developing a GenAI agent that autonomously formulates, prioritizes, and completes a limited set of code improvement tasks enterprise-wide, and we evaluate this agent from two perspectives: first, developing a methodology to estimate effort savings from such autonomous agents by comparing time spent on agent-introduced tasks and agent maintenance efforts against manual execution; and second, comparing the impact of manual versus agent-driven code improvements on future codebase maintenance. Our results demonstrate significant effort savings, even after accounting for effort in agent development, and a Difference-in-Differences (DiD) analysis found that automated improvements had an effect on the reduction of future maintenance effort that was as strong as or stronger than manual improvements confirming the feasibility of autonomous maintenance. While the relaxed completion and timeliness needs of perfective maintenance make it an ideal candidate for autonomy, a degree of autonomy may be achievable in other areas of software development as well.

CCS Concepts

• **Software and its engineering → Software creation and management**; • **General and reference → Empirical studies**.

Keywords

autonomous agents, software maintainability

ACM Reference Format:
Audris Mockus, Arjun Singh Banga, Payal Bhuptani, Sky Jazayeri, Jun Ge, Gurinder Grewal, Emma Lai, Jianmin Li, Feng Liang, Rishab Mangla, Sergey Parshin, Vishal Parekh, Peter C Rigby, Gursharan Singh, Matt Steiner, and Nachiappan Nagappan. 2026. Autonomous Refactoring for Perfective Maintenance at Scale: The Agentic RACER System and its Empirical Evaluations. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3803437.3805211

1 Introduction

The advent of Generative AI (GenAI) is rapidly changing software development. Large Language Models (LLMs) can directly support synchronous tasks such as coding and documentation, and they can also facilitate interaction with software artifacts for information retrieval, symbol navigation, and testing. Prior experiments suggest that, for complex tasks, tool-using agents can substantially improve LLM performance, with gains ranging from 18.1% to 250% [36]. Although the term *agent* is defined broadly [25], we focus on systems that use software engineering tools, exhibit a nontrivial degree of autonomy, and can operate asynchronously.

As with any automation, the productivity impact of an agent depends on its autonomy. A chatbot typically reacts to a user's prompt. Code completion must infer when assistance is useful and then propose a suggestion. Test or patch generation can be more autonomous, for example when triggered by a build system after a change lands or when a test fails. In each of these settings, an engineer still initiates, supervises, or iteratively corrects the agent, which consumes human time and constrains the attainable productivity gains. We therefore ask whether an agent can act

*Mockus is also a professor at University of Tennessee, Knoxville, USA.
†Rigby is also a professor at Concordia University in Montreal, QC, Canada.

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/3803437.3805211

asynchronously, decide what work should be done, select when to do it, and then execute it with limited human involvement. While full autonomy is unrealistic for many software tasks today, we argue that it is achievable for a bounded class of maintenance activities.

We investigate this question in the context of *perfective maintenance*. We study perfective maintenance because it is essential for preserving long-term maintainability as code evolves, yet it is often deprioritized. Codebases naturally accumulate technical debt and become harder to maintain [9], in part because short-term delivery pressures bias decisions away from maintainability [29]. Prior reports suggest that 4% to 14% of development effort is devoted to perfective maintenance [19]. These activities are often well-scoped, metric-driven, and amenable to automation, for example via complexity, coverage, and risk signals. They are also typically not immediately compulsory for system correctness, which makes them suitable candidates for asynchronous autonomy. If automated effectively, such work can free engineers to focus on more complex tasks, particularly because many engineers do not prefer to spend time on routine code improvement [33].

In this work, we build an agentic system that attempts to provide enterprise-wide, end-to-end autonomy for a limited set of code improvement tasks. Based on nine months of deployment data, the system discovers and prioritizes opportunities, materializes them as tasks, generates patches, and routes changes for human review and landing. At present, human review remains the primary control point, but the system is designed to enable further automation over time. The agent prioritizes improvement work using multiple notions of risk, including signals related to complexity, coverage, and centrality. We call the system Risk Aware Code Enhancement via Refactoring (RACER).

Research questions. We structure our study around the following questions.

RQ1 (System feasibility): To what extent can perfective maintenance be executed end-to-end by an agentic system, including opportunity discovery, task selection, patch generation, and submission for landing, at enterprise scale while remaining compatible with existing developer workflows and governance constraints?

RQ2 (Net productivity impact): What is the net change in human effort when using autonomous perfective maintenance, accounting for engineer time on agent-generated work and the engineering effort required to build and maintain the agent?

RQ3 (Downstream maintenance impact): How does autonomous refactoring compare to human refactoring in its effect on subsequent maintenance effort of the modified code?

Contributions. First, we present the design and evaluation of RACER, an agentic system that operationalizes end-to-end autonomy for a bounded set of perfective maintenance tasks. RACER integrates with existing developer infrastructure for task management, version control, code review, and continuous integration, and it employs quality-control gates to produce review-ready changes.

Second, we propose an evaluation methodology for estimating effort savings from autonomous agents by comparing the human effort that would be required to perform comparable work manually to the observed effort associated with agent-produced work, while explicitly accounting for the engineering effort needed to develop and maintain the agent.

Third, we evaluate the downstream impact of autonomous code improvements by comparing subsequent maintenance effort for files improved by humans versus those improved by RACER using a Difference-in-Differences design.

Finally, we report practical engineering challenges encountered in operating RACER at scale, including tooling and model reliability, latency, and reviewer capacity constraints. In particular, because landing remains gated by human review, increasing autonomous output can induce reviewer overload, the so-called wall of diffs, which constrains achievable throughput.

The rest of the paper is organized as follows. In Section 2, we describe the software development context, including the Confucius agentic framework, to motivate how RACER integrates with existing workflows and to specify the empirical setting. In Section 3, we describe the design of RACER, focusing on the features most relevant to autonomy at scale. Section 4 presents the empirical study design for evaluating autonomous maintenance. Sections 5 and 6 present two empirical studies that evaluate net effort savings and downstream maintenance impact, respectively.

2 Context: Software Development at Meta

We first describe the key aspects of the software development process and tools needed to explain the context in which the agent was developed and the data acquisition and analysis of its ultimate performance.

At Meta, we develop software for both our servers and client devices, including specialized hardware devices. This approach allows for fine-grained control over versioning and configurations, and enables quick pushes of new code updates to production. Before any code is deployed, it undergoes rigorous testing, including peer review, in-house user testing, automated tests, and canary tests. Once the code is deployed, engineers closely monitor logs to identify potential issues.

At a high-level, the same systems for tracking tasks and code changes (via version control system) are used company-wide and code is maintained in a single mega-repository. Code reviews, issues, and code changes are tightly integrated as diffs via phabricator [1].

Data associated with the usage of these tools is tracked in a data warehouse that has SQL access. In addition to logs of these tools, numerous code metrics are also computed and stored in SQL databases suited for analytics. Such logs and other metrics are regularly used to asses the efficiency of these and other tools and/or practices that are both ongoing or are considered for a wider deployment. In addition, Meta has a company-wide search engine that, as regular internet search engines, allows search for any keywords and, in addition, has special means to search for source code as well.

In addition to using IDEs, such as VS Code, and numerous other testing and deployment tools, we use central a Continuous Integration system. Developers, after completing and testing their diff submit it for review, creating a patch representing the initial version of the code. Reviewers can suggest improvements, leading to additional revisions until the diff is either approved and incorporated into the codebase or rejected. This process promotes high coding standards, helps detect flaws, and spreads knowledge throughout the organization.

[1] http://phabricator.org

2.1 Confucius

Confucius is a programming framework that simplifies agent development with primitives and abstractions [32]. We build RACER on top of Confucius because it offers an interactive chat-like interface for user interaction. At its core, Confucius is built on Pydantic, a Python schema language for defining custom data structures. The core abstraction class is the Analect, which holds reusable logic and data for inputs, outputs, and runtime. It is similar to a function but provides logging for execution, streamlining reuse throughout the system. Several basic operators are shared among all Analects. These operators implement basic functionalities, including a JSON Parser, a Tracer that automatically collects and logs the execution trace in an internal database, and an I/O function that enables human-in-the-loop interaction.

Confucius Primitives: These are specialized Analects that implement one particular action. They can be applied across various use cases. The Analect is the abstract class of all primitives. One example is the Translator primitive, which converts natural language inputs into structured outputs, such as translating a user's query from English to a command line. Users focus on writing examples without worrying about formatting inputs and parsing results. The Translator also facilitates the translation of commands, queries, and configurations from one language to another.

Tooling DSLs: Originally, Confucius supports DSLs used in network management applications, but was extended to accommodate additional DSLs as new use cases were onboarded.

Applications: RACER is an application to address the specific domain of code improvement.

External Systems: Confucius leverages several external systems to enhance its capabilities, including LangChain [30], MetaGen API [10] for AI model access, FAISS [8] for efficient similarity search, and Instructor [27] for embedding.

3 RQ1. System Feasibility

To what extent can perfective maintenance be executed end-to-end by an agentic system?

3.1 RACER Overview

At a high level, the primary requirement for RACER is to make the perfective maintenance of the codebase autonomous (see Figure 1 for the overall architecture). This encompasses the entire process from determining what needs to be done, to task completion, quality control, and finally, landing of the diffs. Until enough experience with the operation of RACER is obtained to assure that it performs well, we want to have a human engineer in-the-loop to do a code review of the system's diffs in order to land the code. Notably, this requires the system to discover, define, and prioritize work to align with the organization's priorities: work that is is typically accomplished by developers.

3.2 High-Level Architecture

Architecturally, we separate the system into components that a) determine and prioritize perfective maintenance tasks; b) implement (and, potentially, evaluate the quality of) these tasks; c) assign the tasks for human review, and d) monitor, assess, and optimize this set of agents via an administration portal that enables obtaining insights and to track end-to-end progress and goals. These components are interconnected among themselves and use a number of tools and external systems. For example, tasks are defined based on various metrics and priorities from quality control systems and databases. The tasks and diffs are materialized using companywide tools. The code changes are recorded in version control systems and reviewer availability is obtained from code review and other systems. Apart from the task creation agent we also need a coding agent to implement code improvements, and evaluation agent and numerous lint, build, and testing tools to evaluate the quality of the produced change.

3.3 Design Goals

Given the overall complexity and lack of determinism of the entire system we followed several design goals to make it all happen.

First, we rely as much as possible on rich existing sources of data, tools, and practices to avoid reinventing the wheel and to speed implementation. Second, we define interfaces to the Agent framework in order to easily replace or use different agent frameworks. Third, we develop extensive benchmarking and evaluation framework to prevent regressions caused by rapid changes. Fourth, we ensure automatic retry and recovery mechanisms to deal with the scale and duration of processing times. Each component may have component-specific design goals described in the next section.

3.4 Key Features

RACER has several features that are to some extent innovative. First, is the ability to determine and prioritizing what needs to be done or **opportunity discovery**. Second, the ability to **integrate** with a multitude of existing systems and tools. Third, is the use of **replaceable Agentic framework** to rapidly update to latest GenAI technology and enable the use most relevant technologies depending on the component: e.g., task generation, code generation, or evaluation. RACER is also highly **extensible** with a clear structure on how engineers can extend and customize its functionality via runbooks (structured prompts).

3.5 RACER Implementation

3.5.1 Discovery Module. This component is responsible for continuously identifying improvement opportunities within the codebase. Coding standards and quality and risk indicators developed for manual code improvement efforts are currently exploited by the discovery module. In addition to company-wide and product-wide objectives (eliminate static analysis warnings, remove dead code, limit cyclomatic complexity), special needs for specific platforms or embedded systems are also included for these codebases.

3.5.2 Data Sources. Presently RACER uses four data sources to define code improvement tasks. **BigGrep**: A pattern-based code issue detection using the BigGrep service that enables finding deprecated API usage, security anti-patterns, and style violations. **DeadCode**: Helps cleanup of unused functions, variables, and includes in C++ codebases via static cross-reference analysis and integrates with build system for more accurate detection. **QualityInsight**: Obtains information on source code from the quality analysis tools and metrics dashboards used to track the progress of perfective maintenance. These dashboards provide prioritization based on impact and severity. **Cyclomatic Complexity**: Identifies overly complex functions for refactoring via function-level complexity measurement and threshold-based issue identification.

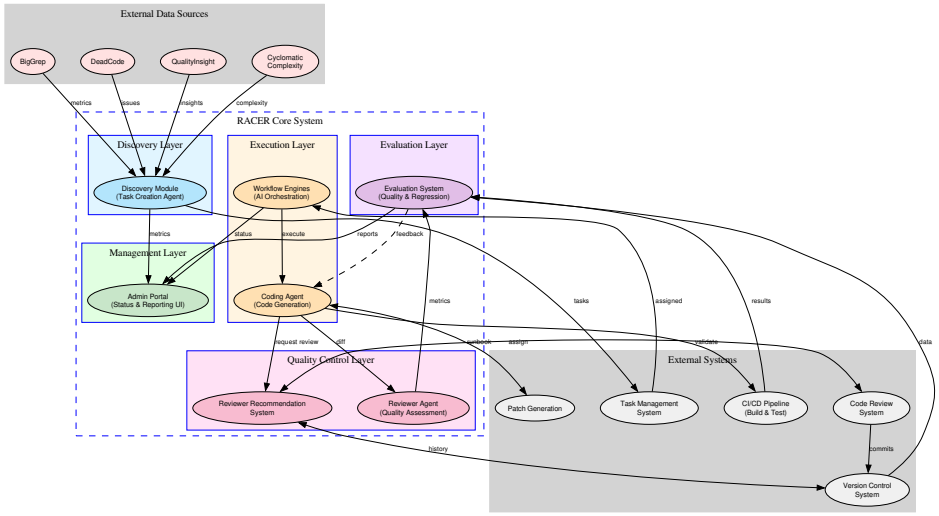

Figure 1: RACER architecture

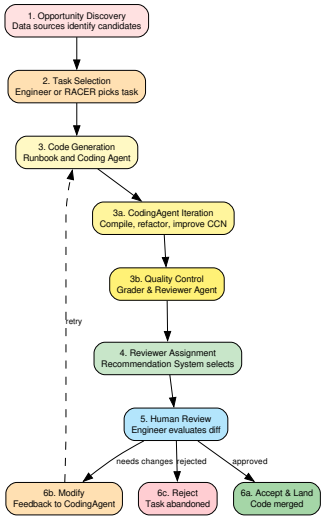

Figure 2: RACER Task Workflow

3.6 Admin Portal - Status Tracking, Reporting, and Management/Configuration

The admin portal serves two major use cases. **Onboarding/Management**: UI-based use case onboarding/enablement and UI-based team/org onboarding with Codehub — an internal portal to projects and codebases. **Tracking/Reporting**: Track the status/health of the codebase, monitor diff generation and trends, identify bottlenecks, and report progress.

3.6.1 Key Interfaces. The integration with internal systems are done via three major interfaces. **Task Management** interface is used to create, update, and close tasks, **Code improvement source** interfaces are used to identify targets for code improvement, and **CI/CD Pipeline** interfaces help integration with build and test systems.

In addition to external systems RACER components also have interfaces. **Workflow Engines** process tasks, **Evaluation System** measures effectiveness of the discovery, and an **API Layer** provides external access to discovery capabilities.

3.7 Customizing via Agent Runbooks

Once target files are identified, the agent needs to complete the task (e.g., produce a refactoring, handle compile warnings, dead code removal, etc.). Runbooks are structured prompts that allow easy extensibility (with over 500 such runbooks created at the time of writing). RACER contains several core (pre-defined) runbooks that serve as examples and are extensively engineered and evaluated to ensure they generate high-quality code.

We employ various tools to facilitate the process of transitioning a runbook from benchmarking to beta to general availability (GA), including a benchmarking system, runbook system, and A/B test system. Quality control is vital to avoid poor diffs. RACER includes criteria, support, and monitoring for use cases transitioning from benchmarking to GA. RACER has a **Grader System** to write heuristic logic verifying task resolution and diff quality and **Reviewer Agent** to dynamically assess diff quality, style, and bad patterns. Diffs undergo quality control before and after generation, with comprehensive checks using governance integrations before entering the reviewer queue.

3.8 Reviewer Engagement/Efficiency - Recommendation and Management

Reviewer recommendation is crucial for diff landing success, as low accuracy can reduce engagement and land ratios. It aims to address several issues: **Reviewer Recommendation**: Build an accurate system to identify the best reviewers for diffs, dynamically adjusting as needed, and using historical diff data to recommend likely reviewers. We already have a powerful reviewer recommendation system [24], but adjustments are made because there is no human author of automated diffs. **Reviewer Incentivization Program** is essential to encourage participation (review) of automated "wall" of diffs. Specifically creating metrics to credit reviewers, forming dedicated reviewer groups for RACER to expedite the review process. Even more important is **Reviewer Queue Management** by incorporating queue awareness into recommendations, ensuring manageable diffs per reviewer. Finally, **Reviewer Feedback Collection** implements feedback mechanisms to improve reviewer recommendations.

3.9 RACER Code Improvement Workflow

In summary, each task creation data source uses its specific methodology to produce an internal issue (see Figure 2). Such internal issues are then converted to structured `TaskMetadata` objects that are, in turn, used to create tasks in the task management system. The bulk processing interface to task management system is then used for creating thousands of tasks efficiently and, to avoid lost computation, it also has means to resume discovery from the last processed item in case of crashes.

The end-to-end process is:

(1) A list of candidate files/functions are identified based on high CCN or other types of risk and a task is created for each.
(2) An engineer can pick any of these tasks and trigger RACER to do refactoring based on the standard prompt (outside RACER), or a task is automatically picked by RACER according to some criteria.
(3) RACER then feeds a prompt to CodingAgent and it iterates until it can improve CCN or fails. This involves various automatic tasks such as compiling generated refactoring, etc.
(4) Engineer is presented with a diff (if the previous step succeeds) and then can accept, reject, modify, or feed it back to CodingAgent. Engineer may commandeer RACER diff as well.
(5) diff is either abandoned or accepted via review.

3.10 Conclusions on System Feasibility

RACER represents a significant step towards autonomous perfective software maintenance at scale. By tightly integrating the existing developer infrastructure, data sources, and governance mechanisms, it moves beyond simple developer-supportive agents to autonomously discover, prioritize, implement, and submit medium-granularity code improvement tasks. The modular design and the use of customizable runbooks ensures the system can rapidly adapt to new GenAI models and coding standards. The emphasis on human-in-the-loop review and quality control tooling is important for managing the non-deterministic nature of AI-generated code changes and maintaining high engineering standards. While initial challenges around tooling stability and output quality were significant, the iterative engineering process, such as shifting to a multi-model approach within Confucius and addressing token limitations, demonstrates the viability of deploying large-scale agentic systems for code improvement in a complex enterprise environment.

> RACER is a fully autonomous AI agent for the entire perfective code maintenance process, from task identification to generating and landing code changes (diffs). It leverages an agentic framework (Confucius), diverse data sources for task discovery, and a strong Quality Control Process to ensure high-quality, human-reviewed changes.

4 Empirical Study Design for Autonomous Maintenance

GenAI tools support various types of developers, from offering basic line completion in source code editors to "vibe coding," where natural language instructions are translated into working software applications. From a productivity standpoint, regardless of the tool, its purpose is to enhance developer output: to help them achieve more within the same quality and time constraints. The developer is essential for initiating and navigating tasks, with GenAI serving a supportive role. This limits the potential effort savings that could be achieved if human involvement were completely eliminated from task initiation and completion.

We can, therefore, compare developer output when using alternative tools (or no tools) to measure a tool's impact on developer productivity. However, this approach does not provide answers on how to measure the impact of *autonomous maintenance*. While autonomous maintenance produces output in terms of code modifications, it may be unreasonable to expect these modifications to be directly comparable to modifications produced by developers. Even in pre-GenAI studies, empirical research would exclude the activity of bots [6] to more accurately assess human development effort.

Since RACER is an autonomous bot, we propose addressing this challenge in two ways. First, autonomy is not completely effort-free. It requires effort to implement and maintain the tool, and it does not completely eliminate engineers' effort, as a review of the code changes produced by the agent is still required. We can then use this tool maintenance and use effort and compare it with the effort needed to manually conduct the equivalent tasks. Second, while it is easy to count and measure code modifications to ensure their similarity between manual and automatic work, the true value produced by perfective maintenance lies in the *effort savings* realized from maintaining the improved code. Thus, as a second approach, we measure the maintenance effort improvements produced by RACER and compare them to those resulting from manual code improvements to ensure that the automatic improvements are not "AI slop."

5 RQ2. Net Productivity Impact

What is the net change in human effort when using autonomous perfective maintenance, accounting for engineer time on agent-generated work and the engineering effort required to build and maintain the agent?

5.1 RQ2. Method

Effort estimates are notoriously difficult in software engineering. In our study, we used Diff Authoring Time (DAT) [1][2] as the outcome measure. This measure has been developed and improved over several years and represents the organization's preferred metric for development speed. It captures exactly how much time it takes to author/write and land a diff. The DAT metric leverages existing tools to obtain a complete and accurate estimate of a constant fraction of time spent working on a diff. It tracks engineers' search, AI interaction, online activity, IDEs, and even terminal activity, using complex empirically-validated algorithms to correlate the observed sessions with tasks, diffs, files, and documentation.

While it is impossible to estimate the exact time engineers spend on each diff (e.g., thinking about the problem away from the computer or discussing it with colleagues), DAT is intended to capture a part of the time engineers spend on individual diffs. Although it does miss effort spent away from computer, there is no evidence

[2] Gradle's DPE 2024 Summit https://tinyurl.com/496225wf.

that DAT would systematically over- or under-estimate effort. In other words, as long as the fraction of time working on a diff while away from a computer is constant on average, DAT is a resonable proxy for total effort. For our analysis, we assume that DAT represents, on average, a fixed percentage of the total engineer effort spent on a diff. This assumption allows us to convert DAT to the numbers of equivalent diffs and to convert diffs completed over a certain period to the numbers of equivalent engineers.

As autonomous software agents accomplish mundane tasks previously done by engineers, the autonomy is not completely free: engineers now have to create, debug, and maintain the agents that accomplish these tasks, even if their effort on these autonomously-done tasks may be negligible. We propose a simple method for estimating effort savings achieved by utilizing an autonomous agent. The key is to subtract the effort spent on agent development and maintenance from the effort saved from its use. The analysis focuses on quantifying effort based on changes made to the agent's code and the savings realized by automating tasks that would have otherwise required manual work:

$$\begin{aligned} \text{EffortSavings} = {} & \text{noAgentEffort (hypothetical)} \\ & - \text{AgentAssistedEffort (actual)} \\ & - \text{AgentEngineeringEffort (actual)} \end{aligned}$$

To follow this approach, we need to describe the nature of inputs and outputs in the productivity calculation and then detail how they are operationalized. The bulk of development effort for the software agent is calculated by tracking the time and resources invested in making changes to its codebase. This includes initial development, bug fixes, feature enhancements, and maintenance.

Effort savings are determined by comparing the time that would have been required to perform diffs manually versus the time actually spent for the diffs done using the agent (while some autonomously created diffs require virtually zero effort, many diffs cannot land without some manual intervention that, in turn, requires effort). For the moment, we assume that the value produced by the manual and automated diffs is exchangeable. Some evidence to support this claim is presented in the second empirical study. For autonomously created diffs, we use two effort estimates: DAT for the diff spent by the author and reviewers, and the time spent on the tasks (DAT includes measurement of time for diff completion and a separate heuristic was used to associate engineers time spent with a task) associated with the automatically-generated diff.

5.2 RQ2. Data Collection

As discussed in Section 2, we track numerous metrics to measure the effectiveness of various tools and practices. We used Phabricator[3] data archives to obtain the necessary measures, such as diff creation and landing times, affected files, author, reviewers, comments, review interval, etc.

Many of the measures, like the experience of an engineer or the past changes to a file, change over time. As our observations are diffs, these measures were calculated at the time of diff landing, i.e., merging into the trunk. So, for example, the measures of author's experience change after each diff.

We collected data on all diffs landed during the nine months. RACER was developed during this period and went General Availability (GA) at the start of sixth month. We exclude diffs authored by bots or done using (deterministic) automation. Since this filtering does not completely eliminate such automation-driven diffs (e.g., authors sometimes run bots under their own ID), we remove all authors (and the corresponding diffs) that are outliers (more extreme than 99% of the authors) for downstream analysis.

[3] http://phabricator.org

Table 1: DAT per diff and author participation (all numbers are rescaled using the first column).

	non-BE	BE (no-RACER)	RACER-change	RACER-no-change
Average DAT/Diff	1.000	0.760	0.337	0.042
Number of authors	1.000	0.405	0.069	0.040

We identified a diff as being RACER-engineering if it modified any file in the sub-folders containing RACER source code and use these diffs to reflect the effort needed to create and maintain RACER, i.e., as effort spent, not saved.

For the period after RACER GA, we identified all Better Engineering (BE) diffs using a set of keywords denoting BE activity present in the diff title or its tags. For example, "better engineering," "refactoring," "dead code," "cleanup," etc.

Finally, we identify diffs done with the help of RACER via corresponding diff tags inserted by RACER (we refer to such diffs as RACER-initiated diff). For better tracking of RACER diffs, additional measurement infrastructure was created to determine if, at landing, the diff code change was modified by an engineer or if it was landed without change.

RACER-initiated diffs are commandeered (taken over) by engineers and may be landed without change (RACER-noChange) or with change (RACER-withChange). RACER can also be used as a regular agent, i.e., it does not create a diff but is the result of an engineer asking RACER to do something (RACER-use).

To interpret DAT differences fairly, we establish two manual baselines. Regular diffs and Better Engineering (BE) diffs can differ in typical scope and intent, and therefore in the effort required to complete them. We therefore compute average DAT-per-diff separately for two manual categories: regular manually completed diffs (neither BE nor using RACER), and BE diffs completed manually without RACER. We then compare these baselines to RACER-initiated diffs, distinguishing cases where the landed patch was modified by an engineer from cases where it landed without modification.

5.3 RQ2. Results

We first compare effort per diff across manual and RACER-assisted categories using the baselines defined in the method section. Table 1 shows that regular manually completed diffs have the highest DAT. Manual BE diffs require 76% of that effort, which is consistent with BE work being more focused and often more routine than feature work. We then compare these baselines to RACER-initiated diffs. When an engineer modifies the patch before landing, RACER-change diffs require 34% of the baseline effort. The largest savings occur when RACER produces a patch that is accepted as-is: RACER-noChange diffs require only 4% of the baseline effort.

Table 1 also provides participation context. We observe that 40% of all engineers performed some BE work, while 7% were involved with RACER diffs. Only 4% of all engineers had RACER diffs land

Table 2: Effort spent on diffs. (the numbers rescaled using the value of the first row)

	Number of Diffs	Total DAT	Period
All (after cleaning)	1.0000	1.0000	Mnth 6-10
BE-non-RACER	0.0533	0.0366	Mnth 6-10
RACER-initiated: noChage	0.0030	0.0005	Mnth 6-10
RACER-initiated: withChange	0.0048	0.0020	Mnth 6-10
RACER-use	0.0037	0.0021	Mnth 6-10
RACER-engineering	0.0041	0.0046	Mnth 1-10

Table 3: Estimated savings (the numbers are rescaled using the same constant)

	Range	DAT	#engineers
RACER-initiated: noChange	Mnth 6-10	$[-0.8, -4.1]$	$[-0.05, -0.27]$
RACER-initiated: withChange	Mnth 6-10	-16.3	-1.06
RACER-use	Mnth 1-10	-16.8	-1.09
Spent on RACER engineering	Mnth 6-10	-36.4	-2.36
BE non RACER-equivalent which would HAVE been spent without RACER	Mnth 6-10	$[111, 126]$	$[7.14, 8.15]$
Total Saved	Mnth 6-10	$[39, 57.4]$	$[2.51, 3.71]$

with no change. These participation rates suggest that BE activity is common, while fully autonomous landings remain concentrated among a smaller subset of engineers and diffs.

> The most dramatic effort savings were achieved in cases where RACER autonomously initiated and completed a code change (RACER-noChange diffs) that landed without any manual modification by an engineer.

We next examine how these savings relate to the overall mix of diffs. Table 2 summarizes the relative frequency and total DAT for each diff category. BE-non-RACER diffs account for 5.3% of the total number and 3.7% of total DAT. Within BE work, 15% of all BE diffs were automatically completed by the RACER agent and then landed (after being commandeered by a human). Among landed RACER-initiated diffs, 35% did not include any human modification to the RACER-produced patch. Taken together, these results show that RACER contributes meaningfully to BE output, and that the largest savings are concentrated in cases where the patch is directly reusable by reviewers.

Finally, Table 3 translates these observations into net effort savings by explicitly incorporating both effort avoided and effort incurred. Effort savings are presented as a range because the conversion from DAT to headcount depends on modeling assumptions. Only the last row and the right column in Table 3 are estimates. Specifically, DAT for the Equivalent-Non-Agent Effort is estimated by multiplying the total DAT of the RACER diffs by the ratio of average DAT over BE diffs to average DAT over all RACER diffs. The right column then converts DAT to engineers by first imputing the number of diffs and then applying the diff-to-engineer ratio.

Overall, these results indicate that RACER provides net effort savings after accounting for agent engineering and maintenance effort. The most dramatic savings occur when the agent chooses and completes tasks autonomously, with smaller savings when agent-produced patches require human modification or when the agent is explicitly invoked by an engineer. Over the four-month period, the effort savings from using RACER exceeded the effort expended to develop, extend, and maintain RACER (over months 1–5) by an equivalent of several hundred engineers, depending on the estimation method for the time spent on automatically completed RACER diffs. We expect savings to increase with wider adoption, and we view this accounting framework as reusable for quantifying net productivity impact of autonomous maintenance tools beyond RACER.

> Savings from autonomous agent significantly exceeded the effort required for its development and maintenance on the order of four months of work of several hundred engineers.

6 RQ3. Downstream Maintenance Impact

How does autonomous refactoring compare to human refactoring in its effect on subsequent maintenance effort of the modified code?

The Difference-in-Differences (DiD) approach is a statistical method used to estimate the causal effect of a specific intervention or "treatment." It is particularly useful for observational studies where a randomized controlled trial is impractical. We use it to compare the impact on human authoring time (DAT) between files that were refactored by human developers ("human-refactored") and files refactored by an autonomous agent ("RACER-refactored").

The Control Group consisted of files undergoing standard human-refactoring. The Treatment Group consisted of files undergoing Agent-Authored Refactoring (RACER-refactored). The DiD approach uses the interaction coefficient between post-refactoring and the refactoring treatment to signify the difference in impact between the traditional (human) and experimental (RACER) treatments.

6.1 RQ3. Measures

Data was collected by selecting all source code files that were either human or agent refactored during the intervention period (Month 6 – Month 8). We then compare the Dependent Authoring Time (DAT) for diffs modifying these files during the pre-intervention period (Months 1-5) and post-intervention period (Months 9-10). Various characteristics of the diffs and authors that might affect DAT are also collected to be included as covariates in the regression model.

Since the outcome (DAT) is influenced by a variety of known factors, we also use more complex collaboration measures based on software supply chains and centrality [2, 26, 28, 31, 35, 39] as controls.

The full set of measures and their mathematical definitions are presented in Table 4. The table lists all control and independent variables we measured, along with the rationale for their inclusion and our hypotheses, where possible, for their effects. All these variables were calculated using the approaches described in Section 5.2.

Interdependencies make code harder to change and are commonly used to predict failures. We use file centrality as a proxy for interdependencies. In our model, the node is the file, and there are multiple edge types. Files with higher centrality are often called, have methods called from many other files, have many authors, are frequently co-changed, and are connected to other central files. We used Katz centrality [13] to account for the importance of connected nodes, not just the number of connections (degree centrality). We calculate centrality using the complete co-change, authorship, and call-graph links to ensure the technical structure transfers to and is affected by the social structure. For instance, a developer modifying a file F, which many other files depend on, inherits some of that centrality. Prior work has shown that communication and dependency networks can model various software engineering outcomes [3–5, 15, 23, 37, 39]. The network projection onto a node (e.g., an author or file) can be summarized by degree or Katz centrality [4, 18, 20, 38].

As *Coordination* needs increase, mistakes are more likely [4, 11]. We represent this via author centrality. Larger author centrality indicates connections to more central files with many authors and the maximum number of authors on modified files. Higher coordination needs may decrease quality and increase development time [4].

Author expertise was measured via tenure at the company and the number of past modifications by the author on the files modified by the diff. Prior work has shown that expertise tends to reduce the risk of issues [12, 21].

Code complexity was measured as the amount of code changed (churn) in a diff. Prior work has shown that higher complexity increases change risk [14].

6.2 RQ3. Statistical Modeling

Statistical models help quantify relationships between outcomes and variables that measure various properties of a diff. We use a multiple regression for the DiD case. 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 no zero values in the log transformation. Furthermore, it is essential to include known control variables in the models that estimate the impact of effort reduction to avoid confounding interpretations that incorrectly ascribe the impact of a confounding variable to the impact of refactoring.

Software engineering measures tend to be highly correlated [16, 17], but models with highly correlated measures are hard to interpret, have unstable estimated coefficients, and possess poor predictive power [7]. To address this, if the correlation between two variables exceeded 0.7, one of them was removed from the model. When making these choices, we ensure that the set of remaining variables is diverse (representing distinct aspects of the diff, author, and modified code) and relies on simple but validated measures already in wide use.

To maximize diversity, we roughly subdivide variables into classes related to interdependencies (also known as diffusion, syntactic, and logical dependencies) [4, 12, 15, 21, 23, 39], coordination needs (also known as work interdependencies) [4, 23], author expertise [3, 12, 21, 34], and code churn [21, 22]. As with all statistical models, we aim to establish a relationship between the response and independent variables, as statistical models do not demonstrate causal relationships. The full set of measures and their mathematical definitions are shown in Table 4.

6.3 RQ3. Results

We present the empirical results for RQ3 by examining the DiD regression model that compares subsequent maintenance effort on files refactored by humans versus files refactored by RACER.

The interaction term (`post:racer_refactored` in Table 5) is the key DiD estimate. We note that the model's overall explanatory power is modest ($R^2 = 0.12$), which is common in observational studies of developer effort where much individual variation remains unexplained. We therefore focus on the sign, magnitude, and significance of the interaction term rather than on the model's predictive accuracy. Intuitively, this term measures whether the *change* in authoring effort after the intervention period differs between files refactored by RACER and files refactored by humans. A negative interaction coefficient indicates that, after accounting for the general post-period shift (`post`) and other measured covariates, RACER-refactored files experienced a larger reduction in subsequent DAT than human-refactored files.

We find that the interaction coefficient is negative and statistically significant. This result is notable because it suggests that agent-driven refactoring is not merely producing superficially similar diffs, but is associated with reductions in future authoring effort that are at least as large as those associated with human refactoring. One plausible interpretation is that RACER systematically targets refactorings that remove recurring sources of friction (for example, complexity hot spots or risky constructs) and that these improvements persist into later maintenance work. Another possibility is that RACER preferentially selects opportunities that are easier to refactor cleanly and therefore more likely to yield downstream benefits. Because selection is not random, both explanations may contribute.

These findings must be treated with caution. Although the DiD design reduces some confounding by focusing on within-file changes over time, it still relies on assumptions that cannot be fully verified in an observational setting. In particular, treated and control files may differ along dimensions that are not fully captured by our covariates, such as local design conventions, unobserved engineering initiatives, or shifts in ownership and priorities. It is also possible that the work performed on RACER-refactored files differs subtly from the work performed on human-refactored files in the post period, even after conditioning on churn and other controls. As a result, we interpret the negative interaction as evidence consistent with a larger reduction in maintenance effort following RACER refactoring, rather than as a definitive causal claim.

The remaining coefficients provide additional context and serve as sanity checks on the model. The `post` term is negative, implying that DAT decreases in the post period for both groups, which is consistent with an overall reduction in authoring effort following the intervention window. The main effect `racer_refactored` is

Table 4: 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, d and D, are diffs, a author, d_a: indicators that a authored d; d_t is the time diff landed.

Category	Measure	Description	Calculation
Interdependencies	$FileCent(f,t)$	The average Katz call-graph centrality of the modified files. More central [37] files may have critical functionality [4, 39]	$\max_{\{f:d_f\}} cent(f,d_t)$
	$NFiles(d)$	The number of files modified. Modifying multiple files in a single diff implies "logical" dependencies and risk [5, 21]	$\lvert\{f : d_f\}\rvert$
Coordination	$AuthCent(d)$	Katz centrality of the diff author based on co-change, authorship, and call-graph links. More central authors work with more central files and have more experience [15, 37]	$cent(a,d_t)$
Expertise	$tenure(a,t)$	Author expertise measured via employee tenure at the time of diff landing. Expertise is associated with defect rate [3, 21, 34]	four categories: 0-1, 1-2, 3-4, and 5+ years
	job_level	Author's job level	four categories: 1, 2, 3, 4+
File complexity	$churn(d)$	Sum of the lines added and deleted over all files modified by diff: More change, more risk [21, 22]	$\sum_{\{f:d_f\}} addedLOC(f,t)+ deletedLOC(f,t)$
Programming language	is_cpp, is_hack, is_py	Diff modifies mostly files containing source code from the corresponding language. Risk rates and review time may vary with programming language.	
Outcomes	$DAT(d)$	The time spent in authoring diff in seconds (excluding time author was waiting for review(s))	

Table 5: Regression model for DAT. $adjR^2 = 0.12$

	Estimate	t value	Pr(>|t|)
(Intercept)	7.65	103.07	0.00
post	-0.20	-29.92	0.00
racer_refactored	0.02	1.70	0.09
hack	-0.24	-38.04	0.00
python	-0.07	-8.62	0.00
javascript	-0.26	-28.89	0.00
lFileCent	0.11	13.60	0.00
lAuthCent	-0.04	-6.75	0.00
lNPastDiffs	-0.28	-66.81	0.00
lNFiles	0.14	21.99	0.00
lChurn	0.19	90.94	0.00
post:racer_refactored	-0.05	-3.24	0.00

not statistically significant, indicating that pre-intervention DAT for RACER-refactored files is similar to that for human-refactored files, supporting the plausibility of the comparison.

Several control variables behave as expected. Language indicators suggest systematic differences in DAT by language mix, with `hack` and `javascript` showing lower authoring effort relative to the C++ baseline. Interdependence, as captured by `lFileCent`, is positively associated with DAT, consistent with the intuition that more central files incur higher coordination and risk-management overhead. Similarly, diff complexity increases DAT: both `lNFiles` and `lChurn` have positive coefficients, suggesting that changes that touch more files or modify more lines require more authoring effort. Author centrality (`lAuthCent`) is negative, consistent with the interpretation that more central developers may be more effective at navigating dependencies and completing diffs efficiently.

Finally, `lNPastDiffs` is negative, which may appear surprising. One possible explanation is that files with frequent prior change have higher "process readiness" for maintenance, for example better test coverage, more familiar ownership, or more established patterns for safe modification. Another explanation is selection: maintenance diffs on highly active files may be smaller or more routine than maintenance diffs on rarely touched files, even when controlling for churn and file count. We do not treat this coefficient as causal, but it suggests that prior change history captures important, unmodeled properties of maintenance work.

> Both human- and RACER-refactored files had substantial reduction in DAT post-intervention. Conditional on our controls and the DiD assumptions, files refactored by RACER exhibited a larger reduction in subsequent maintenance effort (DAT) than files refactored by human developers.

7 Threats to Validity

Our studies are observational and rely on telemetry derived from development tools. We therefore discuss threats to validity and the steps we took to mitigate them.

7.1 Generalizability

Our results are drawn from a single large organization with a monolithic repository, integrated task and code review tooling, and strong governance around change submission. These properties enable end-to-end measurement and consistent workflows, but they may not hold in smaller organizations, multi-repository environments, or settings with different review practices.

RACER targets a bounded class of perfective maintenance tasks that are well-scoped, metric-driven, and typically non-urgent. The net effects we observe may not extend to other maintenance categories (e.g., corrective maintenance) or to tasks that require domain knowledge, cross-team coordination, or deep architectural changes.

The organization also provides rich quality signals and historical data that support discovery and prioritization. In environments with less mature metrics, less consistent coding standards, or different language mixes, agent performance and the resulting effort savings may differ.

Finally, our workflow includes human review as a quality gate. The throughput and net savings we report depend on reviewer capacity and reviewer engagement. Settings with different reviewer norms or different capacity constraints may observe different outcomes.

7.2 Construct Validity

Our primary effort outcome is Diff Authoring Time (DAT), which is a proxy for developer effort. DAT captures a consistent fraction of time spent authoring and landing diffs, but it may omit effort spent away from tooling (e.g., discussions or offline reasoning). If this unobserved effort changes systematically across conditions (manual vs. agent-assisted), our estimates may be biased.

Our classification of diffs into BE, RACER-initiated, and RACER-engineering relies on tags, titles, and path-based identification. Misclassification is possible (e.g., incomplete tagging, inconsistent titles, or manual use of automation under a human identity). We partially mitigate this risk by filtering known automation and removing extreme author outliers, but residual misclassification may remain.

For RACER-initiated diffs, we distinguish cases where the landed patch was changed by a human versus landed without modification. This relies on measurement infrastructure that may not capture all forms of change (e.g., equivalent refactorings, reorderings, or non-functional edits). As a result, the "noChange" category may include some human edits that are not detected, which would make the estimated savings conservative.

For downstream impact, we use post-intervention DAT on files as a measure of subsequent maintenance effort. This is an indirect proxy for maintainability. It does not directly measure defect rates, readability, or architectural quality, and it may be influenced by changes in the nature of work performed on those files.

Our effort accounting focuses on human time and does not include the computational cost and energy consumption of running RACER. LLM inference at scale can incur substantial energy and infrastructure costs, which may partially offset the human effort savings we report. A full cost–benefit analysis would need to incorporate these operational expenses alongside developer time savings.

7.3 Internal Validity

Because RACER selects and completes tasks based on risk signals and operational constraints, task selection is not random. Files chosen for agent refactoring may differ systematically from files refactored by humans (e.g., in centrality, churn, ownership, or reviewer availability). We mitigate this threat in the downstream analysis by using a Difference-in-Differences design with covariates that capture key confounders (e.g., centrality, churn, author characteristics, language), but unobserved differences may remain.

The DiD approach depends on the parallel trends assumption. If agent-refactored files and human-refactored files would have evolved differently even without refactoring, our interaction estimate may be biased. We partially address this by including covariates and using pre and post windows that bracket the intervention period, but we cannot fully rule out violations.

Time-varying factors may also affect outcomes, including changes in tooling, documentation, reviewer behavior, and engineering priorities over the study period. In addition, the agent and its runbooks evolved during the same period, which can confound treatment effects with maturity effects. We reduce this risk by clearly separating measurement windows and by accounting for agent engineering effort explicitly in the net savings calculation, but some coupling between maturation and outcomes may persist.

Finally, reviewer learning effects may influence measured effort. As reviewers become familiar with agent-produced diffs, review latency and comment patterns may change. This could either inflate or reduce apparent savings over time.

8 Concluding Remarks

Generative AI agents enable a shift in software development from interactive assistance to limited forms of asynchronous autonomy. We investigated the feasibility and impact of such autonomy by designing and evaluating RACER, a system that targets a bounded class of perfective maintenance tasks in a large enterprise setting.

RQ1 (System feasibility). We found that an agentic system can execute perfective maintenance end-to-end at scale when it is tightly integrated with existing workflows for task creation, code review, and continuous integration, and when it is constrained by explicit quality gates. In our setting, RACER autonomously identifies and prioritizes opportunities, produces candidate patches, and routes changes for human review. The primary remaining control point is human review, which serves both as a governance mechanism and as a practical limiter on throughput.

RQ2 (Net productivity impact). We proposed and applied an effort accounting approach that explicitly subtracts agent engineering and maintenance effort from the effort avoided through automation. Using Diff Authoring Time (DAT) as our effort proxy, we observed substantial savings for agent-initiated work, including cases where agent-produced patches landed without further modification. These results suggest that autonomous perfective maintenance can deliver net effort savings, even after accounting for the cost of building and operating the system.

RQ3 (Downstream maintenance impact). We evaluated whether agent-driven refactoring yields maintainability outcomes comparable to human refactoring. A Difference-in-Differences analysis based on post-intervention DAT indicates that files refactored by RACER experienced maintenance effort reductions that were at least as strong as those observed for human-refactored files, conditional on our controls. We interpret this result cautiously, as observational designs cannot eliminate all confounding.

Across these studies, we also identified practical engineering constraints that shape achievable autonomy. These include model and tool stability, latency introduced by external tooling, and failure modes where generated changes become too broad. In addition, reviewer capacity can become a bottleneck as autonomous output increases, reinforcing the importance of reviewer assignment, queue-aware scheduling, and quality control that reduces reviewer load.

Overall, our results indicate that autonomous perfective maintenance is feasible for a bounded task class and can produce measurable net savings without degrading downstream maintainability. We view RACER and our evaluation methodology as a foundation for future work on expanding task coverage, improving prioritization using outcome-based feedback, and strengthening quality assurance and governance for autonomous software maintenance. From the technical perspective, RACER can be extended to go beyond preset priorities (complexity, coverage) and autonomously run suitable AB experiments to help choose the most effective code improvements.

References

[1] 2024. Measuring Productivity Impact with Diff Authoring Time. https://engineering.fb.com/2025/01/16/developer-tools/measuring-productivity-impact-with-diff-authoring-time/. Episode 69, Meta Tech Podcast.

[2] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 109–119.

[3] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. 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*. ACM, 4–14.

[4] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. Software Dependencies, the Structure of Work Dependencies and their Impact on Failures. *IEEE Transactions on Software Engineering* (2009). papers/multicompany.pdf

[5] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2009. On the relationship between change coupling and software defects. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 135–144.

[6] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. 2020. Detecting and Characterizing Bots that Commit Code. In *IEEE Working Conference on Mining Software Repositories*. https://arxiv.org/abs/2003.03172

[7] 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. 2013. Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography* 36, 1 (2013), 27–46.

[8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The faiss library. *IEEE Transactions on Big Data* (2025).

[9] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. 2001. Does code decay? assessing the evidence from change management data. *IEEE transactions on software engineering* 27, 1 (2001), 1–12.

[10] David Gutiérrez-Avilés, Manuel Jesús Jiménez-Navarro, José Francisco Torres, and Francisco Martínez-Álvarez. 2025. MetaGen: A framework for metaheuristic development and hyperparameter optimization in machine and deep learning. *Neurocomputing* 637 (2025), 130046. https://github.com/Data-Science-Big-Data-Research-Lab/MetaGen

[11] James Herbsleb and Audris Mockus. 2003. Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering. In *2003 International Conference on Foundations of Software Engineering*. ACM Press, Helsinki, Finland. http://dl.acm.org/authorize?787510

[12] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *IEEE Transactions on Software Engineering* (2013). http://doi.ieeecomputersociety.org/10.1109/TSE.2012.70

[13] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

[14] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. 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)*. IEEE, 11–19.

[15] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. 2008. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 13–23.

[16] Audris Mockus. 2007. Software Support Tools and Experimental Work. In *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, V Basili and et al (Eds.). Vol. LNCS 4336. Springer, 91–99. papers/SSTaEW.pdf

[17] Audris Mockus. 2008. Missing data in software engineering. In *Guide to Advanced Empirical Software Engineering*, J. Singer et al. (Ed.). Springer-Verlag, 185–200. papers/missing.pdf

[18] 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. 2025. Leveraging Risk Models to Improve Productivity for Effective Code Un-Freeze at Scale. *ACM Transactions on Software Engineering and Methodology* (2025). https://api.semanticscholar.org/CorpusID:276950332

[19] Audris Mockus, Peter C Rigby, Rui Abreu, Anatoly Akkerman, Yogesh Bhootada, Payal Bhuptani, Gurnit Ghardhora, Lan Hoang Dao, Chris Hawley, Renzhi He, Sagar Krishnamoorthy, Sergei Krauze, Jianmin Li, Anton Lunov, Dragos Martac, Francois Morin, Neil Mitchell, Venus Montes, Maher Saba, Matt Steiner, Andrea Valori, Shanchao Wang, and Nachiappan Nagappan. 2025. Metrics Driven Reengineering and Continuous Code Improvement at Meta.

[20] Audris Mockus, Peter C. Rigby, Rui Abreu, Parth Suresh, Yifen Chen, and Nachiappan Nagappan. 2023. Modeling the Centrality of Developer Output with Software Supply Chains. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023).

[21] Audris Mockus and David M. Weiss. 2000. Predicting Risk of Software Changes. *Bell Labs Technical Journal* 5, 2 (April–June 2000), 169–180. papers/bltj13.pdf

[22] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*. 284–292.

[23] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can developer-module networks predict failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2–12.

[24] Peter C. Rigby, Seth Rogers, Sadruddin Saleem, Parth Suresh, Daniel Suskin, Patrick Riggs, Chandra Maddila, Nachiappan Nagappan, and Audris Mockus. 2025. Improving Code Reviewer Recommendation: Accuracy, Latency, Workload, and Bystanders. *ACM Trans. Softw. Eng. Methodol.* (May 2025). https://doi.org/10.1145/3736405 Just Accepted.

[25] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. 2025. AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges. arXiv:2505.10468 [cs.AI] https://arxiv.org/abs/2505.10468

[26] Param Vir Singh. 2010. The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 2 (2010), 1–27.

[27] Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A Smith, Luke Zettlemoyer, and Tao Yu. 2023. One embedder, any task: Instruction-finetuned text embeddings. In *Findings of the Association for Computational Linguistics: ACL 2023*. 1102–1121.

[28] Ashish Sureka, Atul Goyal, and Ayushi Rastogi. 2011. Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis. In *Proceedings of the 4th india software engineering conference*. 195–204.

[29] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* 86, 6 (2013), 1498–1516.

[30] Oguzhan Topsakal and Tahir Cetin Akinci. 2023. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International conference on applied engineering and natural sciences*, Vol. 1. 1050–1056.

[31] Song Wang and Nachiappan Nagappan. 2021. Characterizing and understanding software developer networks in security development. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 534–545.

[32] Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, Jerry Yang, and Ying Zhang. 2025. Intent-Driven Network Management with Multi-Agent LLMs: The Confucius Framework. In *ACM SIGCOMM*. https://minlanyu.seas.harvard.edu/writeup/sigcomm25.pdf

[33] Justin D Weisz, Shraddha Vijay Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, Katrin Ellice Heintze, and Shagun Bajpai. 2025. Examining the use and impact of an AI code assistant on developer productivity and experience in the enterprise. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–13.

[34] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. 2008. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering* 13 (2008), 539–559.

[35] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. 2013. Categorizing bugs with social networks: a case study on four open source software communities. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1032–1041.

[36] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. arXiv:2401.07339 [cs.SE] https://arxiv.org/abs/2401.07339

[37] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *ACM SIGSOFT / FSE*. Santa Fe, New Mexico, 137–146. http://dl.acm.org/authorize?309273

[38] Minghui Zhou and Audris Mockus. 2010. Growth of Newcomer Competence: Challenges of Globalization. In *FSE/SDP Workshop on the Future of Software Engineering Research*. Santa Fe, New Mexico, 442–447. http://dl.acm.org/authorize?318913

[39] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*. 531–540.