AI Agent Behavior, Pull Request Quality, and Interaction Patterns in Software Repositories

Zihao Sheng
University of British Columbia Okanagan
Kelowna, BC, Canada
1844716292@qq.com

Wei Li
University of British Columbia Okanagan
Kelowna, BC, Canada
yhszhdw@163.com

1 Introduction

AI coding agents such as GitHub Copilot, OpenAI Codex, Claude Code, Devin, and Cursor are increasingly embedded in modern software development workflows. As these systems generate substantial numbers of pull requests (PRs) across open-source repositories, it becomes essential to understand the conditions under which AI-generated PRs succeed or fail, and how AI-driven workflows shape broader repository interaction structures.

This study examines AI-generated PRs from three complementary angles: (1) PR-level quality prediction using a Random Forest classifier, (2) behavioral failure pattern analysis using K-means clustering, and (3) repository-level interaction structures derived from a weighted AI-activity graph. We show that small, coherent, and stable PRs are strongly associated with high quality, failed PRs cluster into recurring behavioral modes shaped by repository popularity and revision patterns, and repositories self-organize into communities defined by shared AI agents and shared AI-operated accounts. These findings reveal both systematic weaknesses in current AI agents and structural opportunities for improving AI-assisted development workflows.

We address the following research questions:

(1) Which structural and behavioral features best predict the quality of AI-generated PRs?
(2) How do failed AI-generated PRs cluster into distinct behavioral modes, and how do AI agents differ across these modes?
(3) How do repositories interact through shared AI activity, and what community structures emerge from these interactions?

By integrating PR-level modeling, failure-mode clustering, and ecosystem-level network analysis, this study provides a comprehensive view of how contemporary AI coding agents participate in and shape collaborative software development.

2 Methods

2.1 Topic 1: Predicting High-Quality PR Outcomes

2.1.1 *Data Cleaning and Filtering.*

We begin with the main pull request table, where each PR belongs to one of three categories:

(1) PRs with both closed and merged timestamps (successfully merged);
(2) PRs with closed but not merged timestamps (rejected);
(3) PRs lacking both fields (still open).

Topic 1 focuses on successfully merged PRs, yielding an initial subset of 790,139 PRs.

Project repository: https://github.com/Zihao-Sheng/DATA542Project.

Two behavioral variables were selected as core quality indicators: `turnaround_hours` (time from creation to completion) and `review_count`. Exploratory analysis revealed strong skewness in both variables: over 75% of low-quality PRs had zero reviews, and more than 75% of high-quality PRs closed within an hour. To place them on a comparable scale, both variables were normalized to the interval $[0, 1]$ and combined into a composite quality score:

$$\text{score} = 60 \times \text{review\_norm} + 40 \times \text{time\_norm}.$$

To identify factors influencing this quality score, we merged additional attributes from auxiliary tables:

- **Commit behavior:** `commit_count` and `total_change` from `pr_commit_detail`;
- **Task characteristics:** categories derived from `pr_task_type`;
- **Repository metadata:** stars, forks, and programming language.

During this process, we observed many PRs labeled as "high quality" but with no recorded commits. Further investigation indicated two main causes:

(1) Incomplete repository metadata, which led to missing commit detail information for many repositories;
(2) Genuine cases where PRs were merged without meaningful recorded modifications, often shortly after creation.

Because such PRs do not provide reliable behavioral signals, they were excluded. The final Topic 1 dataset contained 24,014 PRs with complete structural, behavioral, and repository-level information.

2.1.2 *Modeling Approach.*

We framed PR quality prediction as a binary classification problem, where high-quality PRs are defined as those with quality score ≥ 90, and lower-quality PRs as those below this threshold.

A Random Forest classifier was chosen for three reasons: (i) its ability to capture nonlinear relationships, (ii) robustness to heterogeneous feature types, and (iii) the availability of interpretable feature importance scores. The model incorporated structural, behavioral, and repository-level variables, including: `commit_count`, `total_change`, programming language, agent identity, repository popularity (stars), and multiple review-related features.

The dataset was split using a 70/30 train–test partition. Hyperparameters (such as number of trees and depth constraints) were tuned using cross-validation on the training set. After training, we evaluated overall accuracy, class-specific performance for high- vs. lower-quality PRs, and extracted feature importance scores to guide interpretation. Partial dependence and grouped probability plots were used to study the marginal influence of key predictors on predicted quality.

2.2 Topic 2: Failure Pattern Analysis

2.2.1 Data Cleaning for Failure Analysis. Building on Topic 1, we first addressed the presence of "pseudo" high-quality PRs—entries labeled as high quality despite containing little or no commit activity. To prevent such cases from distorting the failure analysis, we combined the refined high-quality and low-quality PR subsets into a unified dataset representing PRs with interpretable behavioral signals.

This combined dataset was merged with `pr_commit_detail` and repository-level metadata to obtain complete information on commit behavior, code modifications, review patterns, and repository characteristics. The merged table initially contained 835,913 PRs, but only 7,323 entries retained valid repository associations and commit histories.

The sharp reduction reflects structural limitations of the source data: commit histories were collected at the repository level, and missing or inaccessible repositories produced empty commit records even for successfully merged PRs. Because such PRs cannot be reliably characterized, they were removed. The remaining 7,323 PRs form the final dataset used for clustering in Topic 2.

2.2.2 Clustering Methods. We focused on failed PRs within the cleaned dataset and performed K-means clustering using six standardized features:

- `stars`,
- `turnaround_hours`,
- `commit_count`,
- `review_count`,
- `log_total_change`, and
- `merged` (binary).

All variables were standardized (zero mean, unit variance) to ensure that no single scale dominated the clustering. The number of clusters was selected using a combination of the elbow method and qualitative inspection of cluster profiles.

To interpret the resulting clusters, we computed centroid profiles for each group and visualized them using radar charts, highlighting relative differences across behavioral dimensions. We then cross-tabulated cluster membership with AI agent identity to identify systematic patterns in how different agents distribute across failure modes. A summary heatmap and a 3D cluster visualization by agent were used to present these distributions.

2.3 Topic 3: Repository Collaboration Activity and Interaction Structures

2.3.1 Repository-Level Data Processing. For Topic 3, we constructed a cleaned dataset of repository-level activity indicators derived from PR metadata. For each repository, we computed:

- **PR volume**, representing baseline development intensity;
- **Merge rate**, capturing workflow efficiency;
- **Unique submitting-account count**, reflecting the breadth of distinct (mostly AI-operated) identities initiating PRs;
- **Review density**, measuring reviewer engagement;
- **Average merge time**, describing operational speed.

Raw variables were cleaned, log-transformed when appropriate, and trimmed for extreme outliers. Two composite metrics were then constructed on a 0–100 scale: **Efficiency** and **Popularity**.

Efficiency. Merge activity was summarized using a rate component, merge_rate_score = 100×merge_rate, and a time component, merge_time_score, which assigns higher values to repositories with shorter merge times via a monotone log-rescaling clipped to $[0, 100]$. The final Efficiency metric is:

$$\text{Efficiency} = 0.6\,\text{merge\_rate\_score} + 0.4\,\text{merge\_time\_score}.$$

Popularity. Repository visibility and interaction intensity were captured through log-scaled scores derived from PR volume, unique submitting accounts, and review comment density. These components were combined as:

$$\text{Popularity} = 0.7\,\text{pr\_count\_score} \;+\; 0.1\,\text{contributor\_score} \;+\; 0.2\,\text{review\_comment\_score}.$$

Both composite metrics were strongly right-skewed; only a small subset of repositories achieved high values on both. To focus on strongly AI-active repositories, we applied thresholds of Popularity ≥ 42 and Efficiency ≥ 50, retaining only projects that exceeded both cutoffs.

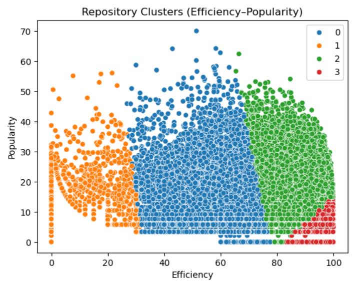

Figure 1: Repositories clustered by Efficiency and Popularity scores.

2.3.2 Network Construction. For each high-value repository, we extracted all associated PRs and used AI-related identity signals to construct an undirected weighted interaction graph. In this dataset, PRs are submitted through AI-controlled accounts, so user IDs primarily represent AI-operated identities rather than individual human contributors. We focused on two levels of AI overlap:

- **Shared agent type:** two repositories are connected if they both receive PRs from the same AI coding agent (e.g., OpenAI Codex, Claude Code, Cursor, Devin, Copilot).
- **Shared AI account (user ID):** two repositories are more strongly connected if the same AI-operated account submits PRs to both projects.

Formally:

- **Nodes** represent repositories;
- **Edges** represent overlap in AI usage between repositories;
- **Edge weights** are assigned as:
  - +1 for each shared AI agent type;
  - +3 for each shared AI user ID.

Each repository node also retains its Efficiency and Popularity scores. A 3D force-directed layout was used to visualize connectivity patterns.

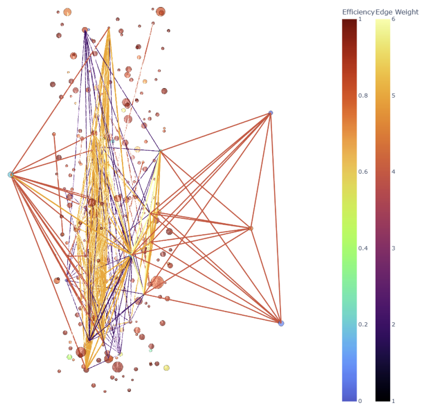

Figure 2: 3D NetworkX graph for interactions between repositories.

2.3.3 Community Detection. To refine the structural analysis, we applied several filtering steps:

(1) Removal of weight-1 edges to eliminate incidental or weak AI-activity links;
(2) Elimination of isolated nodes;
(3) Computation of degree-based metrics to identify hub repositories.

Community detection was then performed using the Louvain modularity-maximization algorithm. The resulting communities demonstrated clear modular boundaries and revealed a stratified interaction ecosystem consisting of:

- Two large communities with broad, multi-language technical scope;
- Two smaller, tightly constrained communities associated with specific authors or organizational namespaces.

A 2D spring-layout visualization confirmed that repositories naturally cluster according to AI-activity patterns.

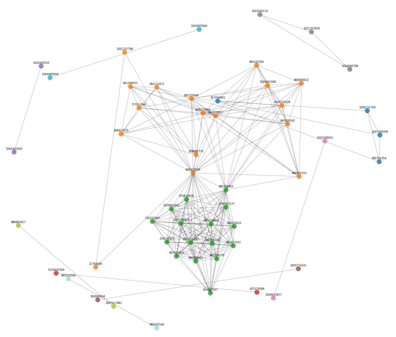

Figure 3: Community structure in the AI-activity interaction network.

3 Results

3.1 Topic 1: Quality Prediction for AI-Generated PRs

3.1.1 Model Performance. The Random Forest classifier achieved approximately 86.2% accuracy on the held-out test set, indicating that the selected predictors contain substantial signal for separating high-quality PRs from lower-quality ones. However, class-level performance showed an important asymmetry: predictions for high-quality PRs were highly accurate, while accuracy for lower-quality PRs was only about 70%. This imbalance suggests a tendency to overpredict high-quality outcomes, which may reflect skewed class distributions or structural similarity between borderline and truly high-quality PRs.

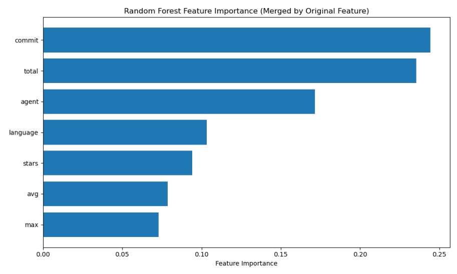

Figure 4: Random Forest feature importance for PR quality classification.

3.1.2 Feature Importance and Partial Dependence. Feature importance scores identify the most influential predictors for distinguishing high-quality from lower-quality PRs (Figure 4). We summarize the key patterns:

Total code change volume. The partial dependence plot for `total_change` (Figure 5) shows a clear negative relationship: PRs with smaller, focused modifications are far more likely to be high quality, while larger changes substantially reduce success probability, likely due to increased complexity and review burden.

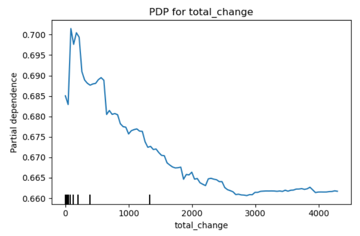

Figure 5: Partial dependence of predicted quality on total code change volume.

Commit count. Single-commit PRs exhibit the highest predicted quality. As `commit_count` increases, predicted quality declines steadily—especially beyond five commits—suggesting that iterative fix cycles or unstable revision processes are associated with weaker outcomes.

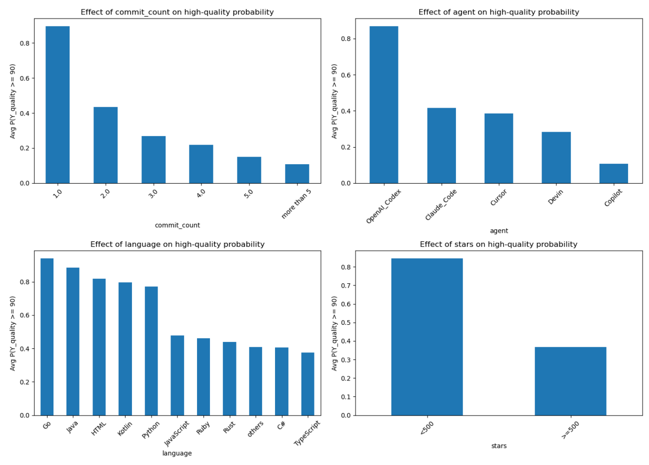

Figure 6: Partial dependence plots for commit count, agent type, programming language, and repository popularity.

Agent type, language, and popularity. Figure 6 summarizes additional partial dependence profiles:

- **Agent type:** OpenAI Codex achieves the strongest predicted performance, followed by Claude Code and Cursor. Devin and Copilot score lower, reflecting persistent differences in agent reliability and alignment with project conventions.
- **Programming language:** Go, Java, HTML, Kotlin, and Python are associated with higher predicted quality, whereas JavaScript, Ruby, Rust, C#, and especially TypeScript show lower success rates, suggesting that language-specific complexity and ecosystem norms influence AI effectiveness.
- **Repository popularity:** PRs targeting repositories with fewer than 500 stars have substantially higher predicted quality, while highly starred repositories show reduced success, likely due to stricter review standards and more complex codebases.

3.1.3 *Summary of Topic 1 Findings.*

Overall, Topic 1 highlights several structural regularities in how AI-generated PRs achieve high quality. High-quality submissions tend to be small in scope, stable across commits, and aligned with established project conventions. Both the coding agent and the technical environment—particularly programming language and repository context—play moderating roles. Quality appears to arise not from any single feature, but from interactions between contribution size, revision stability, language-specific difficulty, and repository-level constraints.

3.2 Topic 2: Failure Patterns in AI-Generated PRs

3.2.1 *Cluster Profiles.*

K-means clustering identified five behaviorally distinct clusters of failed PRs. Radar charts of standardized feature centroids (Figures 7–11) highlight their profiles.

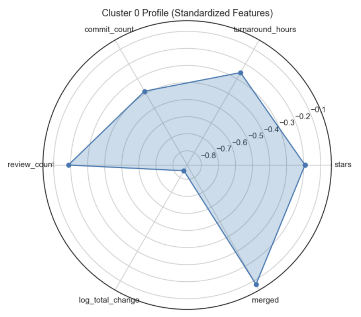

Figure 7: Cluster 0: single-commit, small-change failures in moderately popular repositories.

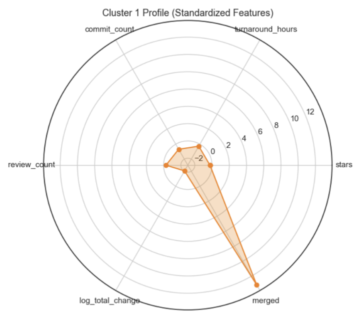

Figure 8: Cluster 1: near-zero-change failures in low-popularity repositories.

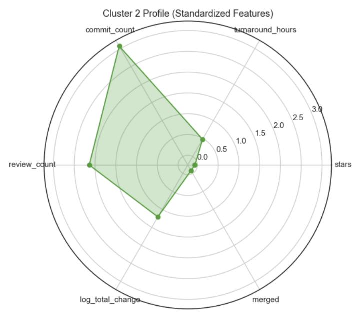

Figure 9: Cluster 2: multi-commit, larger-change failures in highly popular repositories.

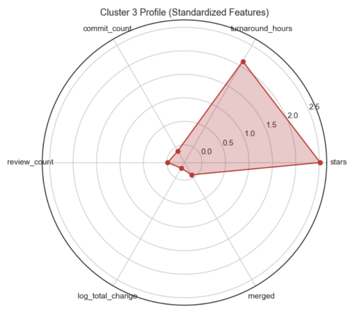

Figure 10: Cluster 3: long-turnaround failures in extremely popular repositories with heavy review queues.

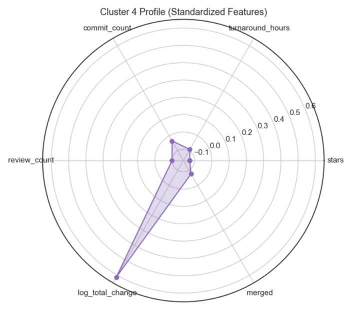

Figure 11: Cluster 4: moderate multi-commit failures in moderately popular repositories.

The clusters can be interpreted as:

- **Cluster 0:** Single-commit PRs in moderately popular repositories, involving small, focused changes and slightly elevated review activity. These PRs are easy to assess and are typically rejected quickly.
- **Cluster 1:** PRs in low-popularity repositories with negligible commit activity and minimal code changes, often offering little observable value.
- **Cluster 2:** Multi-commit PRs targeting highly popular repositories, with substantial revision activity, larger code modifications, and higher review counts.
- **Cluster 3:** PRs in extremely popular repositories with long turnaround times and heavy review queues; these often fail due to staleness or conflict with ongoing development.
- **Cluster 4:** Multi-commit PRs in moderately popular repositories, featuring moderate code changes and balanced review activity but still failing at a high rate.

3.2.2 Agent Behavioral Differences. To understand how AI agents populate these failure modes, we computed the distribution of clusters by agent. A heatmap summarizing these distributions is shown in Figure 12, and a 3D cluster visualization by agent is shown in Figure 13.

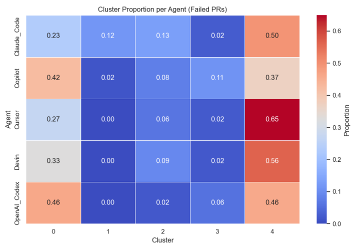

Figure 12: Heatmap of cluster distributions for each AI coding agent.

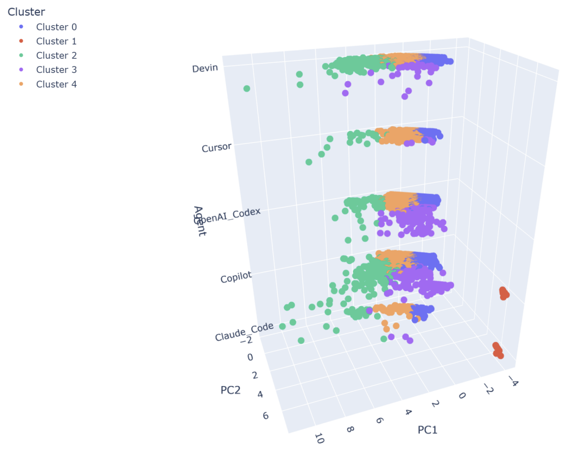

Figure 13: 3D cluster plot of failed PRs, colored by cluster and separated by agent.

Several patterns emerge:

- **Copilot and OpenAI Codex** have strong concentration in Cluster 0, indicating frequent quick, single-commit attempts that are rejected rapidly.
- **Cursor and Devin** are heavily represented in Cluster 4, suggesting a tendency to generate multi-commit PRs that fail later in the review process.
- **Claude Code** exhibits a more balanced distribution across clusters, but still shows notable mass in Cluster 4.
- All agents have minimal presence in Clusters 1 and 3, suggesting that these failure types are primarily driven by repository characteristics (e.g., low visibility or extreme popularity) rather than agent behavior alone.

3.2.3 Practical Recommendations. Based on these five failure categories, we propose several targeted recommendations:

1. **Single-commit PRs to popular repositories (Cluster 0).** These PRs often fail because their changes are too small or underspecified. Adding more substantive modifications and clearer contextual explanations can help maintainers assess value and reduce quick rejections.

(2) **PRs with no recorded commits (Cluster 1).** Many such cases arise from missing repository linkages or incomplete contribution workflows. Ensuring that commit records are properly captured and that each PR contains meaningful code changes improves traceability and reviewability.
(3) **Multi-commit PRs to highly popular repositories (Cluster 2).** High-profile projects often reject PRs that contain excessive revisions or lack polish. Producing clearer, more coherent commit sequences and aligning closely with repository conventions can reduce unnecessary churn.
(4) **PRs submitted to high-traffic repositories with long queues (Cluster 3).** These PRs frequently fail due to staleness. Keeping branches up-to-date, rebasing frequently, and minimizing delays between commits can help mitigate conflicts and timing-related failures.
(5) **Moderate-commit PRs in mid-popularity repositories (Cluster 4).** These contributions show effort but often lack completeness or refinement. Strengthening the final implementation, focusing changes, and reducing irrelevant edits can make these PRs more aligned with maintainer expectations.

3.3 Topic 3: Repository Interaction Structures

3.3.1 Shared Characteristics of Large Communities. The AI-activity interaction network reveals two large communities and two smaller ones (Figures 2, 3, and Table 1). The two large communities share several features:

- Both contain high-visibility "core" repositories surrounded by numerous smaller supporting repositories.
- AI-operated identities frequently contribute across multiple repositories within each community, indicating multi-project workflows.
- The language mix is broad, spanning TypeScript, JavaScript, Python, Go, Rust, C#, and others, consistent with full-stack engineering.
- Licensing is predominantly permissive (MIT, Apache-2.0), with some experimental repositories lacking finalized licenses.

3.3.2 Distinct Roles of the Major Communities. The first large community is dominated by web-facing infrastructure: dashboards, agent interfaces, documentation portals, and LLM interaction gateways. Its linguistic signature is heavily TypeScript- and JavaScript-based, reflecting a focus on user-facing and orchestration layers.

The second large community merges ML engineering frameworks, Python projects, and performance-oriented backend modules. It bridges classical machine-learning workflows with modern agent-augmented practices, forming a link between traditional data/ML systems and AI-enabled pipelines.

3.3.3 Characteristics of Small Communities. The two smaller communities differ markedly from the large ones:

- One community consists of several C-language systems repositories under a single namespace, with extremely high development activity despite having no public stars or forks.
- The other community comprises Python-based agentic computing tools associated with a single developer or organization, characterized by consistent naming and tightly coupled repositories.

These communities operate as localized micro-ecosystems: internally cohesive but relatively disconnected from broader public workflows.

3.3.4 Collaboration Dynamics. Across the full network, several structural patterns emerge:

(1) A small set of repositories functions as hubs, linking otherwise separate technical domains.
(2) Language distribution varies systematically by community, reflecting specialization in front-end, back-end, ML, or systems engineering.
(3) A clear core–periphery structure emerges, with highly visible projects acting as anchors and numerous smaller repositories serving as integration or feature-specific spaces.
(4) Cross-project cohesion is strongest within communities and weaker between them, reflecting specialized workflows and recurring AI activity patterns rather than stable human teams.

4 Conclusion

Across all three topics, our findings present a coherent view of how AI coding agents participate in software development. At the PR level (Topic 1), high-quality outcomes emerge from small, coherent, low-revision changes, whereas larger or unstable PRs show substantially lower success probabilities. These effects vary by agent type, programming language, and repository context, underscoring that AI effectiveness depends jointly on model behavior and technical environment.

Failure analysis (Topic 2) shows that rejected PRs fall into distinct behavioral modes, from trivial single-commit attempts to complex multi-revision submissions in high-visibility repositories. Agents populate these modes unevenly, reflecting systematic differences in revision strategies and alignment with repository expectations. Such cluster-level patterns highlight concrete opportunities to improve agents' planning and workflow adaptation.

The repository interaction network (Topic 3) reveals structured communities defined by shared AI activity. Larger communities specialize in front-end orchestration or ML/back-end engineering, while smaller ecosystems revolve around tightly coupled namespaces or individual developers. These interaction patterns illustrate how AI-generated contributions reinforce existing technical boundaries and shape cross-repository workflows.

Overall, the results show both the strengths and limits of current AI coding agents. While they produce high-quality contributions under focused and contextually aligned conditions, performance degrades in complex environments, long revision cycles, and high-traffic repositories. Future work should target improved context modeling, multi-step revision planning, and stronger alignment with repository-specific norms to support more reliable integration of AI into collaborative development.

Table 1: Highly interacted repositories by community: activity metrics and dominant AI agent.

Comm.	Full name	Lang.	Stars	PRs	Eff.	Pop.	Notes	Dominant agent
A	atariryuma/Everyone-s-Answer-Board	HTML	0	366	96.82	43.42	web UI board	OpenAI_Codex
A	giselles-ai/giselle	TypeScript	154	139	54.92	52.33	agent front-end	OpenAI_Codex
A	hmislk/hmis	HTML	167	421	95.29	58.31	hospital MIS	OpenAI_Codex
A	Tayler01/shadowChat1.0	TypeScript	0	341	96.78	44.44	chat interface	OpenAI_Codex
A	dashort/ride	JavaScript	0	403	97.86	45.83	web app	OpenAI_Codex
A	denuoweb/QuestByCycle	Python	1	314	95.70	43.76	backend service	OpenAI_Codex
A	drivly/ai	TypeScript	29	993	78.67	54.47	production AI service	Devin
A	gabriellagziel/appoint	Dart	0	470	84.53	45.50	mobile app	OpenAI_Codex
A	langfuse/langfuse-docs	Jupyter Nb.	128	70	67.77	43.23	docs / analytics	Cursor
A	mrjoncastro/admin-panel-umadeus	TypeScript	0	939	91.69	51.26	admin front-end	OpenAI_Codex
A	novuhq/novu	TypeScript	37349	80	60.36	45.11	notification infra	Devin
A	patolina1000/-HotBotWebV2	JavaScript	0	331	99.07	44.19	web bot UI	OpenAI_Codex
A	selfxyz/self	Circom	691	63	50.15	43.05	zk / crypto	OpenAI_Codex
A	theopenco/llmgateway	TypeScript	321	120	61.86	46.88	LLM gateway	Devin
A	GlareDB/glaredb	Rust	951	97	83.93	47.82	data engine	Devin
A	antiwork/gumroad	Ruby	6643	148	50.60	47.25	commerce backend	Devin
A	antiwork/helper	TypeScript	537	171	51.88	50.90	helper service	Devin
B	mlflow/mlflow	Python	21402	91	56.62	50.91	ML platform	Copilot
B	AbuAli85/extra-contracts-yy	TypeScript	0	280	85.59	42.80	web / contracts	OpenAI_Codex
B	Aries-Serpent/gh_COPILOT	Python	1	689	95.71	50.29	GH automation	OpenAI_Codex
B	CyanAutomation/judokon	JavaScript	1	947	97.28	52.94	automation tool	OpenAI_Codex
B	InterCooperative-Network/icn-core	Rust	4	809	85.34	51.62	core infra	OpenAI_Codex
B	VimsRocz/IMU	Python	4	1033	92.75	53.66	backend	OpenAI_Codex
B	charlesvestal/extending-move	Python	79	461	73.09	46.95	Move ecosystem	OpenAI_Codex
B	daparthi001/QuantumVestAI_App_Source_Code	Python	0	410	91.42	45.97	trading app	OpenAI_Codex
B	deveydtj/WWF	JavaScript	0	263	90.97	42.28	web app	OpenAI_Codex
B	dmorazzini23/ai-trading-bot	Python	1	706	97.73	50.49	trading bot	OpenAI_Codex
B	gofiber/fiber	Go	37269	51	52.94	42.47	web framework	OpenAI_Codex
B	hbghlyj/kuing.cjhb.site	PHP	0	270	91.10	42.50	site backend	OpenAI_Codex
B	jdfalk/subtitle-manager	Go	1	351	67.18	44.68	media tool	OpenAI_Codex
B	luckydizzier/wrecept	C#	0	449	98.83	46.73	desktop app	OpenAI_Codex
B	ramadhan22/dropship-erp	Go	0	422	91.66	46.21	ERP tool	OpenAI_Codex
B	thesavant42/retrorecon	Python	4	821	95.78	51.75	ML / analytics	OpenAI_Codex
C	splanck/vc	C	0	1124	97.52	52.76	systems / compiler	OpenAI_Codex
C	splanck/vento	C	0	509	99.01	46.16	systems tool	OpenAI_Codex
C	splanck/vlibc	C	0	640	97.87	48.07	libc implementation	OpenAI_Codex
C	splanck/vush	C	0	763	98.53	49.53	shell / tooling	OpenAI_Codex
D	adrianwedd/AI-SWA	Python	0	318	99.57	42.26	agentic app	OpenAI_Codex
D	adrianwedd/Agentic-Index	Python	1	323	94.38	42.39	agent index	OpenAI_Codex
D	adrianwedd/agentic-research-engine	Python	0	450	98.30	45.14	research engine	OpenAI_Codex

Additional Content — Development of a High-Performance Network Visualization Tool

During our work on Topic 3 (Repository Interaction Networks), we observed that existing 2D and 3D visualization tools for NetworkX graphs become inefficient when dealing with large-scale software-repository networks. In practical experiments, graphs containing thousands of nodes and hundreds of thousands to millions of edges either (1) failed to render due to memory limitations, or (2) produced unreadable figures where dense low-weight edges formed dark bundles that obscured the underlying structure.

To address these limitations, we developed a custom high-performance graph-plotting method tailored for large, weighted repository interaction networks.

Key Features of Our Method

Adaptive Transparency for Low-Weight Edges.

Low-weight edges—which typically represent weak repository connections—are automatically rendered with reduced opacity. This significantly reduces clutter and makes high-weight, structurally important edges stand out clearly.

Selective Edge Rendering.

Users can (i) limit the maximum number of low-weight edges drawn per node, (ii) down-sample weak edges, or (iii) disable low-weight edges entirely. This provides fine-grained control over both performance and visual clarity when working with extremely large graphs.

Support for Both 2D and 3D Visualization.

- **2D mode:** implemented using the `PyVis` package, providing a fully interactive web-based interface.
- **3D mode:** implemented using `Plotly`, supporting rotational exploration, hover tooltips, colorbars, and both curved or straight edge rendering.

Community-Based Node Coloring.

The method accepts a community-detection partition (e.g., Louvain). Nodes are automatically assigned distinct colors per community, producing intuitive, visually interpretable cluster structures.

Enhanced Visual Aesthetics for Clarity.

Edge color, transparency, and curvature are dynamically adjusted using hybrid colormap schemes and Bézier-curve techniques. These enhancements produce cleaner, more readable, and more "artistic" layouts even for dense, large-scale graphs.

Usage in This Report

Figures 2 in this paper were generated using this visualization method.

Please see Figure 14 for an example of community-level node rendering produced using the `complex_NX` package. The original

NetworkX graph contained approximately 1,000 nodes and 370,000 edges.

Compared with default NetworkX or Plotly layouts, our approach provides:

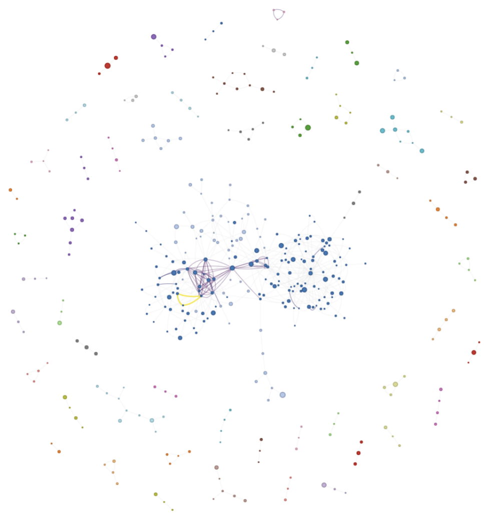

Figure 14: 2D community plot rendered with `complex_NX`.

- clearer separation of communities,
- visually meaningful emphasis on strong connections,
- substantially improved scalability to large datasets, and
- enhanced interpretability of repository interaction patterns.

This visualization tool is generalizable beyond this study and can be reused for a wide range of network-science tasks involving large, weighted graphs.