Pre-Post AI Comparisons: Behavioural Changes in Student Capstone Projects

1 Background

In the last three years, AI coding assistants have gone from novelty to default. Copilot, ChatGPT, and the various IDE-embedded models now write a non-trivial share of the code that lands in active open-source repositories [3, 9]. He et al. trace this transition by mining commit histories from public GitHub projects and identifying AI-authored regions through both timing signals and stylometric markers. The resulting picture is one in which AI involvement is no longer a minority case; in some project categories, it accounts for the bulk of new code over the past year.

The obvious question this raises is about authorship and provenance, which most of the existing literature on AI code detection has sought to answer. We think the more interesting question is about what happens around the code. If AI-assisted teams ship code that is better-documented, better-reviewed, better-scoped, and more consistently formatted than the human-only teams of five years ago, the case for human authorship as a value in itself starts to look thin. Nobody argues for human-written commit messages or hand-rolled linter output; those battles were settled by tooling long ago. If the same thing is happening to code, the right response is not to build better detectors; it is to understand what is changing specifically so we can say which parts are worth preserving and which were never really about human authorship in the first place.

Our data gives us a way to look at this concretely. We have two cohorts of student capstone projects, five years apart: one from before any usable AI assistant existed and one from a year when every team had access to Copilot and ChatGPT. Comparing those cohorts directly would conflate two things: a shift in how students write code with an assistant, and a shift in how GitHub culture, course instruction, and Git tooling changed over the intervening years. To separate the two, we identify the subset of the 2023 cohort whose code reads as AI-generated to a calibrated AI detector and compare their PR-level behaviour against the 2018 baseline. The rest of this paper describes how we identified that subset and what the behavioural comparison shows.

1.1 What AI-generated code looks like

A handful of patterns recur across the empirical literature. None is conclusive on its own, since any individual feature can be reproduced by a careful human or suppressed by a model with the right system prompt. Taken together, however, they form a fairly recognizable fingerprint, and most published detectors lean on some combination of them. The detector we use in Section 3 leans on this fingerprint indirectly, so the fingerprint itself deserves a brief description.

1.1.1 Verbose, descriptive identifier naming

Models trained on well-documented code default to long, self-explanatory names. A function might be called `calculateTotalOrderAmount` where a human under deadline pressure would type `calcTotal` or just `tot` [4]. The effect compounds within function bodies: temporary variables, loop counters, and intermediate results all retain their full descriptive labels rather than the conventional `i`, `tmp`, or `x`. Real codebases are messier than this. Names get shortened, abbreviated, or shaped by whatever convention the team happened to settle on years ago, and the same engineer will often use different naming styles in different files depending on which library they were imitating that day.

Listing 1: AI-generated style: verbose, standardised identifiers

```
def calculate_average_transaction_value(transaction_list):
    total_transaction_amount = sum(transaction_list)
    number_of_transactions = len(transaction_list)
    average_transaction_value = (total_transaction_amount
                                 / number_of_transactions)
    return average_transaction_value
```

Listing 2: Human-written style: concise, idiomatic identifiers

```
def avg_txn(txns):
    return sum(txns) / len(txns)
```

1.1.2 High comment and documentation density

AI-generated functions tend to come with a docstring whether one is needed or not, and the docstring almost always follows a particular shape: a one-line summary, an Args block, a Returns block, and sometimes a Raises block. Inline comments restate what the next line does in slightly more verbose language [9, 4]. Human-written code under a deadline looks nothing like this. Comments are sparse, sometimes outdated, and clustered around the parts the author actually found tricky rather than spread evenly across all functions. Human comments also tend to encode context that lives nowhere else in the code: ticket numbers, references to deprecated APIs, a quick note on why the obvious approach was skipped.

Listing 3: AI-generated style: comprehensive docstring and inline comments

```
def find_maximum(numbers):
    """
␣␣␣␣Finds␣and␣returns␣the␣maximum␣value␣in␣a␣list␣of␣numbers
    .

␣␣␣␣Args:
␣␣␣␣␣␣␣␣numbers␣(list):␣A␣list␣of␣numeric␣values.
␣␣␣␣Returns:
␣␣␣␣␣␣␣␣int␣or␣float:␣The␣largest␣value␣in␣the␣list.
␣␣␣␣Raises:
␣␣␣␣␣␣␣␣ValueError:␣If␣the␣input␣list␣is␣empty.
␣␣␣␣"""
    if not numbers:
        raise ValueError("Input␣list␣must␣not␣be␣empty.")
    maximum_value = numbers[0]
    for number in numbers[1:]:
        if number > maximum_value:
            maximum_value = number
    return maximum_value
```

1.1.3 Consistent whitespace and indentation

Whitespace is one of the strongest single signals available to a stylometric classifier. The model produces indentation depth, blank-line ratios, and spacing around operators consistently because that is what its training data looked like after passing through a linter. Human code drifts in ways that linters tolerate but do not enforce: a stray double blank line between functions, an inconsistent space before a colon, two-space indentation in one helper inside an otherwise four-space file. Suh et al. report ROC-AUC above 0.99 from feature-based classifiers built almost entirely on this formatting signal [7].

1.1.4 Higher code duplication (DRY violations)

Because the model only sees the local context window, it cannot reuse a helper that already exists three files away, so it writes the helper again. Yang et al. [9] measured this directly across 6,000 real GitHub repositories and five different assistants, and found AI-authored commits introduced markedly more duplicated blocks than human-only baselines.

Listing 4: DRY violation: AI regenerates validation logic rather than reusing an existing helper

```
# In file user_service.py
def register_user(email):
    if "@" not in email or "." not in email:
        raise ValueError("Invalid␣email␣format")
```

```
# In file subscription_service.py (AI regenerates rather
    than reusing)
def subscribe_user(email):
    if "@" not in email or "." not in email:
        raise ValueError("Invalid email format")
```

1.1.5 Elevated code churn

AI-authored lines also get reverted or rewritten faster than human-authored ones [9, 3]. Yang et al. track this at the commit level and show that a large fraction of AI-authored lines are removed or heavily modified within two weeks of being committed, which is well above the equivalent rate for human authorship in the same projects. This is consistent with the model optimising for plausible next code rather than for code that survives review.

1.1.6 Security vulnerabilities

Multiple peer-reviewed studies have found elevated rates of security vulnerabilities in AI-generated code. Pearce et al. [5] found about 40% of Copilot completions contained CWE-listed vulnerabilities across 18 tested categories. Fu et al. [1], looking at Copilot-authored code in real GitHub projects, reported similar weaknesses at scale, with the most common categories being CWE-79 (cross-site scripting), CWE-89 (SQL injection), and CWE-22 (path traversal). Tihanyi et al. [8] generated over 330,000 C programs and flagged 62% as containing at least one vulnerability under formal analysis. Whether you stress-test with constructed prompts, mine real-world code, or generate at scale, the rate of insecure output stays high.

Listing 5: AI-generated SQL query susceptible to injection; parameterised form shows the human correction

```
# Insecure: AI generates direct string interpolation
def get_user(username):
    query = f"SELECT * FROM users WHERE username = '{
        username}'"
    cursor.execute(query)

# Corrected: parameterised query
def get_user(username):
    cursor.execute(
        "SELECT * FROM users WHERE username = ?", (username
            ,))
```

1.1.7 Hyper-consistent error handling

AI-generated functions wrap things in try/except even when nothing in the surrounding code suggests the caller wants exceptions caught at that level [9]. The

pattern is so uniform it is almost a tell on its own: multiple specific exception types caught in sequence, each logged with a formatted message, each re-raised. Human error handling is patchier. Some functions catch nothing because the author trusted the caller; others catch everything with a bare `except` because the author was tired. The model's version is more defensible in isolation, but in aggregate it produces noisy logs and obscures the function's actual error contract.

Listing 6: AI-generated style: exhaustive, templated error handling applied regardless of context

```
def read_configuration(file_path):
    try:
        with open(file_path, 'r') as configuration_file:
            return json.load(configuration_file)
    except FileNotFoundError as file_error:
        logger.error(f"Configuration file not found: {
            file_error}")
        raise
    except json.JSONDecodeError as json_error:
        logger.error(f"Invalid JSON in configuration file: {
            json_error}")
        raise
    except Exception as unexpected_error:
        logger.error(f"Unexpected error: {unexpected_error}"
            )
        raise
```

2 Methods

We need a way to identify, within the 2023 cohort, which teams produced code that reads as AI-generated. The detector is a means to an end here, not the object of study. We chose Binoculars [2] after trying two alternatives that failed for unrelated reasons: a RoBERTa-based prose classifier produced verdicts using a hand-picked threshold with no statistical justification, and DetectCodeGPT [6] on a 1.3B DeepSeek base model could not separate the 2018 and 2023 cohorts at all. Binoculars is zero-shot, comes with a principled calibration story, and has reported cross-generator robustness in its original evaluation, which matters because the students were using assistants from the OpenAI family (Copilot, ChatGPT) while our scoring model pair is not.

2.1 How Binoculars works

Binoculars runs two closely related LLMs on the same text and computes the ratio of the observer's perplexity to the cross-perplexity between them. The cross-perplexity term cancels out topic and style effects shared by both models, leaving only the divergence component. A low ratio means the two models agree

on what is likely next, which is the fingerprint of generated text. A high ratio means the observer gets surprised whereas the performer does not, which is how human text looks. We flipped the sign so that higher scores correspond to more AI-like output. We used Falcon-7B as the observer and Falcon-7B-Instruct as the performer, both in 4-bit quantization, which is the exact pair from the Hans et al. paper.

2.2 Pipeline

The pipeline clones every repository for a given GitHub organization, walks the file tree, filters out everything that is not student-authored source code, and feeds what remains through Binoculars in fixed 256-token chunks with a 64-token overlap. Per-chunk scores roll up to file-level means, and file-level means roll up to team-level means weighted by chunk count.

Most of the engineering effort went into the filter. A detector pointed at a raw repository clone will confidently flag every piece of framework scaffolding, every vendored library, every minified bundle, and every auto-generated build file, none of which the student wrote. We exclude these in three passes. A directory blacklist drops `node_modules`, `.git`, `dist`, `build`, `__pycache__`, `venv`, `site-packages`, and similar. A path-substring blacklist catches vendored libraries (Unity SDKs, Android SDKs, JaCoCo assets, auto-generated Javadoc trees) and the four Flutter platform-runner trees (`windows/runner/`, `macos/Runner/`, `ios/Runner/`, `android/.../io/flutter/`). An exact-filename and regex blacklist catches individual auto-scaffolded files: Django's `manage.py`, `wsgi.py`, `asgi.py`; CRA's `reportWebVitals` and `setupTests`; Next.js's `next-env.d.ts`; Android Studio's `ExampleUnitTest.java`; Django's numbered migration files. A regex pass on the first five lines of each remaining file sniffs for generated-file banners that code generators leave behind. The filter caught 161 files across both cohorts.

This part of the pipeline turned out to matter more than expected. An earlier run without the Django and Flutter filters flagged one team (team-2) as LIKELY AI based on scoring Flutter runner C++ entirely, and another (team-7) largely on Django scaffolding and auto-generated migrations. Both were false positives driven by framework code that no human student wrote either. After filtering, team-2 dropped out of the summary entirely (zero scorable chunks remained) and team-7's verdict became based on actual student-authored files.

2.3 Calibration against the 2018 baseline

The raw Binoculars score has no absolute meaning for us; what matters is whether a team sits higher in the distribution than the 2018 baseline. We calibrate the threshold by pooling all 2018 chunk scores, computing the baseline mean and standard deviation, converting each team's mean to a z-score against that baseline, and setting the LIKELY AI threshold at $z \geq \max_{2018}(z) + 0.05$. The 0.05 margin is there so the boundary is not literally on the highest-scoring 2018 team. The LIKELY HUMAN threshold is symmetric at the low end.

Teams with fewer than 15 chunks contribute to the baseline distribution but are forced to LOW_EVIDENCE in the final verdict regardless of their z-score.

This procedure has the property we want: the 2018 cohort cannot be flagged as LIKELY AI because the threshold is set above its maximum. A 2023 team clears the threshold only if it sits above every 2018 team. The calibrated threshold for our data was $z \geq +0.775$, set by `project-12-bus-advisory` at $+0.725$ plus the margin.

2.4 Cohorts

The 2018 cohort comprises eight student capstone teams from `UBCO-COSC499-Winter-2018-Term-1-2`, the last COSC 499 cohort before any publicly available AI assistant existed. The original organization contained ten repositories, but two were placeholder shells with no real code or PR activity, so we dropped them. The 2023 cohort contains 22 teams from `COSC-499-W2023`, the first cohort for which ChatGPT and Copilot were broadly available.

Under the calibrated threshold, the 2018 cohort returned eight UNCERTAIN verdicts and zero LIKELY AI verdicts. The 2023 cohort returned two LIKELY AI verdicts: team-1 at $z = +0.994$ with 485 chunks of evidence and team-7 at $z = +0.810$ with 65 chunks. Four further 2023 teams (team-3 at $+0.703$, team-9 at $+0.755$, team-10 at $+0.390$, team-11 at $+0.547$) sat in the upper UNCERTAIN band without clearing the threshold. Figure 1 shows the full z-score distribution across both cohorts. The Results section treats team-1 and team-7 as the AI-assisted cohort and compares their PR-level behaviour against the 2018 baseline, with the file-level view later revisiting the four upper-band teams as a secondary signal.

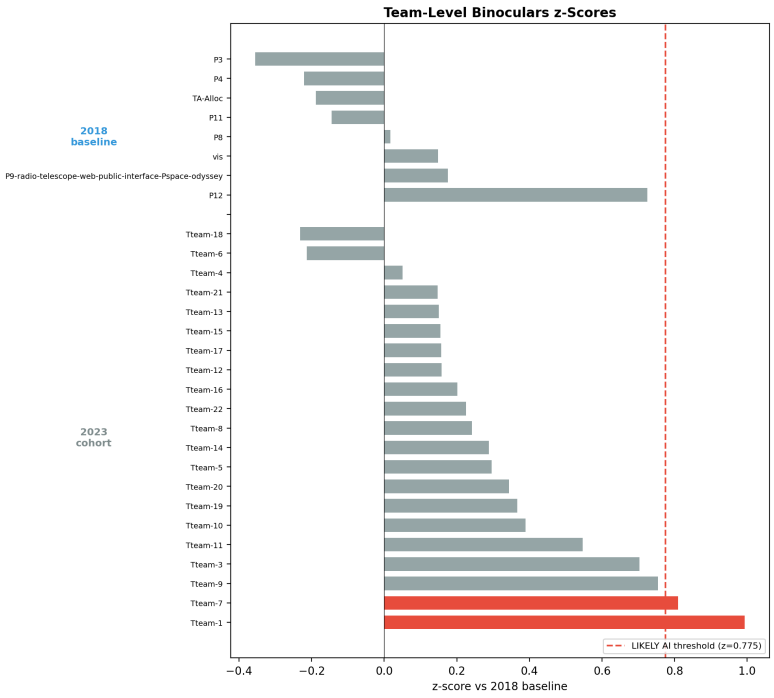

Figure 1: Binoculars z-scores for all 30 teams, grouped by cohort. The dashed red line marks the calibrated LIKELY AI threshold ($z = +0.775$). Two 2023 teams (team-1 and team-7) clear the threshold; all 2018 teams fall below it.

3 Results

The behavioural data comes from a per-PR event log derived from the Git history of each team's repository. Each pull request is labelled with tags describing the size and structure of its changes (small feature, large feature, small refactor, large refactor, number of features per branch, whether the branch name is meaningful or random) and the merge outcome (self-merge, reviewed merge, closed without merge, up-to-date or outdated against `main` at merge time). We compare two groups: the 2 teams flagged LIKELY AI by the calibrated Binoculars detector (321 PRs across team-1 and team-7) and the 2018 baseline (710 PRs across the eight teams). All numbers below are events per PR, which controls for the very different repo sizes across teams.

3.1 Branch naming and scoping

The 2018 baseline has zero meaningful branch names per PR. Branches are called `patch-1`, `branch-2`, `test-fix`. The AI-assisted cohort has 0.96 meaningful names and 0.04 random names, a near-total reversal. At the same time, 2018

Event (per PR)	2018	AI 2023	Ratio
Meaningful Branch Name	0.00	0.96	(undefined)
Random Branch Name	1.00	0.04	0.04×
One Feature Per Branch	0.24	0.93	3.84×
Multiple Features Per Branch	0.76	0.07	0.09×

Table 1: Branch naming and scoping, AI-assisted 2023 teams versus 2018 baseline.

teams packed multiple features into a single branch 76% of the time. AI-assisted 2023 teams do that 7% of the time and put one feature per branch the other 93%.

A descriptive branch name is not strictly necessary for the code to work, but anyone else on the team must know what the branch is about without reading the diff. The shift from `patch-1` to `add-user-authentication-flow` is the kind of change that software-engineering handbooks have recommended for decades. Likewise, one-feature-per-branch is what style guides have recommended for at least a decade. Neither shift is a disputed improvement. The mechanism is less clear: assistants may nudge developers toward bounded, single-intent changes, or branch-protection rules and course instruction may have pushed students the same direction independently. Both could be true. Either way, the 2018 approach of bundling everything into one branch was not something anyone was defending as best practice; it was just easier.

3.2 PR size and content

Event (per PR)	2018	AI 2023	Ratio
Small Feature Size	18.59	7.41	0.40×
Large Feature Size	10.43	2.43	0.23×
Small Refactor Size	19.81	17.93	0.90×
Large Refactor Size	4.98	3.51	0.71×

Table 2: PR content tags. Counts are per PR, averaged across all PRs in each cohort.

AI-assisted PRs are smaller. They contain 40% as many small features and 23% as many large features per PR. Refactoring activity is down too, but less dramatically (90% and 71% of the 2018 rate). This runs counter to the naive expectation: if an AI assistant makes code cheaper to write, you might predict bigger PRs, not smaller ones. The data shows the opposite.

Two complementary explanations fit. Tighter branch scoping mechanically forces smaller PRs; if each branch is one feature and each PR is one branch, PR size drops automatically. The reviewed-merge culture discussed next makes big

PRs unreviewable in practice, so teams learn to ship smaller ones. Either way, the AI-assisted cohort is doing what software engineers tell each other to do at conferences: small-scope changes, not week-long branches that land as a single 2,000-line diff.

Whether this is an improvement depends on what got left out. A 40% small-feature rate could mean the team is producing fewer features overall, or that they broke the same amount of work into more PRs. The current event log does not distinguish those. We return to this in the Limitations section.

Figure 2 shows the per-PR content breakdown side by side.

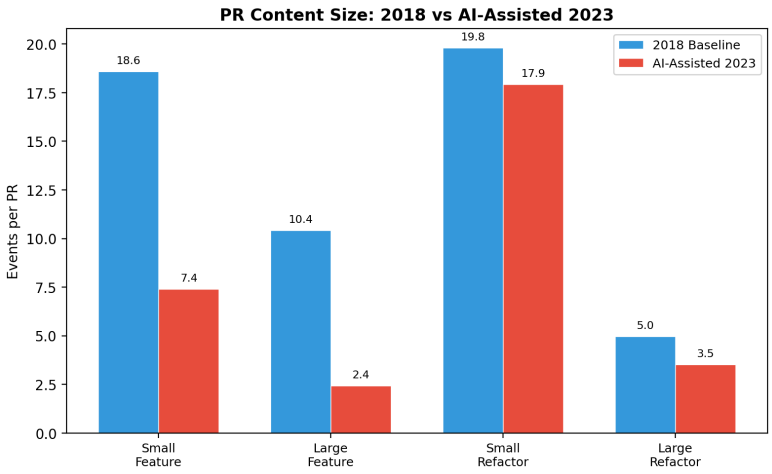

Figure 2: PR content size comparison. AI-assisted 2023 teams ship fewer features per PR than the 2018 baseline, with the gap larger for feature-sized changes than for refactors.

3.3 Review and merge behaviour

Event (per PR)	2018	AI 2023	Ratio
reviewed_merge	0.24	0.54	2.26×
self_merge	0.69	0.39	0.57×
up-to-date	0.45	0.90	1.99×
outdated	0.47	0.03	0.07×
merged	0.93	0.93	1.01×

Table 3: Review and merge outcomes.

The 2018 baseline is what a pre-review-culture workflow looks like. Authors merge their own PRs two-thirds of the time. Reviewed merges are rare. Branches go stale against `main` because no one rebases, and by the time they

merge, 47% are outdated. The overall merge rate is 0.93, meaning that whatever gets opened usually gets landed.

The AI-assisted cohort runs a different workflow. Reviewed merges are 2.26× the 2018 rate. Self-merges are down by 43%. Branches are kept up to date 90% of the time. Outdated branches at merge time are effectively zero. And the overall merge rate is identical to 2018's at 0.93, meaning the more careful workflow is not causing code to get stuck; it is just causing it to get reviewed before it lands.

This is the cluster of changes that most clearly reads as an improvement by mainstream software-engineering standards. Code review, branch hygiene, and up-to-date merges are all things that catch bugs before they reach production, spread knowledge across the team, and make the codebase easier to reason about historically.

Figure 3 shows the full set of workflow behaviours as grouped bars.

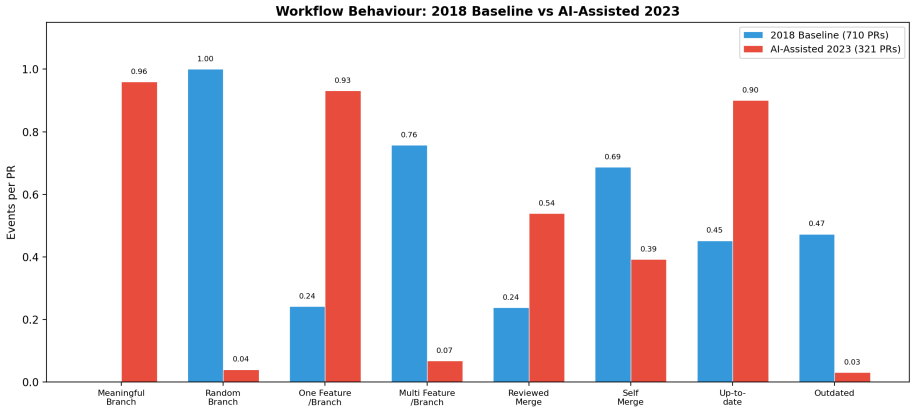

Figure 3: Workflow behaviour comparison across eight dimensions. Blue bars are the 2018 baseline; red bars are the AI-assisted 2023 cohort. The AI-assisted cohort shows higher reviewed-merge rates, near-zero random branch names, near-zero outdated branches, and a strong shift to one-feature-per-branch discipline.

3.4 The file-level view

The team-level verdicts are conservative by construction. A team that used an assistant on 30% of its code gets diluted by the 70% it wrote itself. To get beyond that averaging, we re-scored each file's chunk mean as a z-score against the 2018 baseline and counted the fraction of each team's files that individually met the LIKELY AI threshold.

The file-level view expands what the team-level view catches. team-1 and team-7 are still the only teams whose team-wide averages clear the bar, but

Team	Files scored	Files flagged	% flagged
2023 cohort, top by flagged %			
year-long-project-team-1	133	95	71.4%
year-long-project-team-9	140	69	49.3%
year-long-project-team-7	16	7	43.8%
year-long-project-team-3	125	54	43.2%
year-long-project-team-10	70	20	28.6%
year-long-project-team-11	25	7	28.0%
2018 baseline, top by flagged %			
project-12-bus-advisory	9	2	22.2%
project-9-radio-telescope	192	40	20.8%
visual-course-planner	48	7	14.6%

Table 4: File-level flag density. A file is flagged when its mean chunk score clears the LIKELY AI z-threshold of +0.775.

four additional 2023 teams (team-9, team-3, team-10, team-11) show concentrated file-level AI signal between 28% and 49% of their files, even though their team-wide averages stay in the UNCERTAIN band. That pattern is consistent with partial AI use: some modules generated mostly with an assistant, the rest written by hand, and the team-level average splitting the difference.

The 2018 baseline flags around 20% of files in its most web-heavy projects, which is the upper-tail noise floor on templated code from any era. Six 2023 teams sit above that floor. The true fraction of 2023 teams using assistants non-trivially is almost certainly closer to six out of twenty-two than to the two the team-level verdict catches. Figure 4 shows the full distribution.

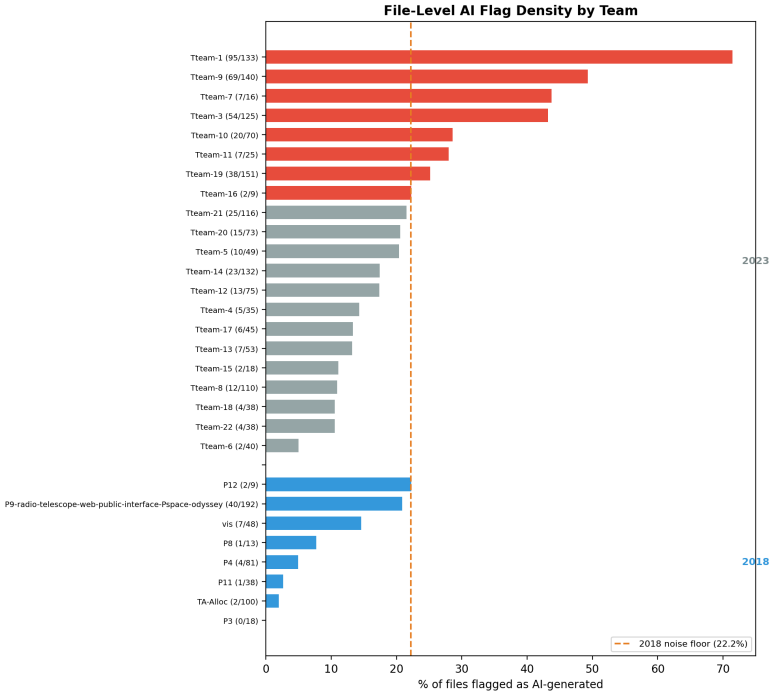

Figure 4: File-level AI flag density by team. The dashed orange line marks the 2018 noise floor (22.2%). Six 2023 teams sit above this line, against two that cleared the team-level threshold.

3.5 Summary of the comparison

The AI-assisted cohort differs from the 2018 baseline on every behavioural dimension measured, and the direction of each difference aligns with what software engineering best practices have recommended for years. Branch names are descriptive rather than numeric. Branches hold one feature rather than several. PRs are smaller. Review rates are higher. Self-merges are lower. Branches are kept up to date rather than left to go stale. Merge rates are unchanged. There is no behavioural dimension in the dataset where the 2018 workflow looks better.

We are not claiming the AI assistant caused all of this. Five years of GitHub evolution, course-instruction changes, and the spread of branch-protection conventions all happened during the same window. The subset of 2023 teams whose code reads as AI-generated also exhibits every workflow improvement the pre-AI literature spent two decades advocating for. Whether that correlation is causal is a question the data cannot answer.

4 What this says about AI-generated code

The results change what it means to ask whether AI-generated code is "good." The code the AI-assisted cohort produced sits within a workflow that treats each generated snippet as a reviewable unit, names branches descriptively, scopes each branch to one feature, keeps branches up to date, and routes everything through review before merging. None of that is a property of the generated code itself. All of it is a property of how humans around the code chose to work with it.

This matters for the common framing of the detection question. The implicit assumption behind detection is that AI-generated code differs from human-generated code in ways that might be harmful, and the detector's job is to flag when suspect content appears so humans can decide what to do about it. Our data suggest the relevant difference lies not in the code but in the workflow in which the code is embedded. And the workflow is, by every conventional metric available to us, better than what the 2018 baseline did without any assistance at all.

One reading is that AI assistants are catalyzing a workflow culture that was always right but hard to adopt without a forcing function. Once code is cheap to produce, careful review, scoping, and documentation become the rate-limiting steps, and teams invest in them accordingly. A second reading is that the AI-assisted cohort self-selects by discipline: the teams willing to use an assistant carefully enough to leave a detectable fingerprint are probably the same teams willing to maintain branch hygiene, and the real variable is conscientiousness rather than the tool. A third, more cautious reading is that the improvement is genuine but narrow. The PR event log does not capture everything that matters, and degradations invisible to it could coexist with everything reported here.

The data cannot separate those three. What it does rule out is any behavioural story where AI-assisted teams work worse than the pre-AI baseline by the standards software engineers have historically defended. On every dimension this dataset measures, they work better.

5 Limitations

Two cohorts, 30 teams total, and 1,031 PRs constitute a small sample, and all comparisons in this paper could plausibly be driven by the five-year gap rather than by the AI assistant. The cleanest way to separate the two effects would be to run the same analysis on every COSC 499 cohort from 2018 through 2024, which would let us see whether the behavioural shifts align with the Copilot/ChatGPT introduction or with an earlier point in the Git-culture evolution.

The PR event log captures workflow, not code quality in any deeper sense. A team that ships small reviewed PRs of shallow code looks identical in this data to a team that ships small reviewed PRs of well-thought-through code. The depth question needs a different instrument.

The detector itself measures a specific fingerprint (regions where two Falcon-7B variants agree closely on next-token distributions) and will miss AI-assisted teams that heavily edit their assistant output before committing. The behavioural comparison is only as good as the cohort definition, and the cohort definition is conservative. Teams that used an assistant carefully enough to dilute the fingerprint are probably in the UNCERTAIN band, not the flagged one, which means the real population of AI-assisted teams is larger than the two our team-level verdict caught. The file-level view gives a rough estimate of how much larger (six teams instead of two), but the confidence in that estimate is not strong.

Finally, "AI-assisted" in our usage means "code that reads to a particular detector as generated by a model in the Falcon family's similarity neighbourhood." It does not mean the student typed a prompt into ChatGPT and pasted the output. The fingerprint and the assistance are correlated but not identical, and that correlation is what we rely on when we talk about cohort behaviour.

6 Conclusion

Student capstone teams whose code reads as AI-generated produce pull requests that are smaller and more tightly scoped, get reviewed rather than self-merged, carry descriptive branch names, and land against an up-to-date main. None of that described the pre-AI baseline. The merge rate is unchanged, so the more disciplined workflow is not blocking code from landing; it is just changing the conditions under which it lands. Every behavioural dimension moved in a direction that established software-engineering practice would call an improvement.

Suppose AI assistants continue to correlate with this workflow shift; the question of whether AI-generated code is acceptable stops being a question about the code and becomes a question about the practices around it. Those practices were contested for years before assistants existed, and the arguments for them (review catches bugs, descriptive names aid understanding, small PRs are easier to reason about) did not depend on the code being human-authored. If anything, the data suggests that assistants are pushing teams toward habits the field endorsed years ago but never reliably saw in practice, at least among the teams that engage with them carefully enough to leave a fingerprint.

Future work. Extending the time series is the most direct move. Two cohorts cannot separate AI effects from era effects; six or seven cohorts probably can, and running the same pipeline on every COSC 499 cohort from 2018 through 2024 is tractable. Beyond that, pairing the PR-level data with outcome measures like defect density, time-to-first-hotfix, and static-analysis warnings per kilobyte would test whether the workflow improvements translate to better code or just better process. Per-language calibration matters too, since the detector behaves differently on Python than on TypeScript or Java, and the cohort language mix shifts year to year. The strongest follow-on comparison would be within-team: if the AI-heavy files inside a single repository can be separated

from the manually written ones, the workflow around each half can be compared directly, which cuts out most of the confounds the between-team analysis carries.

References

[1] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 34(8):1–34, 2025.

[2] Abhimanyu Hans, Avi Schwarzschild, Valeriia Cherepanova, Hamid Kazemi, Aniruddha Saha, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Spotting LLMs with binoculars: Zero-shot detection of machine-generated text. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[3] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Does AI-assisted coding deliver? a large-scale empirical study of quality, productivity, and maintenance effort in open-source projects. *arXiv preprint arXiv:2509.00000*, 2025.

[4] Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Meiyappan Nagappan. Whodunit: Classifying code as human authored or GPT-4 generated — a case study on CodeChef problems. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. ACM, 2024.

[5] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P)*, pages 754–768. IEEE, 2022.

[6] Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2025.

[7] Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. An empirical study on automatically detecting AI-generated source code: How far are we? *arXiv preprint arXiv:2411.04299*, 2024.

[8] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. How secure is AI-generated code: A large-scale comparison of large language models. *Empirical Software Engineering*, 30(2), 2025.

[9] Zhen Yang et al. Debt behind the AI boom: A large-scale empirical study of AI-generated code in the wild. *arXiv preprint arXiv:2603.28592*, 2026.