Measuring the Impact of AI Generated Code in MERN Applications

Student 1: Georgios Panormitis Latos
Student 2: Joey Karlsson
Supervisor: Farnaz Fotrousi

***Abstract*—Generative AI is increasingly being used to create code, however most relevant studies rely on small, isolated tasks that do not fully encapsulate all the complexities of full stack development. We studied the impact of AI generated code in the scope of MERN (MongoDB, Express, React, Node) applications. Across three open source projects, we selected five target files per app and regenerated each file twice using two language model types, more specifically a compact reasoning SLM (Phi 4 14B Reasoning Plus) and a large instruct LLM (Llama 3.1 70B Instruct). Each generated file was integrated as a single file replacement on a dedicated branch, keeping the rest of the codebase unchanged. Quality was measured with complementary static and runtime signals (SonarQube, ESLint, Lighthouse, OWASP ZAP). For the statistical analysis, we applied paired Wilcoxon tests to the raw per file per run values. Overall, AI was competitive with human code on most dimensions, with statistically significant improvements in internal simplicity and effort (Halstead Effort) and composite performance, but made regressions in visual stability (CLS) and security. Specifically, AI reduced Halstead bugs and effort and improved performance, while human baselines showed fewer security vulnerabilities and lower CLS. Comparing models, Llama tended to produce lower complexity code (cyclomatic and average complexity, lower Halstead effort), whereas Phi yielded fewer security findings.**

I. Introduction

Artificial intelligence, particularly in the form of Generative AI like ChatGPT and GitHub Copilot, has gained significant popularity in recent years for its potential to improve software development productivity. These tools offer to automate tasks such as code generation, debugging, and documentation, which can save developers considerable time and effort [1], [2]. While preliminary evidence suggests productivity gains, most existing studies evaluated these Generative AI models using small scale, isolated tasks like algorithmic puzzles or single function code snippets [3], [4].

Full stack development, which involves the integration of frontend, backend, and database components, presents a more complex environment. This complexity introduces challenges such as ensuring code integration, mitigating security risks, maintaining long term code health, and navigating the different coding styles of developers. These are nuances that previous small scale studies often do not capture. There is limited empirical evidence regarding the holistic performance and practical impacts of AI assisted coding tools in realistic, full stack software development scenarios.

This study investigated the capability of Generative AI models in the context of full stack development. We evaluated whether these AI models enhanced efficiency, maintained high code quality, and improved code security and robustness compared to traditional manual coding practices. By addressing this research gap, we sought to provide clearer guidance for integrating Generative AI tools effectively into real world software engineering workflows and to lay the groundwork for future investigations in this rapidly evolving field.

II. Objectives

This study aimed to understand the real impact of AI generated code in realistic full stack (MERN) settings. Our objectives were as follows:

- Evaluate how AI generated code compares to human written code in realistic MERN applications across reliability, maintainability, good practice, performance, and security.
- Compare two representative models (a compact reasoning SLM and a large instruct LLM) to see whether model type or scale meaningfully changes code quality outcomes.

III. Related Work

Prior research on AI generated code has mostly been focused on assessing effectiveness through isolated and controlled experiments. For instance, Khan et al. [1] conducted an evaluation of ChatGPT's code generation capabilities, highlighting its strengths in producing concise, modular code and handling errors, but also noting limitations in addressing visual and graphical coding challenges. Similarly, Agarwal et al. [3] developed the Copilot evaluation harness to evaluate the performance of Large Language Models in real world IDE contexts, covering tasks such as code generation, documentation, bug fixing, test case generation, and workspace understanding.

Tan et al. [2] explored developers' interactions with various Generative AI tools, including GitHub Copilot, focusing on task completion rates, code quality, and developers' perceived productivity. Their findings suggest improvements in task completion and productivity, but the results vary considerably depending on the developers' experience levels and the complexity of the coding tasks involved. At the same time, the research also reveals challenges and areas for improvement in usability, and the integration of these tools into developers' everyday practices. Paradis et al. [11] found that using AI to

assist in a coding environment made developers 21% faster in producing code.

Despite the insights provided by these studies, a research gap remains concerning the evaluation of Generative AI tools in comprehensive, realistic, and integrated development scenarios, specifically in full stack contexts. This research aimed to address this gap by offering a holistic assessment of Generative AI tools in realistic full stack development environments.

Beyond productivity studies, Molison et al. [9] examined the internal quality of LLM generated versus human written code using SonarQube (bugs, code smells, and technical debt) across programming tasks, and argued for reproducible static analysis pipelines in LLM evaluation. While their analysis focused on generated solutions in isolation, our work evaluated AI drop in file replacements inside running MERN applications and complemented static measurements with runtime measurements and active security probing. Nam et al. [12] found that outside the scope of code generation, Large Language Models excelled in boosting developer productivity by helping them read and understand large codebases they had not seen before.

IV. Research Methodology

A. Research Questions and Hypotheses

RQ1: *Did AI generated code produce higher quality code than human written code in a full stack application across reliability, maintainability, good practice, performance, and security?*

- $\mathbf{H_0}$**:** No significant differences across the metrics between AI generated and human written code.
- $\mathbf{H_1}$**:** At least one metric shows a significant improvement for AI generated code compared to human written code.

RQ2: *Between the reasoning model and the instruct model, which produced higher quality code across the same metrics?*

- $\mathbf{H_0}$**:** No significant differences between the two models across the metrics.
- $\mathbf{H_1}$**:** One model significantly outperforms the other on at least one metric.

B. Study Design

Following Wohlin et al. [7], we treated this work as an experiment in software engineering. Their guide structures experiments into scoping, planning, operation, analysis and interpretation, and reporting, and categorizes validity as conclusion, internal, construct, or external, with practical checklists for threats and mitigations. Given our paired, small sample design and non normal metrics, we applied non parametric hypothesis tests (for example Wilcoxon) appropriate for paired comparisons.

This study employed a controlled experimental design, using a comparative, black box evaluation approach. We focused on directly assessing the quality of code generated by two AI models, Phi 4 14B Reasoning Plus and Llama 3.1 70B Instruct, in a realistic full stack development scenario.

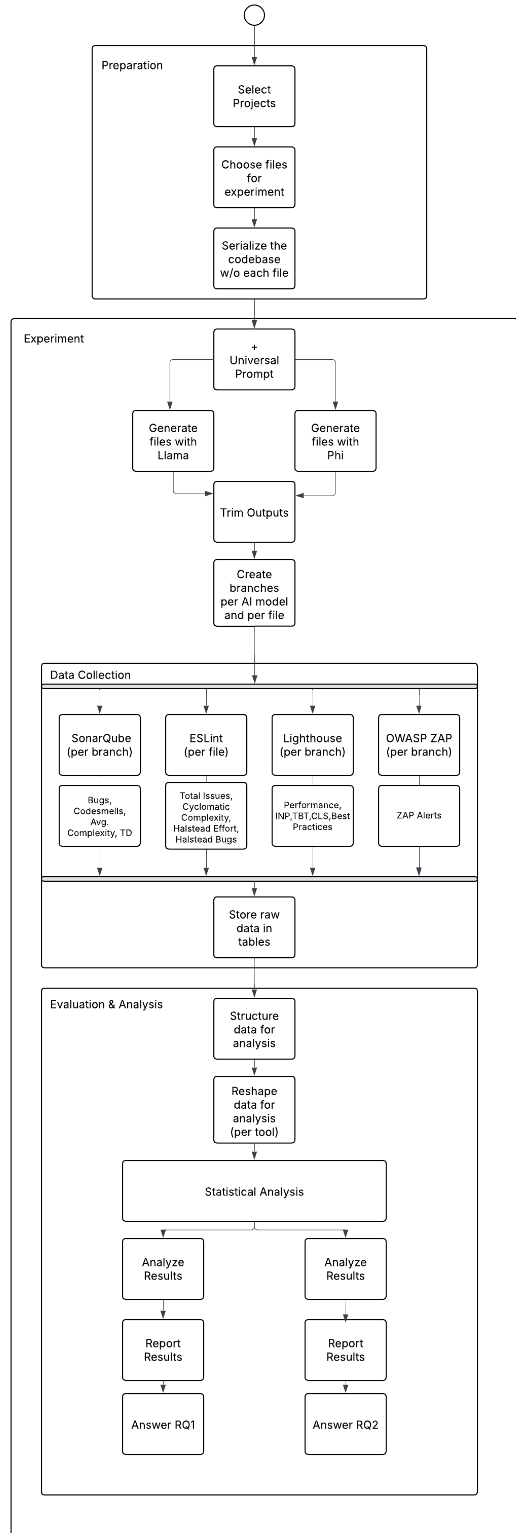

Fig. 1. Experiment process diagram.

We chose the MERN stack (MongoDB, Express, React, Node) because it is one of the most popular and powerful stacks for developing modern day web applications per S.

Aggarwal et al. [6]. It is widely adopted in production, offers a clean client server split, maps naturally to MVC, and uses one language end to end (JavaScript). Its tooling (build, test, lint) and web patterns (REST APIs, auth, state, routing) let us measure code quality under conditions that mirror real world full stack development.

Our experiment began with the selection of three open source MERN stack projects that mimic real world applications. To ensure these applications were suitable for our needs, we established a standard set of selection criteria:

1) The application must use the MERN stack.
2) The application must have a distinct client and server, which is the standard architecture for full stack applications in production.
3) The application must use the Model View Controller (MVC) architectural pattern, an industry standard for separating the concerns of an application.
4) The application must be open source on GitHub and should be working out of the box with the necessary setup.

C. Selected Projects

The MERN stack projects selected for this study based on the above criteria are:

- **1. MERN Thinkboard:** burakorkmez/mern-thinkboard & The files selected for this project:
  - CreatePage.jsx
  - Note.js
  - NoteCard.jsx
  - NoteDetailPage.jsx
  - notesController.js
- **2. MERN Social Media:** ed-roh/mern-social-media & The files selected for this project:
  - Form.jsx
  - MyPostWidget.jsx
  - auth.js
  - posts.js
  - users.js
- **3. MERN Chat App:** burakorkmez/fullstack-chat-app & The files selected for this project:
  - Conversation.jsx
  - Message.jsx
  - auth.controller.js
  - message.controller.js
  - user.controller.js

D. Experiment

Our goal was to test whether an language model could produce drop in replacements for individual files inside our selected applications. For each application we selected five target files (at least two front end) that are modular and interact with the rest of the codebase via existing imports, exports and endpoints.

Selection of these files was based on the ones with the higher logic density and centrality to core features for example controllers that coordinate requests and React components that manage state and routing so a replacement could meaningfully affect reliability, maintainability, good practice, performance and security.

a) Setup: For a given target file, we constructed a full repository context by serializing the entire codebase, including paths and content, except the target file. We then prompted the model to regenerate the missing file at its original path. For creating our custom made prompt we followed Phoenix & Taylor's "Give Direction" (role), "Specify Format" (single, strict file output), and "Avoid hallucinations with reference" (use only repo context) to make outputs reliable and production-safe [8]:

```
"You are a MERN stack developer. Generate the
 ↪  missing file `{filename}` based on the
 ↪  complete codebase provided below.

**RULES:**
- Analyze the codebase to understand existing
 ↪  patterns, imports, and dependencies
- Only use imports and functions that exist in
 ↪  the provided codebase
- Follow the same coding style and structure
 ↪  as similar files
- DO NOT invent or hallucinate
 ↪  imports/libraries that don't exist in the
 ↪  codebase.
- DO NOT assume any other functions/files
 ↪  exist in the codebase apart from the ones
 ↪  i sent you.
- Component should be able to work correctly
 ↪  with the existing codebase without any
 ↪  changes.

The file will be saved at this path: `{path}`
 ↪  so make sure imports are correct.
Generate only the complete code for
 ↪  `{filename}`, no explanations, no markdown
 ↪  formatting, the response will be saved as
 ↪  `{filename}` and it should be good to go.

**CODEBASE:**
{codebase}"
```

b) Generation: We invoked the models via the OpenRouter API and generated two variants per target file (*Llama* and *Phi*). When a response contained any extra text, we manually trimmed it and kept only the file contents. For traceability and reproducibility, we logged every request and the raw response. You can find our replication package in the Appendix (CLI).

c) Branching and artifacts: Each AI generated file was integrated by creating a new branch that was identical to the Human baseline except for replacing exactly one file at the original path. Per application this yielded:

- 1 Human baseline branch (original code),
- 5 *Llama* branches (one per regenerated file),
- 5 *Phi* branches (one per regenerated file).

These branches were the inputs to our metrics analyses except ESLint, which analyzed the files individually. Because only a single file differed per branch, any metric change can be

attributed to the generated file's integration with the existing codebase.

E. Variables

Independent variables: Code Origin

- Manual coding (original human written code)
- Generated code with Phi 4 14B Reasoning Plus
- Generated code with Llama 3.1 70B Instruct

Dependent variables: We organized the outcome measures into five categories that combined static signals (SonarQube, ESLint) with runtime signals (Lighthouse, OWASP ZAP):

- **Reliability**
- **Maintainability**
- **Good Practice**
- **Performance**
- **Security**

Static counts and runtime latencies or alert counts were interpreted as lower is better, whereas Lighthouse scores (for example Performance and Best Practices) were interpreted as higher is better. Because each AI branch differed from the Human baseline by exactly one replaced file, we analyzed matched pairs (Human vs. AI) so that any differences could be attributed to the generated file's integration with the existing codebase.

F. Metrics by Category

i) Reliability

We measured reliability to see if the code behaves correctly and if the page stays stable for users. This metric determines how the page interacts when users use the app, showing up in problems like incorrect outputs or shifting layouts, even if the code is functionally correct.

Bugs: An issue that can lead to incorrect behavior at runtime.

Halstead Bugs: An estimate of defect density based on operators and operands in the code.

Cumulative Layout Shift (CLS): Measures unexpected layout movement while the page is visible to the user. Lower is better for visual stability.

ii) Maintainability

We measured maintainability because code lives for a long time. Files that are complex or very large slow down reviews, make onboarding in companies harder, and increases the chance of mistakes later.

Code Smells: Anti patterns that are not bugs but make the code harder to change or understand.

Technical Debt: Estimated time (in minutes) to bring the code to a clean baseline.

Average Complexity: An aggregate complexity score across the project that rises as control flow gets harder to follow.

Cyclomatic Complexity: The number of independent paths through a function or file. Higher values mean harder testing and maintenance.

Halstead Effort: An estimate of how hard the code is to understand based on token counts.

iii) Good Practice

We checked general hygiene which represents the difficulty of keeping the project healthy day to day. Good practice reduces regressions, keeps the codebase consistent, and makes it easier for new contributors.

Total Issues: The number of ESLint rule violations, covering style problems and possible errors.

Best Practices: A score gathered from checking modern web hygiene practices, such as HTTPS use and safe APIs.

iv) Performance

We measured performance because users notice speed first. Slow pages hurt engagement and make everything else feel worse, even if the features are correct.

Performance score: Composite score that summarizes loading and responsiveness.

Total Blocking Time (TBT): The total time the main thread is blocked long enough to delay input handling.

Interaction to Next Paint (INP): The latency from a user action to the next paint.

v) Security

We included security because users and data can be at risk from malicious attack. While the code may look stable and function correctly, during runtime the code can still expose unsafe endpoints or contain improperly validated input fields.

ZAP Findings: Alerts raised by OWASP ZAP during an active scan, grouped by type of malpractice.

G. Why these tools

We chose SonarQube for bugs, smells, debt, and complexity because it gives strong static signals for JS and JSX at the file level and is easy to automate. We chose to write a script using ESLint for rule violations, cyclomatic complexity, Halstead metrics, and size because it reports per file numbers that match our paired comparisons and is one of the most widely used tools in the JavaScript ecosystem. We chose Lighthouse for CLS, best practices, performance, TBT, and INP because it exercises the app in a browser and produces stable, repeatable scores. You can find that script in the Appendix (Tool) We chose OWASP ZAP for security because it actively probes the running app and finds problems that do not appear in static code scans.

H. AI Tools Used

The experiment used the latest available versions of a reasoning model and an instruct model, more specifically we chose Microsoft's Phi 4 14B Reasoning Plus and Meta's Llama 3.1 70B Instruct. This selection provided a valuable comparison between two distinct classes of models.

- **Phi 4 (14B)** was selected as a leading Small Language Model (SLM). Developed by **Microsoft**, Phi models are specifically engineered for high performance in a compact size. Its strength lies in providing powerful reasoning and coding abilities while being computationally efficient.

[5] This makes it an ideal candidate for assessing the capabilities of smaller, highly optimized models that are becoming increasingly prevalent.

- **Llama 3.1 (70B)** was chosen as it represents the cutting edge of Large Language Models (LLMs). Developed by **Meta**, this model is designed for tasks involving following instructions. Its much larger parameter count (70B) allows for a deeper, more general understanding. Including it in our experiment allowed us to benchmark the performance of a top tier, large scale general model against its smaller, more specialized counterpart.

I. Data Collection

- **SonarQube Data**: For each project we analyzed 11 branches: the Human baseline (original code) plus 10 variants created by replacing exactly one file (5 Llama and 5 Phi). From SonarQube we extracted four branch level metrics (interpreted as lower is better): **Code Smells**, **Bugs**, **Average Complexity**, and **Total TD Mins**.
- **ESLint Data**: For each of the three projects we analyzed 15 files per project 5 human written files and their two AI variants (*Llama*, *Phi*) 5 matched triplets per project total n=15 files across all projects for the pooled analyses. Metrics were produced by a custom Node script that runs ESLint and parses each file with Babel to compute complexity and Halstead figures. From this script we used the following raw per file measures all interpreted as lower is better **Total Issues** from ESLint messages **Cyclomatic Complexity** computed as one plus decision points **Halstead Effort** and **Halstead Bugs** computed from Halstead volume.
- **Lighthouse Data**: Every branch was tested for every function and page using timespan mode to gain a comprehensive set of metrics that spanned across the entire website, rather than any one single page. From Lighthouse we extracted the following metrics: **Performance**, **Cumulative Layout Shift**, **Interaction to Next Paint**, and **Best Practices**.
- **ZAP Data**: Every branch was also analyzed using the ZAP tool to dynamically seek out any possible security vulnerabilities. The counts of every instance were then recorded, with security vulnerabilities being common malpractices like path traversal and cross domain misconfiguration. Since each project contained a different set of vulnerabilities, the metrics between the different projects varied, though they were constant within each project.

J. Data Analysis

You can find our data tables in the Appendix (Runtime Data), (Static Data) After running the generation experiment we were left with structured data ready for statistical analysis, more specifically:

- **SonarQube Data**: We computed per metric deltas as:

$$\Delta = \text{Variant} - \text{Human}$$

(negative Δ indicates an improvement over the baseline). This produced 10 delta rows per project (30 total across all projects). For statistical analysis we followed two questions on the pooled data:
In order to evaluate the AI against the human baseline, we ran a one sample Wilcoxon signed rank test on the deltas ($\Delta = \text{Variant} - \text{Human}$) with the one sided alternative $\mathrm{H}_1: \ \text{median}(\Delta) < 0$ (improvement equals lower is better). We used the Wilcoxon zero method (zero differences dropped) and reported n=30, W, and the p value. In order to evaluate the AI against the human baseline, we paired *Phi* and *Llama* deltas by target (for example **llama authcontroller** vs. **phi authcontroller**) and applied a paired, two sided Wilcoxon test with the zero method, yielding n_{pairs}=15 with corresponding W and p. We set α=0.05 and reported sample sizes and median or mean deltas for both RQ1 and RQ2.
- **ESLint Data**: For each metric, files were aligned by identical **File Name** across variants and reshaped into a wide table with one row per (project, file) and one column per variant (Human, Llama, Phi).
Paired Wilcoxon signed rank tests (two sided) were run per metric on matched pairs, using the Pratt method to handle zero differences and a significance threshold of $\alpha = 0.05$. We computed five comparisons:

  1) Human vs Llama
  2) Human vs Phi
  3) Human vs AI_avg(Llama,Phi) (*per file mean of available AI variants*)
  4) Human vs AI_best(min) (*optimistic upper bound: per file minimum across AI variants*)
  5) Llama vs Phi

  Analyses were performed both per project (n=5 matched files per comparison) and on the pooled dataset across all projects (n=15). For each test we report the sample size n, Wilcoxon statistic W, two sided p value, and a conservative winner label: only when $p<\alpha$ and the median of (*left* $-$ *right*) is nonzero do we declare the side with the lower median as the winner, otherwise we report no significant difference. The full set of tests spans 4 metrics $\times$ 5 comparisons for each scope (3 projects plus pooled).
- **Lighthouse Data**: To perform the statistical analysis, the data from every project was compiled to run a Wilcoxon test holistically across the entire dataset (n=15). First, every branch was compared with its respective baseline branch to get the statistical analysis of differences between AI and human written code, giving us the W statistic, p value, and an overall technical winner. A Mann Whitney U test was performed holistically between each pair of files to give an analysis of differences between Phi and Llama, giving us the U statistic, p value, and an overall technical winner. Although each analysis lists a winner, we only considered it to be statistically significant when $p<\alpha$ where $\alpha = 0.05$.

- **ZAP Data**: Once we had the counts of security vulnerabilities, we performed a statistical analysis by compiling all instances into one lump sum, then comparing every project holistically by matching each branch with its respective baseline, and performing a Wilcoxon signed rank test across all projects, which gave us a W statistic, p value, and an overall winner. To statistically compare the two models, every file was paired and then all files were analyzed using a Mann Whitney U test which gave us the U statistic, p value, and an overall winner. Although each analysis lists a winner, we only considered it to be statistically significant when $p < \alpha$ where $\alpha = 0.05$.

V. Results: Human vs AI generated

A. Reliability

TABLE I
RELIABILITY STATISTICS

Metric	Aggregate P value	Winner
Bugs	0.921	No significant difference
Halstead bugs	0.048	AI
CLS	0.0009	Baseline

- **Bugs:** On average, the AI generated code contained fewer bugs than the original, however the differences in bug counts were negligible, with bug counts at 0 for the most part and rising to 1 or 2 in certain cases, creating little difference between the baseline code and the generated code.
- **Halstead bugs:** The generated code produced a statistically significant reduction in Halstead bugs compared to the baseline, suggesting fewer projected bugs in the generated code. One notable example is the file `Form.jsx` for the Sociopedia app, where the original had a projected bug count of 1.3513, dropping to 0.3975 with Llama and 0.4728 with Phi.
- **CLS:** The human written code produced far fewer large layout shifts, suggesting that the generated code was more prone to visual disruptions. This was important because all three codebases had a baseline CLS of 0, but in most cases the AI generated files created layout shifts.

In some runs, the AI code fixed problems that existed in the original source code, while in other runs the AI code introduced more problems, making it unreliable overall. However, the AI generated code was generally less complex and more robustly written than the human written code, as shown by the Halstead metrics, which suggests that AI generated code was generally simpler. Finally, the AI generated code was more prone to unexpected layout shifts in the frontend, which is expected since the AI does not have vision capability and cannot understand how a web page looks and reacts when loaded.

B. Maintainability

- **Code Smells:** The results show that there is little statistical difference and variation in the number of code smells between AI generated and human written code.

TABLE II
MAINTAINABILITY STATISTICS

Metric	Aggregate P value	Winner
Code Smells	0.822	No significant difference
Technical Debt	0.231	No significant difference
Cyclomatic Complexity	0.977	No significant difference
Halstead Effort	0.011	AI
Average Complexity	0.596	No significant difference

- **Technical Debt:** In some projects, such as within the Chat App and the Thinkboard app, the generated code created much more technical debt, while in others the resulting files maintained the technical debt. When averaged together, the statistics suggest that humans generate more maintainable code, but the results are not statistically significant.
- **Cyclomatic Complexity:** The results showed that there is little statistical difference between the cyclomatic complexity of the AI generated and human written code. In certain cases, we found drastic drops in cyclomatic complexity, like the `Form.jsx` file in the Sociopedia app, which dropped from a baseline of 26 to 3 with Llama's generated file. In other cases within the same codebase, such as the `posts.js` controller, we found Phi increased the cyclomatic complexity from 6 to 12, so overall the results are inconclusive.
- **Halstead Effort:** Generally, the code written by AI was more readable than the human generated code, as well as being markedly less complex, such as in the Sociopedia project, where the baseline dropped from 4395.23 to 18.09 in the file `posts.js`, which suggests better future maintainability and understandability.
- **Average Complexity:** Depending on the project, AI decreased the perceived complexity or increased it, and though the results show that the baseline code was less complex, the result is not statistically significant.

Overall, the data showed that humans and AI created code with similar maintainability, though depending on the project we found that there were large variations. Certain projects allowed the AI to create much more readable code that protected against future problems, while other projects created many more issues in maintainability. For example, within the Chat App and the Thinkboard app, Llama generated controller files that increased the technical debt from 331 minutes to 437 minutes, and from 64 minutes to 364 minutes, which suggests that these specific files are problematic for the AI model.

C. Good Practice

TABLE III
GOOD PRACTICE STATISTICS

Metric	Aggregate P value	Winner
Total Issues	0.616	No significant difference
Best Practices	0.564	No significant difference

- **Total Issues:** Statistically, the generated code created slightly more issues than the human generated code, however the difference is not statistically significant and variation was low.
- **Best Practices:** On average, the generated code had fewer best practices than the human generated code, although the result is not statistically significant and variation was low.

The results for writing generally accepted good code are inconclusive. In our results we found that the human written code adhered to good practice slightly better than the AI generated code, but in our experiments both sets of code were well written in terms of standard practice, with files missing at most one best practices check. Interestingly, the file with the highest total issues was the human written `MyPostWidget.jsx` within the Sociopedia codebase.

D. Performance

TABLE IV
PERFORMANCE STATISTICS

Metric	Aggregate P value	Winner
Performance	0.03	AI
INP	0.416	No significant difference
TBT	0.215	No significant difference

- **Performance:** Both codebases passed most of the metrics within the performance category on Lighthouse, however the AI generated files were able to consistently hit more performance metrics than the human generated files to a statistically significant degree. This was a result of the models serving images in next generation formats where the codebases did not.
- **INP:** In most cases, the human written code did better in minimizing the time from the user interaction to showing the next screen, however the differences were usually minimal except in the case of the Thinkboard app, where the AI generated code consistently cut the time taken by half from 130 ms to around 60 ms across the board.
- **TBT:** The AI generated files slightly edged out the human written files in terms of blocking the main thread, however the difference was not statistically significant. In the Thinkboard app, where the human written files blocked the thread, many of the AI generated files fixed the issue that was present in the baseline, bringing the TBT to 0 ms in most cases.

The results show that in terms of performance, AI generally created better files, however the results are mostly inconclusive. In many cases the AI generated files increased performance, which may be a result of the models using the simplest approaches to most problems, which normally gives better performance over the human written code that may not always be the most efficient solution. This may also be a result of LLMs being trained on well written code in terms of efficiency, and therefore knowing patterns that the humans were not aware of.

E. Security

TABLE V
SECURITY STATISTICS

Metric	Aggregate P value	Winner
Security vulnerabilities	0.0002	Baseline

The baseline code contained many security vulnerabilities to begin with, however the AI generated code very rarely fixed any of the vulnerabilities, and often added more instead. As shown by the statistical analysis, there is a significant increase in the number of vulnerabilities, and often more instances of the same vulnerability. For instance, in the Thinkboard app, we found that the baseline had 23 instances of the server leaking information via `"X-Powered-By"` HTTP response header fields, however when using Llama's generated `createPage` file, the number of instances of this vulnerability rose to 174, suggesting that the AI did not generate with a focus on security and instead focused on the criteria for completion.

VI. RESULTS: PHI VS LLAMA

A. Reliability

TABLE VI
RELIABILITY STATISTICS

Metric	Aggregate P value	Winner
Bugs	0.157	No significant difference
Halstead bugs	0.055	No significant difference
CLS	0.594	No significant difference

- **Bugs:** The results show little statistical significance in the difference in bug counts, and the data agrees with this, since there was little variation in the bug counts, not rising above 2.
- **Halstead bugs:** Although Llama produced fewer Halstead bugs in metrics, the result was not enough to be statistically significant. One notable contradiction to this result is in the Chat App that we studied, where the file `Message.jsx` generated by Phi had a Halstead bug count of 0.03, much lower than Llama's Halstead bug count of 0.104.
- **CLS:** The result was not statistically significant and there was little variation. While the metrics show that the technically better result was Phi, one major outlier is shown in the Sociopedia app in the file `users.js`, where Llama generated a file with a CLS of 0.003, whereas the Phi generated file had a CLS of 0.01, a three fold increase.

Overall, the results were inconclusive, with none of the p values dropping below 0.05, although the Halstead bug count difference came close in favour of Llama. This was especially prevalent in the `posts.js` file under the Sociopedia app, where Llama's bug count was as low as 0.006, whereas Phi's file came out as 0.527. Despite this, it was not enough to declare a clear winner in terms of reliability.

TABLE VII
MAINTAINABILITY STATISTICS

Metric	Aggregate P value	Winner
Code Smells	0.209	No significant difference
Technical Debt	0.212	No significant difference
Cyclomatic Complexity	0.031	Llama
Halstead Effort	0.048	Llama
Average Complexity	0.033	Llama

B. Maintainability

- **Code Smells:** The results show that there is no statistical difference and little variation between the number of code smells between the thinking model and the instruct model.
- **Technical Debt:** In terms of technical debt, we found little statistical difference. Although our Llama model generally created less technical debt in generated files, we also found that generating the note controller file in the Thinkboard app using Llama created a significantly larger technical debt of 364 minutes over Phi's 64 minutes, which was the same as the baseline.
- **Cyclomatic Complexity:** The results showed that across the board, our instruct model created code with less cyclomatic complexity than our smaller thinking model, which suggests that a simpler approach was used. One significant example is the `posts.js` file in the Sociopedia app, where Llama's generated file had a cyclomatic complexity of 1, whereas Phi's generation created a cyclomatic complexity of 12.
- **Halstead Effort:** We found that our smaller thinking model consistently created files with a higher Halstead effort than the larger instruct model that generated the file with minimal prerequisite tokens. This can be seen again in the same `posts.js` file in the Sociopedia app, where the Halstead effort jumped from 18.09 with Llama to 13279.93 with Phi.
- **Average Complexity:** Similar to cyclomatic complexity, we found that across the board our larger instruct model generated less complex files than the smaller thinking model, though the overall variation was low. This suggests that the lack of prior thinking tokens may contribute to creating conceptually simpler code.

Overall, the data showed that using a larger model without explicit thinking enabled generated more readable and maintainable code. A major outlier against this conclusion is the file generated by Llama that created technical debt, however in general, the thinking SLM created more complex code that is harder to read and maintain, most likely due to the thinking tokens contributing to more comprehensive code, while the instruct model generated a solution without extra considerations.

C. Good Practice

- **Total Issues:** Generally, the total issues between the two different models had little variation, and so the result is not statistically significant.

TABLE VIII
GOOD PRACTICE STATISTICS

Metric	Aggregate P value	Winner
Total Issues	0.732	No significant difference
Best Practices	0.564	No significant difference

- **Best Practices:** On average, the code generated by Phi met more best practices requirements than the code generated by Llama, although the result is not statistically significant and the overall variation was low since for the most part all best practice requirements were met.

The results for writing generally accepted good code are inconclusive, even though our results found that Phi had slightly better practice than Llama. It's not statistically significant enough to be a conclusive result, one interpretation is that with the extra thinking tokens before outputting code, the smaller thinking model was able to make and account for more considerations, whereas the larger instruct model output code with minimal prior thought, which led to simpler solutions with less focus on robustness and practice.

D. Performance

TABLE IX
PERFORMANCE STATISTICS

Metric	Aggregate P value	Winner
Performance	0.158	No significant difference
INP	0.330	No significant difference
TBT	0.237	No significant difference

- **Performance:** Both Llama and Phi were generally able to pass most performance audits, however Phi failed more in general than Llama, though the difference is not statistically significant. The most notable example is in the Thinkboard app's `NoteDetailPage.jsx` file, where Phi's generated file spent significant time executing JavaScript code and thereby blocking threads, which is two failed performance audits over Llama's generated file.
- **INP:** Both models generated code with similar times to next paint, and there was little variation, leading to a statistically insignificant result.
- **TBT:** Although statistically our smaller thinking model edged out our larger instruct model, there were a few interesting exceptions. When creating the Chat App files, the models generally kept the blocking time at 0 ms, however in two instances our Phi model generated files that brought the total thread blocking time to above 500 ms. All things considered, the result is not statistically significant enough to declare a clear winner.

In general, both models created files with consistent performance, with no statistically significant winner. This may suggest that creating optimized files is not dependent on whether or not a model has thinking capabilities.

TABLE X
SECURITY STATISTICS

Metric	Aggregate P value	Winner
Security vulnerabilities	0.0027	Phi

E. Security

Although our initial result showed that AI generated files were generally prone to security vulnerabilities, as shown by the fact that they often created files that resulted in more security vulnerabilities than the baseline, our second result shows that our thinking model was significantly more capable of creating files without security vulnerabilities. This suggests that with extra thinking tokens, the model considered more security measures than the instruct model, which is likely more focused on creating files that solved the problem than on extra considerations. This is most notable in the Thinkboard project, where Phi consistently created files with fewer security vulnerabilities than Llama across all files. For example, the instances of Information Disclosure suspicious comments dropped by a minimum of 30, with the most significant drop between the two generated `NoteDetailPage.jsx` files, where the count of this security vulnerability dropped from 140 to 80.

VII. DISCUSSION

Throughout the experiment, AI generated code was competitive with human written code on most static quality and performance indicators, with three consistent caveats:

- (i) Security regressions when generating files.
- (ii) Visual stability (CLS) suffers without explicit guardrails.
- (iii) Results vary widely by file.

Within AI models, the larger instruct model (Llama) tended to produce simpler, lower complexity code, while the smaller thinking model (Phi) showed stronger security outcomes. This points to a trade off, simplicity and speed against thoroughness and robust coding.

A. Reliability

For Human vs. AI, statistically significant wins for AI on Halstead metrics indicate simpler structure in code, which likely explains why Lighthouse performance often improved. The elevated CLS in AI outputs shows a common problem with LLMs, that without visual feedback, generated UI often does not consider space reservation (for example for images and fonts), defers CSS late, or injects content after layout.

B. Maintainability

Most maintainability metrics show no significant difference, yet the specific cases show, for example Chat App and Thinkboard controllers, large degradations in technical debt for specific files. This pattern is typical when an AI model confidently fills gaps with boilerplate that passes linters but misses architecture level constraints like ensuring proper connections between files. For Phi vs. Llama, Llama's lower cyclomatic complexity, average complexity, and Halstead effort suggest it favors more direct solutions while Phi's thinking elaborates more considerations beyond simply creating a solution. Depending on the codebase, either behavior can help or hurt maintainability.

C. Good practice

Best Practices and Total Issues were not statistically different in either comparison, however the file with the highest issues being human written (`MyPostWidget.jsx`) shows that LLMs can be considered at least on par with humans when it comes to writing code with good practice.

D. Performance

AI's significant edge on the composite Lighthouse Performance metric likely stems from simpler solutions involving fewer dependencies and simpler code paths. INP and TBT differences were mostly insignificant, which is consistent with the fact that both AI generated and human written code rarely created real bottlenecks. Notable wins, for example Thinkboard INP and TBT reductions, show AI can fix specific bottlenecking issues in the baseline, but failures in other parts show that this result is not consistent.

E. Security

The strongest argument against AI generated code is security: more vulnerabilities were often introduced than fixed, and the same issue could increase dramatically (for example X Powered By leakage going from 23 to 174 instances). In contrast, Phi vs. Llama shows Phi performs significantly better on security, suggesting that explicit thinking helps the model consider more security measures. Overall these results suggest file generation tends to harm security.

F. Answering Research Questions

This study evaluated the impact of AI generated code within running MERN applications by replacing single, modular files and measuring quality with complementary static (SonarQube, ESLint) and runtime (Lighthouse, OWASP ZAP) signals.

Research Question 1 (AI vs. Human). We find some support for H_1, however across most metrics there was no statistically significant difference. AI generated code showed improvements in code simplicity due to having fewer Halstead bugs and lower Halstead effort, as well as improvements in Lighthouse Performance audits, while regressing on visual stability, based on the higher CLS metric, and security (more ZAP vulnerabilities). Overall, AI was competitive with human written code on reliability and maintainability statistics, better on structural simplicity and performance, but worse on UI stability and security. Unfortunately, we do not reject H_0 as the results overall are mostly indecisive and we do not find enough evidence in support of H_1.

Research Question 2 (Model vs. Model). We also find partial support for H_1. Llama 3.1 70B Instruct produced significantly lower cyclomatic and average complexity and lower Halstead effort, which suggests simpler code, whereas Phi 4 14B Reasoning Plus produced significantly fewer security

findings. Therefore we reject H_0 for security in favour of models with thinking capability.

Implications. In realistic full stack settings, drop in AI file generation can at least match human baselines on many quality indicators and even improve structural simplicity and page performance. However, two consistent drawbacks can be seen from our experiments: (i) security vulnerabilities become more common, and (ii) frontend visual stability goes down. Practically, AI generated changes should be paired with security scanning and review, and UI stability checks should be put into place. From our experiment, we found that model choice can be made to fit developer needs, preferring basic instruct models for simpler and easier to maintain code, while higher token thinking models should be used when security resilience is more important.

Limitations and scope. Results generalize to the studied setup (three MERN apps, five files per app, two models, single file replacements) and should be viewed as exploratory given modest sample sizes. Within this scope, our evidence indicates that AI generated code is viable in assisting with code creation in terms of performance and maintainability, as long as developers compensate for security and UI degradation risks with the appropriate measures.

VIII. Threats to Validity

Internal Validity: We mitigated nondeterminism in generation by fixing temperature to 0, keeping all API parameters constant in a CLI tool, and logging every prompt and response. Nonetheless, residual threats remained: (i) provider or model version drift across days could change outputs even with identical prompts, (ii) our manual trimming of the model responses introduced a small risk of human error, and (iii) integrating each AI file on its own branch may have interacted with project level analyzers (for example SonarQube's project context) in subtle ways. We reduced these risks by regenerating exactly one file per branch, preserving paths and imports, and rerunning all analyses from a clean state per branch.

Construct Validity: We measured quality via well known static (SonarQube, custom ESLint analyzer) and runtime (Lighthouse, OWASP ZAP) signals. Threats include: (i) Our custom ESLint based static analyzer encodes our own design and implementation choices, consequently, some ESLint derived metrics may be under or over represented relative to standard tools. Human factors (configuration drift, version mismatches, or implementation bugs) can further skew counts and reduce reproducibility. (ii) Tool configuration (rulesets, thresholds, versions) can shift counts and scores, (iii) risk metrics such as smells, debt time, complexity, and Halstead figures are theoretical measures for maintainability and reliability, rather than direct measures, and (iv) mixed directionality across metrics (lower vs. higher is better). We addressed this by saving and documenting our scripts and tools, and that lower is better for static counts, latencies, and ZAP findings, and higher is better for Lighthouse scores, and by interpreting results comparatively (paired Human vs. AI) rather than as absolute quality. NIST's rationalization clarifies units for Halstead metrics and separates real world measures from cognitive ones, but explicitly does not establish their empirical validity, our use is therefore comparative. [10]

External Validity: Our findings generalized only to the studied setting: three open source MERN applications, regeneration of five targeted files per app (at least two front end), and two models (Phi 4 14B Reasoning Plus and Llama 3.1 70B Instruct) invoked via OpenRouter. Results may differ for other stacks, larger or older codebases, different file types (for example build tooling and tests), alternative prompts, or other models and providers. The single file, drop in replacement task also underrepresented end to end development and multi file feature work.

Statistical Conclusion Validity: Sample sizes were small per project (n=5 matched files) and modest when pooled (n=15), limiting power. To mitigate this, we used Wilcoxon signed rank tests since it works better for non normalized and smaller datasets than the alternative paired t tests. Pipelines differed slightly by dataset (for example one sided vs. two sided alternatives and zero method choices), and we ran multiple tests across metrics and comparisons, increasing the chance of Type I errors. Therefore we treated results as exploratory and reported effect direction alongside p values.

IX. Contribution and Practical Implications

To our knowledge this study provided one of the first systematic evaluations of AI generated code inside running MERN applications using single file replacements and paired analyses. It showed that AI can match human baselines on many quality indicators and can improve structural simplicity and page performance, while tending to regress on security and visual stability. It also revealed consistent model specific patterns, with Llama 3.1 70B Instruct producing simpler lower complexity code and Phi 4 14B Reasoning Plus yielding fewer security findings. The findings below translate into actionable guidance for both practitioners and researchers:

- Use AI where simplicity and performance matter. Prefer Llama 3.1 70B Instruct when the goal is lower complexity, easier reviews, and better performance.
- Harden security by default. Treat AI generated changes as security risky, enforce header hardening and strict input and output validation.
- Protect visual stability. Reserve space for images and fonts, avoid late CSS, and use placeholders or skeletons to reduce cumulative layout shift whenever UI code is generated.
- Choose the model to fit the risk profile. Prefer reasoning models when security resilience is more important, and prefer instruct models when maintainability and reviewer effort dominate.
- Apply multi view measurement. The combined static and runtime signal set SonarQube, ESLint, Lighthouse, and OWASP ZAP offers a practical template for systemic quality assessment that can be extended to other stacks and file types.

- Explore the model trade off surface. The observed split between simplicity benefits and security risks, and the difference between Llama 3.1 70B Instruct and Phi 4 14B Reasoning Plus, motivates larger studies on when additional reasoning helps and when it hurts.
- Pursue open questions. Promising directions include automated CLS mitigation during generation, security aware prompting and fine tuning, and larger scale experiments that move beyond single file replacements to end to end feature work.

X. Acknowledgment

We want to thank our supervisor Farnaz Fotrousi for her guidance throughout this research.

Appendix

- GitHub Repository - Link
- Static Analysis Data- Link
- Runtime Analysis Data- Link
- Replication Package - Link
- Custom ESLint Analyzer- Link

References

[1] M. Khan, et al., *"Assessing the Promise and Pitfalls of ChatGPT for Automated Code Generation,"* arXiv, 2023.
[2] X. Tan, et al., *"How far are AI powered programming assistants from meeting developers' needs?,"* arXiv, 2024.
[3] A. Agarwal, et al., *"Copilot Evaluation Harness: Evaluating LLM Guided Software Programming,"* arXiv, 2024.
[4] E. Kalliamvakou, "Quantifying GitHub Copilot's impact," GitHub, 2022.
[5] M. Abdin, et al. *Phi-4 technical report.* arXiv preprint arXiv:2412.08905, 2024.
[6] S. Aggarwal et al., *"Comparative analysis of MEAN stack and MERN stack,"* International Journal of Recent Research Aspects, vol. 5, no. 1, 2018.
[7] C. Wohlin, et al., *Experimentation in Software Engineering*, Springer, 2012.
[8] J. Phoenix and M. Taylor, *Prompt Engineering for Generative AI: Future-Proof Inputs for Reliable AI Outputs at Scale*, O'Reilly Media, 2024.
[9] A. S. Molison, et al., *"Is LLM-Generated Code More Maintainable & Reliable than Human-Written Code?,"* arXiv, 2025.
[10] D. Flater, *Software Science Revisited: Rationalizing Halstead's System Using Dimensionless Units.* US Department of Commerce, National Institute of Standards and Technology, 2018.
[11] E. Paradis, et al., *"How much does AI impact development speed? An enterprise-based randomized controlled trial,"* arXiv, 2024.
[12] D. Nam, et al., *"Using an LLM to Help With Code Understanding,"* ICSE, 2024.