ISSN: 2473-4901
v12 n6673

Evaluating LLM-Generated Java Code Quality: A Vibe Coding Experiment with Halstead and McCabe Complexity Analysis and User Reflections

Jon Clark
jon.clark@colostate.edu

Seth Kinnett
seth.kinnett@colostate.edu

Colorado State University
Fort Collins, CO

Abstract

About a year ago, a novel workflow involving an LLM and a developer as a guide became known as "vibe coding." While the potential of this pairing of LLM as an intelligent agent and a human who is responsible for system requirements is attractive, the result in terms of code quality and completeness based on well-established metrics, and the experience of the developer across several LLMs, has not been carefully evaluated. The purpose of this study is to evaluate the effectiveness of vibe coding and its impact on the developer. Software metrics including Halstead Volume, Effort and Time, along with McCabe Cyclometric Complexity will be used as assessments of effectiveness within LLMs and across LLMs. Each developer will offer an assessment of the impact of the vibe workflow in terms of its effectiveness and efficiency via a personal reflection. Findings have revealed both similarities across LLMs as well as some significant differences. Recognizing that LLM abilities are rapidly changing it's likely that the impact on the developer will continue to evolve. The findings in this study have far reaching impact of both education, curricula, systems development, and even employment.

Keywords: Vibe coding, Software Metrics, Halstead, McCabe, Programming Education, Systems Development

ISSN: 2473-4901
v12 n6673

Evaluating LLM-Generated Java Code Quality: A Vibe Coding Experiment with Halstead and McCabe Complexity Analysis and User Reflections

Jon Clark and Seth Kinnett

1.INTRODUCTION

Vibe coding by name has a humble beginning in a post on X.com by Andrej Karpathy (2025, February 6) in which he said the following:

There's a new kind of coding I call "vibe coding," where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also, I just talk to Composer with SuperWhisper, so I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension; I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or web app, but it's not really coding — I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works.

This concept became known as "vibe coding" in a more formal way (Ge et al., 2025) along with an industry framing (Palmer, 2025). Even the *New York Times* asked the question whether AI can write my apps for me (Roose, 2025). Willison (2025) in *Simon Willison's Weblog* claimed that not all AI-assisted programming is vibe coding!

Though his academic and professional accomplishments are extensive, including a Ph.D. in Computer Science at Stanford University in 2015, the above post seems a bit whimsical on a rather serious topic that may have far reaching impact in the IT community affecting development workflows, effectiveness and efficiency, education and even employment. On the serious side, his post rather accurately captures a developer's interaction with an LLM. The purpose of this paper is to explore some of the dynamics of such interactions on both the developer's workflow, as well as the resulting impact on the code produced using accepted metrics by Halstead (1977) and McCabe (1976).

2.LITERATURE REVIEW

To be clear, the potential of AI to automate some of the basic structural considerations of a system design, as well as the coding of its methods, seems to be at hand, even if a bit naive at this point. Considerable foundational LLM work has been done (Chen et al., 2021), along with a comparison of human coding (Hou & Ji, 2025) and a comparison of generative AI versus textbook solutions in an introductory programming course (Bekkering et al., 2025). If this is the case, what are the likely impacts on IT employment? As early as 2024, IS educators were already noting that programmers and other professionals who create for a living were questioning their job security in light of generative AI's rapid adoption. Given, in addition, our dependency on reliable computing, can we be assured that incidents such as CrowdStrike (2024), with its five-billion-dollar impact on 8.5 million devices (Fung, 2024), do not become commonplace? Gill and Sikka (2011) have discussed complexity and its influence on risk. Denny et al. (2024a and 2024b) provide useful insights into prompts, as well as a more formal discussion. In addition, Raihan et al. (2025) provide a literature review of LLMs in computer science education. Sharp et al. (2026) extend this picture with a categorization of AI's role across IS education publications, finding that most current applications emphasize enhancing understanding and skill development rather than wholesale automation. In short, this is a rapidly changing space, one that entering students are already navigating (Frydenberg et al., 2026). See Clark and Kinnett (2026) for a detailed application of the following metrics applied to Java code.

Halstead

Halstead metrics are based on token counts of operators (verbs) and operands (nouns); information theory in terms of bits, the standard measure; and Stroud's (Stroud, 1967) information processing rate taken as a constant of 18 bps. The calculation of the various metrics depends on the counting strategy appropriate to a given language, and many programming languages have well defined strategies established including COBOL, Java, C and C++. Token counts are determined as follows:

n1 = number of distinct operators

n2 = number of distinct operands

n (vocabulary size) = total number of distinct tokens (operators + operands)

N1 = total number of occurrences of operators

N2 = total number of occurrences of operands

N (program length) = total number of tokens (operators and operands)

The derived metrics are as follows:

V (program volume) = $N*\log_2(n)$: program length times bits of information in vocabulary

D (program difficulty) = (n1/2)*(N2/n2) or 1/L: as volume increases, level decreases and difficulty increases

E (program effort) = D*V

T (time to implement) = E/18: effort in bits divided by human binary discriminations per second

I (intelligence content) = V/D

McCabe

McCabe's complexity measures were based on graphs of control flow, where nodes represent program statements and edges (arcs) represent the flow. Obviously, statements that determine decisions produce branches in the graphs and the count of various paths are an important determinant of complexity. These metrics are far more domain specific to procedural programming than Halstead's approach but are not predictive of effort across the stages of systems development.

The control graph produces the following metrics:

E = number edges of the graph

N = number of nodes of the graph

P = number of connected components (program exit points)

The derived metrics are as follows:

v(G) (Cyclomatic Complexity) = E-N+2: number of edges less number of nodes plus the number of connected components

ev(G) (Essential Complexity) = 1<=ev(G)<=v(G): based on reduced control flow graph

Interpretation of v(G) thresholds by McCabe:

- 1-10: simple procedure, little risk

- 11-20:more complex, moderate risk
- 21-50:complex, high risk
- >50: untestable code, very high risk

Essential Complexity, ev(G) is produced by removing all of the structured programming primitives. These include 1) sequence; 2) selection statements, including *if* and *case* statements; 3) iteration constructs, including *while, do,* and *for*.

3.RESEARCH DESIGN AND QUESTIONS

This section describes the research design, participants, and materials used to address research questions. This is a comparative study comparing a collection of known software code characteristics that were generated by Open AI ChatGPT, Anthropic Claude, and Google Gemini LLMs. In order to provide a context for vibe coding that would answer important research questions for this research, we developed a problem statement (Appendix I) that was provided to our novice student developers in CIS360, a course in Systems Development and Design. The developers were randomly partitioned into three distinct groups which were assigned to Open AI ChatGPT, Anthropic Claude, and Google Gemini for their use in developing a simple library application to be implemented in Java. The class drew approximately equally from Computer Science and Computer Information Systems majors, all with prior Java coursework; cohorts comprised 14, 15, and 16 students respectively.

The developers were free to produce any prompts required for their assigned LLM. Each developer was required to submit three files for the exercise: 1) Analysis of the interaction; 2) resulting Java classes; and 3) a statement of Reflections. The Analysis documented the prompts used and the responses of the LLM. The Class file contained all of the generated Java code produced that could then be analyzed relative to style and various complexity metrics. Finally, the Reflections of the developers were captured, and various metacognitive insights were measured across the LLMs including some with lasting impact on the developer's workflow with AI.

The following research questions will be addressed in this research organized into 3 cohorts (Open AI ChatGPT, Anthropic Claude, Google Gemini):

RQ1: What are the significant differences in McCabe cyclomatic complexity v(G), and Halstead volume V, effort E, difficulty D, and estimated bugs B across the 3 cohorts.

RQ2: To what extent do the three cohorts differ in the within-cohort variability of these metrics — that is, how consistent are the outputs that students produced when using the same LLM?

RQ3: What were the significant differences in the three cohorts relative to the respondent reflections? Each student was asked to use the problem statement provided in **Appendix I.**, and to submit three files: 1) LLM Interaction log; 2) Generated Java Classes; and 3) Student Reflections. The interaction log captures the initial prompt used and most if not all of the interaction that resulted in the Java classes being produced. The reflections contain the student's impressions, reactions and impact. The metrics involved from the second and third files are as follows: Java Classes-Halstead and McCabe complexity. Student Reflections were analyzed using Sentiment and Tone Analysis provided by Anthropic Claude. See also Gao et al. (2026) for the complementary nature of software metrics including that of Halstead and McCabe.

4. METHOD AND RESULTS

A series of Kruskal–Wallis tests examined whether Java code metrics differed across the three AI-assistant cohorts (ChatGPT, n = 14; Claude, n = 15; Gemini, n = 16). Non-parametric tests were selected in preference to one-way ANOVA because the cohort sample sizes were small and Halstead software metrics are typically right skewed. Where the omnibus test was significant, post hoc pairwise comparisons were conducted using Dunn's test with Bonferroni correction. Effect sizes are reported as epsilon-squared (ε^2) for omnibus tests and as r ($|Z| / \sqrt{N}$) for pairwise comparisons, using conventional thresholds (ε^2: .01 small, .08 medium, .26 large; r: .10 small, .30 medium, .50 large).

The following tests are based on the detail provided in: **Appendix II (**size and complexity of code); **Appendix IV** (comparison across ChatGPT, Claude, and Gemini); **Appendix V** (Halstead metrics detail); and **Appendix VI (**McCabe metrics detail).

Metric	ChatGPT Mdn	Claude Mdn	Gemini Mdn	H(2)	p	ε²
Lines of code	176.5	221.0	192.5	11.96	.003	.272
Distinct operators (n_1)	19.00	19.00	20.50	6.75	.034	.153
Distinct operands (n_2)	56.00	66.00	64.00	16.21	< .001	.369
Total operators (N_1)	590.0	721.0	654.0	10.72	.005	.244
Total operands (N_2)	300.0	372.0	334.5	11.73	.003	.267
Vocabulary (n)	74.00	84.00	85.00	16.71	< .001	.380
Length (N)	890.0	1,093.0	982.0	11.14	.004	.253
Volume (V)	5,526.5	6,986.8	6,276.9	11.53	.003	.262
Effort (E)	274,767	348,592	329,626	8.89	.012	.202
Time (min)	254.5	323.0	305.0	8.91	.012	.202
Est. bugs (B)	1.84	2.33	2.09	11.53	.003	.262
Methods defined	19.0	23.0	20.0	7.14	.028	.162
Total CC	44.0	51.0	43.0	11.67	.003	.265
Difficulty (D)	52.15	54.10	53.40	1.80	.407	.041
Avg CC/method	2.30	2.35	2.25	1.54	.463	.035
Max CC	11.00	11.00	10.00	4.88	.087	.111

Note. Kruskal–Wallis tests, df = 2, α = .05. ChatGPT n = 14, Claude n = 15, Gemini n = 16. Effect size: ε² (.01 small, .08 medium, .26 large).

Table 1 reports omnibus Kruskal–Wallis results and cohort medians for all 16 metrics. Thirteen showed significant differences. Post hoc Dunn's tests with Bonferroni correction revealed a consistent ordering: Claude produced the largest values, ChatGPT the smallest, and Gemini fell between the two. The Claude–ChatGPT comparison was significant on 12 of the 13 metrics, with uniformly large pairwise effects (r = .48–.70). ChatGPT and Gemini differed significantly on six metrics — primarily vocabulary, operand counts, volume, and estimated bugs — with medium-to-large effects (r = .44–.63). Claude and Gemini rarely differed; the one exception was total cyclomatic complexity, where Claude exceeded both other cohorts with equally large effects (r = .54). One metric broke the dominant ordering: on distinct operators (n_1), Gemini — not Claude — produced the highest median, and the only significant pairwise difference was ChatGPT–Gemini (adj. p = .030, r = .47).

Three metrics did not differ significantly across cohorts: Halstead difficulty (p = .407), average cyclomatic complexity per method (p = .463), and maximum cyclomatic complexity (p = .087). These are precisely the metrics that capture individual method intricacy rather than overall program scale. All three cohorts produced methods with median CC per method of 2.25–2.35, well within McCabe's low-risk threshold, and comparable maximum complexity (medians of 10.00–11.00). Together, these null results indicate that the three assistants produced methods of similar structural complexity; cohort differences reflect how much code was produced and how it was decomposed into methods, not how intricate each method was.

Within-cohort variability

Beyond differences in central tendency, an additional question is whether the three assistants produced more or less *consistent* outputs across students. Brown-Forsythe tests of homogeneity of variance were non-significant for all 16 dependent variables (all *p*s > .13), so the pattern described below is descriptive rather than formally established. Given the modest sample sizes, the dispersion observations should be interpreted as preliminary rather than confirmed.

Two findings stand out. First, on Halstead Effort and its derivative Time (min), Claude showed a noticeably elevated coefficient of variation (CV = .45) relative to both ChatGPT (.30) and Gemini (.36). This reflects a right-tailed Effort distribution in the Claude cohort: the median was 348,592 but the mean was 417,280, and one student's solution reached an Effort of 935,192 — roughly double the next-highest value in the cohort. While Claude's typical output required more effort than ChatGPT's, a small number

of Claude-assisted solutions were substantially more effortful than the rest, widening the cohort's Effort distribution.

Second, interquartile ranges revealed a pattern not visible in the CV statistic. ChatGPT's middle 50% of students clustered more tightly than either other cohort on most size and volume metrics — often by a factor of two or more. Lines of code IQRs were 24.75, 67.00, and 60.75 for ChatGPT, Claude, and Gemini, respectively. The ChatGPT cohort's central body of solutions was consistently the most uniform, with variability concentrated in the tails rather than distributed across the middle of the distribution.

Total cyclomatic complexity showed both higher central tendency and greater dispersion in the Claude cohort. Claude's IQR (15.50) was more than three times that of ChatGPT (4.50) or Gemini (5.75), indicating that Claude-assisted solutions were not only more complex on average but also more variable in how much more complex they were.

Taken together, the within-cohort evidence tentatively suggests that ChatGPT produced the most uniform core outputs while Claude produced larger and more variable ones; Gemini patterned with Claude on relative spread. These descriptive differences await confirmation with larger samples.

Student reflections

Student reflections (n = 45) were analyzed thematically across cohorts (full dimensional comparison in Appendix III). Two themes appeared with near-universal frequency regardless of assigned LLM: the importance of clear, specific prompting (cited in 86–93% of reflections) and the necessity of human oversight of generated code (81–87%). Beyond these shared themes, distinct cohort patterns emerged. Claude assisted with the identification of thematic coding.

The ChatGPT cohort produced the deepest technical critique and the most explicit concerns about hallucination, error loops, and over-reliance. Reflections in this cohort were most likely to invoke prior negative experiences with the tool and to frame human oversight in visceral terms. The Claude cohort showed the greatest architectural awareness, with reflections more often discussing modular decomposition, design strategy, and pedagogical value; the artifact panel interface was widely praised, and several students framed iterative dialogue with the assistant as a learning model. The Gemini cohort reported the highest first-try success rate (69%) – described in several reflections as "scary" or revelatory — and produced the most prominent role-shift commentary, with multiple students articulating a transition from "programmer" to "quality-assurance reviewer."

Four reflections (across cohorts) directly addressed workforce disruption, and concerns about passive acceptance of AI-generated code appeared in all three cohorts but were most pronounced and most explicitly named in the ChatGPT cohort. Cross-tool comparisons were inward-facing in the ChatGPT cohort (compared to past ChatGPT use) but outward-facing in the Claude and Gemini cohorts (comparing the assigned tool to other LLMs). Together, these patterns suggest that LLM choice shaped not only the code students produced but also how they understood their own role in producing it.

5.DISCUSSION AND LIMITATIONS

The central finding of this study is not that the three assistants produced different code — they clearly did — but that the differences were almost entirely in code size and decomposition rather than in structural complexity. Thirteen of sixteen metrics showed significant cohort differences, yet the three that did not — Halstead difficulty, average cyclomatic complexity per method, and maximum cyclomatic complexity — are precisely the metrics that capture how intricate each individual method is. All three assistants produced methods of comparable per-method complexity (median CC per method of 2.25–2.35, well within McCabe's low-risk threshold). Claude's significantly higher total cyclomatic complexity therefore reflects its tendency to generate more methods, not more complex ones. This distinction matters because "more code" is easily mistaken for "more complex code," and the two carry different implications for maintainability, testability, and pedagogical evaluation.

Student reflections provide a qualitative complement that aligns with and partially explains the quantitative patterns. Claude's higher method counts and larger codebases are consistent with that cohort's reflections, which showed the greatest architectural awareness — students discussed modular

decomposition and design strategy more than either comparison group, and several framed iterative dialogue with the assistant as a learning model. ChatGPT's parsimony on size metrics corresponds to a cohort whose reflections were the most technically critical: these students articulated the strongest concerns about hallucination, error loops, and over-reliance, suggesting that a more concise code generation style may place greater cognitive burden on the developer to verify correctness. Gemini's middle position on the metrics paired with the highest reported first-try success rate (69%) and the most prominent role-shift commentary, with students describing a transition from programmer to quality-assurance reviewer. Taken together, the quantitative and qualitative data suggest that LLM choice shapes not only the surface characteristics of the code but also how students understand their own role in producing it.

The within-cohort variability findings, though descriptive, speak to a practical question for instructors: how predictable is the code a given assistant will produce across a classroom of students? ChatGPT's middle 50% of solutions clustered more tightly than either other cohort on most size and volume metrics, often by a factor of two or more — suggesting that students using ChatGPT tended to converge on similar-looking solutions regardless of how they prompted the tool. Claude showed the opposite pattern. Its effort distribution had a noticeably longer right tail (CV = .45 vs. .30 for ChatGPT), meaning that while Claude's typical output required moderately more effort than ChatGPT's, a small number of Claude-assisted solutions were substantially more effortful than the rest — the kind of outliers that complicate grading norms and peer comparison. Claude's total cyclomatic complexity showed a similar spread, with an interquartile range more than three times ChatGPT's. For instructors designing assignments around LLM-assisted coding, this suggests a tradeoff: ChatGPT may produce more predictable outputs that are easier to norm against a rubric, while Claude's greater variability may reflect a wider range of student–model interactions — potentially richer for learning but harder to assess uniformly. Whether these dispersion differences stem from the models' response characteristics, from differences in how students prompted them, or from some interaction of the two remains open and warrants investigation with larger samples.

The size/complexity dissociation of generated code has been of recent interest due to the emergence of LLM-assisted coding and its impact on the reflection patterns of users. Frydenberg et al. (2025) carried out a study of student use of AI-generated Python programs using a framework involving correctness, efficiency, understandability, consistency, and maintainability. Sepidband et al. (2025) and Xu et al. (2026) have both attempted to address the complexity of LLM-based code generation. Finally, Xie et al. (2026) have suggested rethinking code complexity from an LLM perspective.

Since this interest has developed only recently and little research yet exists, the authors decided to use the more general size and complexity metrics proposed by Halstead and McCabe.

Limitations

Several limitations bound the inferential reach of these findings. First, cohort sample sizes (n = 14–16) were small enough to preclude one-way ANOVA and required reliance on non-parametric tests; while Kruskal-Wallis with Dunn's post hoc comparisons is robust, statistical power for detecting smaller effects was limited, and the within-cohort variability analysis is descriptive rather than formally established. Second, the study examined a single programming assignment of moderate scope (a two-class library management system); generalization to larger codebases, different problem domains, or more complex architectural requirements remains an open question. Third, students were drawn from a single course at a single institution, and all had prior Java coursework; results may differ for novice or expert populations. Fourth, the LLMs evaluated represent a snapshot of rapidly evolving systems, and findings tied to specific models may not persist as those models are updated. Finally, student reflections were analyzed using one of the LLMs under study (Claude), which introduces a methodological consideration worth naming, though the reflections analysis was thematic rather than evaluative.

6.CONCLUSIONS

This study compared Java code produced by students using three large language model assistants — ChatGPT, Claude, and Gemini — on a common library management programming assignment. Across 16 software metrics drawn from the Halstead and McCabe families, 13 showed significant cohort differences. Claude's solutions were consistently larger than ChatGPT's on size, volume, effort, and

estimated time metrics, with Gemini occupying a middle position. The three metrics that did *not* differ across cohorts — Halstead difficulty, average cyclomatic complexity per method, and maximum cyclomatic complexity — together indicate that the assistants produced methods of comparable individual complexity. Cohort differences therefore reflect how much code each assistant produced and how that code was decomposed into methods, rather than differences in the local structural complexity of the code itself.

Within-cohort variability analysis added a complementary descriptive finding: ChatGPT's middle 50% of solutions clustered tightly across most metrics, while Claude's effort and total cyclomatic complexity distributions showed longer tails and wider dispersion. Gemini patterned with Claude on relative spread.

Student reflections (Appendix III) revealed shared themes across cohorts — particularly the importance of clear prompting (86–93%) and the necessity of human oversight (81–87%) — alongside cohort-specific patterns. ChatGPT users articulated the strongest concerns about hallucination and over-reliance; Claude users reflected most on architectural awareness and pedagogical use; Gemini users, who reported the highest first-try success rate (69%), were most likely to frame the experience as a shift from programmer to quality-assurance reviewer.

Implications

For instructors, the findings suggest that the choice of AI assistant materially affects the surface characteristics of student code — its size, modular decomposition, and verbosity — without substantially altering its per-method complexity. Curriculum designers may find Claude useful for assignments emphasizing architectural decomposition, ChatGPT useful for parsimony-focused exercises, and Gemini useful for prompting and prompt-quality discussions. The shift from programmer to reviewer that several students articulated points to a broader curricular question worth taking seriously: what skills do students need when code is generated rather than written, and how should programs of study evolve to develop them?

Future work

Larger cohorts, multiple assignments of varying complexity, and longitudinal designs tracking how vibe coding workflows affect long-term skill development are the natural next steps. As the underlying models continue to evolve, replication will be essential.

REFERENCES

Bekkering, T. E., & Harrington, P. (2025). A Comparison of Generative AI Solutions and Textbook Solutions in an Introductory Programming Course. *Information Systems Education Journal*, 23(1), pp. 4–22. https://doi.org/10.62273/YQWP1758

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., … Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374.* https://doi.org/10.48550/arXiv.2107.03374

Clark, Jon D., & Kinnett, Seth J. (2026). Writing Prompts to Identify At-Risk Students in Introductory Programming Courses. *Information Systems Education Journal*, 24(3), 4-15. https://doi.org/10.62273/ISEDJ.24(3)

CrowdStrike, Nov. 2024, Retrieved from https://crowdstrike.com.

Denny, P., Leinonen, J., Prather, J., Luxton-Reilly, A., Amarouche, T., Becker, B. A., & Reeves, B. N. (2024a). Prompt problems: A new programming exercise for the generative AI era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education* (pp. 296–302). ACM. https://doi.org/10.1145/3626252.3630909

Denny, P., Prather, J., Becker, B. A., Finnie-Ansley, J., Hellas, A., Leinonen, J., Luxton-Reilly, A., Reeves, B. N., Santos, E. A., & Sarsa, S. (2024b). Computing education in the era of generative AI. *Communications of the ACM, 67*(2), 56–67. https://doi.org/10.1145/3624720

Frydenberg, M., Mentzer, K., & Patterson, A. (2026). The Rapid Rise of Generative AI Adoption among First-Year College Students. *Information Systems Education Journal*, 24(1), pp. 4–18. https://doi.org/10.62273/HVNN2048

Frydenberg, M., Xu, A., & Xu, J. (2025). Student perceptions of learning through original and AI-generated Python programs from a software quality perspective. *Information Systems Education Journal, 23*(4), 34–56. https://doi.org/10.62273/DVNQ9288

Fung, B, "We finally know what caused the global tech outage—and how much it cost," *CNN Business*, July 2024.

Gao, H., Hijazi, H., Medeiros, J., Durães, J., Lam, C., de Carvalho, P., & Madeira, H. (2026). Complementarity in software code complexity metrics. *Journal of Systems and Software, 232*, Article 112679. https://doi.org/10.1016/j.jss.2025.112679

Ge, Y., Mei, L., Duan, Z., Li, T., Zheng, Y., Wang, Y., Wang, L., Yao, J., Liu, T., Cai, Y., Bi, B., Guo, F., Guo, J., Liu, S., & Cheng, X. (2025). A survey of vibe coding with large language models. *arXiv preprint arXiv:2510.12399.* https://doi.org/10.48550/arXiv.2510.12399

Gill, N. S., & Sikka, S. (2011). Cyclomatic complexity measure for complex system architecture. *Journal of Computer Science, 7*(5), 700–702. https://doi.org/10.3844/jcssp.2011.700.702

Halstead, M. H. (1977). *Elements of software science.* Elsevier.

Hou, W., & Ji, Z. (2025). Comparing large language models and human programmers for generating programming code. *Advanced Science, 12*(8), Article e2412279. https://doi.org/10.1002/advs.202412279

https://simonwillison.net/2025/Mar/19/vibe-coding/

Karpathy, A. (2025, February 6). *Vibe coding* [Post]. X (formerly Twitter). https://x.com/karpathy/status/1886192184808149383

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Palmer, M. (2025, March 18). Vibe coding: A paradigm shift in app development. *Replit Blog.* https://blog.replit.com/what-is-vibe-coding

Raihan, N., Siddiq, M. L., Santos, J. C. S., & Zampieri, M. (2025). Large language models in computer science education: A systematic literature review. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education*. ACM. https://doi.org/10.1145/3641554.3701863

Roose, K. (2025, February 27). Can A.I. write my apps for me? *The New York Times.* https://www.nytimes.com/2025/02/27/technology/personaltech/vibecoding-ai-software-programming.html

Sepidband, M., Taherkhani, H., Wang, S., & Hemmati, H. (2025). Enhancing LLM-based code generation with complexity metrics: A feedback-driven approach. *arXiv preprint*. https://doi.org/10.48550/arXiv.2505.23953

Sharp, J. H., Anderson, J. E., & Lang, G. (2026). AI in IS Education: A Categorization of ISCAP Publications Using Zuboff's Automate–Informate–Transformate Framework. *Information Systems Education Journal*, 24(4), pp. 22–35. https://doi.org/10.62273/QGPE2646

Stroud, J.M. (1967, February), The fine structure of psychological time, in *Annuals of the New York Academy of Sciences*. https://doi.org/10.1111/j.1749-6632.1967.tb55012.x

Willison, S. (2025, March 19). Not all AI-assisted programming is vibe coding (but vibe coding rocks). *Simon Willison's Weblog.*

Xie, C., Shi, Y., Gu, X., & Shen, B. (2026). Rethinking code complexity through the lens of large language models. *arXiv preprint* (v2). https://doi.org/10.48550/arXiv.2602.07882

Xu, Z., Li, Y., Deng, G., Liu, Y., & Xing, Z. (2026). Rethinking complexity metrics for LLM-integrated applications: Beyond source code. *arXiv preprint*. https://doi.org/10.48550/arXiv.2607.01903

ISSN: 2473-4901
v12 n6673

Appendix I. Programming Assignment: Library Book Management System

Your Assigned AI Tool: [CLAUDE / GPT / Gemini]

Assignment Overview

You will use **[your assigned AI tool]** to create a Java console application that manages a library's book inventory. This assignment explores how you interact with AI coding assistants to solve programming problems.

Requirements

Part 1: Book Class

Create a Book.java class with the following properties:

- ISBN (String)
- Title (String)
- Author (String)
- Year (int)
- Availability status (boolean)

Part 2: Library Class

Create a Library.java class that manages a collection of books with these features:

Core Functionality:

- Add books to the library
- Remove books from the library
- Check out a book (changes availability to false)
- Return a book (changes availability to true)
- Display all available books
- Display all checked-out books

Technical Requirements:

- Menu-driven interface (user can select options from a menu)
- Error handling (handle invalid inputs appropriately)
- Must compile and run without errors

What to Submit

1. Java Code Files (2 files required)

- Book.java
- Library.java

Submit as a single ZIP file or via [submission platform].

2. Complete Interaction Log as Word .docx file

Save ALL of your prompts and the AI's responses. This is critical for the study.

How to save your interactions:

- **Claude:** Click the "..." menu → "Share" → "Export as text" or copy-paste entire conversation
- **ChatGPT:** Click conversation title → "Share" → "Copy link" or copy-paste entire conversation
- **Gemini:** Copy-paste the chat sidebar content after you finish the project

Submit as a Word document or PDF with timestamps if possible.

3. Written Reflection

Answer the following prompt in **250-500 words**:

"What did you learn by/about interacting with the assigned LLM?"

List and explain the **3 most significant items** you learned from interacting with the LLM. This could include:

- Java concepts or programming techniques
- How to communicate effectively with AI
- Strengths or limitations of the AI tool
- Strategies for debugging or refining AI-generated code
- Insights about your own learning process

Submit as a Word document.

Important Notes

- You will ultimately submit a .zip file containing
    - Book.java
    - Library.java
- Your_Reflection.docx.**Use ONLY your assigned AI tool** for this assignment
- This is a research study - your interactions help us understand how different AI tools work
- There are no "right" or "wrong" ways to interact with the AI
- Your grade is based on completion and effort, not perfection
- Save everything - your conversation log is just as important as your code

ISSN: 2473-4901
v12 n6673

Appendix II. Metrics Comparison

Metric	ChatGPT (n=14)	Claude (n=15)	Gemini (n=16)
Halstead — Program Size			
Avg Code Lines	176.6	237.9	196.3
Avg Vocabulary (n)	73.4	89.4	86.9
Avg Length (N)	866.8	1147.9	1007.4
Avg Volume (V)	5392	7470.5	6519.5
Min Volume	2428.6	4978	2168
Max Volume	8154.3	12593.4	8764.7
Volume Std Dev	1179.4	1928.4	1640.5
Halstead — Complexity & Effort			
Avg Difficulty (D)	50.2	53.8	52.3
Min Difficulty	33.3	33.1	27.9
Max Difficulty	58.3	74.3	69.1
Avg Effort (E)	275,930	417,280	353,059
Min Effort	80,897	164,725	60,482
Max Effort	475,669	935,192	532,711
Avg Est. Bugs (B)	1.8	2.5	2.2
Min Est. Bugs	0.8	1.7	0.7
Max Est. Bugs	2.7	4.2	2.9
McCabe Cyclomatic Complexity			
Avg Methods / Student	19.4	23.1	19.9
Avg Total CC	43.3	53.8	42.2
Avg CC / Method	2.2	2.3	2.1
Avg Max CC (per student)	10.4	10.9	9.6
Min Max CC	4	9	3
Max Max CC	13	15	12
CC Std Dev (max)	2.1	1.4	1.9
Students with Max CC ≥ 10	11	13	11
Students with Max CC ≥ 15	0	1	0

ISSN: 2473-4901
v12 n6673

Appendix III. Student Reflections

Dimension	ChatGPT Cohort	Claude Cohort	Gemini Cohort
Sample size	14 students	15 students	16 students
Prompt quality lesson	Universal — learned through iteration and failure	Universal — reinforced by structured multi-part prompts	Universal — discovered through contrast (simple prompt, plain output)
First-try accuracy reaction	Moderate; tempered by prior bad experiences	Moderate–High; attributed to prompt quality as much as model skill	Very High; dominant theme; described as "scary" and revelatory
Coding conventions teaching	Present; observational depth	Deepest; conceptual explanations reinforced understanding	Strong; conventions appreciated, emoji additions criticized
Human oversight emphasis	Strongest; most visceral (hallucination, error loops named)	Nuanced; scope-dependent framing	Pragmatic; matter-of-fact caution
QA / role-shift reflection	Present; workflow-level observation	Present; framed as intentional design strategy	Dominant; framed as professional identity question
Metacognitive depth	Moderate; self-discipline framing	High; iterative dialogue as learning model	High; critical reading framing
Interface / UX commentary	Minimal (baseline tool)	Strong; artifact panel widely praised	Moderate; show-thinking feature praised; emoji UX criticized
Emotional tone of reflections	Most varied; excited to skeptical	Analytical and structured	Impressed and occasionally unsettled
Cross-tool comparisons made	Inward — compared to past ChatGPT experiences	Outward — compared Claude to ChatGPT and Copilot	Outward — compared Gemini to ChatGPT
Vibe coding / over-reliance concern	Strongest and most explicit	Present; balanced with advocacy for intentional use	Present; concern about passive acceptance named

ISSN: 2473-4901
v12 n6673

Appendix IV. Halstead and McCabe Metrics for ChatGPT, Claude, and Gemini

ChatGPT Cohort (n=14) — Complete Halstead & McCabe Detail																
Student	Code Lines	n1	n2	N1	N2	Vocab	Length	Volume (V)	Difficulty (D)	Effort (E)	Time (min)	Est. Bugs	Methods	Total CC	Avg CC /Method	Max CC	Most Complex Method
avJava	166	19	56	612	307	75	919	5,724	52.1	298,123	276	1.908	21	46	2.19	11	run
deJava	178	20	52	549	274	72	823	5,078	52.7	267,564	248	1.693	17	40	2.35	10	runMenu
doJava	175	20	53	590	292	73	882	5,459	55.1	300,783	279	1.82	18	44	2.44	11	runMenu
fiJava	187	19	56	590	308	75	898	5,594	52.2	292,259	271	1.864	16	42	2.62	13	main
guJava	250	21	72	847	400	93	1247	8,154	58.3	475,669	440	2.718	26	62	2.38	13	start
joJava	103	17	37	277	145	54	422	2,429	33.3	80,897	75	0.81	19	29	1.53	4	checkoutBook
liJava	164	19	60	599	300	79	899	5,667	47.5	269,187	249	1.889	17	44	2.59	11	run
obJava	210	19	61	623	317	80	940	5,943	49.4	293,380	272	1.981	22	45	2.05	10	runLibrary
paJava	190	16	56	572	300	72	872	5,380	42.9	230,579	213	1.793	21	43	2.05	12	start
shJava	145	18	44	516	258	62	774	4,609	52.8	243,206	225	1.536	18	38	2.11	9	start
soJava	204	20	60	655	328	80	983	6,215	54.7	339,724	315	2.071	19	44	2.32	10	run
tuJava	184	19	60	600	309	79	909	5,730	48.9	280,347	260	1.91	21	45	2.14	11	menu
vaJava	145	18	44	496	248	62	744	4,430	50.7	224,718	208	1.477	17	39	2.29	9	menu
wiJava	172	20	52	550	273	72	823	5,078	52.5	266,587	247	1.693	19	45	2.37	11	run
AVERAGE	176.6	18.9	54.5	576.9	289.9	73.4	866.8	5,392.0	50.2	275,930.2	255.6	1.8	19.4	43.3	2.2	10.4	

Claude Cohort (n=15) — Complete Halstead & McCabe Detail																
Student	Code Lines	n1	n2	N1	N2	Vocab	Length	Volume (V)	Difficulty (D)	Effort (E)	Time (min)	Est. Bugs	Methods	Total CC	Avg CC /Method	Max CC	Most Complex Method
alJava	218	19	65	708	352	84	1060	6,776	51.4	348,592	323	2.259	23	51	2.22	11	run
chJava	200	20	58	627	317	78	944	5,933	54.7	324,292	300	1.978	19	44	2.32	10	run
crJava	285	18	72	903	441	90	1344	8,725	55.1	480,968	445	2.908	27	65	2.41	11	run
ecJava	248	20	74	816	400	94	1216	7,970	54.1	430,831	399	2.657	23	58	2.52	11	run
emJava	366	23	94	1226	607	117	1833	12,593	74.3	935,192	866	4.198	34	80	2.35	11	run
foJava	234	18	71	723	373	89	1096	7,097	47.3	335,577	311	2.366	28	51	1.82	9	run
frJava	281	21	81	942	475	102	1417	9,455	61.6	582,172	539	3.152	27	65	2.41	11	run
knJava	220	19	64	683	352	83	1035	6,598	52.2	344,754	319	2.199	23	47	2.04	10	run
liJava	201	19	63	612	308	82	920	5,849	46.4	271,651	252	1.95	19	44	2.32	11	run
maJava	284	22	84	972	477	106	1449	9,749	62.5	608,949	564	3.25	25	68	2.72	12	run
paJava	221	20	64	721	372	84	1093	6,987	58.1	406,108	376	2.329	18	53	2.94	10	main
roJava	195	19	59	626	317	78	943	5,927	51	302,535	280	1.976	18	45	2.5	11	run
seJava	254	21	70	797	396	91	1193	7,764	59.4	461,170	427	2.588	24	56	2.33	11	run
tuJava	164	16	66	510	273	82	783	4,978	33.1	164,725	153	1.659	19	33	1.74	9	main
woJava	197	19	62	590	302	81	892	5,655	46.3	261,687	242	1.885	20	47	2.35	15	main
AVERAGE	237.9	19.6	69.8	763.7	384.1	89.4	1,147.9	7,470.5	53.8	417,280.2	386.4	2.5	23.1	53.8	2.3	10.9	

Gemini Cohort (n=16) — Complete Halstead & McCabe Detail																
Student	Code Lines	n1	n2	N1	N2	Vocab	Length	Volume (V)	Difficulty (D)	Effort (E)	Time (min)	Est. Bugs	Methods	Total CC	Avg CC /Method	Max CC	Most Complex Method
amJava	81	16	39	239	136	55	375	2,168	27.9	60,482	56	0.723	11	15	1.36	3	addBook
baJava	228	21	77	810	417	98	1227	8,116	56.9	461,519	427	2.705	26	46	1.77	8	handleUserChoice
coJava	266	20	77	860	468	97	1328	8,765	60.8	532,711	493	2.922	23	41	1.78	10	main
crJava	191	21	60	613	312	81	925	5,864	54.6	320,194	296	1.955	18	41	2.28	10	runMenu
daJava	151	19	69	728	378	88	1106	7,144	52	371,805	344	2.381	17	30	1.76	10	main
deJava	208	19	63	658	325	82	983	6,250	49	306,274	284	2.083	19	46	2.42	11	run
heJava	110	20	59	456	239	79	695	4,381	40.5	177,473	164	1.46	14	36	2.57	9	main
loJava	248	22	84	808	427	106	1235	8,309	55.9	464,611	430	2.77	24	45	1.88	10	run
meJava	176	21	63	625	320	84	945	6,041	53.3	322,173	298	2.014	18	43	2.39	10	start
roJava	257	21	74	806	410	95	1216	7,989	58.2	464,762	430	2.663	23	55	2.39	12	run
saJava	180	20	59	609	314	79	923	5,818	53.2	309,657	287	1.939	17	43	2.53	11	main
taJava	194	21	65	650	331	86	981	6,304	53.5	337,079	312	2.101	21	44	2.1	9	main
toJava	189	19	77	591	338	96	929	6,117	41.7	255,105	236	2.039	22	38	1.73	10	processChoice
tuJava	178	20	57	589	293	77	882	5,527	51.4	284,123	263	1.842	18	43	2.39	9	main
wiJava	235	22	61	740	383	83	1123	7,159	69.1	494,452	458	2.386	23	51	2.22	10	runMenu
yaJava	249	23	82	830	415	105	1245	8,359	58.2	486,518	450	2.786	24	58	2.42	11	run
AVERAGE	196.3	20.3	66.6	663.3	344.1	86.9	1,007.4	6,519.5	52.3	353,058.6	326.8	2.2	19.9	42.2	2.1	9.6	

ISSN: 2473-4901
v12 n6673

Appendix V. Halstead Code Metrics

Java Code Metrics — Halstead Software Science | All 45 Students | Spring 2025														
Cohort	Student	Code Lines	n1 Distinct	n2 Distinct Operan	N1 Total Ops	N2 Total Operan	Vocabulary (n)	Length (N)	Volume (V)	Difficulty (D)	Effort (E)	Time (min)	Est. Bugs (B)	Cohort Rank (Vol)
ChatGPT	avJava	166	19	56	612	307	75	919	5724.3	52.1	298123	276	1.908	10
ChatGPT	deJava	178	20	52	549	274	72	823	5077.8	52.7	267564	248	1.693	4
ChatGPT	doJava	175	20	53	590	292	73	882	5459.4	55.1	300783	279	1.82	7
ChatGPT	fiJava	187	19	56	590	308	75	898	5593.5	52.2	292259	271	1.864	8
ChatGPT	guJava	250	21	72	847	400	93	1247	8154.3	58.3	475669	440	2.718	14
ChatGPT	joJava	103	17	37	277	145	54	422	2428.6	33.3	80897	75	0.81	1
ChatGPT	liJava	164	19	60	599	300	79	899	5667.1	47.5	269187	249	1.889	9
ChatGPT	obJava	210	19	61	623	317	80	940	5942.6	49.4	293380	272	1.981	12
ChatGPT	paJava	190	16	56	572	300	72	872	5380.2	42.9	230579	213	1.793	6
ChatGPT	shJava	145	18	44	516	258	62	774	4608.5	52.8	243206	225	1.536	3
ChatGPT	soJava	204	20	60	655	328	80	983	6214.5	54.7	339724	315	2.071	13
ChatGPT	tuJava	184	19	60	600	309	79	909	5730.1	48.9	280347	260	1.91	11
ChatGPT	vaJava	145	18	44	496	248	62	744	4429.9	50.7	224718	208	1.477	2
ChatGPT	wiJava	172	20	52	550	273	72	823	5077.8	52.5	266587	247	1.693	4
Claude	alJava	218	19	65	708	352	84	1060	6775.9	51.4	348592	323	2.259	7
Claude	chJava	200	20	58	627	317	78	944	5933.4	54.7	324292	300	1.978	5
Claude	crJava	285	18	72	903	441	90	1344	8725.1	55.1	480968	445	2.908	12
Claude	ecJava	248	20	74	816	400	94	1216	7970.4	54.1	430831	399	2.657	11
Claude	emJava	366	23	94	1226	607	117	1833	12593.4	74.3	935192	866	4.198	15
Claude	foJava	234	18	71	723	373	89	1096	7097.4	47.3	335577	311	2.366	9
Claude	frJava	281	21	81	942	475	102	1417	9454.8	61.6	582172	539	3.152	13
Claude	knJava	220	19	64	683	352	83	1035	6598.2	52.2	344754	319	2.199	6
Claude	liJava	201	19	63	612	308	82	920	5848.9	46.4	271651	252	1.95	3
Claude	maJava	284	22	84	972	477	106	1449	9748.8	62.5	608949	564	3.25	14
Claude	paJava	221	20	64	721	372	84	1093	6986.8	58.1	406108	376	2.329	8
Claude	roJava	195	19	59	626	317	78	943	5927.1	51	302535	280	1.976	4
Claude	seJava	254	21	70	797	396	91	1193	7763.8	59.4	461170	427	2.588	10
Claude	tuJava	164	16	66	510	273	82	783	4978	33.1	164725	153	1.659	1
Claude	wiJava	197	19	62	590	302	81	892	5655.1	46.3	261687	242	1.885	2
Gemini	amJava	81	16	39	239	136	55	375	2168	27.9	60482	56	0.723	1
Gemini	baJava	228	21	77	810	417	98	1227	8116.2	56.9	461519	427	2.705	13
Gemini	coJava	266	20	77	860	468	97	1328	8764.7	60.8	532711	493	2.922	16
Gemini	crJava	191	21	60	613	312	81	925	5864.4	54.6	320194	296	1.955	5
Gemini	daJava	151	19	69	728	378	88	1106	7144.1	52	371805	344	2.381	10
Gemini	deJava	208	19	63	658	325	82	983	6249.5	49	306274	284	2.083	8
Gemini	heJava	110	20	59	456	239	79	695	4381.1	40.5	177473	164	1.46	2
Gemini	loJava	248	22	84	808	427	106	1235	8309	55.9	464611	430	2.77	14
Gemini	meJava	176	21	63	625	320	84	945	6040.7	53.3	322173	298	2.014	6
Gemini	roJava	257	21	74	806	410	95	1216	7988.9	58.2	464762	430	2.663	12
Gemini	saJava	180	20	59	609	314	79	923	5818.4	53.2	309657	287	1.939	4
Gemini	taJava	194	21	65	650	331	86	981	6304.2	53.5	337079	312	2.101	9
Gemini	toJava	189	19	77	591	338	96	929	6117.4	41.7	255105	236	2.039	7
Gemini	tuJava	178	20	57	589	293	77	882	5527.3	51.4	284123	263	1.842	3
Gemini	wiJava	235	22	61	740	383	83	1123	7159.2	69.1	494452	458	2.386	11
Gemini	yaJava	249	23	82	830	415	105	1245	8359.2	58.2	486518	450	2.786	15
ChatGPT AVG		176.643	18.929	54.5	576.86	289.929	73.4286	866.79	5,392	50.2	275,930	255.571	1.8	
Claude AVG		237.867	19.6	69.8	763.73	384.133	89.4	1147.9	7,470	53.8	417,280	386.4	2.5	
Gemini AVG		196.313	20.313	66.625	663.25	344.125	86.9375	1007.4	6,520	52.3	353,059	326.75	2.2	

ISSN: 2473-4901
v12 n6673

Appendix VI. McCabe Code Metrics

Java Code Metrics — McCabe Cyclomatic Complexity | All 45 Students | Spring 2025								
Cohort	Student	Methods	Total CC	Avg CC / Method	Max CC	Most Complex Method	Risk Level	Cohort Rank (Max CC)
ChatGPT	avJava	21	46	2.19	11	run	High	4
ChatGPT	deJava	17	40	2.35	10	runMenu	High	9
ChatGPT	doJava	18	44	2.44	11	runMenu	High	4
ChatGPT	fiJava	16	42	2.62	13	main	High	1
ChatGPT	guJava	26	62	2.38	13	start	High	1
ChatGPT	joJava	19	29	1.53	4	checkoutBook	Low	14
ChatGPT	liJava	17	44	2.59	11	run	High	4
ChatGPT	obJava	22	45	2.05	10	runLibrary	High	9
ChatGPT	paJava	21	43	2.05	12	start	High	3
ChatGPT	shJava	18	38	2.11	9	start	Moderate	12
ChatGPT	soJava	19	44	2.32	10	run	High	9
ChatGPT	tuJava	21	45	2.14	11	menu	High	4
ChatGPT	vaJava	17	39	2.29	9	menu	Moderate	12
ChatGPT	wiJava	19	45	2.37	11	run	High	4
Claude	alJava	23	51	2.22	11	run	High	3
Claude	chJava	19	44	2.32	10	run	High	11
Claude	crJava	27	65	2.41	11	run	High	3
Claude	ecJava	23	58	2.52	11	run	High	3
Claude	emJava	34	80	2.35	11	run	High	3
Claude	foJava	28	51	1.82	9	run	Moderate	14
Claude	frJava	27	65	2.41	11	run	High	3
Claude	knJava	23	47	2.04	10	run	High	11
Claude	liJava	19	44	2.32	11	run	High	3
Claude	maJava	25	68	2.72	12	run	High	2
Claude	paJava	18	53	2.94	10	main	High	11
Claude	roJava	18	45	2.5	11	run	High	3
Claude	seJava	24	56	2.33	11	run	High	3
Claude	tuJava	19	33	1.74	9	main	Moderate	14
Claude	woJava	20	47	2.35	15	main	Very High	1
Gemini	amJava	11	15	1.36	3	addBook	Low	16
Gemini	baJava	26	46	1.77	8	handleUserChoice	Moderate	15
Gemini	coJava	23	41	1.78	10	main	High	5
Gemini	crJava	18	41	2.28	10	runMenu	High	5
Gemini	daJava	17	30	1.76	10	main	High	5
Gemini	deJava	19	46	2.42	11	run	High	2
Gemini	heJava	14	36	2.57	9	main	Moderate	12
Gemini	loJava	24	45	1.88	10	run	High	5
Gemini	meJava	18	43	2.39	10	start	High	5
Gemini	roJava	23	55	2.39	12	run	High	1
Gemini	saJava	17	43	2.53	11	main	High	2
Gemini	taJava	21	44	2.1	9	main	Moderate	12
Gemini	toJava	22	38	1.73	10	processChoice	High	5
Gemini	tuJava	18	43	2.39	9	main	Moderate	12
Gemini	wiJava	23	51	2.22	10	runMenu	High	5
Gemini	yaJava	24	58	2.42	11	run	High	2
ChatGPT AVG		19.4	43.3	2.2	10.4			
Claude AVG		23.1	53.8	2.3	10.9			
Gemini AVG		19.9	42.2	2.1	9.6			