The True Cost of AI Assistance to Programming of Software

Daniel Berry

Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1, Canada
dberry@uwaterloo.ca

1 The Cost of Correcting Incorrect Software

I have a set of slides titled "Writing code during requirements determination: A good head start or a costly bad bet?", that I use in my fourth year undergraduate requirements engineering (RE) course. In it, I tell the proverbial story of the start of many a computer-based system (CBS) development. In it, the boss of the development, hereinafter, just "the boss", gives to the programmers[1] the boss's order, "You people start the programming while I go find out what the customer wants." Understand that almost all of students in the course are in the university's cooperative education program in which they work one term of the three per year in a paid job in their planned professions, often at an industrial company, and receive academic credit for doing so. So, a majority of the students in the course have done some serious CBS development professionally several times.

I ask them "How many of you have participated in such a boss's order?". I clarify "as a programmer". Invariably a number of hands go up. Occasionally, one of the students pipes up with, "Yeah, we might as well get a head start on the programming." I then remark that it's a very common occurrence, arising from the mistaken assumption that the sooner programming starts, the sooner it ends. I ask each that raised er[2] hand, "Did it work as planned — giving your team a head start on the programming?". Usually, at least a few answer, "No!", to which I say that we will now learn why. If anyone answers "Yes", I ask a follow-up question "Did your team have to change *any* of the already-written code after the boss learned what the customer really wanted?". I have never had anyone say "No" to this follow up question.

To all that said that they had to change some code, I then say "What percentage of the code had to be changed after the boss returned with what the customer wants, the requirements specification?" I clarify that changed code includes all ripple effects.

Usually no one answers. So, then I announce that I would call out percentages and each student should raise er hand when I call out the percentage of code that E saw being changed. I call out slowly "10%, ... 20%, ... 30%, ... 40%, ... 50%, ... 60%, ... 70%, ... 80%, ... 90%, ... 100%, ..." and with a smile "110%". A few hands go up at "10%"; some more go up at "20%". About the same number of hands go up for each of "30%" and "40%". The number drops off at "50%", and rarely is there anything at a higher percentage. I *do* get some laughter at "110%". I say that we will be conservative and take only 10% as the fraction of the code that has to be changed after learning what the customer wants. I conclude the story by saying "Let's see how much the head start saved. Keep in mind that in this case, the boss returned with the full set of requirements only *after* the code had been written."

I remind the students of some earlier slides that they had seen. One, that shown in Figure 1, from Barry Boehm in 1981, shows that when a requirements error is detected during the writing of the code, correcting the error costs 1.5 to 3 times what programming the code correctly from the start costs [6]. By the 1990s, according to the graph in Figure 2 by Steve Schach, correcting the defect costs about 10 times what just writing the code correctly costs [30]. I remind them also that, as shown in the graphs in Figure 3, requirement defects are harder to correct than architectural defects, which are harder to correct than design defects, which are harder to correct than implementation defects [1].

[1] This article uses "programmer" to describe a person who produces code from requirements specifications. Others use "developer" for the same. However, this article calls the whole process of building a CBS from conception through deployment "developing the CBS". In this context, "developer" would be ambiguous. When this article paraphrases or quotes other work, it uses that other work's vocabulary.

[2] "E", "em", and "er" are gender non-specific third-person-singular pronouns in subjective, objective, and possessive forms, respectively.

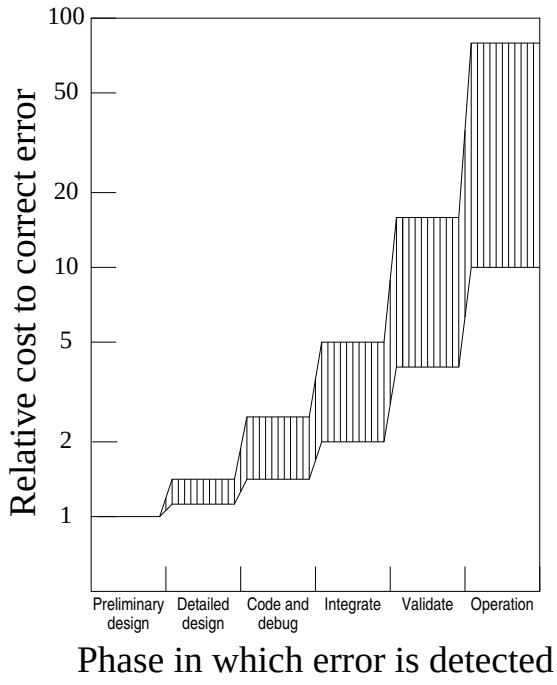

Fig. 1. Cost to Correct Errors Over Lifecycle

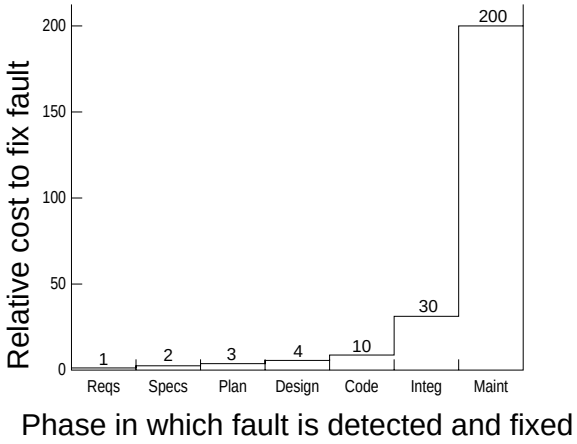

Fig. 2. Cost to Repair Defects Over Lifecycle

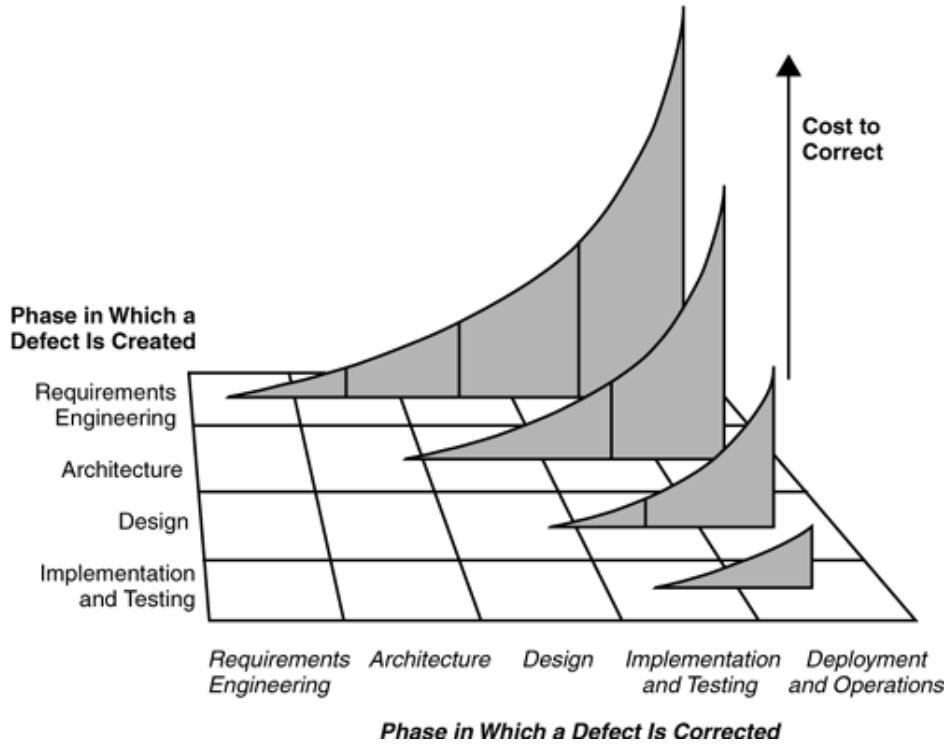

Fig. 3. Cost to Repair Defects vs. When Defect Introduced and Discovered

Source	**Phase Requirements Issue Found**			
	Require-ments	Design	Code	Test
[Boehn, 1981]	1	5	10	50
[Hoffman, 2001]	1	3	5	37
[Cigital, 2003]	1	3	7	51
[Rothman, 2000]		5	33	75
[Rothman, 2000] Case B			10	40
[Rothman, 2000] Case C			10	40
[Rothman, 2002]	1	20	45	250
[Pavlina, 2003]	1	10	100	1000
[McGibbon, 2003]		5		50
Mean	**1**	**7.3**	**25.6**	**177**
Median	**1**	**5**	**10**	**50.5**
[NASA, 2010 System Cost Factors]				
Method 1	1	8	16	21
Method 2	1	3–4	13–16	61–78
Method 3	1	4	7	157–186
[Langenfeld, 2016]	1	1.6	4.9	6.7
[Hamill, 2017] Mean	1		5.1	24
[Hamill, 2017] Median	1		3	27
[IBM SSI, Pressman]		1	6.5	15
[Extrapolated & Normalized IBM SSI, Pressman]	1	5	32.5	75

Table 1. Normalized Cost-to-Fix Estimates

More recent data are summarized in Table 1, which is Table 1 of a 2010 NASA report titled "Error Cost Escalation Through the Project Life Cycle" [2, 13], displaying data from studies reported from 1981 through 2003, extended by its own 2010 data from total system developments that include hardware, by data from later studies, and by data from what appears to be another source[3] [6, 7, 12, 14–16, 22, 23, 26, 28, 29]. Because the mean is skewed by the unusually large values in the Pavlina row, the median is a more reliable average to use than the mean. Thus, going forward, the scale factor for correcting requirement errors during programming will be 10.

I say to the students that I will again be conservative and say that correcting a requirements defect detected during programming costs ten times what having programmed the code correctly from the start costs.

Then I return to the aftermath of the boss's order. I claim that if as little as 10% of the code written in advance of knowing the full requirements has to be changed after the full requirements are known, the cost of writing the code has doubled. First, I say that since each change arises from having implemented wrong requirements or not having implemented right requirements, the cost multiplier to apply is indeed 10. Then, I explain that if C is the cost of writing the advance version, the cost of correcting the advance version when 10% of it has to be changed is $(10 + 0.1 \times C)$, and the total cost of writing the code is $C + (10 + 0.1 \times C) = 2 \times C$.

It gets worse if more than 10% has to be changed. I remind them that it *can* get *much* worse. Data show that 50–85% of all lifetime defects in deployed SW can be traced back to requirement errors [4–6, 9, 21]: missing, wrong, and extraneous requirements, the kinds of requirement defects that not knowing what the customer really wants causes.

I conclude that obeying the boss's order amounts to a very bad bet! It's practically guaranteed to end up at least doubling the cost of writing the code and developing the CBS. It's better to wait until the boss returns with a full specification of what the customer wants to start the programming with full knowledge of what the code is supposed to do. If you don't want the programmers to be idle while the boss is finding out what the customer wants, then have the programmers join the RE team with the boss. The RE team will have more brain power than before, and programmers can help the RE team know when the requirements specification is complete enough that the code can be written without the programmers' having to ask questions.

I even add that this cost analysis says something about how bug fixes and maintenance should be done. In short, "Start All Over!" It is cheaper in the long run to throw out the buggy code, redo and finish requirements analysis, and start programming from scratch. However, no CBS development project manager is willing to throw out er investment in written code, even if it is clearly buggy, *even though* the data are clear that E should!

I have been doing this lecture in the RE course for about 15 years now. Students are able to answer questions about it during the course's final exam. However, from what happens during the students' capstone projects, in which most students, in teams, develop a running CBS of their own choice, it is not at all certain that lessons of the lecture sink into their minds to affect their CBS development behavior. C'est la vie d'un éducateur!

2 Origin of the Hypothesis

When ChatGPT was released to the world in 2022 [25], almost immediately we began to see people using it and other LLMs to help them write code [17]. I saw reports that about one quarter of the ChatGPT-generated code is wrong and has to be corrected [18, 19]. I began to wonder how an analysis similar to that of the bass's order would go for code generated by an LLM, such as ChatGPT, in response to a human's prompting. The code will not be perfect. So, the prompter will have to correct it, but at what cost?

Because of the historical ten-fold cost factor to correct a defect, if indeed, one quarter of an LLM-generated code is wrong, then a human's correcting the code should cost 2.5 times what the human's writing the code emself from scratch. But factor will be *even* worse than 10 fold. The unspoken assumption of the old data, is that humans are

[3] I found lots of mentions of an undated study by the IBM System Science Institute (SSI). All gave exactly the same data. Those with citations cited Dawson *et al* [8], but that paper, while mentioning the IBM SSI, does not have a full citation to the original source. Since none of the authors is from IBM, it is unlikely that this paper is the original source. Because of the possibility that the IBM SSI report does not really exist, I tried to find evidence that the report does exist, if not the report itself. A clue is that the data from this elusive IBM SSI study are identical to those that Rothman [28] cites as from Pressman's 1992 Edition of *Software Engineering, A Practitioner's Approach* [27, p. 559]. Pressman attributes the IBM SSI data as from some course notes produced by IBM's SSI in 1981 [15]. The IBM SSI data are shown as the second last row of the table. The IBM SSI data and this second last row are missing the Requirements value. The last row of the table is this second last row with an estimated Requirements value equal to $\frac{1}{5}$ of the Design value, as in the median row, and finally normalized so that the Requirements value is 1.

correcting code written by themselves and others in their development teams. So, the correctors have some familiarity with the code that they are correcting[4]. The assumption was unspoken because no one was even thinking of the very possibility that something other than a person would write the code, and people corrected their own code, because anyone else doing it would have to spend time studying the code to understand it well enough to find the origin of any defect. Code generated by an LLM does not meet this assumption. Therefore, the human who prompted for the code will have to study the code thoroughly to even begin to consider how to correct its defects. Moreover, when the human starts to study the code for defects, E has *no* idea what the defects in the code are. They could be anywhere, including nowhere, on the outside chance that the code is correct to begin with. Only careful study of the code can say. Nevertheless, people I respected as honest scholars were reporting that they were successful at engineering a series of prompts that would persuade an LLM to generate correct code.

I finally concluded that LLMs could be successfully prompted to produce code for any set of requirements, but along the way, the generated code would have lots of defects. These would have to be found, with no idea what they even are. Only then can they be corrected, and the correction will be of unfamiliar code, written by someone or something else. This whole process of finding and correcting the LLM-generated code will be significantly more expensive than finding and correcting code written by humans in the same development project.

I therefore hypothesized;

High AI Copiloting Cost (HAICopC) Hypothesis:

The cost for a set of humans

to correct the code generated by an LLM
to implement a set of requirements

is significantly larger than

the cost for a set of humans

to correct the code programmed by the same set of humans
to implement the same set of requirements.

The HAICopC Hypothesis is my theory and is a prediction that would have to be subjected to empirical tests to confirm the prediction, just as Einstein's general relativity theory predicted that light would be observably bent while passing by a heavy star, and years later, experimental physics provided the data to confirm the prediction. I began to think about the difficulties to conduct empirical tests to soundly test this hypothesis. See the appendix for a brief discussion of some possible empirical tests of the hypothesis, none of which is totally satisfying.

In the meantime, I wrote a question for the RE course's Spring 2023 final exam to get the students to think about the situation and come to their own conclusions [3, See Question 6, "Cost Estimation, Bad Bets, and ChatGPT"]. Since my conclusion was only a theory, I accepted *any* answer from a student if the student supported it by evidence from the course materials or er own experience.

The first chance I got to publicly reveal the theory was shortly after the final exam at a panel titled "Requirements Engineering and Large Language Models" at the RE'23 conference in Hannover, Germany. After the panel members had largely been enthusiastic about the help that LLMs could give to RE and to programming, I spoke up, as my wont, as the first questioner from the audience. Markus Borg, the panel rapporteur, captured and summarized my words and the words of the people who replied to me very well:

> Dan [Berry] offered a critical observation and a thought-provoking hypothesis about the speeding up of engineering tasks. He argued that while LLMs might accelerate human work, they may also lead to more mistakes early on in projects. The catch is that correcting these mistakes later, in code no human ever wrote, could demand more than the 10-fold effort that correcting one's own code has been shown to require. He hypothesized that LLMs might do a good job, but organizations will spend more effort than they would if they did things the old-fashioned way from the start. Dan urged us to go beyond assessing the quality of LLM outputs. The community needs to compare task completion times with and without LLM assistance.
>
> Başak [Aydemir] replied by sharing her experience using an LLM to generate exam questions. She agreed with Dan's idea, noting that it took her longer to adapt a domain model drawn by ChatGPT for an RE course

[4] Even if a project uses indepedent quality assurers, who are not in the project, to *find* defects, the *correction* of the found defects is left to the project members.

than to do it herself. Despite this, she enjoyed the process and acknowledged that it was her first trial.

In contrast, Alessio [Ferrari] held a different stance. Although he acknowledged the need for scientific evaluation, he argued that working with LLM support is much faster and highly effective for quick prototyping.

Markus [Borg] responded that the benefits of LLMs are primarily in providing fast and succinct access to information. He envisioned using A/B testing in CodeScene to measure the time efficiency of LLMs in users' information-seeking compared to current visual analytics and dashboards available in the tool.

I had presented the same hypothesis as a panelist at another panel, in the Empirical RE (EmpiRE) workshop of the same RE conference. There, the discussion focused on the difficulties, and even the impossibility, of testing the hypothesis in a statistically significant manner with artifacts large enough to match industrial CBS development realities. We estimated that it would be years until sufficient numbers of similar large projects, some programmed by hand and some programmed with the help of an LLM, had produced enough mineable data with which to test the HAICopC Hypothesis. In the mean time, we would have to make do with reports by individuals who happened to pay attention to the time they spent programming both ways.

I *was* heartened that several in both panels had reported observing what I had hypothesized on a small scale. Having alerted the community of the HAICopC Hypothesis, all I could do now was to search for and wait for data that would confirm or refute the hypothesis. Starting in the Spring of 2025, I began to see blogs reporting timings and difficulties with AI copiloting.

3 Saranyan Vigraham's Post

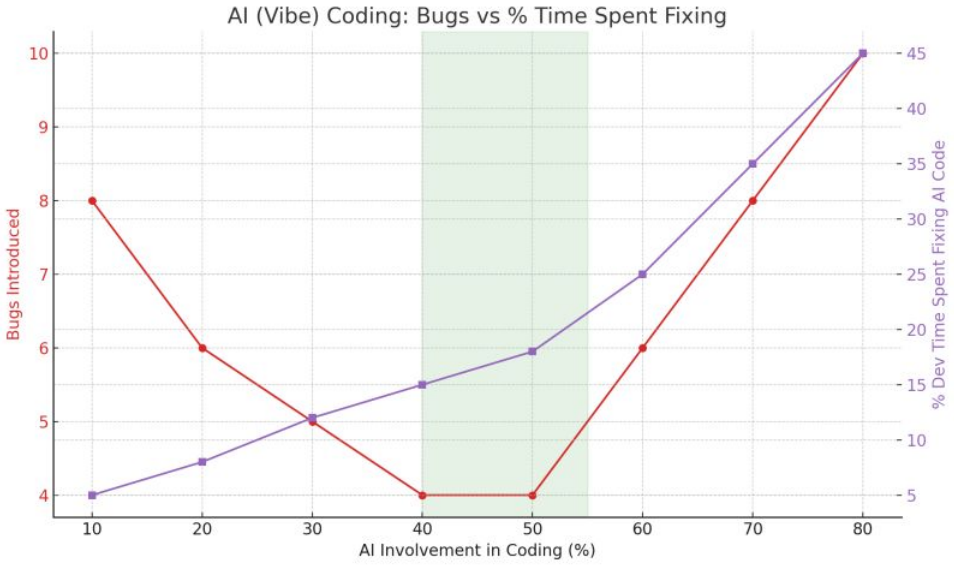

Fig. 4. Bugs and Repair Time Percentage vs. Percentage AI Involvement in Coding

Saranyan Vigraham, Director of Engineering at Meta, had kept data for a variety of Vibe Coding CBS development projects that differed in the level of AI involvement, from 10% to 80% [33]. Rather than following a predetermined task complexity progression, Vigraham conducted his study across 8[5] different projects ranging from feature development

[5] Vigraham's post says "10 different projects", but I count only 8 by three different methods: There are only 8 items in the list, "from 10% to 80%", of percentages; there are only 8 points in the y-axis in the graph of Figure 4; and there are only 8 tasks in the two bulleted lists of tasks he gives.

to migration tasks. His graph, which plots both (1) bugs introduced and (2) percentage of development time spent correcting these bugs against the percentages of AI involvement, is shown in Figure 4. Past 50% involvement of AI, both the bug count and the time spent correcting the bugs begin to grow, when before 40% involvement of AI, the time spent correcting the growing number of bugs was decreasing. So, he concluded that the "sweet spot … [was] 40–55% AI involvement. Enough to accelerate repetitive or structural work, but not so much that the codebase starts to hallucinate or drift".

Vigraham's data do not address the HAICopC Hypothesis. Because they arise from a controlled comparison of AI-assisted and manual developments of identical tasks, they provide important insights into how AI involvement affects development costs. Crucially, Vigraham found that task complexity did not increase linearly with AI involvement percentage. Instead, higher AI involvement itself created complexity through cascading effects; more bugs led to more rewrites, which led to increased debugging time, transforming originally straightforward tasks into complex ones.

Vigraham planned a sequence of eight development tasks, T1, T2, …, T7, and T8. In each task, T1, …, or T8, he would involve the AI in approximately 10%, …, or 80%, respectively, of the task. In order that

- he could meaningfully *measure* percentages of AI involvement as the percentage of the tasks given to the AI and
- trends in the bug counts and bug correcting times would reflect increasing percentages of AI involvement, as he progressed through the increasing percentages of AI involvement,

he applied the tasks in the planned order[6].

Vigraham reveals this ordering of tasks when he says:

> Where AI shines:
> - Boilerplate and framework code
> - Large-scale refactors
> - Migration scaffolds
> - Test case generation

> Where it stumbles:
> - Complex logic paths
> - Context-heavy features
> - Anything requiring real systems thinking [and new architectures etc.]
> - Anything stateful or edge-case-heavy

Notice that there are eight task categories, one for each percentage of AI involvement. Vigraham's sweet spot is right there between the last of the tasks for which AI shines and the first of the tasks for which it stumbles. With this ordering, the Vigraham's data can be seen to begin to address the HAICopC Hypothesis.

In the old days, BC (before copiloting), any programming shop would have debugged boilerplate and framework code, migration scaffolds, and test cases for its suite of CBSs in its artifact library (AL). No one would program these from scratch if E could find what E needed in the AL. In addition, refactoring was not really considered programming. It is moving chunks of the code around in algorithmic ways that preserve behavior. There are tools that help do the moving correctly once the tool user identifies the code chunks to be moved. To the extent that the refactoring tools need to do searching, LLMs are very helpful. Thus, these tasks of the development would not be introducing the defects that would figure in the data used to arrive at the 10-fold cost factor that make the boss's order a bad bet. These data would arise from the real programming, that involving complex logic paths, context-heavy features, anything requiring real systems thinking [and new architectures etc.], and anything stateful or edge-case-heavy.

Seen in this light, the part of Vigraham's data that arise from what was considered real programming, and that thus address the HAICopC Hypothesis, is that for 50% through 80% AI involvement, where both the bug count and the time to correct the bugs are growing.

[6] Of course, this would mean that he did not have give any task to the AI and correct its bugs repeatedly. He could give a task to the AI, then do the rest of the job, including correcting the AI's bugs, manually, log the data for the task's percentage, go back to the just after the most recent involvement of the AI (the last beginning of working manually), give the next task to the AI, and repeat in this manner until all the tasks had been given to the AI and he had eight sets of data. Note also that for most LLMs, the output for any input is not repeated when the same input is given again. So, the set of bugs to correct would likely change for each repetition of a task, making the data for the different percentage involvements not comparable.

Interestingly, the news is not all bad. As mentioned, three of the tasks for which the AI shines are, at any programming shop worth its salt, are searches of the shop's AL. Traditionally, one of the impediments that makes reuse of artifacts in a shop's AL difficult is the lack of good search tools [20]. Once a shop's AL gets big enough that it can help with most new developments, it is too big for manual search. Before LLMs, search engines were based on natural-language-processing (NLP) techniques. These typically do not achieve better than 85% recall, i.e., they find only 85% of what they need to find. These search engines are very picky about the relevance of a query. If the user gives the wrong set of words as the query, the desired item cannot be found, even though the query makes perfect sense to humans. LLMs have proven to achieve 95+% recall for the same search tasks. LLMs are far less picky about the relevance of a query. Just about any query that makes sense to a human allows an LLM to find just about *everything* related in any way to the query. So maybe we should focus on using LLMs to build super-duper reuse libraries and abandon the attempt to get LLMs to do the kind of thinking that only a human can do. We would be using LLMs for only tasks in which AIs shine and avoiding using them in tasks in which AIs stumble, hallucinate, and drift. Note how focusing on building only a super-duper artifact reuse library (ARL) solves the hallucination problem. Each programming shop could set up standards for adding artifacts to its ARL — only fully vetted, tested, and debugged artifacts with all their test cases, documentation, etc. are allowed to be committed to the ARL — standards that would ensure that the LLM does not begin to hallucinate when it is searching for the right artifact to reuse.

In fact, with improved reuse libraries whose search functions achieve close to 100% recall, and precision, of relevant artifacts, programmers would have to write *less* code and only entirely *new* code. They would never have to write mind-numbing, repetitive or structural code that they have written dozens of times before and could instead focus on exciting new, challenging stuff.

That LLMs are very good at tasks that can be achieved by searching with high recall and precision, but are very poor at tasks that require thinking is borne out by an empirical evaluation of several LLMs performing a variety of software development tasks conducted by Miserendino, Wang, *et al*. In particular, they found that AI agents excel at locating the code that is committing a describable bug, but they fail to identify the root causes of the bug, even as they may find all the pieces of the code that contribute to the bug [24].

Nevertheless, Vigraham's data do not completely address the HAICopC Hypothesis for the reason given as a threat to his conclusions by one of the comments to his blog, that by Aadharsh Kannan, who describes emself as an AI/ML-focused leader with deep economics expertise. E asked, "wouldn't it be more appropriate to track total dev time as opposed to time fixing code? Even in a context heavy mode, Maybe sum(Manual Dev Time + Manual Fix Time) > sum(AI Dev Time + Manual Fix Time) even if % Dev Time fixing code is higher on the latter." Vigraham replied that he did track total development time as well, but he could not find a way to uniformly use it as an objective metric. As Vigraham noted, "Total dev time goes up because of this" increased AI involvement, and he observed that "sometimes bounded experiments are better to have strong control over the code that gets generated, or else the AI makes the same mistake over and over", suggesting that unbounded AI assistance can create repetitive error patterns that compound debugging costs.

4 Victor Schwartz's Post

Victor Schwartz, with a computer science degree and years of programming experience from an early age, does not does not consider himself to be a production engineer that builds large-scale CBSs [31]. He reports that when he started to build his most recent product, AI tools were magical. With them, he was able to build working front-ends in minutes and prototype complex features in hours. He "Went from zero to demo faster than ever." For building prototypes he found AI tools be be "genuinely transformative". Once the demo yielded a go-ahead, he would normally turn the working prototype over to production engineers to build the production version. But this time, he thought that he could push ahead, prompting the AI to build the production version. Then he hit the brick wall of reality. Now that he needed to

- set up scalable API pipelines,
- design proper data schemas,
- handle system interdependencies, and
- manage increasing complexity,

the AI tools started hallucinating. He realized too late, after many months of refactoring, that these activities have a steep learning curve that is way beyond the AI tools' capabilities. His hard-earned conclusions are that AI programming tools are great, even incredible and game changing for the activities of

- front-end development,
- rapid prototyping, and
- getting basic API calls working.

They can even make it possible for a non-techie to build a prototype to *show* what E wants without having to explain what E means to a techie with whom E has difficulty communicating. However, AI programming tools are woefully inadequate for

- making production-level architecture decisions,
- designing complex CBSs,
- making CBSs secure[7], and managing interdependent components at scale.

His final recommendation is to use AI programming tools only to validate ideas and prototype quickly, and then, once the prototype is approved, to turn the production of the final product over to a team of professional, human production engineers who can really think and manage complexity.

While Schwartz does not provide data as does Vigraham, Vigraham and Schwartz agree on the nature of tasks for which AI is not helpful. Both say that AI does not help deal with

- high complexity (complex logic paths and anything stateful or edge-case-heavy vs. manage increasing complexity and set up scalable API pipelines),
- high interconnectivity (context-heavy features vs. handle system interdependencies),
- new structures of any kind (anything requiring real systems thinking [and new architectures etc.] vs. design proper data schemas).

They list different sets of tasks for which AI *is* helpful. Vigraham focuses on the initial steps of the development of a production system, while Schwartz focuses on prototyping, which precedes the initial steps of the development of a production system. Nevertheless, they both list the writing of the basic code that many CBSs in a programming shop's domain share (boilerplate and framework code vs. front-end development and getting basic API calls working).

5 Uplevel Data Labs Study

Grant Gross from CIO reports on a study carried out by Uplevel, a code analysis firm, that attempted, among other things, to determine if GitHub's AI Copilot helped developers to be more productive [10, 32]. The study measured a developer's productivity by capturing the frequency with which E merged new units of code into a repository and the development time for each unit merged into the repository. The higher is the frequency and the lower is the time, the higher is the productivity. The study compared the productivity data for about 800 developers over a three-month period after they started using Copilot with their own productivity data for the same three-month period a year earlier, before Copilot even existed.

The hypothesis driving the study was that the unit development time would decrease, the frequency of mergings would increase, and the defect rate would decrease. In fact, the study found *no* change in the developers' productivity while the use of Copilot led to 41% more defects. Apparently, in the meantime, developers were *saying* in surveys that Copilot "is really helpful for our productivity". Interestingly, some developers were making the discovery that they had "to be more of a [code] reviewer".

While Uplevel's hypothesis was not supported, these results are at least consistent with the HAICopC Hypothesis. From what the study reports, it appears that the developers with Copilot *were* doing things faster than without Copilot. However, Copilot was putting many more defects into the code than developers alone did, and these had to be corrected by the developers. The time to correct the defects was wiping out the time savings afforded by the use of Copilot. Some developers' mentioning that they were reviewing code more suggests that they noticed that they had to study the code that they had not written in order to correct the larger than usual number of defects that Copilot was putting into the code it had generated.

[7] Schwartz says to see all the vibe coder apps getting hacked.

6 Hale's Post About Wessling's VeraCode Report

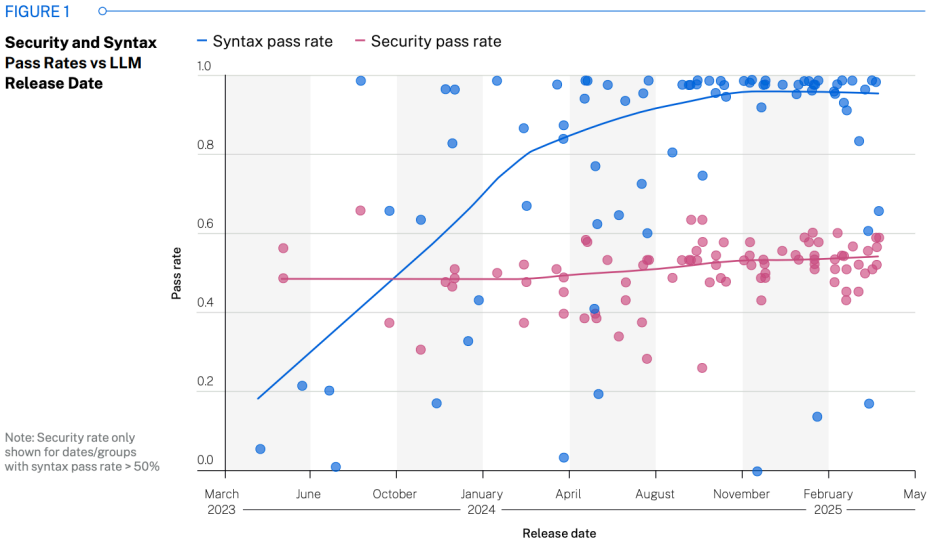

Fig. 5. Wessling's Figure 1 Showing Syntax and Seurity Pass Rates vs LLM Release Date

Craig Hale [11] reports a study by Jens Wessling [34] at VeraCode of 100+ LLMs assisting with 80 different coding tasks that shows that 45% of the LLM-generated code had serious security flaws. The failure rate varied according to the programming language used to write the code, with Java suffering the worst rate at 72%. This report shows that from March 2023 through May 2025, while the syntax error rate of LLM-generated code improved from just above 80% to just under 5%, the security flaw rate stayed almost the same, decreasing form about 50% to about 45%. See Figure 5 showing Wessling's Figure 1. During this same time period, developers are reporting that they are developing code faster than they did before.

While these data to not address HAICopC directly, their inverse is consistent with HAICopC. If the developers were spending the time to track down the security flaws in the LLM-generated code, they would undoubtedly be reporting a lot more time spent in developing the code. Indeed, one comment, from the Arkitekt, to Hale's post said "And the AI cheerleaders wonder why devs don't like using it… when you have to go through generated doe with a fine tooth comb, its [sic] faster to just doing it right yourself the first time. SMH" [11].

7 Conclusion

It was observed long ago and confirmed with data that correcting a code defect arising from missing, wrong, or extraneous requirements costs at least ten times what programming the code to the correct requirements. This observation explains why starting to writing code for a CBS in advance of knowing the full set of requirements for the CBS ends up not providing any real head start. Instead, correcting the code that will in all probability be incorrect due to missing, wrong, or extraneous requirements will cost at least as much as the so-called head-start programming. Thus, the cost of the programming with the so-called head start will cost at least double that of just waiting to start programming until the correct requirements are fully known.

This same observation predicts that the cost of developing a CBS with the assistance of an LLM generating code to requirements driven prompts will be even more than double the cost of developing the CBS manually. We know that the LLM generated code will be buggy, with some estimates saying that a quarter of the code will be wrong. We know that, unlike when one is correcting er own code, the programmer using the LLM will have to study the generated code thoroughly to even begin to correct the defects.

This all leads to the HAICopC Hypothesis:

The cost for a set of humans

to correct the code generated by an LLM
to implement a set of requirements

is significantly larger than

the cost for a set of humans

to correct the code programmed by the same set of humans
to implement the same set of requirements.

It is difficult to test this hypothesis in an internally and externally valid experiment. It will take years until we have enough data on large this anecdotal evidence developed with the assistance of LLMs. However, anecdotal evidence agreeing with the prediction is beginning to emerge. This article has reported some of this anecdotal evidence. I am interested in hearing about others. It will be necessary to plan to mine long-term project data about manual and LLM-assisted developments to provide an empirically sound verdict on the HAICopC Hypothesis.

Appendix

To test the HAICopC Hypothesis with an experiment that has both internal and external validity seems to be impossible. A test with internal validity would have two equal-sized groups, GM and GA, each consisting of dozens of randomly assigned programmers.

Each member, mM, of GM,

1. would program a CBS manually from a reqirements specifiation, S to produce a program, P, to implement S,
2. would be given a modification, M, of S, and
3. would manually modfy P to implement M.

In the meantime, each member, mA, of GA,

1. would receive a program, Q, generated by ChatGPT to implement the same S,
2. would be given the same modification, M, of S, and
3. would manually modfy Q to implement M.

Each participant of either group would keep track of er time programming, if any, and of er time correcting. These times would be used to test the HAICopC Hypothesis.

In order to be able to afford to have enough participants for statistical significance and strength, and to be able to control all the variables that need to be controlled, the CBS would have to be small enough that the whole experiment could be done in 2 to 3 hours[8].

Unfortunately, such a small CBS is not representive of industrial reality, and the test has no external validity. Making the CBS bigger and more representive makes the experiment require more time. This in turn, makes getting volunteers harder. Statistical significance and strength suffer from fewer participants, or participants have to be paid for their time, reducing affordability, or the number of people paid.

For maximum reality, the CBS should be from an actual development in some company. But, what company is willing to pay to develop a CBS multiple times in different ways to test a hypothesis for the benefit of the advancement of science? Even if a company were willing to have the CBS implemented twice, once for each treatment, there would be no statistical significance or strength.

Acknowledgements

I thank Reinhold Burger, Paul Eggert, Jason Hinek, Joe Petrik, Victoria Sakhnini, and Richard Schwartz for their pointers to relevant blogs, their comments on previous drafts of this article, or both. I thank Victor Schwartz and Saranyan Vigraham for confirming to me by e-mail that I undertood the gists of their posts, and I thank Saranyan especially for sending me additional data and taking personal time to correct my description of some points that I got wrong.

[8] If the hypothesis were supported, mA's modfifcation time would be longer than mM's modification time, perhaps by an amount equal to mM's programming time, thus possibly making mA's total time about the same as mM's total time.

References

1. Allen, J., Barnum, S., Ellison, R., McGraw, G., Mead, N.: Software Security Engineering: A Guide for Project Managers. Addison-Wesley Professional, Upper Saddle River, NJ, USA (2008)
2. Anonymous: Error cost escalation through the project life cycle. Tech. rep., NASA (2010), https://ntrs.nasa.gov/api/citations/20100036670/downloads/20100036670.pdf
3. Berry, D.M.: Se463 / cs445 spring/summer 2023 — final exam (2023, viewed 15 June 2025), https://student.cs.uwaterloo.ca/~se463/ExampleExams/exam-s23.pdf
4. Berry, D.M., Lucena, M., Sakhnini, V., Dhakla, A.: Scope determined (D) and scope determining (G) requirements: A new categorization of functional requirements. In: Ferrari, A., Penzenstadler, B. (eds.) Requirements Engineering: Foundation for Software Quality. pp. 75–84 (2023), https://doi.org/10.1007/978-3-031-29786-1_6
5. Berry, D.M., Lucena, M., Sakhnini, V., Dhakla, A.: Scope determined (D) versus scope determining (G) requirements: A new significant categorization of functional requirements. Tech. rep., Cheriton School of Computer Science, University of Waterloo (2023), https://cs.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/GvsDprelimTechReport.pdf
6. Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ, USA (1981)
7. Cigital: Case study: Finding defects earlier yields enormous savings. Cigital.com (2003), https://web.archive.org/web/20071003044003/http://www.cigital.com/solutions/roi-cs2.php
8. Dawson, M., Burrell, D., Rahim, E., Brewster, S.: Integrating software assurance into the software development life cycle (sdlc). Journal of Information Systems Technology and Planning **3**, 49–53 (2010)
9. Gilb, T.: Principles Of Software Engineering Management. Addison-Wesley, Wokingham, UK (1988)
10. Gross, G.: Devs gaining little (if anything) from AI coding assistants (2024), https://www.cio.com/article/3540579/devs-gaining-little-if-anything-from-ai-coding-assistants.html
11. Hale, C.: Nearly half of all code generated by AI found to contain security flaws – even big LLMs affected (2025, viewed 15 August 2025), https://www.techradar.com/pro/nearly-half-of-all-code-generated-by-ai-found-to-contain-security-flaws-even-big-llms-affected
12. Hamill, M., Goseva-Popstojanova, K.: Analyzing and predicting effort associated with finding and fixing software faults. Information and Software Technology **87**, 1–18 (2017), https://doi.org/10.1016/j.infsof.2017.01.002
13. Haskins, B., Dick, B., Lovell, R., Stecklein, J., Moroney, G., Dabney, J.: Error cost escalation through the project life cycle. In: MANAGING COMPLEXITY AND CHANGE! INCOSE 2004 — 14th Annual International Symposium Proceedings (2004), https://incose.onlinelibrary.wiley.com/doi/pdf/10.1002/j.2334-5837.2004.tb00608.x, a better version of [2]
14. Hoffman, C.: Mitigating software development risk. PowerPoint Presentation (2001)
15. IBM Corporation: Implementing software inspections. Course Notes (1981)
16. Langenfeld, V., Post, A., Podelski, A.: Requirements defects over a project lifetime: An empirical analysis of defect data from a 5-year automotive project at bosch. In: Daneva, M., Pastor, O. (eds.) Requirements Engineering: Foundation for Software Quality. pp. 145–160 (2016), https://doi.org/10.1007/978-3-319-30282-9_10
17. Lin, P.: How to use chatgpt for coding and programming (2023), https://www.computer.org/publications/tech-news/build-your-career/chatgpt-for-coding
18. Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.B.D., Lo, D.: Refining chatgpt-generated code: Characterizing and mitigating code quality issues. ACM Transactions on Software Engineering Methodolody **33**(5), 116 (2024), https://doi.org/10.1145/3643674
19. Liu, Z., Tang, Y., Luo, X., Zhou, Y., Zhang, L.F.: No need to lift a finger anymore? assessing the quality of code generation by chatgpt. IEEE Transactions on Software Engineering **50**(6), 1548–1584 (2024), https://doi.org/10.1109/TSE.2024.3392499
20. Maarek, Y., Berry, D., Kaiser, G.: An information retrieval approach for automatically constructing software libraries. IEEE Transactions on Software Engineering **17**(8), 800–813 (1991), https://doi.org/10.1109/32.83915
21. Marandi, A., Khan, D.: Analytical phase wise analysis of defect removal effectiveness to enhancing the software quality. International Proceedings of Economics Development and Research **75**(10), 40–46 (2014)
22. McGibbon, T.: Return on investment from software process improvement. DACS (2003), www.dacs.dtic.mil
23. McGibbon, T., Ferens, D., Vienneau, R.L.: A business case for software process improvement (2007 update), measuring return on investment from software engineering and management. A DACS State-of-the-Art Report (2007), https://csiac.dtic.mil/state-of-the-art-reports/a-business-case-for-software-process-improvement-2007-update-measuring-return-on-investment-from-software-engineering-and-management/, a later version of [22]
24. Miserendino, S., Wang, M., Patwardhan, T., Heidecke, J.: SWE-Lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? arXiv (2025), https://arxiv.org/abs/2502.12115
25. OpenAI: Introducing ChatGPT (2022), https://openai.com/index/chatgpt/
26. Pavlina, S.: Zero-defect software development. Dexterity Software (2003), https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/zero-defect-software-development-r1050/
27. Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, New York, NY, USA, 3rd edn. (1992)

28. Rothman, J.: What does it cost you to fix a defect? and why should you care? Catapulse.com (2000), https://www.jrothman.com/articles/2000/10/what-does-it-cost-you-to-fix-a-defect-and-why-should-you-care/
29. Rothman, J.: What does it cost you to fix a defect? StickyMinds.com (2002), http://www.stickyminds.com/stgeletter/archive/20020220nl.asp
30. Schach, S.R.: Software Engineering. Aksen Associates & Irwin, Boston, MA, USA, 2nd edn. (1992)
31. Schwartz, V.: Victor Schwartz's Post: I went all-in on AI coding tools to build my side project (2025, viewed 15 June 2025), https://www.linkedin.com/posts/victor-schwartz_i-went-all-in-on-ai-coding-tools-to-build-activity-7335676769817001986-DsEL
32. Uplevel Data Labs: Can GenAI actually improve developer productivity? (2024), https://resources.uplevelteam.com/gen-ai-for-coding
33. Vigraham, S.: Saranyan Vigraham's Post: LLM Vibe Coding AI activity (2025, viewed 15 June 2025), https://www.linkedin.com/posts/saranyan_llm-vibecoding-ai-activity-7329881611460575232-gH5w
34. Wessling, J.: We asked 100+ AI models to write code. Here's how many failed security tests. (2025, viewed 15 August 2025), https://www.veracode.com/blog/genai-code-security-report/