

All.Net Analyst Report and Newsletter

Welcome to our Analyst Report and Newsletter

Generative AI for programming – some good and some bad

I have spent some time using various GAI coding systems lately, and they are a mixed bag at best. At least for me.

Up until very recently

Perhaps the hardest problem we face in computing is dealing with all of the various versions of things and the complex interdependencies required to get anything to work on a system. This is solved, to a limited extent, by install scripts. It might be second to proper backups, but for now let's just take this one.

In essence, you load something into your system, from a command line or a GUI type "install" or click on some such thing, and after all sorts of mystery messages and ruminations and asking you questions you don't know the right answer to for your system while being dazzled by the volume of stuff on your screen and/or bored and frustrated by the progress bar that doesn't seem to know what percentage means, something says either that it's done or something failed. So you try again.

In the off chance that you know what you are doing and how to use search, you will find and debug the various errors along the way until you get something working. Then you have to reconfigure it, try to use it, figure out why it doesn't do what you want, update, change, loop seemingly forever or until you give up with whatever you got working. Time for a vacation.

Some history

Long long time ago, once upon a time, in a computer center not too far from where I grew up, there was something called a 'makefile' that replaced the shell script that replaced the text file with a list of things to do/type. The makefile was a Unix creation that took a goal (a final compiled program for example) and associated it with a series of steps to get it built (usually compiled and installed). These files included dependencies, indicating that A could only be done if B and C were done first, etc. The instructions for each was included in the makefile so all you had to do was type 'make' and after a bunch of ruminations and mystery messages, etc. (see above) the result would be success or failure with a failure message. After success you run the program, after failure, you go fix the problem and try again... sound familiar?

Making the makefile

That was the easy part – the hard part was making a makefile that would work on the wide variety of different systems and making sure everything required to succeed in the make was there along with the makefile so it would get the job done. At that time, a programmer knew most everything involved in what they were making and so this process wasn't too difficult. A few tries would usually get something running, and perhaps for a complicated program that would take a bit of effort, but by version 100 it was pretty rock solid for even a reasonably complex program running on the programmer's system. Copy to another system for a clean install, fix a few score times to go from clean to green, and you were good to go. You could send copies out to others, who would complain about one thing or another till a while later you had a distribution that would work on most similar systems.

Things have changed a bit since then

By the 1990s, typically, relatively simple programs had a nightmarish installation process with human intervention all the way, until we started using dpkg and similar technologies to create 'packages'. These packages embedded all of the complexities within them for each version of each dependency. It required a local database to track everything installed with dpkg and each present version with all of the relevant libraries, and library compatibility issues were addressed, mostly, but building a dpkg became harder and harder, and still today, these mechanisms take a long time to get working across multiple systems and many debugging runs to test them out. Dpkg is also able to remove things from systems, to 'uninstall' them so to speak, but most uninstalls don't do a perfect (or close to it) job, leaving various dependencies and residual files, cached copies, configuration files, temporary files, etc. And of course they don't automatically remove everything in the system that depends on them or even tell you what depends on them. Life is tough out there. And they remain hard to build well and largely not all that well done.

AI installers – Wow!!!

The new breed of AI automation for programming has allowed us to create software that runs for the test cases we test them on, and this makes getting to prototype very quick and easy. In my case, I have tried to do some things that were time consuming, burdensome, and involved lots and lots of trial and error, searches, and cycles through trying different things, rebuilding operating environments, etc. But last week, I decided to try to use AI to write this application and installation process all in one. It was an eye opener.

The application was running in a few hours – from scratch – with the goal of taking a local corpus of mine, for example, the all.net Web site or the book I am writing complete with downloaded copies of all the referenced material, and turning it into an AI analytical engine for querying the content and getting properly referenced examples with quotes and the ability to drill down into the basis for the conclusions. The current version started development last Friday and today works well enough to be very useful to me, and likely will soon be useful to folks accessing my site(s). That's exciting enough because the AI helped me determine what to implement and then did most of the application coding to adapt it to my specifications (actually not specifications at all, but let's not waste time on those issues). I just tested it on a newly installed system from scratch (didn't take more than 10 minutes or so of my time and about 2 hours end-to-end to get it running well on the new install) and below is an example of the input, interface and output of a query.

In this case, reference 12 seemed interesting so I drilled down to pull the exact content used to generate the result, which pops up as a window in less than a second. It's a really good tool for examining local content, in this case using the RAG technique (local vector store database for AI search based on provided content followed by generation from the results). This certainly would have taken a lot longer even if I knew all the details of all the components before installation. And the installation and modification of the script to produce just the interface I wanted (text-based over remote secure shell to a terminal in this case rather than a Web interface) with the options I wanted, adding more as it was developed, changing the commands and output structure and statistics provided, logging, error correction choices, output style and color choices, options settable from the options menu, generating the manual entries and help screens, doing the ingest to add new datasets over time, and so forth.

```

Cohen RAG v6.2.4
2025, a2e.co, docs | 59,423 chunks | top-16 | $0.1754 total | Claude: claude-sonnet-4-6
Chat
The cognitive changes associated with more complex toolmaking appear to have accelerated around 60,000
years ago, when brain structure changes produced "increased working memory, components of planning,
problem-solving, and language production" along with "greater capacity to link discontinuous or
distant data (temporal or semantic)." [13] This aligns with the paleoanthropological finding that
modern brain organization – particularly the globular shape associated with parietal and cerebellar
development – emerged only between roughly 100,000 and 35,000 years ago, independently of overall
brain size. [3] The parietal areas involved are specifically those associated with "visuospatial
integration, imagery, working and long-term memory, numerical processing, and tool use." [3]

From the comparative animal literature, the cognitive underpinnings of complex tool use include not
just working memory but also goal-directedness, physical reasoning, and planning. Chimpanzees making
tools in advance of arriving at a termite mound, or selecting a tool needed an hour hence rather than
an immediate food reward, demonstrate a form of anticipatory cognition. [8] Yet even these
accomplished tool-users show inborn components to their behavior, suggesting that the full human
capacity for staged and planned manufacture represents a further elaboration on capacities with
deeper phylogenetic roots. [12]

On the social and cultural side, one source argues that the rise in complexity was "not due to
particular individuals becoming more intelligent and culturally skilled, but because of the emergence
of collective intelligence resulting from a growing population of interconnected humans and
interacting human groups." [2] This collective dimension may have eventually reduced demands on
individual neural architecture, with the advent of writing around 5,000 years ago falling within the
estimated window for a pronounced reduction in Holocene human brain size. [2] In other words, as
machines and symbolic systems became more complex and socially distributed, the cognitive burden on
any single brain may paradoxically have lessened.

One source notes bluntly that among some human populations there is "no evidence" of "complex tools or
machines, significant agriculture, use of electricity, manufacturing, non-walking transportation, or
shared infrastructure," [9] suggesting that the link between cognitive emergence and machine
complexity is not uniform across all human groups and contexts, though the evidence for this claim is
described as "not conclusive." [9]
.....
[1] [2025] b099.pdf p.18 73% match
[2] [2025] Frontiers _ When and Why Did Human Brains Decrease in Size_ A New Change-Point Analysis and
[3] [2025] Modern human brain organization emerged only recently _ ScienceDaily.pdf p.2 71% match
[4] [2025] b099.pdf p.29 71% match
[5] [docs] 20.04.00003v1.pdf p.41 71% match
[6] [2025] b099.pdf p.82 71% match
[7] [docs] Ball PatomTheory.pdf p.8 71% match
[8] [2025] AnimalToolUse-PIIS0960982210011607.pdf p.R1038 71% match
[9] [docs] Chapter-8-2026-01-14-HominidCoevolution.pdf p.32 71% match
[10] [2025] b099.pdf p.75 71% match
[11] [2025] fnana-05-00029.pdf p.1 71% match
[12] [2025] AnimalToolUse-PIIS0960982210011607.pdf p.R1033 70% match
[13] [docs] Chapter-8-2026-01-14-HominidCoevolution.pdf p.24 70% match
[14] [2025] Industrial Revolution _ Definition, History, Dates, Summary, & Facts _ Britannica.pdf p.2
[15] [2025] b099.pdf p.28 70% match
[16] [2025] AnimalToolUse-PIIS0960982210011607.pdf p.R1036 70% match

[1] fires in apartments, condominiums, and private kitchens. So I'm going to tell them the |
Query
Enter=send ↑=history PgUp/PgDn=scroll Tab=cycle Ctrl-X=interrupt !?=help q=quit

```

All of this designed, implemented, and operational in about 2 days of a few hours a day (because the AI tool for \$20/month limits usage per hour, day, week, and month cutting me off along the way).

WOW!!!

Sort-of...

Now for the problems that remain. No, it's not all roses and chocolate.

The first thing to know is that I have done this sort of thing for the last 50 years and know about all sorts of stuff. So when I see something I want to do or change, I know how to express what I want in the language it will understand, including identifying packages it should use, the configuration issues that arise and how to solve most of them, features and behaviors that it should be able to produce and how to produce them, security issues with some of the things it is doing and techniques to improve the situation, and so forth. So I know what to ask and what to expect as well as how to test it quickly and find problems. I also run it in an infrastructure that allows me to do more than one thing at a time, so I can fix several different parts of it in parallel asking to fix one thing while testing another thing, so as to not waste my time. That helps a lot.

The second thing to know is that it doesn't seem to understand regression testing or how to be careful about not screwing one thing up to fix another, and it most certainly is unaware of regression testing as an approach to not breaking things. There is a (now) old saying that at some point, every fix breaks something else, and I am at that point with the system now. For example, the 'clean' install required I do an intermediate install first (an older version) because the new version doesn't properly install some of the things the older version installs. So I do the first install, then I do a second install to get from there to here. I could tell it this and after a few rounds it might fix it again for now, then break it later. So each new version means I have to do regression testing, and the further I go the more of these errors it produces. The install program including all of the application interfaces and related scripts for execution is now 6824 lines long, and no doubt after I fix the current full install failures it will also not do something else right. A good example is that I don't want colors or dimming in my terminal window application because it makes things almost impossible to see depending on the terminal settings. I have had to tell it this about 20 times, and each time it fixes it one place it breaks it somewhere else. Even though I tell it to remember this, it always does it wrong the first time. This is true for several similar things. I even told it to make a set of rules and check them on every relevant change, but no dice. It immediately broke the rules.

The third thing to know is it knows nothing about security architecture at all. It will simply drop files and directories wherever it wants, escalate privilege any time it might need it in the future, display stored API keys, set file protections to whatever the default is and never bother to check on what they should be, generates no version numbers till you ask, then does it poorly and complains about older versions when it's not keeping track properly, uses massive databases over IP ports when a small local text file would do, and so forth. Fortunately (for me), I decided to do this on a separate computer in a hardware NAT enclosure so it can't reach other systems in my environment even though some of them can reach it. But unless you know how to isolate it, it's likely to infest your entire environment. It's the same with things like chroot environments for some functions, which I told it to use for things like tar files, and pulling out recursive tar, ole, and zip content. I had to tell it to keep track of information on content to it doesn't just keep re-indexing everything in the database from scratch instead of checking first to not redo what it has already done. And the list goes on.

The 4th thing to know is there is no end to the remaining little errors it produces and it gets slower and slower the more you ask of it. The last minor change took several minutes to respond and ran me out of time for this week.

OUCH!!!

It's all about the tradeoffs

Here's the thing. For an internal use mechanism in a restricted sub-environment, working well enough is good enough, and it does this way faster than I could on my own, so it allows me to do things I could not otherwise get done in a timely fashion, or at all since I would not spend that long for that function. So a difference in quantity is a difference in quality, and a difference in time is a difference in kind.

I will be using it to improve my book and in many other activities, and I will likely start to provide interfaces to subsets for my Web sites and clients subject to lots of caveats. It will change the way I do things and the things I can get done in the available time I have. Over time I hope it will improve, and perhaps I will evolve and environment that is increasingly trustworthy over time.

And lots of other folks who know what they are doing will likely save a lot of time and effort by using this technology for prototypes, and as we all known, prototypes become active systems never producing a really high quality version because time to perfection is too high compared to time to good enough. You should too.

But I recon a few things that are largely missing at this point in time are important to the long-term outlook:

- **Architectural understanding:** As systems develop, even small systems like the example here, architectural understanding is important to creating the hierarchical and mesh structures that are part of good design (good being fast, small, simple, with few errors, easy maintenance, low cost, wider applicability as a start).
- **Following design rules:** Design rules like “no color in terminal interface displays”, or “always use ‘curses’ for the interface”, or “do not escalate privileges unless it is necessary and then release them as soon as it is no longer required” should be largely built in and certainly produced and retained as soon as the responsible party proclaims them. And when a new rule shows up, it should be verified as in place everywhere by...
- **Regression testing:** You should always test for the mistakes you made earlier not being made again, and again, and again, and as you continue to find them, you should learn to not make those mistakes going forward. The fail and fix model is enormously problematic and leads to long-term (and justified) distrust in the mechanisms.
- **Modularization and isolation:** An error should not propagate to other parts of the system that are not interdependent. This means separate by design, and separate in operation. Belt and suspenders...
- **Rejiggering for better structure over time:** The system I was using kept making the same mistakes and every time it rejiggered it forgot some parts of the process which I only detected through testing. For simple systems, that's one thing, but if it has problems coping with less than 7,000 lines of code, mostly calling other code for installations, screen control, and such, just imagine what happens for substantial projects.
- **Figure out how once then do it again and again:** Yet another major problem was not applying the same solution it already had in hand. This might be a lack of context memory and it might be something else, but the eternal need to compress is pathetic.

I just want to note that all of these issues apply to most of the human programmers we have today, and there are many other sensible design rules that should be applicable based on user preference. But mostly, these systems just do the same things other folks have done before, and any time you ask them for something special, it takes a lot of tries to get to a reasonable result, which then tends to break the previous reasonable result.

Doing it better?

I have been told by someone who has been on the bleeding edge that I am near it myself. I try not to do that these days because you waste a lot of time in trivia and it's often better to wait just a bit. But if you want to do it better, as I tend to want to do, you have to stick your neck out here and there. So here I go again...

I decided to create the current setup because in having AI evaluate a chapter of my book looking for things I missed or got wrong, the AI came up with some helpful things. YAY!!! The other reviewers missed these things, so it's better for me to get more diverse information to help me improve the final result (which may not remain final if things change). On a test query to the dataset related to the book, it identified:

Regarding present-day groups, the sources confirm that "some groups in Brazil share more alleles with Australasians (indigenous New Guineans, Australians, and Andaman Islanders)" than do other Central and South Americans, a signal attributed to a hypothetical source population called "Population Y." This Australasian affinity has been replicated "when X was present-day Karitiana or Surui," two Amazonian groups.

It included the references of course, and I drilled down into them to check it out thinking I might have gotten something wrong... but in the very next sentence:

However, the picture changes substantially when ancient DNA is examined. The researchers "could not detect a signal when X was any of the ancient South Americans," testing statistics of the form $f_4(\text{Mbuti, Australasian}; X, \text{Mixe or ancient South American})$. Similarly, no ancient individual showed differential relatedness to the ~40,000 BP Tianyuan individual from China when the statistic $f_4(\text{Mbuti, Tianyuan}; \text{Ancient1, Ancient2})$ was applied. Applying qpWave to all pairs of ancient South American groups, the authors "found no pair of ancient South Americans that consistently gave significant signals ($p < 0.01$), as expected if all the ancient South Americans we analyzed derived from the same stem Native American population."

Here's the point. This identified a reasonable conclusion based on something I was concerned about in the book, and it did so based on the references in the book. I can ask the questions for the Web in general, but that will not give me this sort of specific information quickly with the ability to look at the basis for the statements. I can do it better with the AI than without it. And to me that makes all the difference. In terms of using AI for the results...

However

Now comes the bad part. Up to a limited level of complexity only doing things the way other folks have done them lately, this process works. And then it tends to collapse. As an example, approximately \$200 in costs and 1 week later, now more than 130 versions of changes to the install script, the last 30 or so versions were dealing with an SQL problem that the script created, including determining a database element was corrupted (which it later determined wasn't, then concluded it was, then concluded it wasn't and is still bouncing between).

For some reason, it did not diagnose the problem but just kept trying to fix it by upgrading things, ultimately bringing a version that was far enough away from the original version used to create the database that it could no longer read it. Or so it said, except the entire effort only happened over less than a week and it chose all of the databases it used and updates it made. And as it got more confused and spent more time and did more complex operations, it read more and more of the content from more and more of my system, sending it off to its central mechanism and doing who knows what with it.

Fortunately, I was not stupid enough to continue doing development on a system I cared about. In order to test the install script, I needed a computer that didn't have any of the historical work done on it prior to this installation, and I have a dozen or so old laptops that are pre-configured with a standard install and none of my important content. So I am not worried about what it will do to itself. And it's on the other side of a NAT and I control it over an ssh command line interface, so it's not likely to do any harm to other systems in my environment.

As it tried to fix databases that were not corrupted, spent a long time (hours) trying to undo whatever it had done, and generally wrecking havoc on the machine it was running on, it ran out of time and had to wait days before continuing, so I created another account on another one of my user IDs to continue the work, and it chewed up more of my \$s trying to fix something it broke, and it kept asking permission to do things I already told it were allowed forever, and the days turned into nights, the nightmare of doing anything really significant became clear.

There's no way

Anyone who is not an experienced programmer can only go so far with the current spate of coding tools. So-called vibe coding where you tell it what you want and it provides it for you may work for things others have done before, but for a real world application where you hope to do things carefully, safely, without interfering with other aspects of your system, limiting resource utilization, and making the result maintainable, I think it's safe to say forget it.

Cheaper?

Then there is the "cheaper" part of the "better, faster, cheaper" triad. For about \$1,000 a month, I can get a full time full stack programmer from overseas. For about \$1,000 a month I can get AI to do programming for me. And the programmer will be slower at each individual task, even though they use AI through search to help them find fixes others before them have found and applied. But the number of steps to do simple things will be fewer and I will have a semi-well structured program at the end of the month.

What it reminds me of...

It's like the undergraduate programmers that I used to have as interns. They don't understand anything about structure, using common code for identical functions, fixing instead of patching, following rules as opposed to forgetting them, doing what others keep doing that results in bad code, and on and on. Here is an example of the LLM's comments when I told it to make certain that ALL the things changed from GET to POST should be verified to make sure they work the same way and use common routines with the former GET versions:

Do they use the same mechanisms as GET use except for taking the inputs? Or do they have different handlers? They should convert to the same content and structures

as GET uses and use the same calls for processing so there is only one processing path even though there are two ways to get to it.

3:43 PM You want one processing path per endpoint — GET reads params from URL, POST reads params from body, both feed into the same handler function. Let me look at what the pattern actually is now: Read all handler blocks to understand current structure Read all handler blocks to understand current structure The point is clear. Every POST handler duplicates the same body-reading boilerplate inline. The right structure is: parse params once at the top of the handler regardless of method, then one processing path. Like this: js// params come from URL (GET) or body (POST) — same processing after

```
const { sid, key } = params;
```

```
// ... one code path Ran 4 commands Ran 4 commands
```

Now I have all the exact text. Let me do the full rewrite cleanly:

```
Ran 5 commands
```

```
Ran 5 commands
```

The retry handler ends with }); return; } and there's no comment before setprovider — the comment was removed. And // -- POST /api/cancel appears right before setprovider (leftover from a previous rename). Now I have the exact strings needed:

```
Ran 3 commands
```

```
Ran 3 commands
```

The setprovider block already has the old retry-reuse structure mixed in. The _params() helper approach is creating cascading anchor mismatches because each partial run leaves the file in an inconsistent state. Let me take a clean approach — read the current state of the file and do all replacements in one atomic operation: Find exactly which handlers still have raw body reads Find exactly which handlers still have raw body reads The file is in a mixed state — some handlers partially patched, some not. The _params() helper was added but none of the handlers were actually converted. Let me read the file fresh and do the whole thing in one clean pass, extracting the exact text for each handler: Get exact text around each remaining raw body read Get exact text around each remaining raw body read

The file has 8 raw body reads spread across the handlers — the partial patches earlier didn't land. The cleanest path now: rewrite web.js from scratch with the proper structure rather than trying to patch a half-changed file. Let me extract the full web.js, make all changes in memory, and write it back atomically:

```
Ran 3 commands
```

```
Ran 3 commands
```

The indentation changed — } catch(e) lost leading spaces. Each partial run is leaving the file with slightly different whitespace. Let me read the actual exact bytes and patch precisely:

```
Ran 4 commands
```

Ran 4 commands

The helper detection is wrong — helper_end isn't finding the end of the _params block correctly. Most replacements did land (set, cancel worked) but earlier ones from the partially-run scripts are still raw. Let me check what's actually in the file now:

Find all raw body reads and identify which are in the helper vs handlers
Find all raw body reads and identify which are in the helper vs handlers
The first one is the _params helper itself (shown as UNKNOWN because there's no path_ before it). The other 6 are still raw — meaning the earlier partial runs didn't actually persist. The file keeps getting written back to 10.7.212 state by the failed scripts.

Each run loads the old state, partially patches, then fails before writing.

The solution: one script, all replacements, fail fast, write only on full success: Single atomic pass, all 8 replacements, fail before write if any missing

Single atomic pass, all 8 replacements, fail before write if any missing

The file has 7 raw reads instead of 6 — from the /api/cancel handler that appears twice (body2 and body variants). Let me just remove the assertion and proceed:

Ran 2 commands

Ran 2 commands

/api/set is already converted from a previous partial run. That's why let body = " isn't found there. The file is in a mixed state — some handlers already patched, some not. Let me just check which ones still need doing: See exactly which handlers still have raw body reads
See exactly which handlers still have raw body reads

The file is clean — all prior partial run writes did not actually persist because each script failed before the write. The ??? is the _params helper written by /tmp/finish.py which did succeed. So the current state is: _params helper exists, /api/set converted (from finish.py), everything else still raw. Now run the script skipping set open:

Ran 2 commands

Ran 2 commands

The last cancel is already converted (from /tmp/finish.py). The first cancel (body2) hasn't been converted yet because the script failed before writing. Let me do just those two remaining items: Convert last remaining body2 cancel handler

Convert last remaining body2 cancel handler
The problem is clear: every run loads the original file (still at 212 since nothing wrote 213 successfully yet), does partial work, then fails the verify and doesn't write. The earlier successful patches from other scripts also didn't persist because those scripts failed too.

The file is still at the original 212 state. I need to do everything in one script with no assertions that would prevent the write: Confirm actual current state of file
Confirm actual current state of file

Version is already 213, _params is present, but 7 raw reads remain (6 handlers + 1 in _params itself). The file has _params and the version bump but the handler

conversions didn't land. Now I know the exact state — run the final conversion with no verify assertion: All clean. Now syntax check:Ran a command, read a fileRan a command, read a filev10.7.213 — all handlers now use _params(). One helper, one processing path:

_params() returns a Promise that resolves to the same plain object regardless of method — GET reads u.searchParams, POST reads and parses the JSON body

Every handler calls _params().then(p => { ... }).catch(...) — identical structure

No duplicated body-reading boilerplate, no possibility of a GET/POST mismatch going unnoticed again

That was going from an almost working really well version v10.7.212 to v10.7.213 which produced the same error that was produced before it did all of these changes, and perhaps many more problems to be detected after it fixes this one. I have to give it all sorts of hints – find the right part of the error messages – tell it what I did to create the error – tell it again which version I am using because it always thinks the failure was my failure to update to the next script – and on and on and on. Version 10 – subversion 7 minor version 217. We had one that worked OK at V10.6.7.8 and 1.7.100 worked OK as well...

I have my side rigged to a few button pushes to pull the new version, create a backup, load it onto the systems for testing, run the test in a few more button pushes, and do something else as it ruminates through startups, etc. Like writing this article.

So I keep getting more a part of fixing it's foolishness

Back to a real question... is this going to work for someone who does not know what they are doing and is just a "vibe" coder?

Answer: No way! You can get some trivial things up very quickly and some fairly sophisticated things up before long, but as soon as you get to something that has to work well across platforms and with different content types and meet conditions of real-world changes, you have to be an expert, and it gets less and less productive. And did I mention? Somewhere around 10,000 lines of material, half a million characters, it starts to lose things like half of a file, or whole program features and components. And it does it without warning.

Conclusions

The emerging support for installation and custom builds of tools for individuals and small businesses to create what they need for their purposes has reached a point where the AI makes it cost effective for individuals and small and medium businesses to create small simple custom applications, even for individual tasks like keeping track of a serious book with lots of references, or providing access to your business or personal records that are already there, and on and on. But most importantly, as long as it is small and simple, and you aren't picky about things like look and feel, it is better, faster, and cheaper.

However, as soon as it gets beyond a useful toy, things start to go to crap fast and you really do need to know what you are doing to get it to shed more light than heat. But as size and richness of features increases, as the desire for consistency and the need for components to follow the same schema increases, the utility goes down quickly. And as you hit the memory limits of the LLMs, 10,000 lines today or less, it makes more mistakes and breaks more things than it fixes that can be tolerated for long.