The Evolution of AI-assisted coding features and developer interaction patterns

21 Dec, 2024

Yes, I agree that's a fancy title. But consider this - what started as simple autocomplete suggestions has evolved into something far more powerful – we now have AI that can generate entire functions on the fly, scaffold complete files with proper architecture, and even bootstrap entire codebases from scratch. The tools have evolved from being helpful typing assistants to becoming collaborative coding partners.

The progress has been pretty insane. Whether you are a new developer or a seasoned professional, I think understanding and adapting to these features/tools is the need of the hour. As these tools advance, it's becoming a bit complex to figure out which ones to use to maintain tight control. when to let AI take the wheel.

How much control should we give to these AI assist coding features? I present an analogy here. You can think of AI-assisted coding features to be the gears of a car. In first gear, you have maximum control over the engine but move slowly - that's the autocomplete. Shift up through the gears (conversational assistant, cursor chat, agentic mode), and you trade granular control for more speed and automation.

In this post, we explore through the evolution of AI-assisted coding via three perspectives -

1. Historical Narrative - What breakthroughs happened that led us to here. I think this is essential for us to gain a deeper appreciation and understanding of these tools. Some of you are too spoilt and take these tools for granted.
2. Interaction Patterns - How we use these tools, what UX they unlock, and some observations of emerging patterns in their usage.
3. The Gears Analogy - Control vs Automation - How much control and trust are you willing to trade for speed.

Fair warning: The blog will jump a bit between introducing new AI features and the historical narrative.

Autocomplete

The code suggestions era

I started my programming journey in late 2018. My first programming language was Python. I learnt it from Coursera's Python for Everybody Specialization and some sentdex and Corey Schafer videos. Back then, Kite was making waves with its "machine learning" based code suggestions. However, I rawdogged my code in Pycharm and Spyder IDE.

Around the same time, Microsoft released Intellicode in (2018-present) in VsCode and Visual Studio IDEs which could provide contextually-aware suggestions about the next methods, variables, essentially prioritizing the most likely suggestions in the "Intellisense" list.

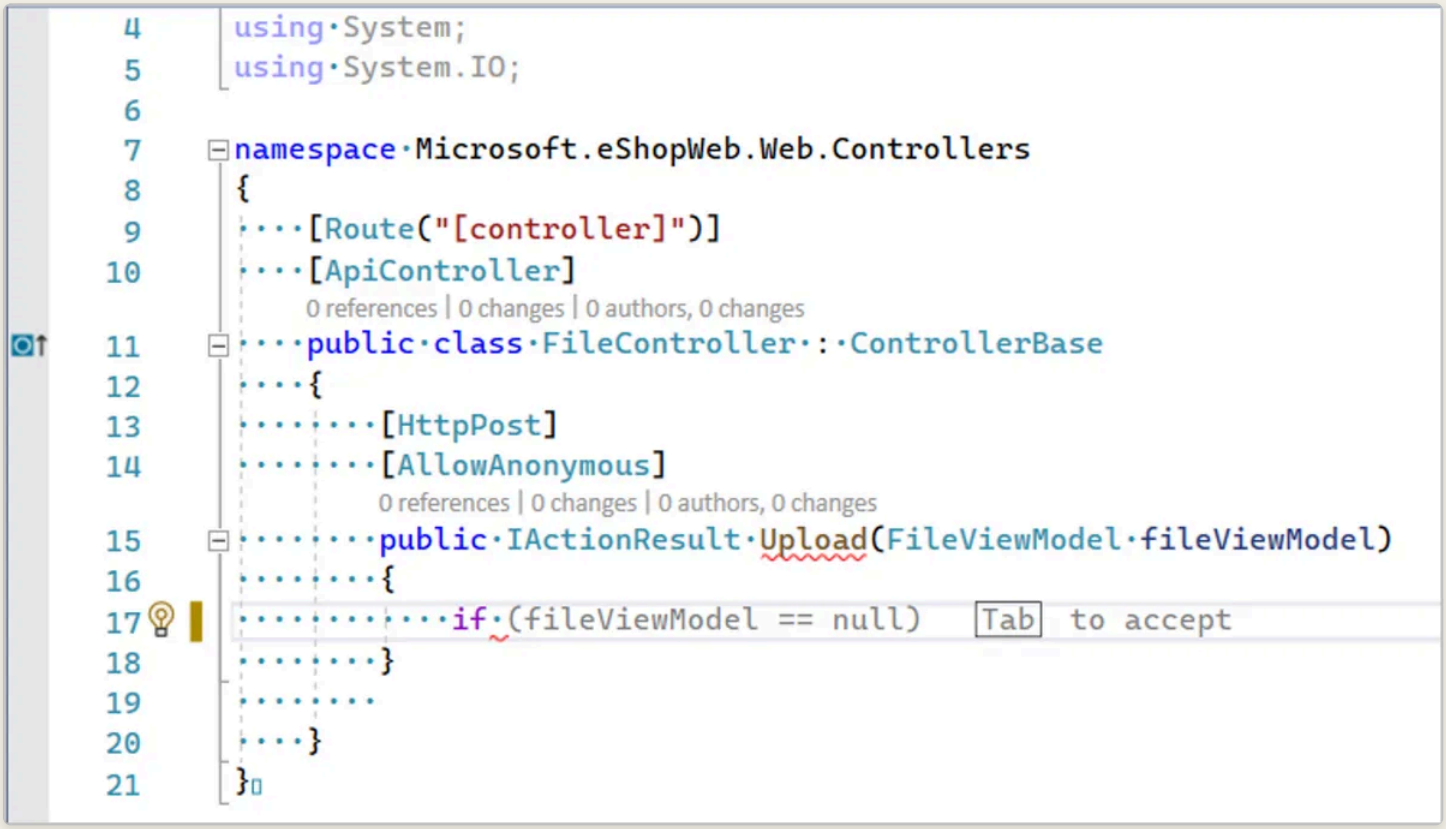
They trained their models on several open source projects hosted on Github. They initially used Markov Chains and slowly moved to deep learning based suggestions in

2020 (GPT-C). You can read more details in this blog post from one of the individual contributors of that project.

Other prominent features were “Repeated Edits” detection - detecting pattern in your edits and recommending automating similar changes throughout the codebase, “Quick Actions” and refactoring suggestions.

It was also possible to train Intellicode on a company’s repository to align it’s suggestions with the team’s coding standards. (enterprise part I believe)

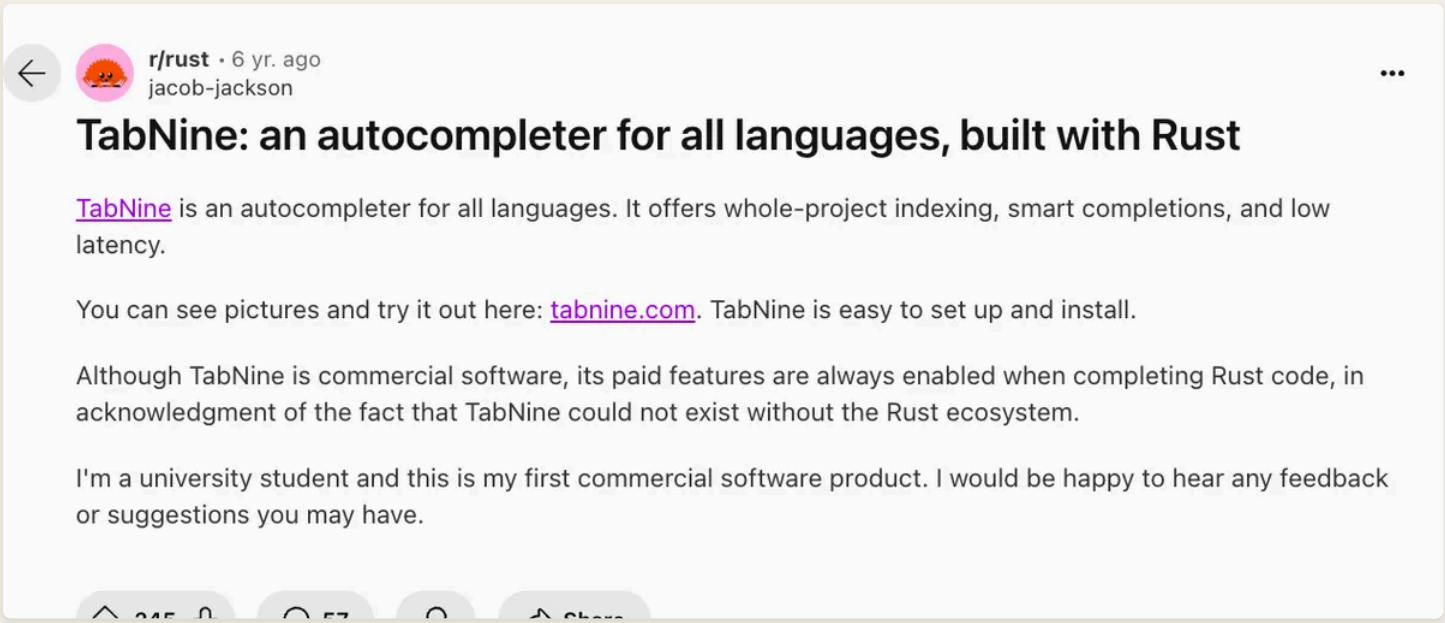
Towards single line autocompletion



Microsoft eventually the introduced “whole line prediction”. Whole line code prediction meant predicting the next logical line of code based on context. Remember the grey-color predictions in the editor?

It's worth noting that early code completion models weren't language-agnostic like today's tools. Each had its specialty: Intellicode focused on C# and .NET, ReSharper on C#, and Eclipse IDE on Java.

Multi-line autocompletion



The first real breakthrough came with Tabnine by Jacob Jackson. It was the first code-editor to integrate GPT-2 for code completion. It led the groundwork for the future AI code-editors. I am not 100% sure but it looks like TabNine was the first editor to support language-agnostic multi-line code completions based on local context.

GPT-2's training on diverse codebases enabled the first truly language-agnostic code completion model. Karpathy's tweet shows how the tab completions used to look like. here's a video as well. It showed suggestions of next line along with the percentages.

You may notice that auto-completion has always been Tab.

Enter Block level code completions

We apparently didn't have block level code completions like generating entire functions or methods based on user input and context-awareness code gen or understanding user intent

till the release of OpenAI Codex model. OpenAI codex powered Github Copilot. It was able to perform the above features with decent accuracy.

From OpenAI's Codex model to GitHub Copilot

When OpenAI released GPT-3 in June 2020, GitHub knew developers would benefit from a product that leveraged the model specifically for coding. So, we gave input to OpenAI as it built Codex, a descendant of GPT-3 and the LLM that would power GitHub Copilot. The pair programmer launched as a technical preview in June 2021 and became generally available in June 2022 as the world's first at-scale generative AI coding tool.

To ensure that the model has the best information to make the best predictions with speed, GitHub's machine learning (ML) researchers have done a lot of work called prompt engineering (which we'll explain in more detail below) so that the model provides contextually relevant responses with low latency.

Though GitHub's always experimenting with new models as they come out, Codex was the first really powerful generative AI model that was available, said David Slater, a ML engineer at GitHub. "The hands-on experience we gained from iterating on model and prompt improvements was invaluable."

Source: How Github Copilot is getting better at understanding your code

Fill in the middle

The autocompletions we have talked about so far were limited to left-to-right operation. These models could only "append". They couldn't look ahead into the suffix for context. Earlier models couldn't fill the gap.

```
def calculate_total(items):
    # [gap here]
    return total
```

```python
# complete the self-attention code
def self_attention(x):

    # 1. compute the query, key, value
    query = x
    key = x
    value = x

    # 2. compute the attention scores
    attention_scores = torch.matmul(query, key.transpose(-2, -1))


    return x
```

This limitation was later addressed by the Fill in the middle technique introduced by OpenAI. FIM stands for fill in the middle. Sometimes referred as bidirectional awareness i.e understanding both before and after the cursor. Many of you reading this blog might be hearing of FIM for the first time.

I recommend reading this blogpost by Codeium to understand the intuition for FIM. OpenAI paper for training details.

Codeium was the first to integrate FIM in their extension beating Github Copilot who also started supporting it by May ‘23 at scale. You can start seeing terms like “prompt engineering” and “context-window” from the blog

Modern autocompletion has evolved further through:

- LSP (Language Server Protocol) integration for real-time syntax info and symbol references
- Robust codebase indexing for tracking function relationships
- Enhanced context awareness through AST and symbol table analysis

The journey from simple text completion to context-aware code generation shows how far we've come, but this was just the beginning of what's possible with AI-assisted coding.

We have already reached May'23 in the timeline. Of course, we didn't want to stay limited to just autocomplete. We started getting coding assistants sometime back from late 2022. It's time to talk about coding assistants now, we will return back to autocompletion later.

Towards (Conversational) Coding assistants and AI-first Editors

Nov 30, 2022 - ChatGPT research preview featuring GPT-3.5

ChatGeeBeeDee 3.5 was probably the first time people people started realising the potential of code generation via LLMs. You could talk to this chatbot that could explain you entire blocks of code. It could provide accurate code if you provided it with a good enough prompt and relevant context.

Fast-forward March'23, we got the GPT-4 model which was amazingly good at code generation. Copilot integrated GPT-3.5/GPT-4 for copilot chat.

Post July '23, we got Llama-2 release which led to a cambrian explosion in the open source local LLM scene - in domains of code generation fine-tuning, character roleplay etc.

The Opportunity for AI code-editors

The pattern of work for a long time was dumping code in chatGPT (Plus), writing decent prompts and then copy-pasting code back to our editors. It required you to spot the correct places to paste too. I personally was ok with this workflow for a long time xD as I got to look at the code more and carefully read through the differences.

The other drawback was the "context-switching" penalty not just limited to changing window but also the fact that you may start browsing haha. The launch of GPT3.5/4 prompted a marketspace for various AI-editors. There were many problems to solve for better developer experience, primarily being

- Eliminate context switching between chat platforms and editors
- Implementing direct code edits from the LLM output
- Figure out ways to provide context more effectively to the LLMs
- Codebase indexing / better context awareness to feed to LLMs

Enters Cursor

The challenge was to be able to do all the above in a smooth and low-latency manner. I recommend checking out cursor's Problems-2023 blog.

Cursor was the first to jump ship on the same for implementing the chat experience and the diff-edit generation workflow (where they automatically apply the edits and show it in diff format). In hindsight, this was essential to leverage the GPT-4 hype. Shipping fast and offering a great UX was the key.

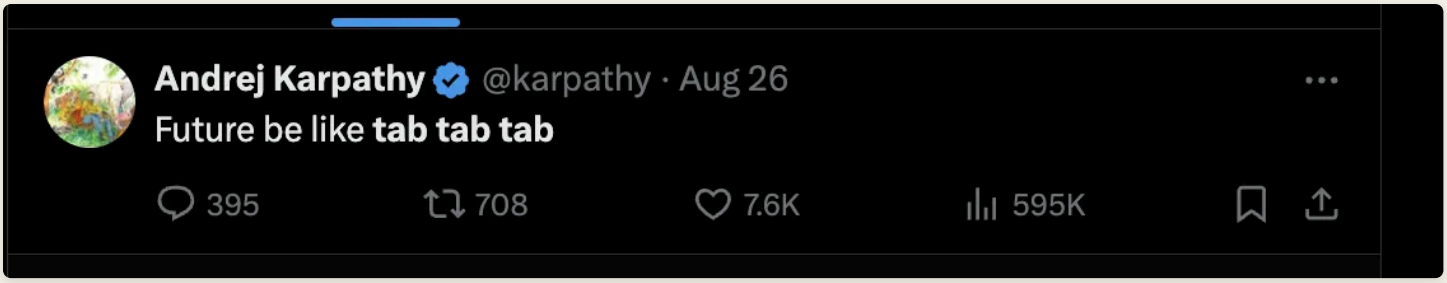
Shortly later, Cursor added features like partial accept and reject, codebase context feature that would semantically search across your codebase for your query. Later, they added CMD+K (line edit), CMD+L (chat) workflows.

Image source: Cursor Changelog

There were several other projects like Continue Dev (supported OSS models as well), Codeium but they didn't implement the edit experience as early as Cursor. There diff-apply (now fast-apply) plus chat interface was also smooth and preferred by people.

Back to autocompletion → Copilot++

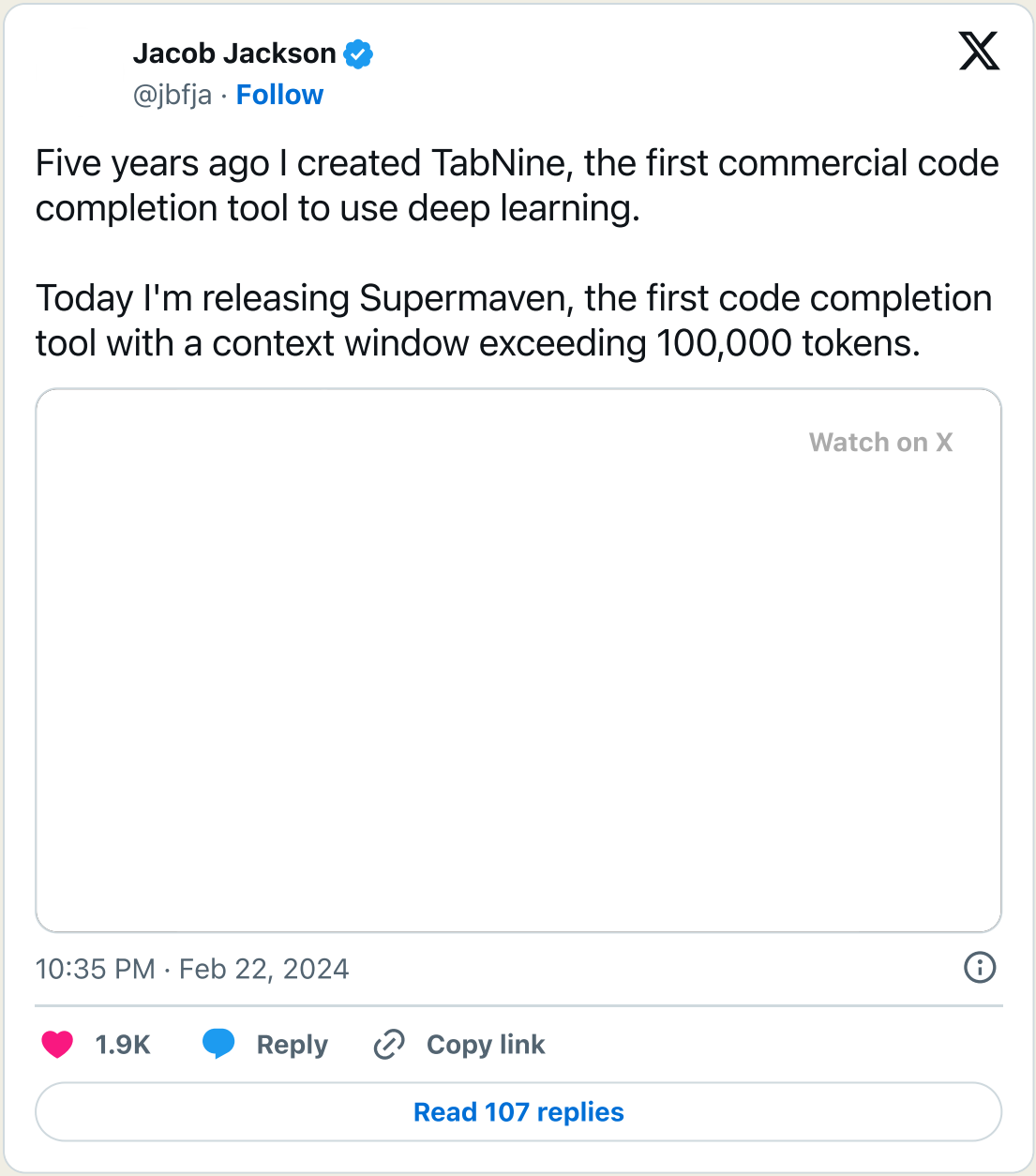
Cursor's autocomplete is the most advanced autocomplete I have used so far. It's like the boss of autocompletions we have discussed - it's trained (or finetuned?) in such a way that it predicts the **next-action of the developer**. Very intent-driven. "I made this change and the next line the model should go is line 18 ... and the model should know it". You can try refactoring name of a variable and it will suggest a multi-line refactoring. You could write comments for a function signature or a part of code and it will appropriate suggest completions (with FIM ofc).



Karpathy switching to Cursor was probably an inflection point for them. But IMO, the main inflection point was Cursor was the release of Sonnet 3.5 late in June ‘24. It was much better at code generation than any other models and was more agentic (better at instruction following and function calling). The solidified UI/UX experience, copilot++ helped them ride Sonnet’s wave.

Supermaven (autocompletion mainly)

Supermaven (by Tabnine founder) had a competitive (to Cursor) autocomplete too. I haven’t used it personally so based on hearsay and reading other’s inputs, it was at par or slightly worse than copilot++ but certainly faster. SuperMaven’s speciality at launch was a 100K token context window (more than GPT-4 and Sonnet (200K context) didn’t exist yet) which helped provide a tonne of context for better and faster autocomplete generations. They trained their own model with modifications over vanilla self-attention.



They later added Copilot++ like next-action prediction capabilities (**including cross-file jumps which probably Cursor** can't do yet), 1M context window.

Supermaven didn't have a chat experience integrated - it was an extension. They could not add several features due to VSCode's API limitations for extensions. **They merged with Cursor recently. I am excited for what's about to come in Cursor w.r.t to their codebase indexing and autocomplete experience.** Jacob mentions in this interview

> *0:49-1:14: Their visions and approaches to AI coding were converging and had significant overlap*
>
> *2:11-2:14: Super Maven would eventually need to build their own IDE due to VS Code extension limitations, duplicating Cursor's work*

> *2:35-2:52: The teams had complementary strengths - Cursor focused on user experience while Super Maven specialized in AI models*
>
> *28:12-28:19: Direct quote from Jacob: "throughout Super Maven I always thought if there was one other team that I would like to join forces with it would be cursor"*

I am expecting major upgrades to the tab autocompletion loop - it will probably be able to predict your actions across multiple files at once. supermaven's 1M long context model should improve the autocomplete as well as context detection in cursor chat / composer agent mode.

OSS Code editors

2024 saw the emergence/more popularity/funding of OSS code-editors and extensions like Continue dot dev (autocomplete, OSS models, code explanation but no apply of edits), Aider-chat (llm chat + edits via terminal), Pear AI, AIDE, avante.nvim plugin... Personally, I have found only aider-chat interesting to use a bit. I love their blog.

Patterns

The lower the gear in a car, the more control you have over the engine but you can go with less speed. If you feel in control, go to a higher gear. If you are overwhelmed or stuck, go to a lower gear. AI assisted coding is all about grokking when you need to gain more granular control and when you need to let go of control to move faster. Higher level gears leave more room for errors and trust issues. **Prompts** are the steering wheels (gear 2 onwards).

An observation from a small sample of people - more senior people or people working at the cutting edge prefer lower gears. Non-technical people are attracted more by the higher gears (ironically).

1st Gear Autocomplete

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
)



// KeyValueStore holds the data in a map.
type KeyValueStore struct {
    store map[string]string

}
```

There's a lot of emphasis on code generation via LLMs but the **model level UX unlock** ("baked into the weights") that truly makes editors feel like a magical comfy place is the **autocomplete**. Autocomplete is probably the first or second most used feature in AI code editors - you will find yourself working in an existing codebase more often than coding one from scratch.

If you ask senior engineers (who still write code and not just do meetings), they will say they mostly use autocomplete. People working in specialised domains where the LLMs are still not good also probably benefit the most from autocomplete.

When you are working on an existing codebase, you are making more edits than writing brand new code. In production codebases, you would find yourself pattern matching to write new features often. Production code requires more surgical precision than broad strokes.

You are mostly making edits across files, targeting to insert a feature somewhere or fixing some bug, possibly refactoring. Here the **localised context** (remember supermaven 1M context model?**)** fed into the model is important.

The point I am trying to make is - autocomplete speaks to your desire of a more granular **control** over your actions. It's like driving your car in the **first gear**. The lower the gear, the higher the control on the engine of the car. And a good autocomplete model fulfills your control preferences by figuring out the exact context and intention.

I love how the Cursor engineers are framing autocomplete as a more generalized "next action prediction" model problem.

> *To start, we've extended Copilot++ to predict your next location. Combine this with next edit prediction, and the model can play through a sequence of low-entropy changes:*
>
> *We press tab 11 times and all other keys 3 times. We call this **Cursor Flow** (for obvious reasons).*
>
> *We're working on predicting the next file you will move to. The next terminal command you will run. The next edit, conditioned on your previous terminal commands! A **next action prediction model**.*
>
> *Furthermore, the model should surface information the moment you need it. Whether it be the right piece of code or documentation.*
>
> *Cursor should feel like an extension of your will. The moment you think of a change, the language model requires minimal intent to execute it instantly.*

2nd Gear - Conversational LLM / standalone chat

A lot of programming work boils down to three things

- knowledge
- localised context
- understanding

If you want to add a feature in an existing codebase and you are ready to implement, the next step is figuring out where the **fuck** you want to make the change. Bolded the "fuck" because I have PTSD from digging humongous Java codebases to find the files where I want to make the change. (I couldn't use AI either so the fastest way was to run the server and hook up Intellij Debugger. Set up a few breakpoints and then grok the flow of the program).

This is where features like cursor chat comes handy. You can ask questions across the codebase using the @codebase feature that uses semantic search across codebase to search for relevant files. Post that, you could ask questions to improve your understanding of the codebase.

Alternatively, if it's allowed to copy paste, you can copy relevant files or dumping entire codebases into claude's 200K context or 1M long context models like gemini 2.0 flash / 1206 exp and then asking questions as to where one can find what.

Autocomplete + Chat together are like steroids especially when you are editing or debugging stuff. I feel this is a "cognitive level" unlock as you are improving your understanding of the code.

There's a lot of emphasis on code-generation in 2024/5 and shipping but not a lot on how we can use these tools to deepen our understanding of codebases and ultimately tech itself. Topic for another post I guess.

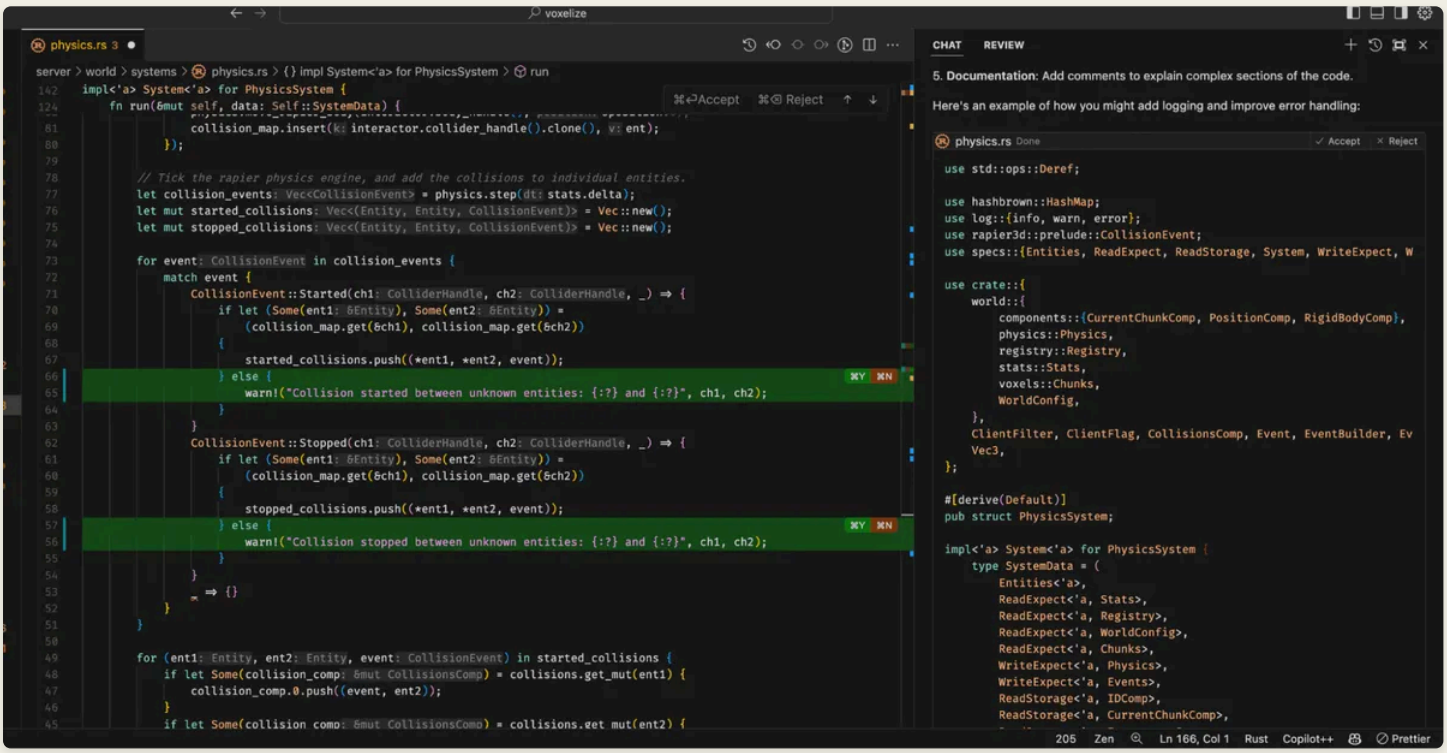
3rd Gear - Cursor Chat + Apply Edit, Windsurf Cascade Chat

I would place copy-pasting generated code from ChatGPT/Claude AI platform in the 2nd gear. It gives more control to the user since they have to manually determine the position of changes and it's slower. Also involves a context-switch which is very susceptible to changing tabs and going onto internet detours.

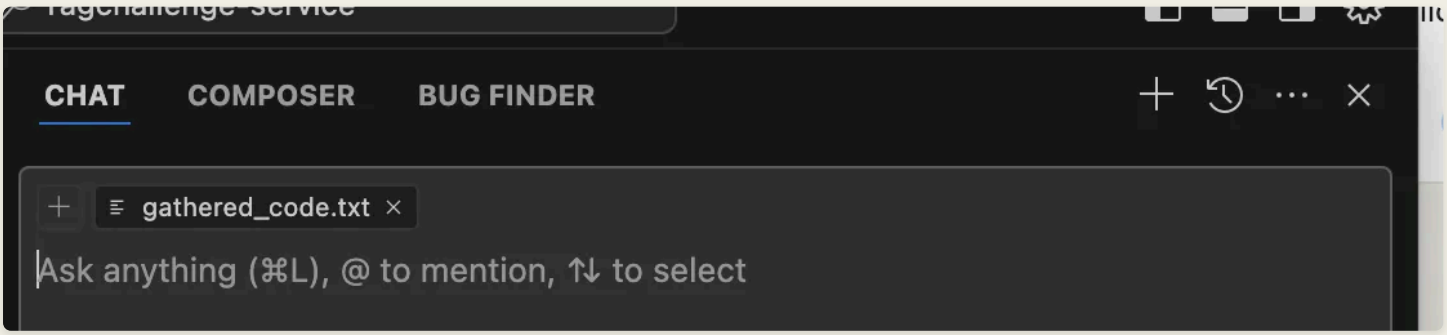
The 3rd gear was probably introduced by Cursor. Parsing LLM output and applying the edits in the **correct places.** Emphasis on correct places as it saves so much labour plus solves for the context-switch issue.

The diff format also helps me quickly review through the code. This is fast plus I get to review the code. The control is still there and your trust issues aren't triggered because you can still verify the changes (and reject changes)

> *This has actually been a pretty hard problem to solve due to stuff lazy coding from models like GPT-4 although Sonnet probably made it easier for the developers. Their blog on this is no longer up but it was mostly solved initially by using unified diff. They later added speculative editing to make it faster.)*



People who are writing a lot of new features or doing 0 to 1 - gear 3 is a sweet spot.



My recommendations for Gear 3 is to divide your task into subtasks and then work on each of them individually. **Make sure to refresh the context window for each task. Hint: Plus button**

If you are working in a more mature codebase, you will operate the most in Gear 1 to 3.

4th Gear - Agentic features - Windsurf Cascade and Composer Agent mode

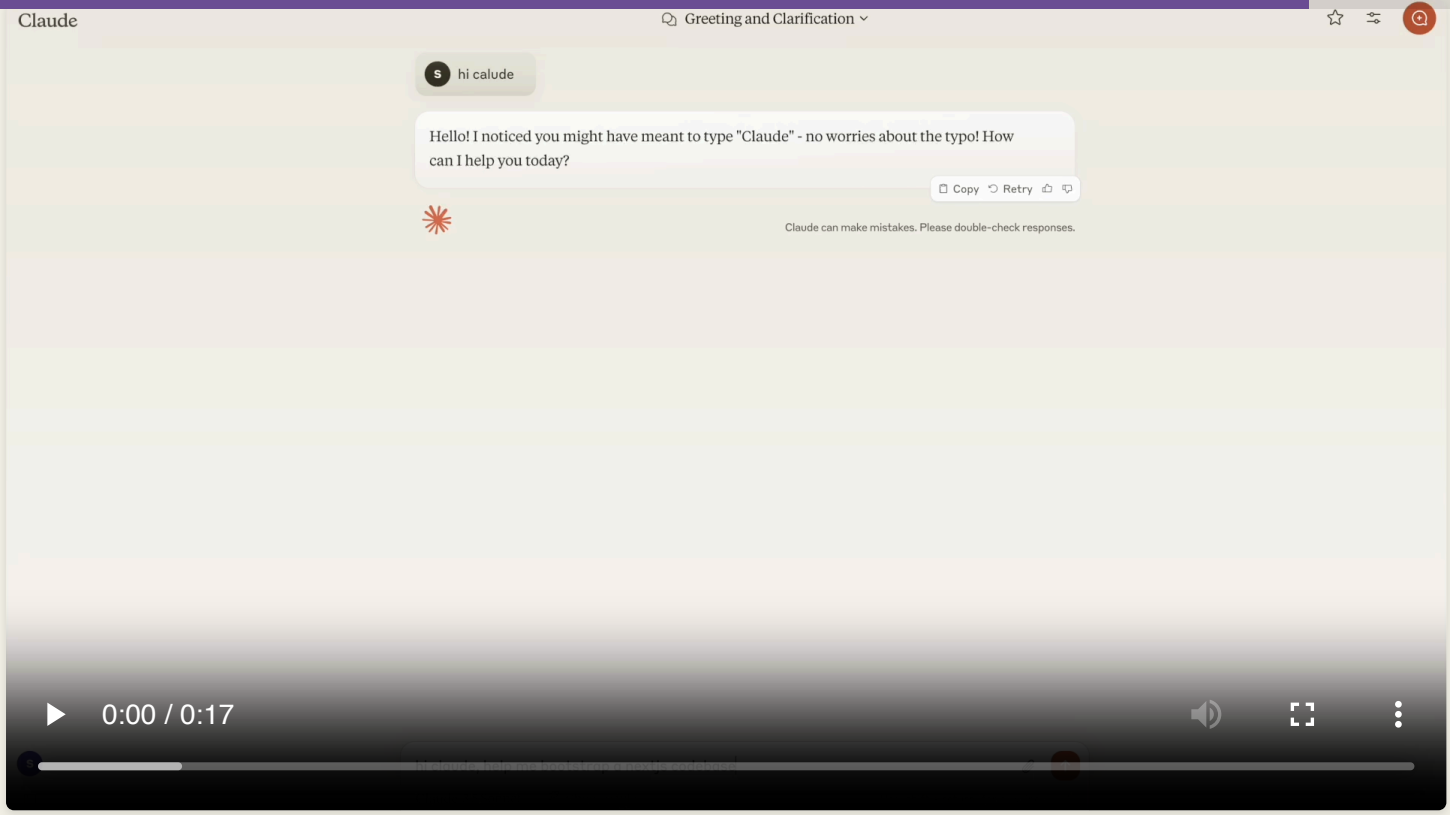
Honourable mention: Composer - normal mode is useful for multi-file editing and refactoring. It detects context on it's own but not very reliable to be honest. Composer agent mode solves for this though.

Things get a bit spicy here. This gear is only possible because of models like Claude 3.5 Sonnet with great agentic abilities. "Agentic" - in simple language, refers to the model's ability to be able to figure out, plan and proactively perform subtasks on it's own to complete your main task.

More specifically, the model can plan subtasks, do some stuff, be aware of intermediate states, look at the intermediate context, adjust it's course of execution and complete the end user task.

Agentic abilities are "baked in the weights" via pre-training and fine-tuning for function calling. When people say a model is agentic, they mean it's able to follow your instructions accurately ("good instruction following") and it is good at tool/function calling.

To be good at agentic code-generation, the model also needs to be very good at code generation obviously. Sonnet is a great balance of great coding skills and SOTA agentic performance. Sonnet new is even better at agentic codegen. It is more pro-active than old Sonnet. TLDR: Agentic mode is a capability unlock at the model level.

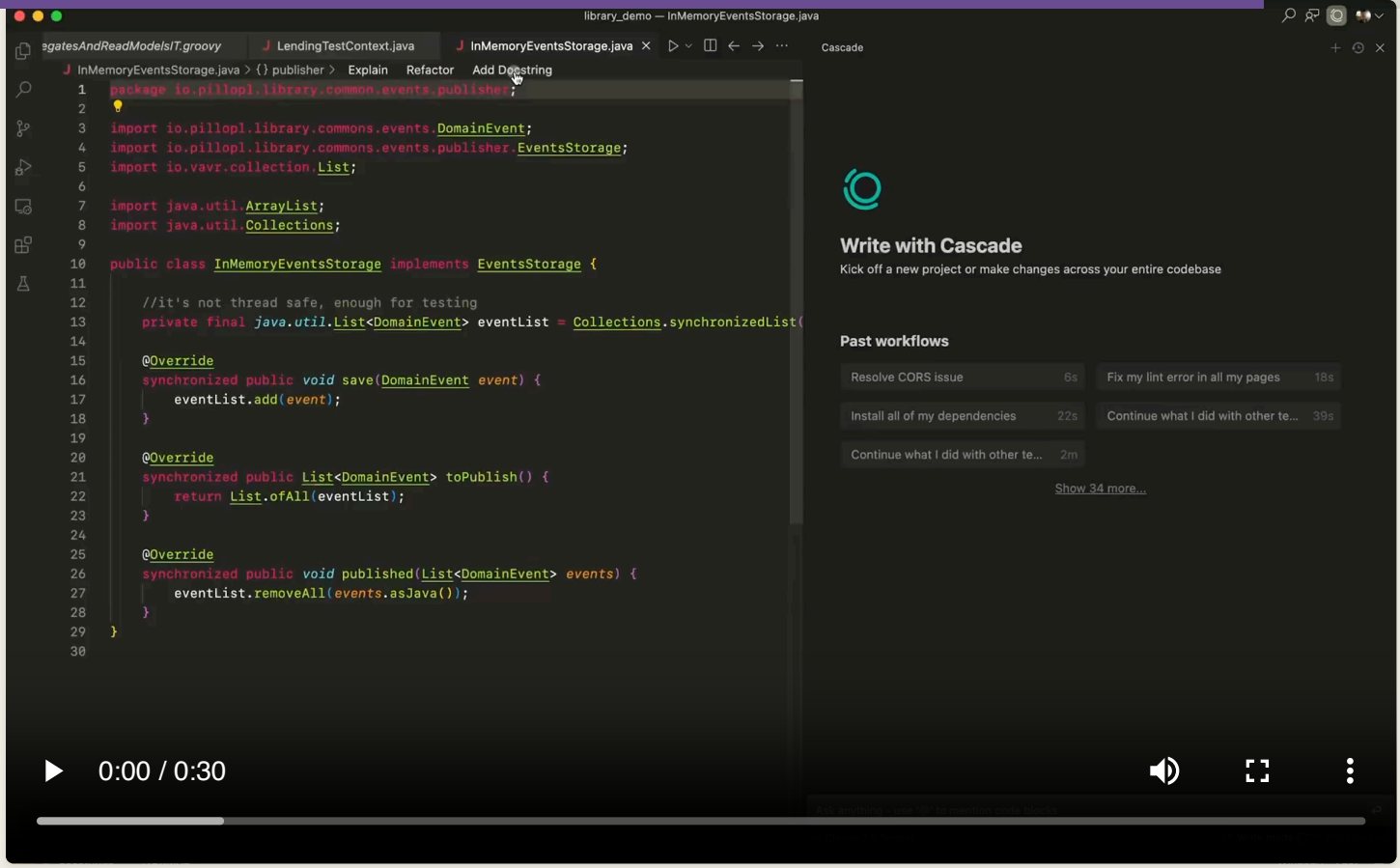


You can see in above video how Sonnet keeps calling “Artifact” several times to complete the task. I believe it’s a tool call to open up Artifacts. This is agentic.

Windsurf by Codeium seems to be the first editor to introduce the proper agentic mode at the editor level. It’s **primary mode of function** is “**Cascade**”. When I had first used it, it felt like taking a glimpse into the near future of coding.

Edit: My experience with Windsurf was from a few weeks back. They have now added a Chat mode too where you can brainstorm and apply edits manually.

You can basically tell some high level instructions and the model will keep planning and doing a bunch of tasks, running commands through terminal, creating files etc. to complete your task. It shows it’s own “agency”. Windsurf provides a great interface to see all the things the model is doing and shows the diff edits like Cursor does. UI/UX wise, I think it’s better than Cursor Agent mode. Also slightly better at detecting the context (opinion based on some personal usage from a few weeks back and hearsay from friends)



"Agent mode" in editor is extremely useful for refactoring files and making multi-file creation/edits. It's usefulness is more when you are in the 0 to 1 phase as compared to being in the middle of dev stages. There can be errors often though in terms of file detection or just undesirable code (lots of code being generated so context window may be exhausting, knowledge cutoff issues)

You are trading off a lot of control here to ship faster. The editors implement diff views so atleast you can review the changes though. I am comfortable using agent mode in cursor when I roughly know which files to edit.

It seems less experienced programmers are more attracted to Windsurf because of the appeal of Cascade (although I feel the sweet spot for them is gear 3). Windsurf is very competitive to Cursor however it still needs some polish and support for minor features (like fetching docs from a link). Cursor is more power user centric and provides more gears to operate with. That said, Windsurf tends towards reducing decisions on the user's side.

Personal observations

In programming, getting stuck is common. When I get stuck, I would like to try a different mode of operation - where I have more control and is slower. That way having

the option to switch to a lower gear like chat from agent mode offers a good way to pace myself / slow down and think on the issue. Beginners are more susceptible to get into the error loop...

Agent modes also eat up your requests quota faster. Less experienced people/non technical will suffer more here because they are going to be stuck in loops more often. Which is why again I prefer chat+edit mode where you can control your requests quota too implicitly.

One pattern of usage I like is - use agent to make draft of features (as it can create files at correct places) and then carry on with chat + autocomplete.

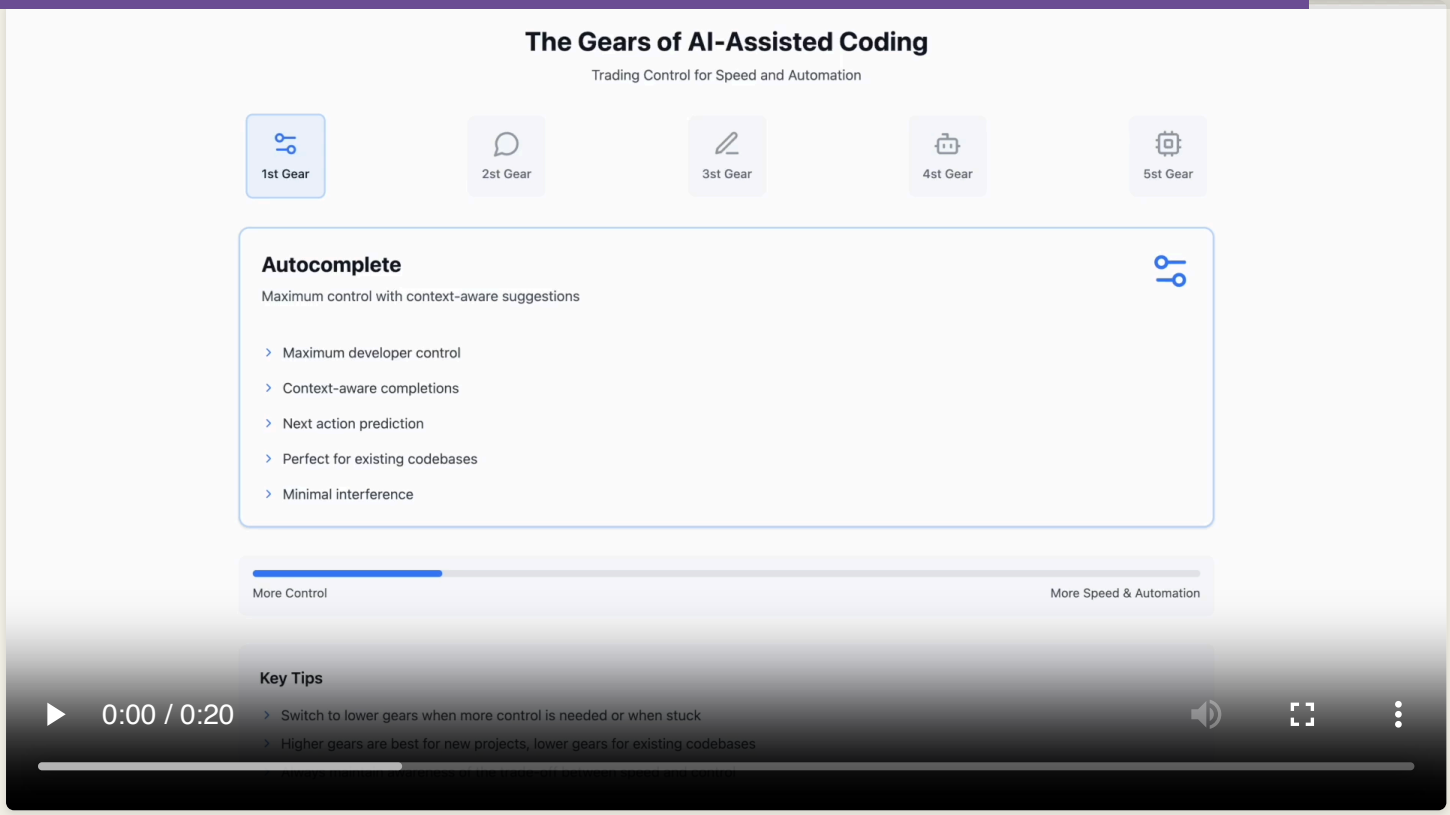
5th Gear - Coding agent tools

I had always been skeptical at agents but after the launch of new sonnet and working on a project recently which had an agentic component to it - I realised if you n**arrow down the scope of problem you are solving, agents are effective**. The value you get is in reduction of human hours.

I have personally tried bolt.new. It works on the same principle. The scope is clear - 0 to 1 bootstrapping with a opinionated frontend framework (at which the LLM is also good). bolt.new is very good for 0 to 1. Once the codebase matures, you start seeing diminishing returns.

We also saw the advent of Replit Agent, Devin although I have not personally tried these. These are more broadly scoped - end to end agents. “ai software engineer” if you will. Devin for instance can navigate through your codebases, take notes, make changes, report back. It can work in async (but it’s slow based on what i have read)

Agents are all about how losing control to do stuff faster. The scope for error and mishaps is also the most here especially in the broadly scoped ones. **I personally prefer editors (gear 1 through 4) instead of coding agent tools.**



Common pitfalls with agentic tools and considerations

You can ship fast with cursor agent or windsurf cascade. Same for standalone tools. You are only fast till you get stuck and when you get stuck, it can be hard to debug sometimes (especially for lesser experience programmers or non technical people). The models can get stuck in loop often due to outdated knowledge (for instance sonnet cannot debug nextjs 15 breaking changes to nextjs14).

You will have to read docs and stackoverflow/github discussions, search for the solution the traditional way. Change gears - Gear 2 and 3 are very effective here. My tip here is to improve your understanding of the code at this point. Understand the problem you are trying to solve to prompt better or solve the bug manually.

AI assisted coding can be very fast until it isn't

Most slowdowns happen in the last mile of features - hence you need to be aware of what's going on in the code. Otherwise debugging is gonna be painful. Screenshot from the "the 70% problem: Hard truth about AI assisted coding". I highly recommend reading this blog as it offers actionable insights to get better at AI Assisted coding.

then reality sets in.

The two steps back pattern

What typically happens next follows a predictable pattern:

- You try to fix a small bug
- The AI suggests a change that seems reasonable
- This fix breaks something else
- You ask AI to fix the new issue
- This creates two more problems
- Rinse and repeat

Tunnel vision

Avoid tunnel visioning. If you are stuck somewhere, take a break. Zoom out and think broadly. Discuss with LLMs or teammates about alternative approaches and then proceed again.

Learn basics if new to an area

I also recommend you atleast learn the basics of whatever you are working on in order to prompt well. Learning basics means you are able to convert unknown unknowns to known unknowns.

Time to wrap up the post now. It's been very long.

Conclusion

AI-assisted coding is only gonna improve from here. It's high time we utilize it the best. We already have very strong models and they are only going to get better. Lots of work

needs to be done on integrating context side. Sorry, this word comes 30+ times in this blog but context is really the key.

I hope the gears analogy will help you in some way in your daily workflows. Thanks for reading. Please share if you liked it.

#AI #featured #technical

69
