AI Blindspots - A Collection of Insights

Stop Digging

March 3, 2025

Outside of very tactical situations, current models do not know how to stop digging when they get into trouble. Suppose that you want to implement feature X. You start working on it, but midway through you realize that it is annoying and difficult to do because you should do Y first. A human can know to abort and go implement Y first; an LLM will *keep digging*, dutifully trying to finish the original task it was assigned. In some sense, this is desirable, because you have a lot more control when the LLM does what is asked, rather than what it *thinks* you actually want.

Usually, it is best to avoid getting into this situation in the first place. For example, it's very popular to come up with a plan with a reasoning model to feed to the coding model. The planning phase can avoid asking the coding model to do something that is ill advised without further preparation. Second, agentic LLMs like Sonnet will proactively load files into its context, read them, and do planning based on what it sees. So the LLM might figure out something that it needs to do without you having to tell it.

Ideally, a model would be able to realize that "something bad" has happened, and ask the user for request. Because this takes precious context, it may be better for this detection to happen via a separate watchdog LLM instead.

Examples

- After having made some changes that changed random numbers sampling on a Monte Carlo simulation, I asked Claude Code to fix all of the tests, some of which were snapshots on exact random sampling strategy. However, it turns out that the new implementation was nondeterministic at test time, so the tests would nondeterministically pass/fail depending on sampling. Claude Code was incapable of noticing that the tests were flipping between passing/failing, and after unsuccessfully trying to bash the tests into passing, started greatly relaxing the test conditions to

account for nondeterminism, instead of proposing that we should refactor the simulation to sample deterministically.

Black Box Testing

March 3, 2025

Black box testing says that you should test the functionality of a component without knowing its internal structure. By default, LLMs have difficulty abiding with this, because by default the implementation file will be put into the context, or the agent will have been tuned to pull up the implementation to understand how to interface with it. Sonnet 3.7 in Cursor also has a strong tendency to try to make code consistent, which that it will try to eliminate redundancies from the test files, even though black box testing suggests it is better to keep the redundancy to avoid reflecting bugs from the implementation directly in the test.

Ideally, it would be possible to mask out or summarize implementations when loading files into the context, to avoid overfitting on internal implementation details that should be hidden. It would be necessary for the architect to specify what the information hiding boundaries are.

Examples

- I asked Sonnet 3.7 in Cursor to fix a failing test. While it made the necessary fix, it also updated a hard-coded expected constant to instead be computed using the same algorithm as the original file, instead of preserving the constant as the test was originally written.

Use Static Types

March 6, 2025

The eternal debate between dynamic and static type systems concerns the tradeoff between ease of prototyping and long term maintainability. The rise of LLMs greatly reduces the pressure to choose a language that is good at prototyping, since the LLM can cover up for boilerplate and refactors. Choose accordingly. You will want an agent setup where the LLM is informed about type errors after changes they make, so they can easily tell what other files they need to update when doing refactors. Be careful about your token costs.

Unfortunately, the training corpus highly emphasizes Python and JavaScript. Their typed offerings are workable, but as both are gradual type systems you will need to carefully setup the typechecker settings to be strict (or carefully prompt your LLM to setup the settings properly).

In principle, Rust should be a good target language for LLMs. However, LLMs have shown some limitations in working with its strict type system.

Use MCP Servers

March 6, 2025

Model Context Protocol servers provide a standard interface for LLMs to interact with their environment. Cursor Agent mode and Claude Code use agents extensively. For example, instead of needing a separate RAG system (e.g., as previously provided by Cursor) to find and feed the model relevant context files, the LLM can instead call an MCP which will let it lookup what files it wants to look at before deciding what to do. Similarly, a model can run tests or build and then immediately work on fixing problems when this occurs. It is clear that Anthropic's built-in MCP servers are useful, and you should use agent mode when you can.

Epistemic status: this next section is theoretical, I have not attempted it in practice yet.

A further question to ask is whether or not you should be writing your own MCP servers. One of the built-in MCP servers is a shell, so in practice you can turn on YOLO mode in Cursor and add some instructions to your Cursor rules about what commands to run and this actually works reasonably well. But it's really dangerous! Arbitrary shell commands can do anything, and there are plenty of shell commands in your LLMs pretraining set that will trash your environment. So you are mostly praying that your LLM

doesn't go off the rails some day (or you're a careful user and audit every command before running them—let's be real, who's got time for that).

The alternative is to write an MCP server that exposes the commands that you want your model to have access to (note, this is a bit different from what most people are thinking of when they write MCPs to access various existing platform APIs). In principle, this should make it easier to control what tools actually end up getting called, but as of March 2025 support for this in Cursor is poor. In particular, there is no direct way to have different MCP servers on a per project basis, and the overall direction the ecosystem is going with MCP servers are to provide tools that are universally applicable, rather than the equivalent of `npm run` as an MCP server.

Examples

- When I ask Sonnet 3.7 to typecheck a TypeScript project and fix errors, in agent mode it will use MCP to run the command, get the output, and decide what to do from there. There is much more convenient than having to manually copy paste terminal output into the chat window. It's important to prompt this appropriately (either in Cursor rules, or via an MCP), as the LLM is prone to hallucinating what command you should run.

Mise en Place

March 6, 2025

In cooking, *mise en place* refers to the practice of organizing and arranging all of the ingredients that will be needed during a shift, so that one does not have to scramble to find the things you need when you are on task.

Mise en Place for your LLM is ensuring that all of the rules, MCPs and general development environment for your model may need are properly setup before you have a task. In my experience, Sonnet 3.7 is not very good at fixing a broken environment; vibe coding to fix a mutable, live production environment is like randomly copy pasting commands from StackOverflow and hoping that something good will help. More likely than not, you're going to just irreparably destroy your environment. So make sure that

everything is setup correctly, so that Sonnet doesn't go down a big rabbithole trying to debug why your environment isn't working when as an agent it tries to run some tests.

Examples

- Due to some `npm link` shenanigans, I had a broken VSCode environment where I wasn't picking up imports to another local project correctly. When I asked Cursor to run lint and fix tests, it got fixated on trying to fix this brokenness.

Respect the Spec

March 7, 2025

When designing changes, it is important to keep in mind what parts of the system you can change and what parts you cannot. For example:

- If you expose a public API, you should prefer not to make a BC breaking change to the API, even if it would make your life easier if the API was different.
- If you interact with an external system, you have to conform to the API that actually exists, not some wishful thinking version that directly gives you what you want.
- If a test is failing, you should not delete it to make the test suite pass, you should understand if the test is right or not.

The common theme for all of these is that some parts of the system are part of the specification, and while *occasionally* the right thing to do is to change the spec, for day-to-day coding you would like to respect the spec.

LLMs are not very good at respecting the spec. They'll happily delete tests, change APIs, really do anything while they're coding. Some of these boundaries are common sense and could potentially be encoded in your prompt, but some you will only discover when the LLM comes up with some new and fascinating way to hack the reward function. This is one of the most important functions of reviewing LLM generated code; ensuring that it didn't change the spec in an unaligned way.

Examples

- Sonnet was failing to fix a test, so it replaced the contents of the test with `assert True`.
- I was trying to make a file well typed. One public function was returning a dict with the key `pass`, and on a first pass Sonnet 3.7 tried to use the TypedDict class syntax, but this doesn't work because pass is reserved keyword. To fix this, Sonnet decided to rename the key to `pass_`.

Memento

March 7, 2025

In the movie Memento, the protagonist is unable to form new memories, and has to resort to an elaborate system of notes to remember what he has done in the past to uncover who killed his wife.

Like in Memento, the LLM you are working with has no memory. Whenever you ask it to perform a task, it must reconstruct enough context to do what it needs to do. This will be the prompt (e.g., the Cursor rules for your project), context that is explicitly/implicitly attached, and anything the model decides to ask for in agentic mode. That's it! Your model is speed running understanding your codebase from scratch every time you start a new chat.

"The marvel is not that the bear dances well, but that the bear dances at all." It is incredible that LLMs have this capacity to reunderstand your codebase from scratch every request. But it is a fragile thing; if the model fails to get the right files into its context, it can quickly start ripping up the floors, having gotten into its head the wrong idea of what you are trying to do.

Help the model do the right thing: make sure there documentation it can reference, and make sure the model can find it (e.g., via prompting, or just putting it where the model expects it to be). Avoid asking for major changes (where misalignment on the actual goal is more likely to be harmful) without contextualizing how it should fit together with the project as a whole.

Examples

- I asked Sonnet 3.7 to come up with a project plan to come up with a way of doing end-to-end testing of an already existing project. Sonnet interpreted this to mean that the raison d'être of the entire repository was testing.

Scientific Debugging

March 9, 2025

When there is a bug, there are broadly two ways you can try to fix it. One way is to randomly try things based on vibes and hope you get lucky. The other is to systematically examine your assumptions about how the system works and figure out where reality mismatches your expectations. I generally think that the second approach of scientific debugging is better in the long run; even if it takes you more time to do, you will walk away with a better understanding of the codebase for next time.

A non-reasoning model is not going to do the scientific method. It is going "guess" what the fix is, and try to one shot it. If you are in an agent loop, it can rerun the test suite to see if it worked or not, and then randomly try things until it succeeds (but more likely, gets into a death loop).

The general word on the street is you should use a reasoning model for debugging (actually, people tend to prefer using models like Grok 3 or DeepSeek-R1 for this sort of problem). Personally, when I do AI coding, I am still maintaining a pretty detailed mental model of how the code is supposed to work, so I think it's pretty reasonable to also try to root cause it yourself (you don't have to fix it, if you tell the model what's wrong it will usually do the right thing).

Examples

- One of the most horrifying death loops an agent LLM can get into is trying to fix misconfigurations in your environment; e.g., a package is missing for some reason. One funny problem with Sonnet 3.7 is that it expects `pip` to be available, which it's not in the default venv created by `uv`, and it is incapable of figuring out what went wrong here, wasting a lot of tokens.

The Tail Wagging the Dog

March 10, 2025

The tail wagging the dog refers to a situation where small or unimportant things are controlling the larger or more important things. A common reason this occurs in software engineering is when you get too absorbed in solving some low level problem that you forgot the whole reason you were writing the code in the first place.

LLMs are particularly susceptible to this problem. The problem is that in the most common chat modality, everything LLM does is put into the context. While the LLM has some capability of understanding what is more or less important, if you put tons of irrelevant things in the context, it will become harder and harder for it to remember what it should be doing. Careful prompting at the beginning can help, as is good context hygiene. Claude Code does something smart where it can ask a subagent to do a task in a dedicated context window without polluting the global context.

Examples

- If not carefully prompted, if you ask an LLM to think about how they would do something, they will often forget that they were only supposed to think about it and will go straight to trying to do the thing they were thinking about.

Know Your Limits

March 12, 2025

It is important to know when you are out of your depth or you don't have the tools available to do your job, so you can escalate and ask for help.

Sonnet 3.7 is not very good at knowing its limits. If you want it to tell you when it doesn't know how to do something, at minimum you will have to explicitly prompt it (for example, Sonnet's system prompt instructs it to explicitly warn a user about

hallucinations if it is being asked about a very niche topic.) It is very important to only ask the LLM to do things that it actually can do, especially when it's an agent.

Examples

- Sonnet 3.7 deeply, deeply believes that it has the capability to make shell calls. If you are doing agentic coding and there is no bash command available, if Sonnet decides that it needs to make a file executable, it will start creating random shell scripts on your filesystem to try in some weird proxy of the thing they actually wanted to do. It's common for Sonnet to say, "I am going to do X" and then generate a tool call for Y, which is totally different. The best thing to do in this situation is to improve the prompt (not perfect, Sonnet will forget) or just give Sonnet a tool that does the thing it wants to do.

Culture Eats Strategy

March 12, 2025

Culture Eats Strategy (For Breakfast) says no matter how good your strategy is, the culture of your team isn't capable of executing it. If your problem is execution, look to change the culture instead of trying to come up with increasingly elaborate strategies.

By default, your LLM lives in a certain part of the "latent space": when you ask it to generate code, it will generate it with a style that is based off of how it was fine tuned, as well as its context window up until the point (this includes the system prompt as well as any files it's read into the context.) This style is self-reinforcing: if a lot of the text in the context window that uses a library, the LLM will continue to use that library—conversely, if the library is not mentioned at all and the LLM is not fine-tuned to reach for it by default, it will not use it (there are exceptions, but this is a reasonably good description of how Sonnet 3.7 will behave.)

If the LLM is consistently doing things you don't like, you need to change its culture: you need to put it in a different part of the latent space. This could be adding a rule to your Cursor rules (modifying the prompt), but it can also be refactoring existing code to follow the style you want the LLM to follow, since LLMs are trained to predict the next token in context. The fine-tune, the prompt and the codebase are the culture. One you

can't change, and the codebase is a lot bigger than the prompt and ultimately will have a dominating effect.

Examples

- Sonnet 3.7's house style is to prefer synchronous over asynchronous Python. To get it to reliably write new code using async, it was necessary to force the issue by refactoring existing code to use async.

Rule of Three

March 13, 2025

The Rule of Three in software says that you should be willing to duplicate a piece of code once, but on the third copy you should refactor. This is a refinement on DRY (Don't Repeat Yourself) accounting for the fact that it might not necessarily be obvious how to eliminate a duplication, and waiting until the third occurrence might clarify.

LLMs love to duplicate code. Think about how, absent any other prompting, if you ask an LLM to modify a program, it will spit out a new copy of the program with the changes requested. For an LLM to decide to refactor code to remove duplication requires a moment where the LLM proactively decides to go out of their way to do some cleanup. (Actually, Sonnet 3.7 also has some of this inclination, but if you look at the Claude Code system prompt, they specifically instruct the model to not be *too* proactive, since it's easy for the model to lose track of what it's doing.) To make matters worse, if code is copy pasted everywhere in your codebase already, the LLM will assume that you want it to continue copy pasting the code everywhere. It's up to you to ask the model to reduce duplication.

Examples

- I asked the LLM to write tests for some functionality, and it ended up duplicating lots of logic between tests. I had to explicitly ask it to factor helper methods when doing so. (Another difficulty: when the LLM is writing a new test, it needs to know about the

helper to use it. In agent mode, you better hope it looks at the right file to find out about the helper.)

Preparatory Refactoring

March 3, 2025

Preparatory Refactoring says that you should first refactor to make a change easy, and then make the change. The refactor change can be quite involved, but because it is semantics preserving, it is easier to evaluate than the change itself.

Current LLMs, without a plan that says they should refactor first, don't decompose changes in this way. They will try to do everything at once. They also sometimes act a bit too much like that overenthusiastic junior engineer who has taken the Boy Scout principle too seriously and keeps cleaning up unrelated stuff while they're making a change. Reviewing LLM changes is important, and to make review easy, all of the refactors should happen ahead of time as their own proposed changes.

Ideally, the LLM would be instructed to not do irrelevant refactors (NB: anecdotally, Cursor Sonnet 3.7 is not great at instruction following, so this doesn't work as well as you'd hope sometimes). Alternately, a pre-pass on the code the LLM expects to touch potentially should be done to give the LLM a chance to do whatever refactoring it wants to do before it actually makes its change. Sometimes, context instructing the model to do good practices (like making code have explicit type annotations) can make this problem worse, since arguably the model was instructed to add annotations. Accurately determining the span of code the LLM should edit could also help.

Examples

- I instructed an LLM to fix an import error from local changes I made in a file, but after fixing the imports it also added type annotations to some unannotated lambdas in the file.

Stateless Tools

March 3, 2025

Your tools should be stateless: every invocation is independent from every other invocation, there should be no state that persists between each invocation that has to be accounted for when doing the next invocation. Unfortunately, shell is a very popular tool and it has a particulary pernicious form of local state: current working directory. Sonnet 3.7 is very bad at keeping track of what the current working directory is. Endeavor very hard to setup the project so that all commands can be run from a single directory.

Ideally, models would be tuned to prefer not to make tool commands that change state, even if they are available. If state is absolutely necessary, continuously feeding the current state into the model could help improve coherence. The RP community probably has quite a lot of lessons here.

Examples

- A TypeScript project was divided into three subcomponents: common, backend and frontend. Each component was its own NPM module. Cursor run from the root level of the project would have to cd into the appropriate component folder to run test commands, and would get confused about its current working directory.

Bulldozer Method

March 3, 2025

The Bulldozer Method suggests that sometimes you can achieve results that seem superhuman simply by just sitting down, doing the brute force work, and then capitalizing on what you learn by doing this to get a velocity increase. AI coding is the epitome of brute force work: you can just brute force large refactoring problems if you are willing to spend enough tokens, or you can have the LLM build the workflow you will

use to brute force the problem. Look for opportunity in places where previously people had written off a problem as "too much work". Make sure you inspect what the LLM is actually doing though, because it will happily keep doing the same thing over and over, unlike a human who would get bored and look for a better way.

Examples

- A historical annoyance people have had with strongly typed languages like Haskell or Rust is when you make a change to some core function, and then you have to refactor half the universe to account for it (fearless refactoring, they say, because the type checker will help you fix it). In many cases, the "read compile error, fix the problem" loop can be completely automated by an agentic LLM.
- I had some test cases with hard coded numbers that had wobbled and needed updating. I simply asked the LLM to keep rerunning the test and updating the numbers until they passed.

Requirements, not Solutions

March 3, 2025

In human software engineering, a common antipattern when trying to figure out what to do is to jump straight to proposing solutions, without forcing everyone to clearly articulate what all the requirements are. Often, your problem space is constrained enough that once you write down all of the requirements, the solution is uniquely determined; without the requirements, it's easy to devolve into a haze of arguing over particular solutions.

The LLM knows nothing about your requirements. When you ask it to do something without specifying all of the constraints, it will fill in all the blanks with the most probable answers from the universe of its training set. Maybe this is fine. But if you need something more custom, it's up to you to actually tell the LLM about it. If you ask the LLM something too underspecified and it misunderstands you, it's best to edit the original prompt and try again; because previous conversation stays in the context, leaving the incorrect interpretation in the context will make it harder for the LLM to get to the correct part of the latent space that lines up with your requirements.

By the way, if you are *certain* some aspects of the solution should work a particular way, it's very helpful to tell the LLM about it, because that will also help you get to the right latent space. And it's also important to be *right* about this, because the LLM will try very hard to follow what you ask it to do, even if it's inappropriate.

Examples

- If you ask Sonnet to make a visualization, chances are it will generate you an SVG. But if you specify that it needs to be interactive, it will give you something else.

Use Automatic Code Formatting

March 4, 2025

Automatic code formatting tools like gofmt, rustfmt and black help enforce a consistent coding style across your codebase. LLMs are generally not very good at following mechanical rules like "lines with no content on them should have no trailing spaces even if the indentation level is non-zero" or "make sure a line is wrapped at 78 columns." Use the right tool for the job.

This applies to lint fixes too: prefer lints that can be automatically fixed.

Keep Files Small

March 4, 2025

It has been long debated about at what size a code file is too big. Some say that it should be based on single responsibility principle (one class per file), others say that large files can be situationally OK and it depends on if it is causing problems.

Do not make files that are too large, if your RAG system for feeding code context can only operate on a per-file level, you will blow out your context; also, IDEs like Cursor will

start failing to apply the patch created by the LLM (and even when it succeeds, it can take quite a long time to apply the patch anyway—for example, on Cursor 0.45.17, applying 55 edits on a 64KB file takes). At 128KB, you will have trouble getting Sonnet 3.7 to actually modify the entire file (Sonnet's context window is only 200k tokens).

There is also little excuse for not keeping your files small; the LLM can handle all the drudgery of getting the imports right on your split out file.

Examples

- I asked Sonnet 3.7 in Cursor to move a small test class from a 471KB Python file to another file. Although the individual edits were small, Sonnet 3.7 had trouble with the file size.

Read the Docs

March 4, 2025

When you're learning to use a new framework or library, simple uses of the software can be done just by copy pasting code from tutorials and tweaking them as necessary. But at some point, it's a good idea to just slog through reading the docs from top-to-bottom, to get a full understanding of what is and is not possible in the software.

One of the big wins of AI coding is that LLMs know so many things from their pretraining. For extremely popular frameworks that occur prominently in the pretraining set, an LLM is likely to have memorized most aspects of how to use the framework. But for things that are not so common or beyond the knowledge cutoff, you will likely get a model that hallucinates things. Ideally, an agentic model would know to do a web search and find the docs it needs. However, Sonnet does not currently support web search, so you have to manually feed it documentation pages as needed. Fortunately, Cursor makes this very convenient: simply dropping a URL inside a chat message will include its contents for the LLM.

Examples

- I was trying to use the LLM to write some YAML that configured to call a Python function to do some evaluation. Initially, the model hallucinated how this hookup should work. After providing the manual as context the model understood how to fix the error and also updated the output format of the function.

Walking Skeleton

March 6, 2025

The Walking Skeleton is the minimum, crappy implementation of an end-to-end system that has all of the pieces you need. The point is to get the end-to-end system working first, and only *then* start improving the various pieces.

In the era of LLM coding, it has never been easier to get the entire system working. By using the system, it becomes obvious what the next steps should be. Get to that point as soon as possible. The LLM can't dogfood the code it writes.