Agentic Test-Driven Development

Liam Keegan, SSC

Last updated: 2026-05-26

Tentative Outline

- Classical tests intro
- Coding with LLMs intro
- Classical TDD intro
- Agentic TDD: what changes, what stays the same
- Common failure modes
- Best practices
- Hands on
  - Could be fully hands on
    - But hard to coordinate, each student will rapidly diverge
  - Could be just me coding
    - Students interacting with me as we discuss what to do next, pros/cons of different approaches

Advantages of coding with LLMs

- Can generate a lot of code very quickly
- Can get results without understanding every detail

In what scenarios is this an advantage?

- Rapid prototyping of new ideas
- Writing one-off scripts
- Anything where being wrong is not a huge deal
- Working with a language and/or domain where you are not an expert

Disadvantages of coding with LLMs

- Can generate a lot of code very quickly
- Can get results without understanding every detail

In what scenarios is this a disadvantage?

- When your project gets bigger
- When correctness and/or understanding is critical
- If you need other humans (including yourself) to review and understand the code

Levels of understanding

When you write code by hand, you mostly understand how it works.
There is of course a spectrum of understanding, maybe some parts were copy and pasted from stackoverflow, maybe the details of some parts are a bit unclear.
But by and large, if the thing works as intended, and you wrote it, you pretty much understand how it works.

When you produce code using an LLM this is no longer necessarily true.

Levels of understanding

1. I understand every line of code
2. I understand most of the code
3. I understand the tests, but not the code
4. I understand the big picture, but not the tests or the code
5. I don't understand the big picture, or the tests, or the code

Appropriate understanding

Even before LLMs, it was inefficient to be at level 1 for all the code you write.

- for a javascript animation in your website, it may be fine for you to have no idea how it works
- for the main algorithm from your thesis, you need to have a very deep understanding
- most code lies somewhere in between

How to increase your understanding

Avoid being passive! Don't just ask it to do something, but debate different possible solutions, discuss pros and cons, eventually agree on a design / implementation.

- Question anything you don't understand
- Ask it to explain why it made particular design choices
- Suggest simplifications, better approaches
- Look for existing code that can be re-used instead of writing new duplicated logic

If you're an experienced software developer used to reviewing other peoples code, this is actually pretty similar!

How to increase your understanding

What if you're not an experienced software developer?

- Question anything you don't understand
- Ask it to explain why it made particular design choices
- Ask it what alternative design choices would make sense here
- Ask it if this can be simplified
- Ask it what constraints or invariants it is assuming
- Get another LLM (or the same one with fresh context) to review it

How to increase correctness

Tests are the key to correct code. If you have a good test suite:

- Tests specify the desired behaviour of your code
- Anything not tested is not constrained

How do tests help LLMs

LLMs work best (like people) when

- they have clear goals
- they have clear ways to tell if they've reached their goals
- they can iterate and get feedback quickly

Code bloat

By default, current LLMs do a pretty great job of implementing what you ask for without breaking stuff or crashing. However, this means

- defensive programming style
  - they take into account a lot of possible invalid inputs or state
- backwards compatibility
  - they try to avoid breaking existing code by adding backwards compatibility layers and logic

In addition they have

- enterprise programming style
  - overcomplicated APIs
- only local understanding
  - re-implement existing functionality, miss non-local effects of their changes

At first glance this seems harmless, maybe even good? Yes we have a bit more code than strictly required, but it works, nothing got broken...

Recursive code bloat

The problem is that with every iteration of an LLM on the codebase, they have to take into account more and more possible weird states, and more and more backward compatible branches and forks in the logic, and as the codebase grows their local understanding of it becomes more and more of a limitation.

You need to actively fight against this bloat (which is not unique to LLMs, but it does happen much faster with them than with humans).
You can do this by asking for simplifications, finding duplications, discussing what assumptions are being made that are not needed, or what backwards compatibility is actually required, etc.

A great tool here though is to have a good test suite. Then instead of it guessing what inputs it may have to deal with, or what changes needs backwards compatibility, you can just point it to the tests. If they are reasonably complete, then any change that doesn't break tests is ok, which can allow a lot of simplification.

Failure modes

- local reasoning
  - LLMs only have a local view of the code
  - this makes them (relatively) bad at architecture / API decisions
  - making changes that are consistent with / take into account conventions/ preserve invariants from a large codebase
  - note they are bad at this relative to their ability to implement code, they can still be pretty good!
- code bloat
  - defensive coding
  - backwards compatibility
  - unnecessary complexity
  - "enterprise" coding style
- cheating
  - in particular when writing tests

Older failure modes

These failure modes used to be very common but seem to have been largely fixed in recent models:

- incorrect code
  - code that doesn't compile
  - code that contains syntax errors
- hallucinations
  - uses an invented API call that doesn't exist
  - or invents a language feature
- convincing "at a distance" code
  - looks good at first glance
  - but lots of small details are wrong

Getting what you ask for

LLMs are amazingly good at giving you what you ask for. But not always at giving you what you need.

TDD

TDD works in a tight feedback loop of three steps

- write failing test
- write code to make the test pass
- refactor

The key point is to think about the test before the implementation,
which also forces you to think about the API before the implementation,
and to work in small incremental steps.

Test suite as oracle

With LLM code, having a good test suite becomes even more valuable

- although not if the tests are just written after the implementation by the LLM!

You should iterate with the LLM on the tests, in particular

- test that check the code does what it should do with some valid input (happy path)
- also what happens with invalid inputs (unhappy path)
- end goal is a test suite that more or less specifies the code

If you understand the test suite, and it covers all the key behaviours of the code,
then you can be fairly confident in the correctness of the generated code, even without understanding the code!

Anecdote: one OpenAI developer still writes their tests by hand, the LLM then writes all the code.

Context

The LLM that wrote the code has all that code in its context.

To get a less biased opinion, clear the context and/or use a different model:

- `/review` in claude/codex does a review with fresh context window
- `/clear` then ask it e.g. to review the tests and if they cover all relevant cases
- use another model to review code and tests, and ask it questions

1960 - 2020: Humans write code

Software development has been around for many decades.

In that time, new programming languages and methods were developed, and many things changed - but the basic paradigm of humans writing code didn't really change.

2023: LLMs can write code

It turns out LLMs can (sort of) generate (sometimes working) code from plain english prompts.

Andrej Karpathy
@karpathy

The hottest new programming language is English

Jan 2023
https://x.com/karpathy/status/1617979122625712128

2025: Vibe coding

You can (more or less) code fun projects, without understanding the code, just by talking to the model.

Andrej Karpathy
@karpathy

There's a new kind of coding I call "vibe coding" [...] it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works.

Feb 2025
https://x.com/karpathy/status/1886192184808149383

2026: Agentic engineering

LLMs are now good enough at coding to be used by professional developers in serious projects.

Andrej Karpathy
@karpathy

programming via LLM agents is increasingly becoming a default workflow for professionals [...] to differentiate it from vibe coding, personally my current favorite "agentic engineering" [...]

Feb 2026
https://x.com/karpathy/status/2019137879310836075

Keeping up

If you feel like it's hard to keep up with this dramatic rate of change, you're not alone!

Andrej Karpathy
@karpathy

I've never felt this much behind as a programmer [...] I have a sense that I could be 10X more powerful if I just properly string together what has become available [...] some powerful alien tool was handed around except it comes with no manual and everyone has to figure out how to hold it and operate it [...]

Dec 2025
https://x.com/karpathy/status/2004607146781278521