



A Gopher's Guide to Vibe Coding

Daniela Petruzalek

Developer Relations Engineer @ Google Cloud

@danicat83



Introduce yourself, set the tone for the talk

In this talk we are going to explore the concept of vibe coding from the point of view of a Gopher: a Go developer. Our community is made up of people that takes great pride in their craft, especially in regards to writing idiomatic, readable, testable and maintainable code. On the other hand, we often hear stories about how people nowadays are not even reading the code anymore, thanks to the ability to delegate all the code writing to a large language model. Can those two apparently opposite things coexist together? I believe so, and in this talk I'm going to show you how.

Who am I?

Developer Relations Engineer at Google, originally from Brazil 🇧🇷, but living in the UK 🇬🇧 since 2019.

My background is in backend and data engineering. Have been doing Go development (on-and-off) since 2017.



Self-introduction

Agenda

Introduction to Vibe Coding

Types of Coding Assistants

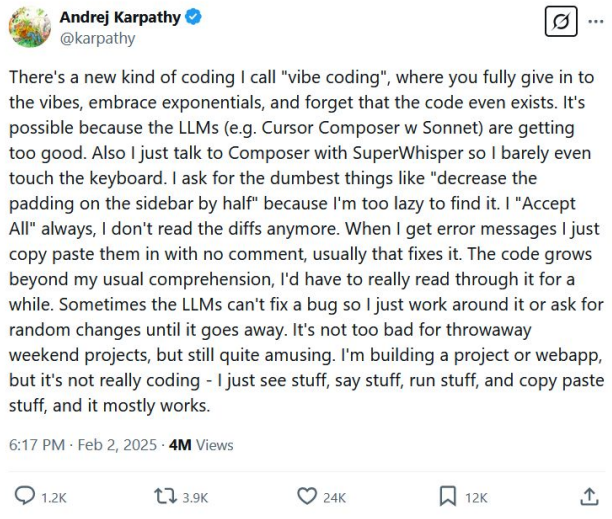
AI Assisted Developer Workflow

Live Demo: Jules & Gemini CLI


Practical Tips & A Peek at the Future

Talk flow:

- Brief intro to vibe coding to get everyone in the same page, followed by a quick assessment of the main types of coding assistants we have in the market today
- A deep dive into my developer workflow, grounded on the main problems with the current approaches and how I try to address each of them
- Finally, a few demos and tips & tricks on improving your model's responses, finishing with a few Go features and libraries to look forward to



This is the often attributed as the original definition of vibe coding, although the concept is probably much older than that. While it's an interesting though, I believe most people nowadays are adopting a different definition, a bit more mature and grounded: (next slide)



A coding approach that relies on LLMs to generate working code by providing natural language descriptions rather than manually writing it

There are of course, people who object this definition and will say that from the moment you start reviewing the code it stops being vibe coding. I am on the group that defines vibe coding as the LLM writing the majority of code for you, but you are still responsible for reviewing and adjusting it to your needs. This is what differentiates professionals from amateurs, and this the care I would expect from proper engineers to avoid incidents like the one we have seen recently with the Tea app:

<https://www.bbc.co.uk/news/articles/c7vl57n74pgo>

And I suspect many more are to come. As engineers, we need to do better. Maybe one day we will be able to stop caring about security, but that day is definitely not today.

Objections to this definition: "If an LLM wrote every line of your code, but you've reviewed, tested, and understood it all, that's not vibe coding in my book—that's using an LLM as a typing assistant."^[1]

Types of Coding Assistants



Code Completion

Traditional IDE autocomplete with "Ghost Code"

Gemini Code Assist or GitHub Copilot on VS Code



Contextual Chat

Conversational interfaces on the IDE via chat add-on

Gemini Code Assist or GitHub Copilot Chat on VS Code



Coding Agent

CLI interfaces with a REPL, context is the entire project, "pairing"

Gemini CLI, Claude Code, Aider



Autonomous

"Fire-and-forget": completely async

Google Jules, Devin AI, GitHub Copilot Agent


From code completion to fully autonomous, nowadays we have a wide range of tools at our disposal. I am focus the remaining of this talk on the last two, as the first two tend to be more traditional IDEs adapted to do AI, instead of being tools thought for the AI-native world.

Biggest Challenges Today

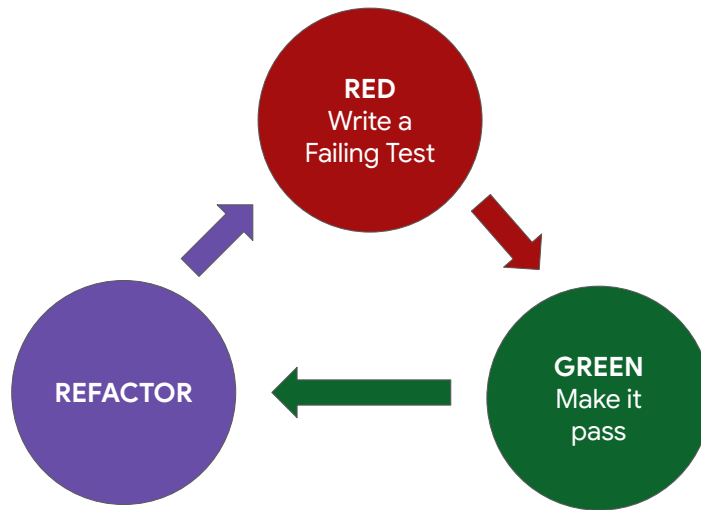
- LLMs are non-deterministic by nature
- LLM generated code is mostly “not great”
- Prototypes and full systems are different beasts
- Coding environments are not entirely fit for purpose
- Code correctness - does it do what is supposed to do?
- How to ensure code quality and maintainability over time

This ordered by degree of control, not line length, I promise!

We can't do much, as consumers of the LLM technologies, about the first two, but we can improve the bottom four with our own expertise, processes and better tooling. We are going to tackle this next.



Vibe Coding with the
“vanilla” models can be
equal parts amazing and
infuriating... Can we make
it better?



Does anyone knows what this cycle is about? Hopefully you do! This is the Test Driven Development loop.

Vibe Coding is “TDD on Steroids”

- RED: Start with a feature request (failing test)
 - Keep features small
 - Reduce ambiguity => improve the prompt
- GREEN: Make it work (make the test pass)
 - No code unrelated to the feature should be written
- Refactor (only allowed on green)
 - Code review, test coverage, linters, documentation, ...

IHMO proper vibe coding is nothing more than TDD on steroids - you establish a work loop and operate under the same constraints - no refactoring until the feature works, but no new features until the feature is properly tested and adhere to the best coding practices.

Improving Response Quality

- Give the model more up to date / relevant information
 - Prompt / Context Engineering
 - Retrieval Augmented Generation (RAG)
- Grounding with Tool Calling (e.g. Web Search, APIs)
 - Model Context Protocol (MCP)
 - Not only retrieve data, but perform actions

To address model limitations, these are the most common techniques nowadays

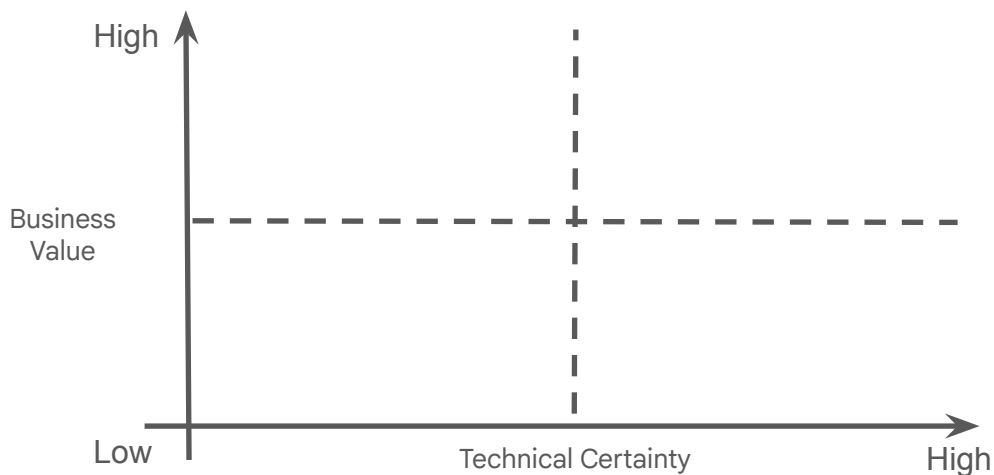
High emphasis to MCP, which is to AI models like REST is to http servers. Or as some people like to say, the USB for agents. I prefer the REST analogy because I believe that just like in the past 10+ years the industry has been heavily focused on making everything “API first”, for the next 5 to 10 years I believe we will be retrofitting every single API to be “AI-Enabled” with MCP. Ok, maybe not everything, but at least the most important ones.

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)



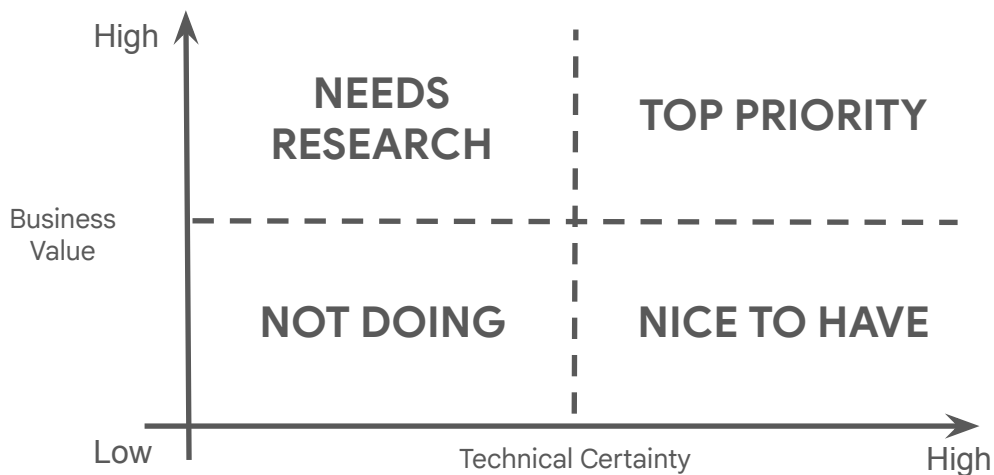
This is maybe the first time in the industry where this joke doesn't apply, as everyone quickly converged to mcp

My Current Workflow



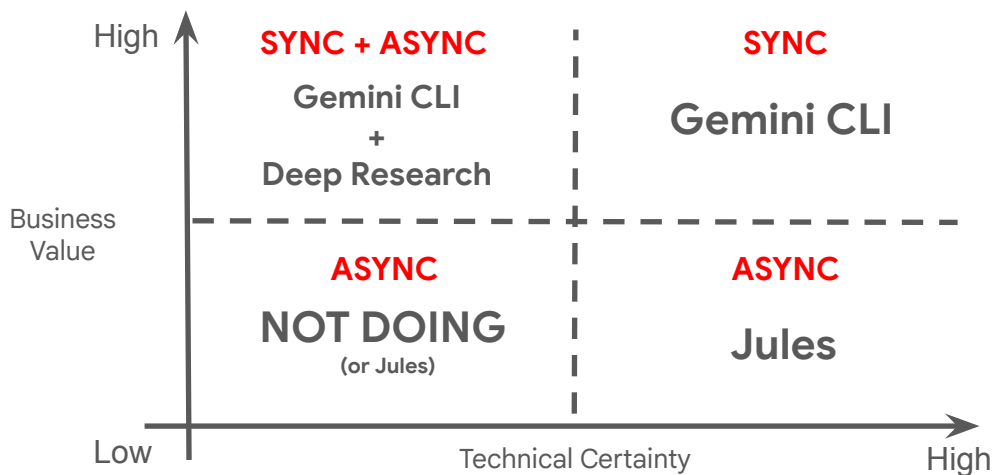
Finally, there is a matter of which of the 100's of AI tools to use. By working at Google I obviously end up using more Google tools than anything else, but still, believe it or not (lol) we have way too many options. So I adapted this quick prioritization exercise to help me to choose the best tool for each task. It works by classifying each task you need to do over those two axis - technical certainty and business value

My Current Workflow



Once you have everything on the board, you can adopt this classification... for example, high business value with high technical certainty should start immediately as you can quickly deliver to the business. On the other end, low business value and low technical certainty are those things that sound clever at first but they are probably never going to be implemented, so they are more a distraction and should be removed from your backlog.

My Current Workflow



Taking those priority groups, here is a simple mapping of my work. I will only be hands on keyboard all times on a high/high class, because that one is clearly money on the table so I want to deliver it as fast as possible with the highest possible quality, as for the others, I will use a mix of async techniques and sometimes (high business value) do a deeper analysis to reduce ambiguity and hopefully improve technical certainty (the famous spike)

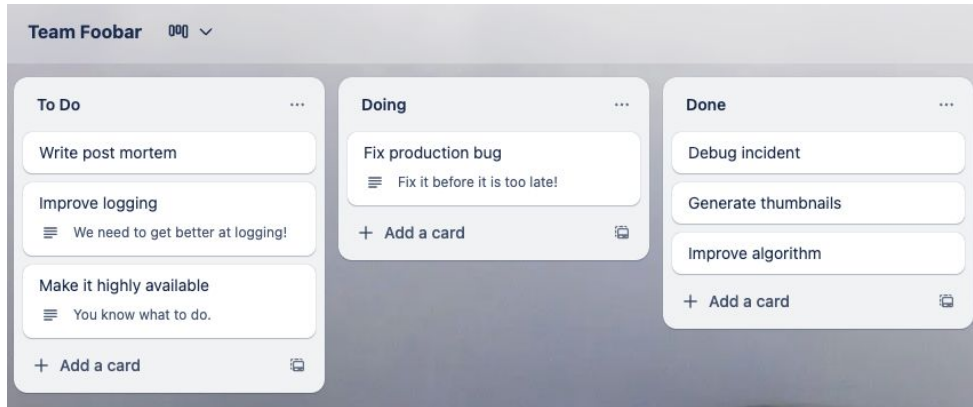
Demo: Jules

<https://jules.google/>

<https://github.com/danicat/testquery/issues/5>

I know I have been speaking a lot so let's see the first of these tools in action - here is Jules

Does your team write stories like this?



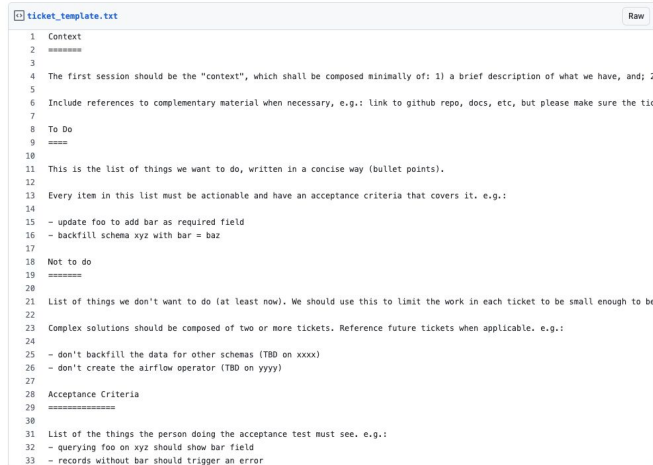
While we wait for jules to do its magic lets reflect a little bit... how does your team write stories? Does it look like this?

You open the card and the description is empty? Or simple re-estates the title in different words? I have been in teams like this for more years of my career than not. Unfortunately, doesn't matter how brilliant the developers are, they are usually very sloppy with the documentation. The problem with this is that any of these cards have a hard dependency - a hard dependency on the mind of the developer who wrote it. And if that developer is YOU, and YOU need to open the card two weeks later, it is very likely that your request for memory will be a cache miss... the ticket content is lost, for eternity!

Of course, just like all of you I also hate the whole "As a someone, I need to blah and blah, WHEN THEN etc etc"... the BDD language kills my motivation to live (and code). And don't get me started on ginkgo tests please :P

This is why in every team I join I advocate for something like this...

Good Prompts = Good Stories

A screenshot of a text editor window titled 'ticket_template.txt'. The editor shows a template for a ticket with line numbers 1 through 33. The content includes sections for 'Context', 'To Do', 'Not to do', and 'Acceptance Criteria'.

```
1 Context
2 =====
3
4 The first session should be the "context", which shall be composed minimally of: 1) a brief description of what we have, and; 2
5
6 Include references to complementary material when necessary, e.g.: link to github repo, docs, etc, but please make sure the tic
7
8 To Do
9 ====
10
11 This is the list of things we want to do, written in a concise way (bullet points).
12
13 Every item in this list must be actionable and have an acceptance criteria that covers it. e.g.:
14
15 - update foo to add bar as required field
16 - backfill schema xyz with bar = baz
17
18 Not to do
19 =====
20
21 List of things we don't want to do (at least now). We should use this to limit the work in each ticket to be small enough to be
22
23 Complex solutions should be composed of two or more tickets. Reference future tickets when applicable. e.g.:
24
25 - don't backfill the data for other schemas (TBD on xxxx)
26 - don't create the airflow operator (TBD on yyyy)
27
28 Acceptance Criteria
29 =====
30
31 List of the things the person doing the acceptance test must see. e.g.:
32 - querying foo on xyz should show bar field
33 - records without bar should trigger an error
```

<https://gist.github.com/danicat/854de24dd88d57c34281df7a9cc1b215>

This is a ticket template that I wrote some good 7 years ago. It adds some context to the task, explains what to do and what not to do, then finishes with a very simple acceptance criteria list. Sometimes I even simplify it further by removing the headings, but the essence is the same: a description of the task, some references and how do I know the task is finished. This is brilliant for one single reason: now anyone who opens this ticket can do not, not just past you (because current you also can't)

A Good Prompt Template

1. Task: description of what you want to achieve, including context, constraints, etc. For example: use SDK x, library y.
2. References: links to documentation, reference implementations, etc.
3. Testing: how to test the successful implementation; example commands with output.

And guess what? Good prompting is very similar to good story writing!

Let's try something practical!

(what could go wrong?)

(again?)



The Prompt

Your task is to create a Model Context Protocol (MCP) server to expose a "hello world" tool. For the MCP implementation, you should use the official Go SDK for MCP and use the stdio transport.

Read these references to gather information about the technology and project structure before writing any code:

- <https://github.com/modelcontextprotocol/go-sdk/blob/main/README.md>
- <https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle>
- <https://go.dev/doc/modules/layout>

To test the server, use shell commands like these:

```
(
  echo
  '{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"protocolVersion":"2025-06-18"}}'
;
  echo '{"jsonrpc":"2.0","method":"notifications/initialized","params":{}}';
  echo '{"jsonrpc":"2.0","id":2,"method":"tools/list","params":{}}';
) | ./bin/godoctor
```

Here is an example of a prompt with all these elements

Typical Error Modes



Fully Recoverable

Typos, compilation errors (e.g. syntax), edit errors (e.g. duplicate lines)

Model usually recovers on its own given enough time



Intervention

Wrong reasoning (e.g. tests vs code), placeholder code, hallucinations

Model **might** recover, but you will be better off giving it a nudge



Full Stop!

Refactor loops; insisting on not following directions

Model **will not** recover and potentially spend lots of \$\$\$\$. Take over and course correct

While the CLI is working, I just want to highlight that problems WILL happen. If you are like me when you see a work in progress you are probably afraid to interrupt it for the fear of leaving it in a dirty state. But in case of “pairing” with LLMs there are scenarios that you **MUST** interrupt or you are going to: 1) waste lot of time, 2) waste lot of money and 3) acquire unhealthy levels of frustration

Tips for a Successful Experience

- **Don't be afraid to interrupt the AI.** The AI will sometimes propose actions or code that you don't agree with. Cancel and course correct.
- **Encourage tool use.** If the AI seems lost or is making up information, encourage it to use its available tools to do web searches, look for examples, inspect files, etc.
- **Have you tried turning it off and on again?** In the (not so) rare case that the AI is misbehaving, clearing the context and giving it a clean start with a better prompt is often the best option.

A Peek at the Future of Go

Go SDK for MCP: <https://github.com/modelcontextprotocol/go-sdk>

MCP Support for gopls: <https://tip.golang.org/gopls/features/mcp>

Google Gen AI SDK: <https://github.com/googleapis/go-genai>

OpenAI Go API Library: <https://github.com/openai/openai-go>

Gemini OpenAI Compatibility:
<https://ai.google.dev/gemini-api/docs/openai>

Here are some highlights we can expect over the next few months from the Go community, the first two being two things the Go team is actively working on

(give a tour of gopls)

The Go SDK for MCP is a partnership between Anthropic and Google and is the sdk I've just used on my example.

Useful MCP Servers

Playwright: navigates web pages, takes screenshots, ...

<https://github.com/microsoft/playwright-mcp>

Context7: retrieves documentation from a crowdsourced repo

<https://context7.com/>

GoDoctor: Go coding assistant

<https://github.com/danicat/godoctor>

Speedgrapher: prompts and tools for technical writing

<https://github.com/danicat/speedgrapher>

Don't forget to go back to the demo / prompt you left running two slides ago!

What About Building Your Own?

[How to Build a Coding Assistant
with Gemini CLI, MCP and Go |
Google Codelabs](#)



Here is the link to a code lab where you can build your own mcp server from scratch and play with all these concepts.

In Summary

For successful vibe-coding sessions:

- Develop your prompting (and story writing) skills
- Prioritise your tasks and use asynchronous coding for the less critical ones
- Actively manage your context window
- Equip your toolbox with MCP servers: find what is useful or build your own (it is easy and fun to do)

Thank you!

<https://danicat.dev>

[linkedin.com/in/petruzalek](https://www.linkedin.com/in/petruzalek)

@danicat83

Google Cloud

