DHBW Ravensburg

Disciplined Vibe Coding

The unreasonable effectiveness of language

Prof. Dr.-Ing. Mark Schutera | July 8, 2026 · *orange disciplined dino*

[1, 3, 6]

1

Alignment

On Humansplaining and LLM Understanding

Pareto hits the vibe. Hard.

[1,3,6]

Naive vibe coding is brittle, then fails. Software engineering discipline (and a better model) fixes each mode.

Misalignment. you and the agent mean different things by the same word.

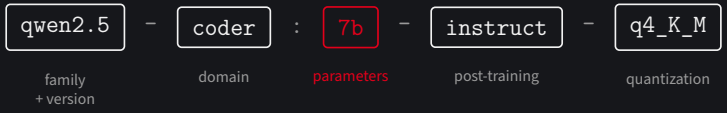
Verbosity. you ask for one thing and get 400 lines, while other functions silently break.

Non-functional. the high-score save looks wired, but nothing persists.

Decay and Debt. each build-loop makes project and module complexity harder to manage.

[1]

Reading a model card



Two things the name never carries. The **context window** (`num_ctx`) is the token budget the model attends to at once, paid for in **GPU memory**. And whether the model **reasons** on a scratchpad before it answers.

Feel free to switch to more capable models now.

qwen2.5 base model and release.

coder fine-tuned for a domain, here code.

7b parameters: seven billion weights in memory (15GB) while the model runs.

instruct follows instructions and chats; a base model only continues text.

q4_K_M quantization: about 4 bits per weight, shrinking memory to 4.4 GB.

An agent is a prompt, its tools, and a model

Plan and **build** are the two built-in **agents**. Each bundles a system prompt, a set of allowed tools, and a model. You define your own the same way: one markdown file in `.opencode/agent/`.

A **subagent** is an agent a primary one spawns to handle a single self-contained task in its **own fresh context window**, then hands back a short summary. The side quest never clutters your main session.

```
> @007 shake it, do not stirr.
```

Switch primary agents with `/agent` or `Tab`; call a subagent by name with `@`.

Delegation is a tool call. Enter agent orchestration.

Models and agents

Run it	What it does
`/models` `ctrl+x m`	Switches the model the agent runs on, from your configured providers.
`f2`	**Recent model:** toggles back to the model you used just before, for a quick side-by-side on the same task.
`/agent` `ctrl+x a`	Switches the active agent, build versus plan, each with its own tools and system prompt.
`tab`	**Cycle agent:** steps to the next agent without opening the list.
`right`	**Cycle subagents:** moves between the subagents a session has spawned, to read what each is doing.

Plan and Build mode in opencode

Plan mode reads and reasons; Build mode writes and runs. Like gating, the split mitigates surprises. Toggle with `Tab`.

Plan

Read only. opencode explores, explains, and proposes; it never edits files or runs commands.

Use it to scope the work and agree on the approach before any change lands.

Build

Full access. opencode edits files and runs tools to carry out a plan. Here small models default into action.

Use it once the plan is clear to both parties. You can build across several sessions.

Agentspanning-Capabilities or Skills

A **skill** is a markdown file of instructions and context. Invoke it by name; the agent opens that file and follows it, turn by turn. That is the whole mechanism.

Those instructions do more than steer the model. A skill can call an API, run a script, or hand a slice of work to a subagent, so one named file becomes the interface to agentic workflow automation.

Invoke with /skills, then pick one.

Compact, token-frugal skills win: the whole file is read into context every time the skill runs.

write-a-skill

Every skill so far taught the agent one job. `write-a-skill` teaches it to author the next one in the same shape, so the toolkit grows by its own rules. Name one job, list **3 to 5 bold-lead steps**, close on a single principle, all short enough to read in a breath. If the job needs an “and”, split it, and wrap it in another skill. **skillception**.

```
> write-a-skill for "writing skills"
Or wait, I will just nano this myself.
```

Writing a skill, is a skill. Of course it is easy to forget, that we can also still do stuff ourselves - manually.

The plan toolkit, read only

Align

`askuserquestion`

Reach a design you both hold before a line is written.

Model

`domain-modeling`

Fix one word to one meaning, then name the pieces.

Scope + slice

`to-prd`
`to-issues`

Write the brief, then split it into small issues.

Spike

`prototype`

Throw one away to settle a single design question.

askuserquestion

The arc starts with **askuserquestion**. It does not write a plan; it reaches one. The skill interviews you one question at a time, proposes its own answer to each and says why, and walks every branch of the decision tree, until it has no questions left.

```
> run askuserquestion on feature_brief.md
Q: do power-ups stack? (I'd say no)
> no
Q: does score reset on death?…
no open questions. aligned.
```

One question at a time, and it stops only when alignment is reached, not before. In general following proposals is adviced, until it is not.

And just because a model has no questions left, does not mean everything is perfectly clarified.

domain-modeling

`askuserquestion` settled the vocabulary. You have no `GLOSSARY.md` yet; `domain-modeling` rereads the chat and code and writes one line per term. Now code and tests speak the same words every session. Naming the thing is the alignment.

```
> domain-modeling
```

GLOSSARY.md (after the run)

```
obstacle: box to clear
hitbox: what collides()
duck: lowers the hitbox
ramp: speeds up w/ score
```

Remember how the substrate of these models is similarity within an embedded feature space. So whenever possible go with their wording to increase performance and save context tokens.

[1,3,6]

[2]

`to-prd`, the product requirements document

Start with letting the agent write this document based on the survey, then step in and adjust, extend, and clean up.

```
Acceptance lines
- the high score survives a page reload
- the score ticks up each step
Out of scope
- no multiplayer, no leaderboard
- no power-ups this pass
```

to-prd fixes verbosity: the aligned chat becomes a short PRD in `issues/PRD.md`. Two parts carry the weight, the out-of-scope list and one acceptance line per slice, a checkable sentence like “the high score survives a page reload”.

to-issues

`to-issues` splits `issues/PRD.md` into independently **grabbable** slices, one markdown file each. Every issue carries its acceptance line as its “how tested”.

```
issues/01-banana-on-screen.md
 tested: the banana runs across the screen
issues/02-persist-high-score.md
 tested: the high score survives a reload
 blocked-by: 01
```

Reject a horizontal first slice and re-slice until the first one shows a number. Check it actually writes the slice.mds into `issues/`.

Spike by `prototype`

What if you can not answer a design question? `prototype` answers it with a **throwaway spike**. The answer is the deliverable; the scratch code is deleted.

Will this library do the job, is the API fast enough, does the real data fit the schema? A 20-line script answers all three. Record “library works, 120 ms median, dates are strings”, delete the script.

The delete step is mandatory. A spike you keep and build on rots into decay: it was never designed to last. It was a standalone accepted cost.

Vibe an infinitely running banana

Same endless runner, now under this discipline. Run the whole planning chain on `feature_brief.md`, back to back, and write no code.

- `askuserquestion` until it stops asking.
- `domain-modeling` into `GLOSSARY.md`.
- `to-prd`: Set up a product requirements document.
- `to-issues`: A first vertical slice that serves as a silver bullet.

The deliverable is a **structured executable plan**. If the agent starts writing code, stop it.

The banana runner, planned this time.