DHBW Ravensburg

Disciplined Vibe Coding

The unreasonable effectiveness of language

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

Objectives

By the end of this lecture series you will have learned,

- The algorithmic foundations and technical intuition of **large language models** (LLMs).
- Put LLMs to use as an **engine for vibe coding**.
- How to set up a sophisticated **structure for disciplined** vibe coding.

The rhythm will be theoretical primers followed by hands-on vibe coding.

Preparing the Workbench

We teach on small local models by design. The hands on sections run a **small model** served by OLLAMA through the `opencode` terminal client. One bootstrap script installs the whole stack and pulls qwen3:1.7b.

There is still value in reading docs as a human:

- opencode.ai/docs
- docs.ollama.com

To **Set Up** clone the repo and run its bootstrap script. The README walks you through macOS, Linux, and Windows.

github.com/Quillstacks/disciplinedvibe

Before we begin, three questions.

1

Who has access to an LLM and uses it regularly?

2

Who knows how these systems work?

3

Who knows how to program in the classical sense?

[1, 3, 6]

1

Foundations

History, Theory and Mechanics

The hottest new programming language is English

[1,3,6]

Software 1.0

Humans write explicit instructions in code, and the machine executes them.

Software 2.0

Gradient descent writes the weights of neural networks from data.

Software 3.0

The program is a natural-language prompt, and the LLM compiles it to behaviour.

LLMs are a **new computing substrate**, with prompts as programs and context windows as RAM.

Raise the floor, keep the bar

Raise the floor. Everyone can now build software they could not build before. Fast.

> State your intent, accept the diffs, run the result, embrace the exponentials and iterate without necessarily touching or reading every line. That working style is **vibe coding**.

Keep the bar. AI engineering means you adjust your workbench to this abstraction layer.

> Most professional work requires rigor. As you will still own the code, the quality, and are accountable for whatever you ship. That working style is what we term **discipline**.

Making the machine understand words.

Word-level vocabulary

`banana` → `[24 907]`

One token, one entry, one id. A larger vocabulary yields longer subwords, representing complex semantics and patterns directly.

Subword vocabulary

`ba` `na` `na` → `[3021, 512, 512]`

Three tokens from two entries, `ba` and `na`; the repeated `na` reuses id `512`. Fewer merges keep the vocabulary small and character-based, which improves generalization.

In the beginning was the token: a unit of text, a word or subword. It is the atomic unit a model reads and writes; a vocabulary is the fixed set of tokens it knows. Tokenizing is how text is sliced into entries of that set.

LLMs use vocabularies of 30,000+ tokens.

[1, 3, 6]

[1–4]

Making the machine understand language

LLMs learn the structure and semantics of language by predicting the next token,

$$\hat{y}_{t+1} = \theta(x_{1:t}).$$

The free lunch, an implicit ground truth inside every token sequence, enables unsuperised learning of θ on unlabeled data,

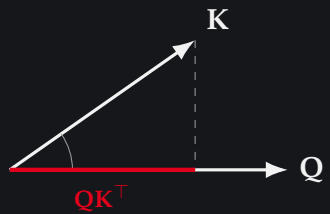
$$\mathcal{D} = \{(x_t, \tilde{y}_t)\}_{t=1}^{N},$$
$$x_{t+1} = \tilde{y}_t.$$

An LLM: the stochastic parrot that predicts the next token.

Attention is all you need

[1, 3, 6]

Attention compares token vectors: the closer a query and a key point in the same direction, the more that token attends to the other, which reflects **semantic similarity**.



The red projection is the dot product $\mathbf{QK}^\top$.

Q, query: what the token is looking for.

K, key: what a token offers to be matched against.

V, value: the information a token passes on once matched.

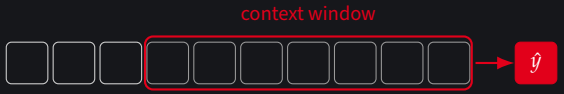
Notice how these computations can be done in parallel, and accelerated by GPUs.

[6]

Context is all you have

[1, 3, 6]

Each token turns into a query and a key, and attention scores every one against every other, a grid of $\mathbf{QK}^\top$ interactions. This runs token by token over one flat sequence, the **context**.



Context is limited - think Goldfish. Its size is capped by GPU memory.

Every token in the window competes for attention:

- your prompt,
- system prompts,
- the files you shared,
- tool and search results,
- the conversation so far,
- and the model's own replies.

Irrelevant tokens still enter the $\mathbf{QK}^\top$ scores and crowd out the signal; doubling the tokens quadruples the compute.

Still a parrot: prompting

[1, 3, 6]

Prompt injection

Steer the model by engineering the input prompt, **injecting instructions** or context that guide and nudge it toward a specific task without retraining.

"The product is awesome is a good review.
The product breaks easily."
→ [LLM generates the response]

This is zero-shot, the task lives entirely in the tokens you place in the context, so a good prompt is a good program.

[10]

Still a parrot: Instruction tuning

[1, 3, 6]

Instruction Behavior Training

Fine-tune the model to follow an instruction. At this point the *free lunch* is over. An instruct dataset pairs each instruction with a label: Self-Instruct, InstructGPT, FLAN.

Instruction: "Is this review positive? The product breaks easily."
→ Response: "No, it is negative."

Prompting bends behaviour at inference; fine-tuning writes it into the weights. Prompt-to-tune is a recurrent pattern in LLM development.

[11–13]

Vibe a poem on bananas

If not done yet, **set up** your local LLM: clone the repo and run its bootstrap script.

github.com/Quillstacks/disciplinedvibe

- Get comfortable with Open Code
- Write at least one poem, what works well, what does not?
- Share your best poem in the chat.

Not very useful outside the classroom, but it hints at the **semantic capability** of LLMs and their speed, and at the applications they open up.

The muse for your four-line banana poem.

[1, 3, 6]

2

Vibing

When Language becomes Action

Designing Interfaces by Injection

The model only reads and writes tokens. A tool still needs an interface, so we make one the same way as before: by injecting its description into the prompt, then reading the call back out of the reply.

- The **system prompt** specifies the tool, its arguments, and the format for calling it.
- The model writes that call as ordinary text; a **parser** spots it and runs the real tool.
- The result comes back as tokens and get injected as observations, and the model continues.

System prompt

```
tool weather(city)
call <tool>...</tool>
```

LLM Tool use

```
<tool>
weather(Ravensburg)
</tool>
```

MCP and plug in outside tools

The **Model Context Protocol** is a standard way to connect the agent to outside tools and data: a database, a web service, a file store, a program, each exposed through a small server.

MCP is a protocol. The agent host connects, discovers them at runtime, and calls them. Write to the standard once, and any MCP client can use it.

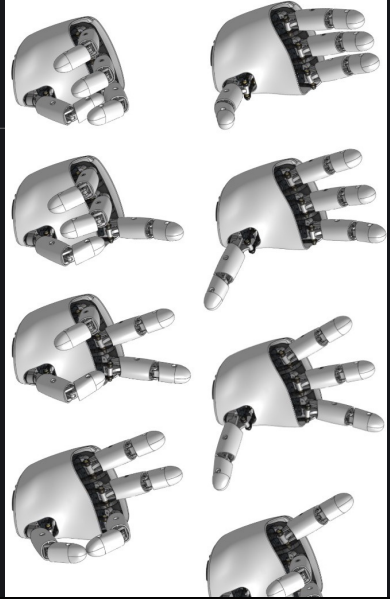
This is the structured answer to agent-ready APIs. The engineering way with proper auth and hardened.

The one Acting

Agentic, from Latin *agere*, “to act,” the root of *agent*, one who acts.

- Emits its action in a fixed structured format using tags.
- Stops at `</action> or </tool>` once emitted.
- A parser then runs the action.

An action can be many things. Communication and subagent orchestration. Information gathering through search, queries, and retrieval. It uses tools, from API calls to code execution. Or it acts on the environment, driving digital or embodied interfaces.



The Harness

The machine got an interface, now it's time for the human to get one to take control.

> The **harness** is everything wrapped around the raw model to make it reliable: the client, the system prompt, the skills, the tests, the permission gates, the loop that runs them.

`opencode` is your primary harness interface: type `/` to open the command menu and pick by name.

It has a set of **built-in tools** which can be called directly: `bash`, `read`, `write`, `edit`, `glob`, `grep`, `webfetch`.

Getting started

Run it	What it does
`/init`	Scans the project and writes an `AGENTS.md` of your stack, scripts, and conventions, so every later session starts oriented and in **context**. Run once per repo.
`/help`	Opens the in-app help: the command list, the current keybinds, and where the docs live. Your first stop when a key surprises you.
`/exit` `ctrl+x q`	Ends the session and drops you back to the shell. `ctrl+c` on an empty prompt quits too.

Steer by Context

Run it	What it does
`/compact` `ctrl+x c`	Summarises the conversation into a short brief and continues from it, reclaiming context-window space without losing the thread. Don't wait too long.
`/new` `ctrl+x n`	Starts a fresh session with an empty context. Your project stays; the conversation so far is dropped. Reach for it when the topic changes.
`/sessions` `ctrl+x l`	Lists your earlier sessions and jumps to the one you pick.
`ctrl+r`	**Rename** the current session to a memorable title, so you find it again in the list.
`esc`	**Interrupt** the agent mid-run the moment it heads the wrong way, instead of waiting for it to finish.

Gating and sandboxing

Gating keeps the human on the loop, the sanbox stops agent escapes. Latest when you auto, you want a sandbox.

Gating

opencode asks before it edits or runs: `allow`, `ask`, `deny`, set per tool in `opencode.json`. Ships in the box.

Stops surprises. It cannot stop a command you already approved.

Sandboxing

A container or VM. A project mounted, network restricted, `opencode` running inside it. You build it yourself.

Stops escapes. If something goes wrong it wrecks the box, not your machine.

Reasoning or the Inner Monologue

[1, 3, 6]

Prompted at inference time

reasoning coaxed out by the prompt, no weights changed.

Chain-of-thought prompting appends a cue like *"Let's think step by step,"* steering decoding toward a plan rather than a snap answer, so the model breaks the task into sub-tasks.

We can interleave those reasoning steps with actions and observations so thinking and tool calls advance together, called ReAct.

Trained into the weights

reasoning trained not nudged.

Models like DeepSeek-R1 and OpenAI's o1 are trained with **human feedback reinforcement learning** to reason before they answer.

> R1 emits a thinking section between the tokens `<think>` and `</think>` before its answer.

[17–19]

Interface and input

Run it	What it does
`/editor` `ctrl+x e`	Opens your `$EDITOR` to compose a long message with full editing, then sends it back to the prompt.
`ctrl+v`	**Paste** clipboard content into the prompt.
`ctrl+c`	**Clear** the prompt box in one stroke.
`/details`	Toggles a tool call's full input and output, to inspect what ran in detail.
`/thinking`	Toggles the model's reasoning `<think>` blocks, if the model has this capability.

Vibe an infinitely running banana

Now vibe an endless runner in a single `index.html`. Then ask the LLM to run it in your browser and play, then iterate.

- How easy is it to get one specific thing done?
- Watch your context and pay attention to the impact of a saturated context.
- Reflect on how productive this feels versus how productive you are.

Feel the floor rising already? Lines of code have never been cheaper.

Screenshot of a banana again.

[1, 3, 6]

3

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.

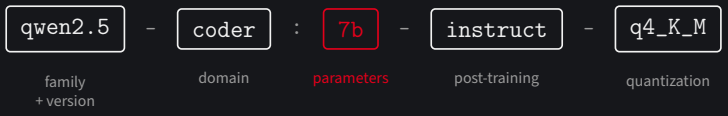
Verbosity. you ask for one thing and get 400 lines, while other functions silently break.

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.

[20]

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]

[21]

`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.

[1, 3, 6]

4

Building

aber mit höchster Disziplin.

And so we build

That was the **day shift**: plan, align, and slice by hand, human-in-the-loop. You now hold a backlog of trustworthy slices molded into issues.

You just finished a product owner workflow, ready to hand-off to the Devs. Remember the silverbullet.

test-driven-development

Each acceptance line in `issues/` becomes a test that fails before it passes. The skill drives the red, green, refactor loop one small change at a time, running `node --test` after each. **Write the failing test before the code that satisfies it**, so each feature earns a real acceptance.

```
> test-driven-development on
01-banana-on-screen
RED: banana not shown, test fails
GREEN: banana renders and runs, test passes
```

Fix the code, (never) the test, and watch the runner execute. This concept can obviously be extended to integration and end-to-end tests as well.

[1,3,6]

[22]

diagnosing-bugs

[1,3,6]

Bugs will show up. Fixing it is mandatory, but then hardening your codebase, by fortifying the fix with tests is best.

- Reproduce the bug with a test.
- Fix and document the bug.
- Confirm the fix by rerunning your test.
- **Add test to suite.**

```
> diagnosing-bugs on
‘‘some error message’’
```

[23]

Vibe an infinitely running banana

Same banana, now under the discipline of tests. Take one acceptance line from the plan and run the whole build chain on it, back to back.

- `test-driven-development`. Write a test, and a feature against it.
- `diagnosing-bugs`: Encounter a bug, fix it, harden against it.

The deliverable is a **tested banana core** you trust without reading every line. And what remains is taste.

The banana runner, now building.

References

1 Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *arXiv preprint arXiv:1301.3781* (2013).

2 Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation”. In: *EMNLP* (2014).

3 Joseph Weizenbaum. “ELIZA—a computer program for the study of natural language communication between man and machine”. In: *Communications of the ACM* 9.1 (1966), pp. 36–45.

4 Daniel Jurafsky and James H. Martin. “Speech and Language Processing”. In: (2023).

5 Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: *OpenAI* (2019).

6 Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

References

7 Colin Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *arXiv preprint arXiv:1910.10683* (2019).

8 Hugo Touvron et al. “LLaMA: Open and Efficient Foundation Language Models”. In: *arXiv preprint arXiv:2302.13971* (2023).

9 Ashish Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems* 30 (2017), pp. 5998–6008.

10 Tom B. Brown, Benjamin Mann, Nick Ryder, et al. “Language Models are Few-Shot Learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

11 Xinyi Wang, Jason Wei, Yuxin Wang, et al. “Self-Instruct: Aligning Language Model with Self Generated Instructions”. In: *arXiv preprint arXiv:2212.10560* (2022).

12 Long Ouyang, Jeffrey Wu, Xu Jiang, et al. “Training language models to follow instructions with human feedback”. In: *arXiv preprint arXiv:2203.02155* (2022).

References

13 Hyung Won Chung, Le Hou, Shayne Longpre, et al. “Scaling Instruction-Finetuned Language Models”. In: *arXiv preprint arXiv:2210.11416* (2022).

14 NVIDIA et al. “GR00T N1: An Open Foundation Model for Generalist Humanoid Robots”. In: *arXiv preprint arXiv:2503.14734* (2025).

15 Mustafa Shukor et al. “SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics”. In: *arXiv preprint arXiv:2506.01844* (2025).

16 Tejas Kumar. *Harnesses in AI: A Deep Dive*. 2026.

17 Jason Wei et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *arXiv preprint arXiv:2201.11903* (2022).

18 Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *arXiv preprint arXiv:2210.03629* (2022).

References

19 DeepSeek-AI. “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning”. In: *arXiv preprint arXiv:2501.12948* (2025).

20 Addy Osmani. *The 70% problem: Hard truths about AI-assisted coding*. 2024.

21 Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

22 Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

23 Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

What you learned

[1, 3, 6]

Foundations

You understand the mechanics of LLMs and have a technical intuition about them.

Vibe

You are able to run an LLM as an engine for vibe coding, and know how to steer it.

Discipline

You have the means to make your vibing dependable and aligned.

Give in to the vibes, keep the discipline

Prof. Dr.-Ing. Mark Schutera | DHBW Ravensburg