

Joy of Vibe Coding

minimising the emotional rollercoaster

Ekaterina Yakunenko
https://delay-rag.ru/

Translation from Russian
Original version published in December 2025

Hi!

I don't know what to call this document — calling it a manual, methodology, or handbook would be too bold. It summarizes my nearly six-month experience of so-called «vibe coding» and lays out the method I came to follow myself, after making plenty of mistakes.

I'm sure you can find other recommendations elsewhere that suit you better. And that's wonderful! I sincerely believe there are no — and cannot be — strict rules in this field, and if someone criticizes the methods that feel familiar and comfortable to you, they're wrong.

I practice what could be called AI-assisted coding (copying the code suggested by an LLM into an IDE and running it in the terminal) without knowing programming languages, and I can't precisely assess how relevant my experience is to other coding methods (e.g., with agents). But I really hope this material proves useful and helps brighten your vibe coding experiments!

UPD from the future: *as of May 2026 and practicing vibe coding mostly with Claude Code CLI I still find most of the advice extremely useful and not rotten at all!*

About the author

My name is Ekaterina Yakunenko. I have worked as a lawyer for more than 6 years, and in June 2025 I tried coding with neural networks for the first time, building a scraper to collect rulings of the Russian Federal Antimonopoly Service (FAS) from its open database.
This scraper became the foundation for my projects:

- ✦ Telegram bot @lastminute_legal_bot which checks ad creatives for compliance with the Russian Federal Law On Advertising, using a knowledge base of FAS rulings via RAG technology;
- ✦ a RAG pipeline for the bot;
- ✦ an open dataset with annotated FAS practice on advertising cases, which you can download here or here.

I write about my projects in my Telegram channel and invite you to subscribe and share your experience of legal vibe coding in the comments!

UPD from the future: *the scope of my activities and projects expanded considerably since that time, and you can find everything about them on my web-site https://delay-rag.ru/*
It's all in Russian, but that should not be a problem for advanced AI users!

This material can (and should!) be distributed under the CC BY-NC-ND 4.0 license.

The Two Pillars of Comfortable Vibe Coding

Understanding the inherent properties of LLMs
context window limitations
quality prompting

The product life cycle
the sequence of actions during development



A Few Basic Reminders

On the context window

The context window is the maximum number of tokens the LLM can hold in attention at once. The LLM can only work with the part of your conversation and code that actually fits within the *effective* context window at any given moment.

As your chat with the model grows longer, new messages start pushing earlier inputs out of memory. This is true for any interaction with LLM, but for code writing it can show up as the model forgetting the original architecture, variable names, or the technology stack you chose.

For coding, this is critical: program code requires strict coherence — if the model forgets a function definition at the start of a file, it can't use it correctly at the end. To compensate for memory gaps, the model starts to hallucinate and invents methods inapplicable to the task, breaks the logic of working code, or proposes superficial, lazy solutions.

And since a vibe-coder doesn't know programming-language syntax and doesn't see where the LLM is offering illogical solutions, errors only surface during testing. And where will the vibe-coder go to fix them? Right back to the same chat with the overloaded context window. This kicks off an endless and exhausting debug* loop, which makes it easy to grow disillusioned with the whole idea of coding something yourself.

On prompting

Everyone knows a prompt should be clearly worded, with minimal grammatical errors, defining the role, the task, and the context. In vibe coding, programming in a sense begins with a quality prompt — but in natural language. Your prompt is directly converted into the program's functions and how well they work.

An LLM can't read minds, so abstract requests give birth to abstract code. When I was coding my bot, I wrote a very long and detailed prompt about my vision of the functionality, but I missed a lot. For example, I asked it to "log all user actions," but the model interpreted that abstract request in its own way and logged completely insignificant things (and didn't write any code for the significant ones). Setting up the logging system turned out to be a non-trivial, bulky task that I struggled with for a long time.

An effective prompt should help the LLM translate legal logic into the language of system architecture. You need to describe the logic and data-processing algorithms in detail, write strict *if/then* scenarios, and specify the desired form of the result in advance.

The more precisely you describe the desired functionality and structure of the output, the lower the chance the model will fill in the nuances for you.

**Debug — fixing errors (bugs) in code.*

How NOT to talk to an LLM

"Write code for my law firm's website using this reference: https://www.pwc.com/"

- ✦ That's like asking to build a skyscraper from a photo of the facade. The model can't "scan" the entire internal structure of a complex site; without a detailed spec you'll get an empty shell that doesn't address your actual needs.

"Make a VAT calculator"

- ✦ Too abstract for a precise tool. Without specifying jurisdiction, the exact calculation rules, conditions for applying different rates, and the calculator's use cases, you get code that calculates a "spherical tax in a vacuum."

"I want a more solid design"

- ✦ "Solid," "stylish," and similar adjectives describe a subjective feeling, not a technical parameter. The model doesn't understand emotions; you need to translate them into design language: "use a dark blue palette, serif fonts, and remove the animation."

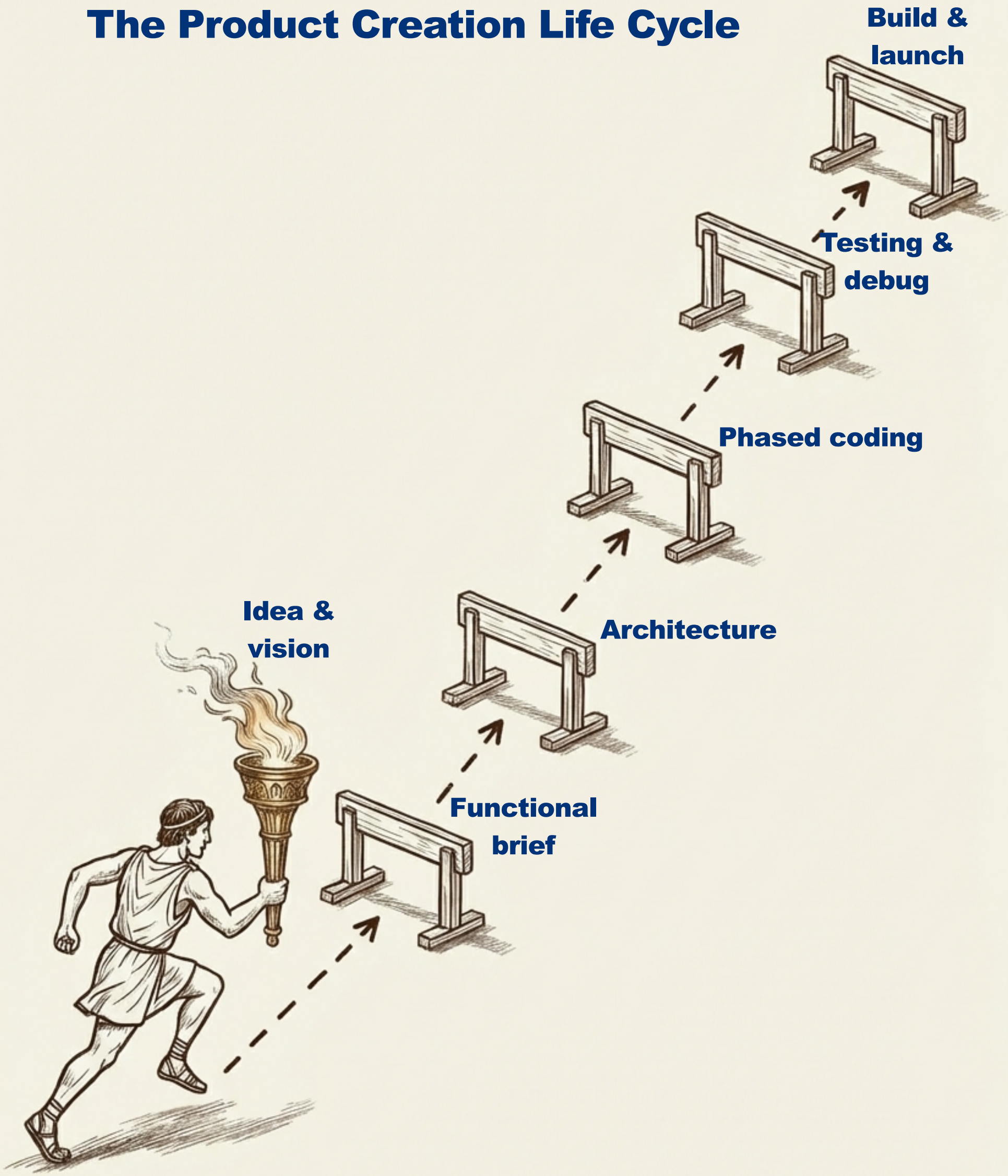
"I'm getting this error, fix it" (screenshot without error text)

- ✦ Without the error text (traceback) and the piece of code that causes it, the model will heal blindly, which most likely produces new bugs.

"Here are some documents, make a RAG out of them"

- ✦ RAG (search over a knowledge base) is a complex architecture, not a "magic" button. The prompt skips the stages of choosing a database, the text-chunking method, and search configuration, so the result will either be a hallucination or a non-working script.

The Product Creation Life Cycle



General notes

In my experience, building a product in alignment with life-cycle stages turned out to be the most effective way to fight context-window limitations. Working in stages also gives you a clear structure for prompting.

In the race from idea to product launch you have to leap several barrier-stages. Completing each stage gives you a key artifact that you can confidently bring into a new chat or session with a clean context window.

Let me note right away that this isn't necessary for simple programs and one-off scripts: you'll feel in practice the connection between the complexity and multi-layered nature of a task and the decline in context-window effectiveness that applies to your project. With modern "flagship" models, in most cases 2–5 moderately simple prompts are enough to get the desired "quick" result.

Structuring your vibe coding practice by life-cycle stages is useful when your project will be used by someone other than you, the user journey branches, you need a special interface, etc.

It's also fine to apply all life-cycle stages but fit them into a single chat or session (if, for example, you use Gemini via aistudio.google.com or Claude Code with 1 million token context window).

Below I'll offer my approaches to each stage. Note that I don't give specific prompts; rather, I outline the range of tasks you assign to the LLM at each stage (they may be sequential within one chat or not all applicable).

➡ indicates actions recommended to the reader

✦ indicates an example prompt (task) for the LLM

Stage 1. From your vision to a brief

Goal of the stage	Turn "I want something like…" into dry, structured text — a functional brief — and "tighten up" the idea. The model here acts as a business analyst or product manager.*
Prompting	➡ describe your idea, the logic of its implementation, and provide references in detail ✦ don't write code — analyze the idea, ask clarifying questions, find downsides and pitfalls, and choose a realistic implementation scenario ✦ develop and describe the user journey in detail ✦ outline the technology stack for the project
Artifact	Functional / technical brief (project passport, etc., to your taste) in .md (markdown) format.

** For some reason I don't use role assignment in my own practice, although it's a universal recommendation from neural-network developers. Don't be like me — use it.*

Stage 2. Architecture and development tactics (roadmap)

Goal of the stage	Finalize the technology stack*, define the file structure and the order of development. The model acts as a "system architect."
Prompting	➡ load the brief from Stage 1 ✦ don't write code — work through the technology stack (offer different options with detailed explanations of their features) ✦ develop the project's file structure (files with code and other tools, files of necessary data) and justify the need for that number of files; describe the functionality of each file and the connections between them ✦ pick the optimal development order, compose a roadmap, write prompts for each stage, and provide instructions for setting up the necessary environment on your device ✦ assemble the artifact (blueprint or roadmap, to your taste)
Artifact	blueprint.md (selected tech stack, project file/folder tree, data schema, step-by-step development plan (checklist), setup instructions)

** Technology stack — the set of specific programming languages, databases, and tools that work together as a single foundation for your program.*

Stage 3. Phased development (here's the actual vibe coding)

Goal	Write the code and assemble all files and data according to the structure planned in Stage 2.
Working principles	➡ work on each stage from the plan in a new chat ➡ load the brief and blueprint into every chat — that's the baseline context. It's worth creating a project if your LLM supports that feature, and placing the brief and blueprint as project materials ➡ load files with previously written code into each subsequent chat (adding them as project files would require constantly updating them when the code changes) ➡ don't move on to a new stage until the previous one passes its tests
Prompting	➡ reference the brief and blueprint, indicate the stage you're working on, and use the prompts generated in Stage 2 ✦ write code that will test the file being worked on, log execution errors (traceback), and fix those errors ✦ all secret materials (keys, passwords, links to private data, etc.) go to .env — don't hardcode* them
Artifact	Files with code in the chosen programming language.

** Hardcode — to include directly in the code text.*

Stage 4. Testing and debugging

Goal	Identify the maximum possible execution errors and fix them.
Working principles	➡ a complex error not obviously related to another error → new chat (within the project, to avoid duplicating the brief and blueprint) ➡ attach all files with written code to the new chat — not just the one causing the error ➡ debugging is iterative work. But if you feel stuck or going in circles, try a different model
Prompting	➡ describe the errors — after which actions they occur — attach screenshots, traceback ✦ "diagnose the causes and write the minimum necessary code to fix them" — I strongly recommend this formula, since LLMs in practice show inappropriate proactivity ✦ preferred form for presenting code changes (entirely new code / fully rewritten function / highlighted part of the function in "before — after" format (I prefer the last option))
Artifact	The final working program.

Stage 5. Build and launch

Goal of the stage	Bring the program to a state ready for regular use by end users; prepare documentation.
Prompting	✦ **audit the code for hardcoded keys and passwords** (see next section) ✦ detailed instructions for building exe/app, deploying to a server, depositing on GitHub, or another way of executing the program ✦ create dependency files (requirements.txt for Python or package.json for JS) and documentation (README with installation, launch, project structure, troubleshooting instructions, etc.) ✦ set up versioning (Git) — optional ✦ choose a license for the use of your project by others — optional
Finish line	Executable program, documents (dependencies, environment, README), GitHub repository, Git.



Some closing security considerations

4 rules for vibe coding that's not just comfortable but secure as well



Due diligence on third-party libraries

- ✦ LLMs occasionally hallucinate, inventing nonexistent libraries or suggesting packages whose names differ by a single letter from popular ones. Hackers exploit this by placing malicious code under similar names (typosquatting).
- ✦ Before issuing the install command, verify the library's existence and purpose by asking the LLM for a link to its official repository (PyPI or npm).
- ✦ Check the publication date and number of library updates: if the library is 2 days old, installing it is a critical risk.

Limiting agent permissions

- ✦ Coding multi-agent systems (Claude Code, Codex, Antigravity) and IDEs with built-in agents (Cursor, Replit) are built for professional developers who can tell when something has gone wrong.
- ✦ Agents gain access to the file system and can initiate commands to download or delete files, which can lead to unpleasant consequences.
- ✦ Don't disable the action confirmation mode ("Ask before run") — you should confirm every command that modifies or deletes files.

Moving sensitive data to environment variables

- ✦ Environment variables (.env) should hold any data that, if obtained by a third party, allows them to gain unauthorized access to your servers, database, or finances, or to decrypt user sessions.
- ✦ Such data may include:

  - ➡ API keys (a unique secret code that serves as both an "identity card" and a "payment card" for your program, allowing it to use paid resources of third-party services (such as LLMs or data aggregators) and charge fees to your account for each request);
  - ➡ database connection strings (Database URL — looks roughly like 'postgres://user:password@localhost:5432/mydb');
  - ➡ mail-gateway credentials (SMTP — logins and passwords for mailboxes);
  - ➡ bot tokens (Telegram, Slack, Discord — give full control over the bot);
  - ➡ private keys and certificates (SSH, Private Keys — never hardcode them or paste the contents of an automation key into the environment: in .env you only specify the path to the key file, never the key itself);
  - ➡ messenger session strings (Session String — a generated code that is a digital fingerprint of an already completed sign-in). Possessing it allows an attacker to instantly clone your account on their device, completely bypassing the password and two-factor authentication (SMS code). See case.

- ✦ Store keys only in special protected environment files (.env) that don't reach the public domain.
- ✦ Set up .gitignore — a service file that tells the version-control system (Git) which files should remain only on your computer and must never end up on GitHub. It needs to be created before you make the first "save" (commit) in Git, otherwise it will still remain in the change history and hackers will be able to retrieve it.
- ✦ Require the model to set this structure up from the very beginning, as part of Stage 2.

Verifying terminal (command-line) commands

- ✦ The terminal (command line) is a tool for direct control of the computer, often with "superuser" rights. Entering a command blindly (copy-paste) can, in the worst case, lead to irreversible consequences such as the complete deletion of the project (or other data important to you) without the possibility of recovery.
- ✦ Be especially cautious with commands containing:

  - ➡ rm (remove) and especially rm -rf — file and folder deletion commands. There's no "Recycle Bin" in the terminal: deleted data disappears forever and instantly;
  - ➡ curl | bash or wget — commands for downloading and immediately running scripts from the internet. You must be confident in the source's reliability;
  - ➡ chmod (especially chmod 777) — changes file access rights. The 777 setting allows reading and writing the file by anyone.

- ✦ Use the "second key" rule: never run a complex command if you don't understand the meaning of every flag (the symbol after the dash). Ask another model what the command will do.
- ✦ If a command seems too radical (deletes lots of files or changes system settings), ask the model to suggest a safer way to solve the task.

Thank you for your interest in this material — and I wish you comfortable, safe, and inspiring coding with LLMs!



Disclaimers

The material is distributed under the CC BY-NC-ND 4.0 license, which lets you freely copy and distribute the material in any format, provided you credit the author. Commercial use and the creation of derivative works based on it are not permitted.

The images used in the material were generated by the NanoBanana Pro neural network; the concept was devised by the author.

LLMs were used while preparing the material — for the framework of the texts on the context window and prompting, for assembling data for the security section, and for general grammatical editing and translating into English.

Contact the author

yakunenko.ekaterina@gmail.com

https://t.me/delay_RAG

https://delay-rag.ru/