LLMs for the working programmer - Handout

Workshop by Manuel Odendahl (https://the.scapegoat.dev) - AI engineering world fair 2024

Find this document at https://github.com/go-go-golems/go-go-workshop

---

Welcome to my workshop about LLMs for the working programmers. The focus of the workshop is on concrete techniques that work for your day to day job. I have been using LLMs for *everything* I do since copilot/ChatGPT came out, and hopefully these notes will inspire you.

During the workshop, I will present a few of these techniques on the screen, but the most important is for you to experiment with them. Use the following notes and examples as starting points for your own explorations.

Using LLMs for programming is a skill that is learned through practice. We are at the bleeding edge of a new way of interacting with computers and noone really knows what is possible.

Overview

After showing a few pragmatic, "common sense" techniques, I will dive into a conceptual framework for working with LLMs. The higher level concepts will be illustrated with copious ideas and prompt examples, but because noone wants to read full transcripts and I don't want this handout to be 500 pages long, I encourage you to try them out on your own. In fact, that's why you came to a workshop!

- We will first look at LLMs as "translation engines" and what that entails for us working programmers.
- We will then go over what I think the fundamental skill a centaur software engineer should have: divergent thinking and consistently working out higher level abstractions.
- Finally, I will show some examples of prompts that will hopefully shatter your understanding of what it means to program with LLMs. Finally, let us all discuss and freak out about the future of software engineering as a profession.

Preliminary setup

All of the following techniques are meant to be used with stock models, through a chat web UI or through API calls using your own tools. They are all based on the idea of zero-shotting a solid first step and then refining it through conversation.

Concrete techniques

Here is a set of concrete techniques.

- **Regenerate often**. This will give you an idea of how "wide" the probabilistic spray of a model is.
- **Start new conversations often.** This helps make sure that your prompts are solid and helps maintain a clean and small context window.
- Don't be afraid to **use many different system prompts** or just straight out / commands in chat.openai.com etc...

Example: Creating /command system prompt

Write a system prompt that defines /commands that I can give the LLM o that I can control how it outputs information.

I want a /wiki to format in wiki format, a /concise to concise bullet point lists, /rfc to format as a code RFC, /github issues to create github issues, /dsl to create a YAML DSL.

Add a / command to create LLM system prompts.

/systemprompt Create a prompt that guides me through the process of building /commands, so I can give it to my mother. It should explain what /commands are and how to use them, and help her along.

Then use:

Creating a /recipe command

Paste the previously generated prompt

Create a /recipe prompt to create recipes

/recipe vegetarian pasta dish

- **Experiment with new models** coming out all the time. This doesn't mean just bigger models, but also smaller faster models. Get a feel for what they can or cannot do and how they behave (or misbehave).
- **Practice solving the same problem multiple time** as you learn more (something called "code katas" in agile literature).
- **Don't hesitate to edit your or even the LLMs messages.** "Gaslight" them your heart's content. Let them think they output the code you had to fix.
- **Generate helper scripts** to help you work (edit clipboard, store things in history, search transcripts, etc...). Don't do anything tedious by hand anymore.

> Helper script to manage clipboard history
>
> ChatGPT transcript
>
> Write a shell script that takes the current clipboard content, opens a terminal with $EDITOR so I can edit it, and when closing, updates the clipboard content.
>
> I use macosx.
>
> ---
>
> keep a history of previous clipboards and add a flag that allows for listing and another for selecting past clipboards.

- **Summarize your transcripts into reusable context fragments** (see for example the /command prompt which helps me edit my sessions down to wiki entries or RFC documents I then put into my knowledge base).
- **Use "ridiculous" domain examples to more clearly separate "the linguistic pattern (reasoning)" and to avoid being "fooled"**. I like using dinosaurs, tolstoy, marine biology, etc... Have fun!

  > T-Rex operating system
  > Write an arduino RTOS for a remote controlled tyranosaurus rex, with real-time preemptive task switching.

LLM as translation engines

No one will ever understand LLMs: they literally are computational interactive artifacts created out of all of the words they have been trained on. That is so far beyond human comprehension that we are not really dealing with software technology anymore, but with *cultural* technology. Learn more about art, history, read, literature, management speak, other languages, poetry, philosophy, mysticism...

Fundamentally, I think of LLMs as **translators between "types of language".**

There is no "one natural language", nor are there "formal languages". The LLM doesn't care, all it works with are tokens. It so happens that we can train them to be better at formal output, but we all know how robust that is. Instead we have (just a few):

- Spoken language
  - With your mom
  - With your friends
  - With your boss
  - On a TV interview
  - At a fundraising gala
- Programmer jargon
- Github issues
- RFCS / guidelines / HOWTOs
- Code
  - Code sketches
  - Code design patterns
- Titles, subtitles, taglines
- Slides
- Log files and data dumps
- Markup languages
- Keywords, tags, labels
- Poetry
- Theater
  - Greek drama
  - Reality TV show
- DSLs (which is creating specific target "languages")

Each of these types of language have their own structure, vocabulary, syntax, whether it is clearly apparent as a formal syntax or not. Even when a formal syntax is used, the "lexical units" (the words and symbols used) have a much stronger pull than what we are used as programmers. Calling a variable "a" or calling it "sumOfYearlyRevenue" has a huge impact, even though we are used to the computer not caring.

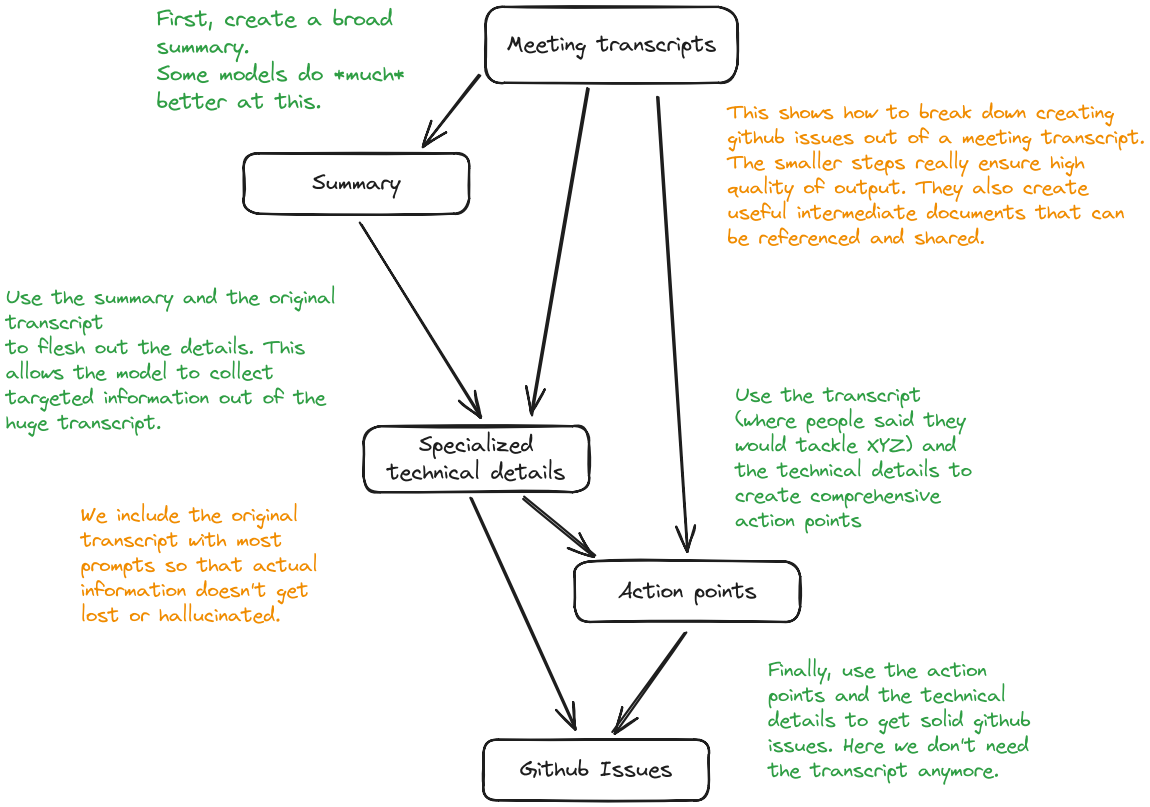
LLMs are fantastic at translating from one type of language form to another, assuming that the "reasoning" step is not too large, or that "fringe linguistic hallucination" can be spotted and corrected quickly. Now this might sound obvious, but this means that **to effectively use LLMs, we have to decompose our tasks into "language translation steps".**

This doesn't just mean decomposing a problem from a technical / engineering point of view (what we would call system design or software architecture or project management), but from a "language" point of view. A lot of what makes software successful is not code, but documentation, RFCs, communication amongst colleagues, meetings, pair programming sessions, communicating with stakeholders, budget plans, presentations and marketing copy. As such, the LLM programming centaur should **widen their scope and tackle engineering from a human language perspective**.

Examples

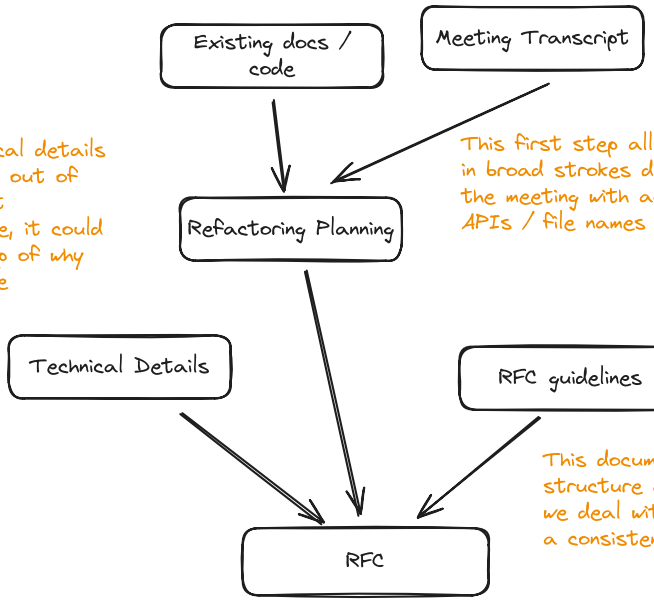
Here are a few examples of "translation steps".

Converting meeting transcripts to github issues



Converting meeting writeups to RFCs

We discussed how to refactor a feature during the meeting, and we want to create a concise and solid RFC describing what lies ahead.



These are the technical details we already extracted out of the meeting transcript previously. For example, it could be a detailed write up of why we want to modify the pagination widget.

This first step allows us to fill in broad strokes discussed in the meeting with actual code / APIs / file names / etc...

This document describes the structure and process of how we deal with RFCs, and ensures a consistent result.

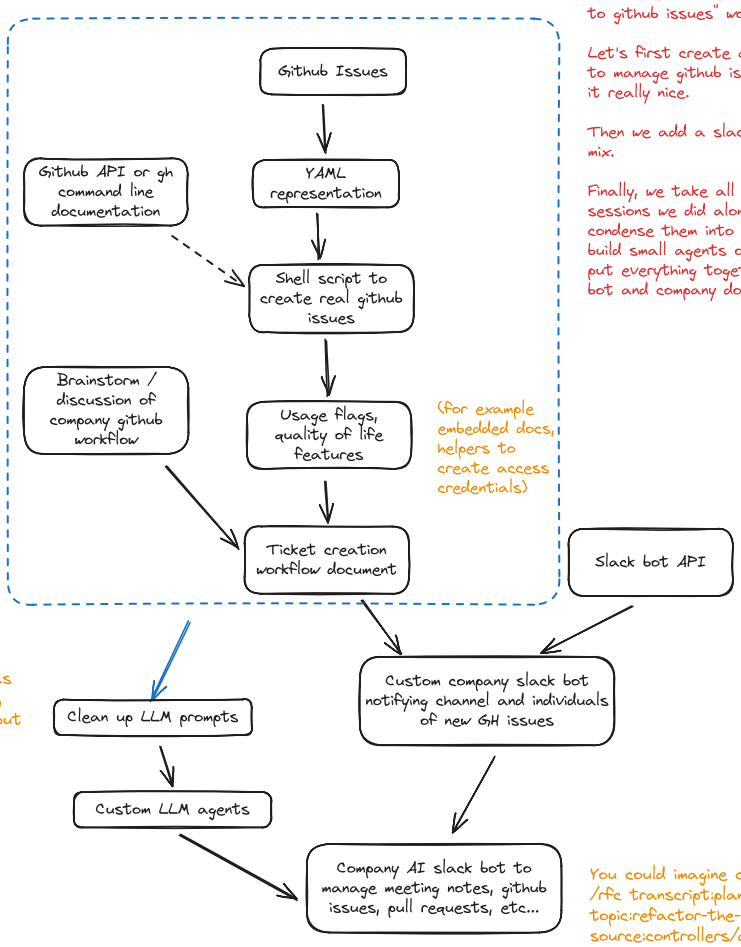
Github issues to automated github management system

Now let's go meta. We want to automate that entire "transcript to github issues" workflow.

Let's first create a simple script to manage github issues, and make it really nice.

Then we add a slackbot into the mix.

Finally, we take all these nice LLM sessions we did along the way, condense them into LLM prompts, build small agents out of them, and put everything together into a slack bot and company documents.



For higher quality models who have been trained on recent data and can "remember" it well, this might not be necessary. It however can't hurt, because hunting down hallucinated code mistakes is time intensive, compared to the time spent collecting documentation.

often, workflow documents require collaboration and brainstorming. Instead of writing things up by hand, just add the transcript to the mix.

(for example embedded docs, helpers to create access credentials)

Now take all the prompts you wrote as part of the process, clean them up, and put them together into "agents"

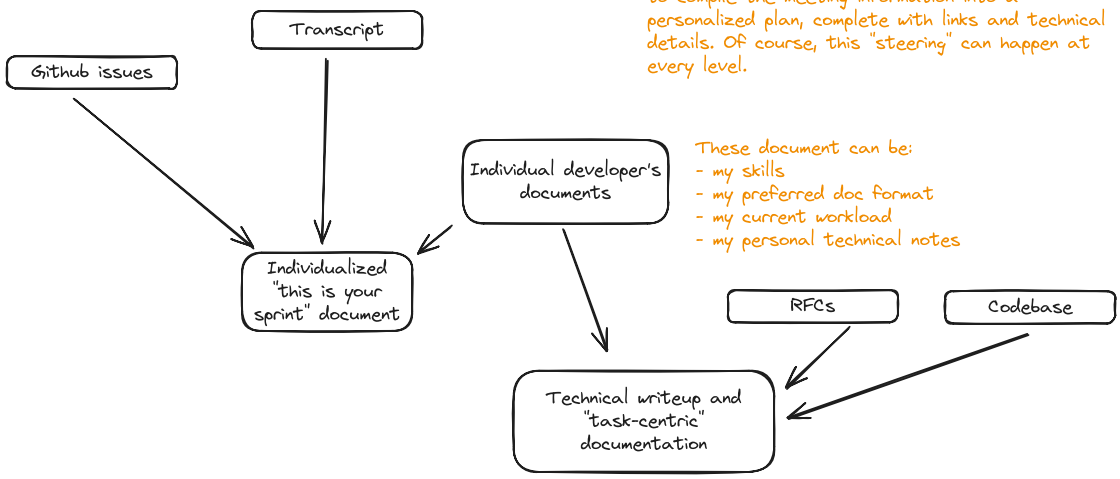
These are agents like:
- transcript to summary to github issues
- transcript + code to RFC

You could imagine commands like:
/rfc transcript:planning-meeting topic:refactor-the-db-schema source:controllers/db.rb

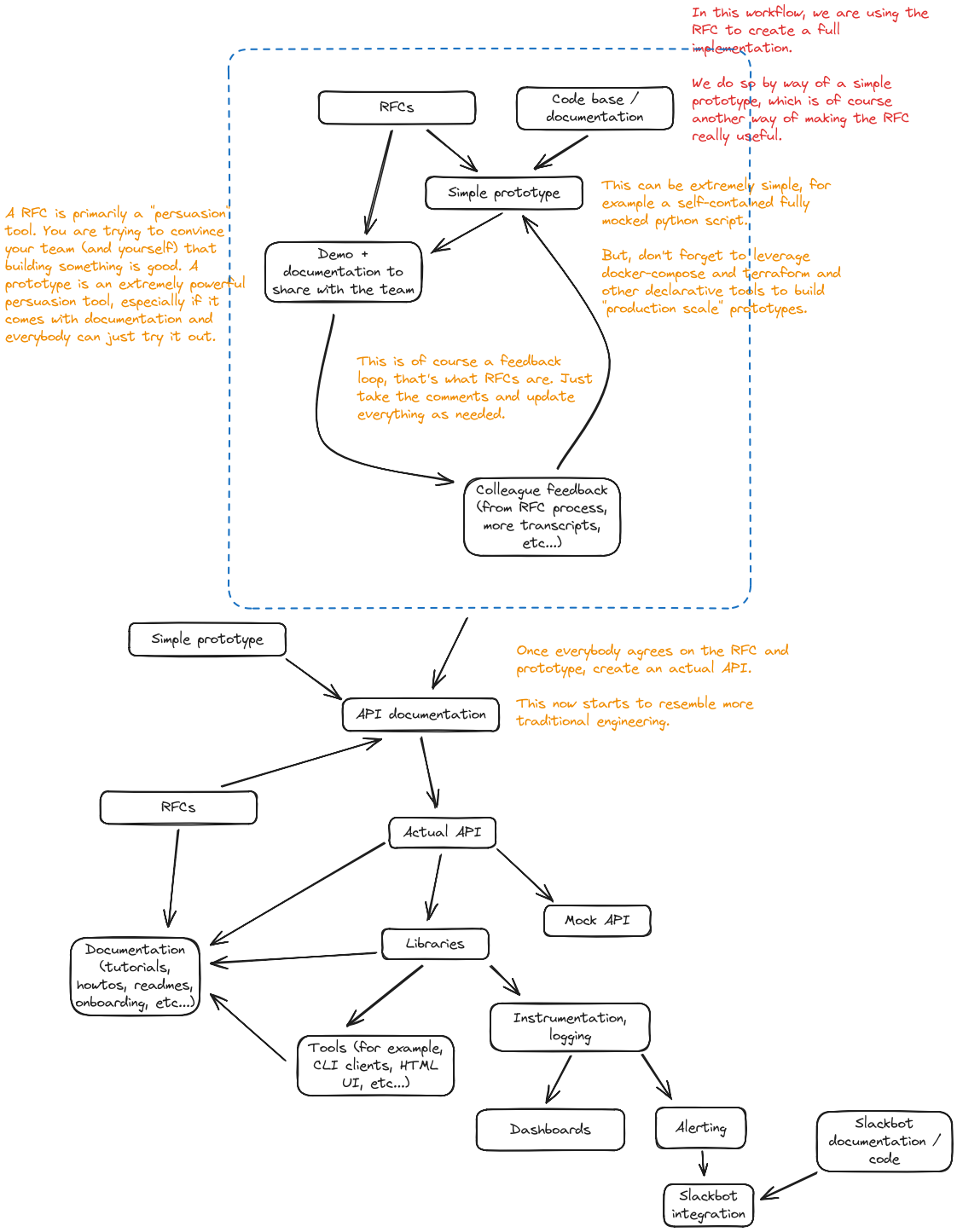
Transcript to personalized developer writeup

Let's take this meta, but in a different direction.

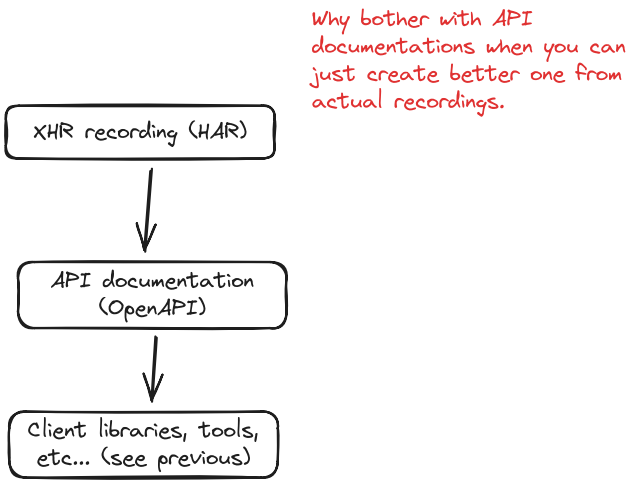
Each developer has a different way of working and thinking. Let them describe what they would prefer to have as a document, then use the LLM to compile the meeting information into a personalized plan, complete with links and technical details. Of course, this "steering" can happen at every level.



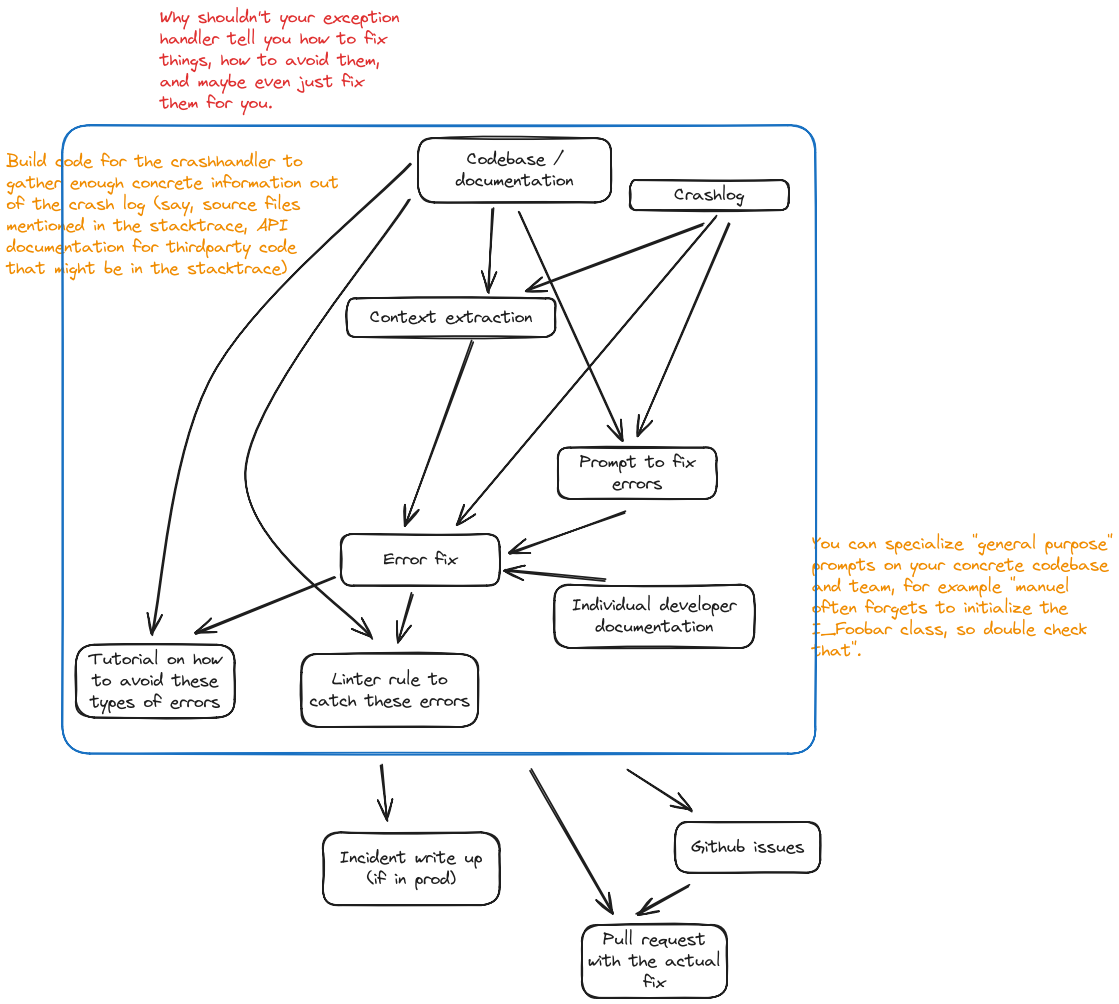
RFC to full implementation



XHR requests to full API



Crashlog fixer

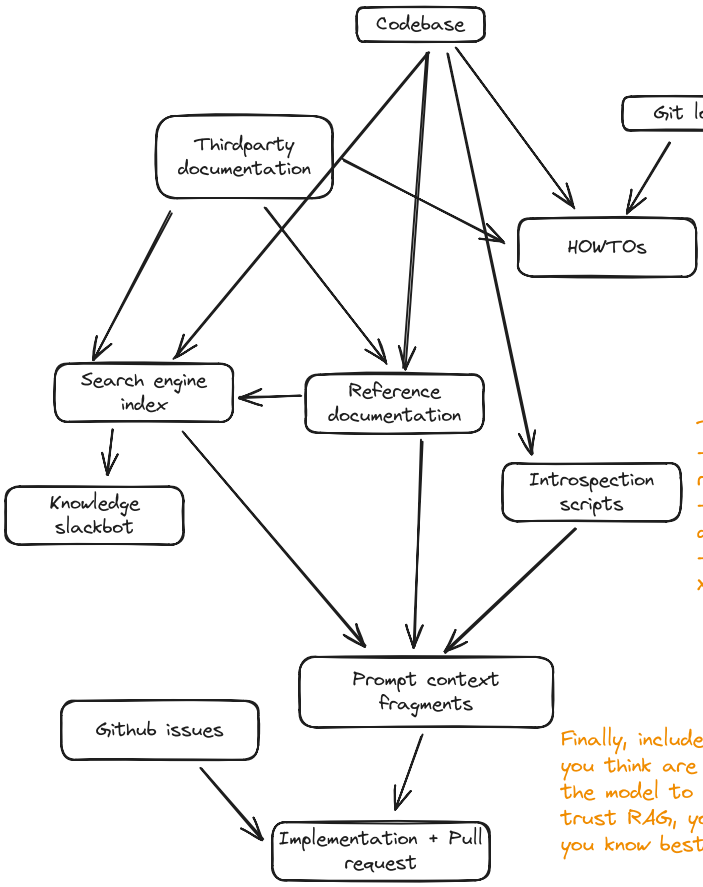


Prompt context creation

One of the most powerful techniques for using LLMs for programming is to create concise and effective prompt fragments that can be reused as needed.

Instead of yeeting 100kLOC of code at the model, treat it like a human. It was trained on human communication, after all!

Create API specs, howtos, overview READMEs. Gather the API and schemas from the live system or the current codebase by using scripts.



Finally a reason to write clean git log history and commit descriptions.

Because what is a clean git log if not a tutorial on how to do somethiing.

These are scripts like:
- give me the current DB schema related to customers
- give me all public functions and docstrings in src/foobar
- get me the last 15 log entries for XYZ

Finally, include the context fragments you think are relevant when prompting the model to build something. Don't trust RAG, you are a capable engineer, you know best!

Learning a new topic

Phind - Learning a bout the reactive UI patterns

I want to learn about the reactive UI pattern.

I am a PHP developer. Write a skeleton prototype of reactive programming using PHP.

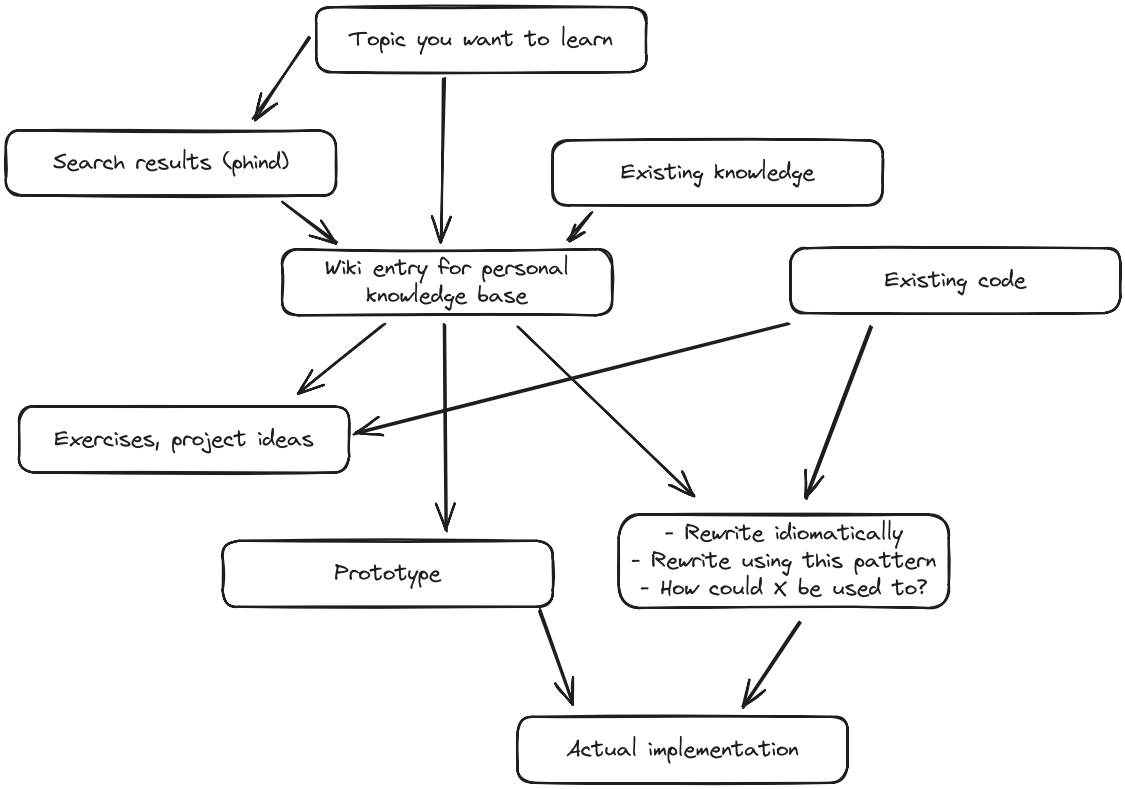
Now, write it without any external library, using the fundamentals.

Use it to design a little interactive text terminal adventure game.

Write a tutorial on implementing reactive UI from scratch in PHP.

Make a list of exercises for me to extend this game and learn more about reactive UI patterns.

For exercises 1-4, explain how the reactive UI pattern applies.



Domain specific languages

Now that we know how to leverage existing language types when prompting LLMs, why not have the LLM create its own languages. This is really using the DSL pattern.

Ask the model to:

- **create a "YAML DSL" for the problem you are trying to solve**
  - cheaper in token counts
  - strongly constraints the model (if it can't express some erroneous thought process in the DSL, then it won't)
  - can easily be checked, even by non-developers
  - can easily be visualized and validated programmatically
- **regenerate and modify until you like the DSL**
  - This really is software architecture and system design. This is where your knowledge, wits, experience, creativity as an engineer come into play. A well designed DSL makes it seem as if the problems disappear
- **create a documentation for the DSL**
- **create the interpreter for the DSL**
- finally, **solve variations of the original problem by using the newly created DSL**

This technique of "letting the LLM roam free" is extremely powerful. Because the LLM created the language it thinks it should target, it is a language that usually is easily targeted by the LLM. The language created is also very consistent, compared to what a human can create, and can often pull in concepts from computer science literature that one might not have thought of.

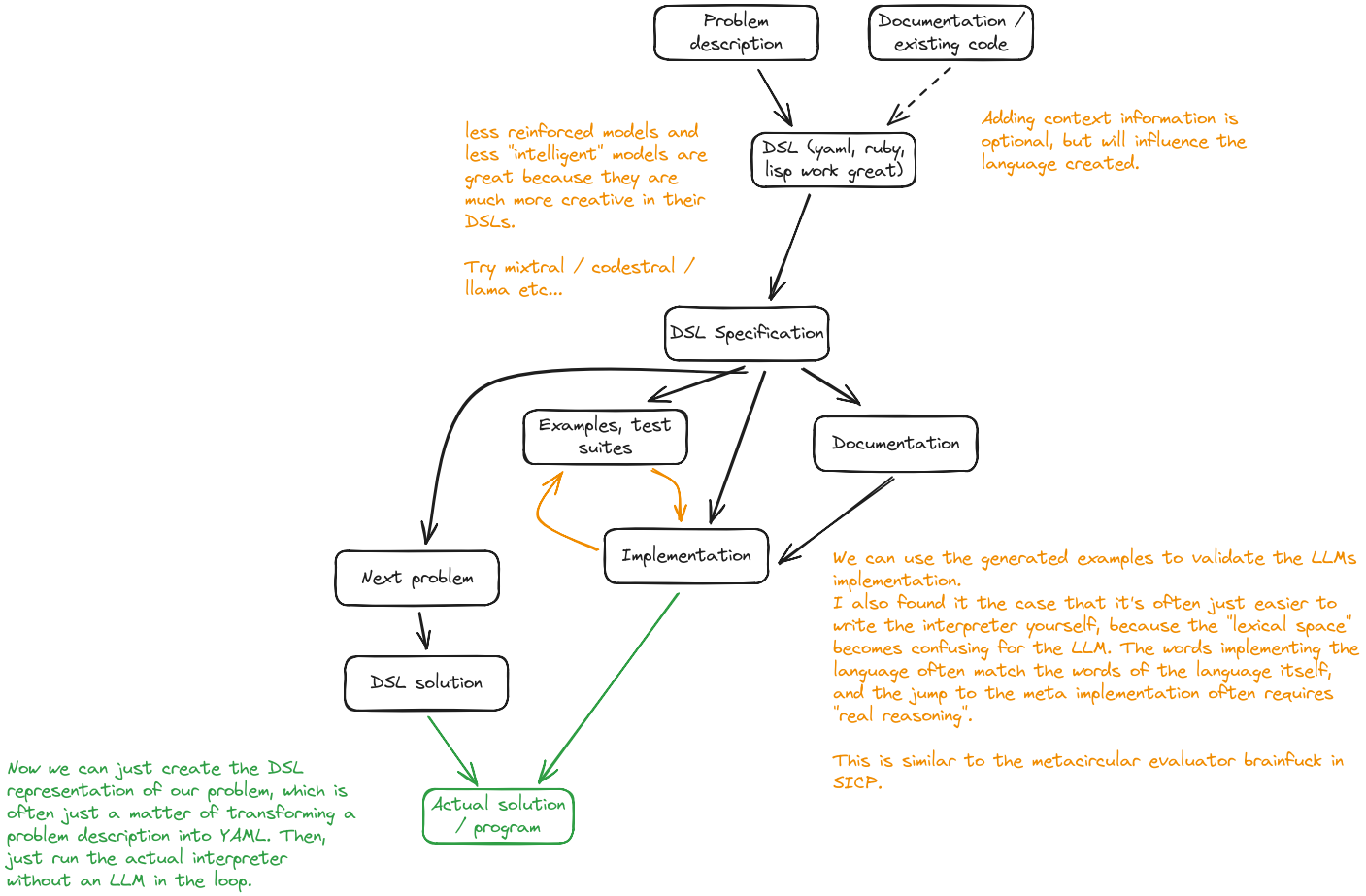
Designing and implementing a DSL

ChatGPT Creating adventure games

Design a YAML DSL to represent text adventure game logic and content.

Write pseudocode for interpreting user actions and updating the game state.

Create an alien inspired game where you have to find the keycard and the source code to dismantle the main computer system and kill the alien.



World simulation

The model's reality is what you tell it it is. There are literally no limits. If you can put it in tokens, it will exist.
Here are a few examples:

You are an application that does X.

- Output your UI in concise YAML DSL form.
- Start
- *interact with the UI*. Output the list of actions you take as a result as a YAML DSL.
- Implement yourself using X

Headshot resizer

You are an application that resizes head shot profiles, resize it for conferences, select the head location. I want to put up my picture form my phone and it will be stoerd in the DB, and you can go there later and just get the link for that website. --- Output your UI as next.js widget short notstion in YAML. --- Don't output JS< but instead a shortcut concise YAML notation. Only UI widgets.

I upload an image. Output a YAML list of the actions that you take as a result on the backend.

No, the upload only stores the image and precomputes the headlocation. The resizing and all that is done when I decide to. Update the action list.

Now let me resize the image to 64x64 and store that as the "linkedin" template. Then, update my linkedin profile with the generated link.

Sketch out the React components for the UI.

Implement the backend actions as next.js

You are a reality show where the participants are pieces of code / variables / programming concepts.

ChatGPT - Code Survivor

You are a working piece of code that does X, using these tools.

What skills do a 10x LLM centaur make?

I find it hard to narrow down the skills needed to be really effective at LLM assisted programming. If I were to make a guess, I think it is a combination of:

- **Deep practical and fundamental knowledge** - Knowing how to build something and do it repeatedly in light of changing circumstances helps recognize what patterns to apply to decompose a problem into steps within the reach of a specific LLM, and recognize when the LLM goes awry.
- **Constantly practicing** - This is more of a skill like playing the guitar or other crafts, and less something you can learn analytically and out of books. You have to develop your own style and intuition on how models behave. Practice solving every problem you encounter with LLMs, or at least think "how could I decompose this so that I can solve parts of it with an LLM"
- **User centric thinking** - This might sound paradoxical, but the output of an LLM only has value if it is "experienced" by a human. If you don't know what the target human wants (be it a working app, or some clear documentation or a great TPS report), you will not have a direction for decomposing the problem into "natural language translation steps".
- **Divergent thinking** - LLMs are a cultural technology. You have to think outside the box to actually explore the box. Thinking of "can I use greek tragedy to make the LLM create better code reviews" is a step that is not easy to make. Look at what non-programmers are prompting.
- **Abstract thinking / meta thinking** - For each thing you build, think about the implications on the overall problem you are solving. Can you build the thing that builds the thing that solves the problem? Can you write the software that does the thing you just had the LLM do? Can you write the software to help you write the software that builds the software that helps humans write software that writes software?
- **learning when to step away** - You can only go so fast. You can generate 80 versions of a piece of software, and even have it working: the temptation is big to continue. But often, the best engineering decision you can make is to go walk in the woods.
- **Learn when you should do something yourself** - LLMs can't really reason, they certainly can't reason creatively. They will not create new algorithms because they thought really hard about it until they had a eureka moment. In fact they can't often solve a simple programming task like a SQL query to compute YOY revenue. Instead of fighting the model and going against the grain, just write the query yourself. It's way more fun, and one of the reasons we became programmers in the first place.
- **Learn about DSL and language design** - Language is the alpha and omega of LLMs. There is nothing more. Learn about programming language design and philosophy of language and linguistics and mysticism (so much of mysticism is about words) and literature and foreign languages and grammar and knowledge representation and cognitive linguistics. They now are all "programming frameworks." Of special interest are self-referential languages like Common Lisp, ruby, forth or to a lesser degree haskell.