Structured planning in LLM-assisted software development

From concept to plan to code.

Brett Viren

February 22, 2026

Overview

- LLM-assisted software dev strategies
- Motivation for the strategy
- Motivation for the package.
- Formulating a concept (plan for a plan).
- Constructing plans
- Plan driven development

Sampling of LLM-assisted software dev strategies

Serial, ad-hoc

Human prompts, LLM codes, human tests, iterate.

Serial planning

Human prompts, LLM writes design doc, iterate, doc becomes prompt, LLM codes, human tests, iterate.

Structured planning

Human prompts, LLM writes plan-for-a-plan, human edits, LLM writes detailed plans and tasks with dependencies, human reviews and edits, LLM works on tasks following dependencies:

- Human-driven serial steps or serial or even parallel full-auto.

This is basically the “waterfall model” but the “phases” take minutes instead of weeks/months. Instead of a team, you have Claude + Beads.

Why structured planning

- Up-front architectural design can catch fundamental problems.
  - Missing requirements and conflicting direction encourage wrong results.
- Writing complex prompts is difficult and a little stressful.
  - Writing specs “offline” allows for edits, contemplation, consistency.
  - Direct “prompt to code” means fixing mistakes in code rather than verse.

The “toenails” tool

Goal: develop a tool named **toenails**[1] to:

- Help humans and LLMs understand source a code base
- Perform “semantic” and “symbolic” queries against source code.

There are many such “semantic search” type tools like this already. Some companies even.

- I walk through the entire process from idea to working tool.
- It took about 4 hours including setting up beads, some false starts.

[1] Name derives from Lisp being described as “toenails and oatmeal”.

Formulating the concept

I knew roughly what I wanted including some specifics but many unknowns.

- Tree-sitter for code parsing to produce a **graph model** of the source.
  - Query for “symbol search” and following relationships.
- Support for one or more embedding model(s) to produce RAG vectors.
  - Query for “semantic search”
- Build in incremental updates as code base develops.
- Wanted workflows and results accessible for humans and LLMs
  - CLI, text, JSON, TUI, web UI.

Generate initial `plan.md`

I described to an LLM roughly what I wanted without too much care about my English and cohesion of thoughts.

> *Make an initial plan.md file wiht a to develop a Python based command line tool for performing hybrid AST and semantic search of source code. I will describe the basics for what I want and you will produce a plan to develop this tool. This initial planning will result in a "plan for a plan" that will defines the tasks required to refine the plans for implementation. I will now describe the features wanted for this tool. First,*

Iterate on the plan-for-a-plan

I then touched up `plan.md` by hand, but not by much. It was already pretty good considering I used Gemini Flash 2.5 for this.
Going forward though, Gemini Flash was lack luster and I moved to Claude Sonnet 4.6.

Enter Beads

Beads is a local issue tracker with dependencies between issues. Humans and LLMs easily use the `bd` command for all usage.

```
$ bd init --prefix tn
$ bd create  # make an issue
$ bd dep add tn-abc tn-xyz
$ bd list
$ bd ready
```

Initially for “bug tickets”, it’s become one way to coordinate LLM-assisted development.

Flesh out the plan

To Claude Sonnet 4.6

> *Read plan.md and convert its information into Beads epics and issues using the "bd" command. Include dependencies between issues using beads (bd dep add). Many of the tasks are not yet fully defined in the plan and issues should be made to describe what is needed to provide their specification.*

```
All 41 issues created and exported. Here's what was built:

  9 epics (one per phase, tn-k25 through tn-zq2):
  - Phase 0: Plan Refinement — 7 decision-type issues for specs that don't yet exist
  - Phases 1–8: 32 implementation tasks

  19 issues are immediately ready to work ...

  82 dependencies were wired, encoding the natural ordering:
  - details ....

  Run bd ready to see the 19 actionable issues, or bd blocked to inspect the dependency graph.
```

Claude does more planning

Work on beads issue tn-k25 starting with its children.

Claude uses bd to discover the main issue and its children and then:

I'll mark them in$_{progress}$ and research/draft each specification. Let me start all of them in parallel.

Claude updates three issues with results and then:

Now I'll research the library decisions while drafting the design specs. Running research in parallel with the design work.

Claude launches "explore agents".

While the library research runs, I'll write the design specifications for the five non-library decisions (tn-k25.1–tn-k25.5).

Did some web searches on various Python packages and concludes "right choices" for some technology.

- **These choices definitely need human review**

Wrote 7 design docs by updating Beads issues and offered to close them.

- I decline until I review.

Enter human

At this point I go through the design docs in the `tn-k25.*` issues to make sure I'm happy.

- This part is laborious but it's where human judgment must come in.
  - Whatever is missing/wrong here likely wastes a lot of time later.

```
$ bd show tn-k25.1
$ bd edit tn-k25.1 --design
$ bd edit tn-k25.2 --design
...
```

- Directly update each design doc.
- Chat with Claude to settle any uncertainties.
- Make new `tn-k25.*` children issues as needed.
  - Made a new one depending on `tn-k25.*` that centralizes the DB schema.

The choice of vector DB

I ask Claude

> *In tn-k25.3, chroma is selected. What is the purpose of the chroma.sqlite file and why is it separate and not combined with the toenails.db file?*

Claude explains that it's an internal chroma choice and admits that may not be a good choice and offers `sqlite-vec` (which would have been my pick) and offers to switch.

> *Yes, revise the design to use sqlite-vec and a single Sqlite file. Also extract and centralize all DB schema into its own issue.*

Claude makes a new `tn-k25.8` with the central Sqlite schema.

Start knocking down the backlog of issues

Work on tn-w2i and its children tn-w2i.2 and tn-w2i.3. tn-w2i.1 is already done.

Usual watching and hitting Enter, sometimes giving a “Yes, but”.

Carry on

Find new work.

```
$ bd read
$ bd epic status
```

> *Work on tn-0zl and its children.*

This detects dependency on the design issues and asks to close them. Yes, since we have reviewed them.

- Will Claude refer to them in the future? How if we lose context?

Continue each “epic” in original order. Claude “learns” to predict next epic to work on so “development” becomes:

- Read epic and issues.
- Reconnoiter with trial code snippets.
- Install new deps
- **Write code**.
- Write tests.
- Run tests, fix bugs.
- Close issues.

Compaction happens

Context fills and Claude summarizes it and restarts. And then runs:

```
bd list --status=in_progress
```

And picks up where it left off.

Iteration ends

```
$ bd ready

No open issues
```

Usage testing

```
$ uv run pytest
$ uv run toenails index .
...
indexed 34 file(s) (34 full) in 0.6s
```

Actual runtime was 10s but it included downloading models.

```
$ uv run toenails search parser
[0.10] 338025c3a066c1b5d62e5c6249997f5ae4b9540e4442bb766099258ccfbbc5df:9-10  fun
       def parser():
           return SourceParser()
...
$ uv run toenails find parse
callers of parse
  ...

callees of parse
  ...
$ uv run toenails find parse [--json|--tui]
$ uv run toenails web
```

Things not perfect

- Output formatting needs polishing.
- Need CLI and config file for using different embedding models.
- Ways to share models between different toenails projects.

Things I forgot from the start

- Instruct Claude to update a ticket with summary of the work prior to closing it.

https://github.com/brettviren/toenails

FIN

Backups