Comparing LLMs for structured planning

No code was harmed

Brett Viren

March 2, 2026

Topics

- Intro to **structured planning**.
- Rehash of the prior experiment.
- Setup for the planning test.
- Some results and observations.
- Drill down to one “research task”.
- Some initial conclusions and bias forming.

Structured planning

Structured planning (just my name) is an approach to using LLMs for **large scale** software development. Basic workflow of a project:

1. Human + LLM write a "**plan-for-a-plan**" document (eg Markdown file).
2. Iterate: human edits and/or human+LLM refines.
3. LLM breaks down **plan-for-a-plan** into detailed **plans** and **tasks** as **Beads Issues**.
4. Iterate: human edits and/or human+LLM.
5. Execute: human develops and/or human+llm and/or just LLM.
6. Automate: let LLM walk up dependency tree working issue after issue.

Each Beads issue has an ID like `<prefix>-<hash>` and is composed of:

Sections	title, description, design, notes, comments.
Labels	"epic" (major aspect), "feature", "task", "bug", …
Dependencies	parent/child `id-abc`, `id-abc.1`, `id-abc.2` or `id-abc` "blocks" `id-xyz`.

Structured planning caveats

Not an original idea

Many people pursuing variations on this theme

- Beads is the kernel for Gas Town which scales this idea up orders of magnitude.
- Others may write a single `project.md`, generate a `plan.md`, generate `tasks/<task>.md`.

I do think the constraint of the Beads CLI is beneficial. A file-system can be "anything".

Main point

Human prompts the LLM to write prompts for LLMs.

- Distinct from "immediate mode" loop of human prompts, LLM codes.

Allow the human to think of planning, design, tasks at human pace and order instead and avoid "prompt paralysis".

Prior work

I used 4 hours of my weekend and just Sonnet to apply this approach to build "**toenails**", a "semantic code search" to completion. It is a working tool with CLI, TUI and Web UI, code parser filling graph and vector DBs providing symbolic and semantic queries on a code base.

- See previous presentation for a walk through.

Main conclusion:

- The structured planning method worked well.
- Tool essentially worked after all issues completed.
- Gaps show through in result, after testing find polishing needed.
  - Better and more consistent content formatting of query results.

Process questions this test raised:

- Would a plan-as-you-go approach (more "agile" than "waterfall") work better/faster?
- How much does the choice of LLM matter?
  - This work tries to address this.

Comparing different LLMs for the planning phase

Give identical prompt (see backups) to different LLMs and evaluate the results.

- Define a **concept** for a novel software project.
- Initialize a greenfield Python package repo with **uv**.
- Install **Beads** 0.57.0 with **claude** CLI integration.
- For each LLM: structured planning and work one “research” task.

The LLMs:

qwen	`unsloth/Qwen3.5-122B-A10B-GGUF:Q4_K_M` in llama.cpp running on both two RTX 4090 in wcgpu0.
haiku	4.5 on Azure
sonnet	4.6 on Azure
opus	4.6 on Azure

The project used as fodder for this test

The initial prompt starts with:

> *We will develop a software application that provides a "memory" database with interfaces for LLMs and humans.*

- Scope is a super-set of the prior **toenails** project.
- I am mostly interested in the planning phase here and may not push to actual coding.
- See backups for the full prompt.

Structure planning usage by the LLMs

LLM	Input	Output	Create	Read	Total	cost
qwen	1,734,719	6,475	0	0	1,741,194	0
sonnet	10	564	50,217	206,142	256,933	26¢
opus	11	860	39,428	246,205	286,504	39¢
haiku	79	13	40,448	129,456	169,996	6¢

- Ran tests in the order listed.
- Create/read count tokens in/out of Azure / Anthropic cache (“read” cheaper than “input”).
- `ccusage` may under-estimate the cost.
- Qwen ran on $10k of local hardware so was not exactly “free”.

Quantified comparison of products from the planning

LLM	epics	tasks	total	bytes	time
Qwen	7	21	28	20k	4
Haiku	12	23	35	25k	1
Sonnet	10	22	32	36k	5
Opus	12	22	34	37k	8

- An **epic** is an umbrella issue representing a major development with children **tasks**.
- The **bytes** simply counts the data size for all generated issues dumped to JSON.
- The **time** is in minutes and not measured carefully.
  - Qwen sometimes halted and I needed to issue “continue” prompts.
  - Times include time for me to react an ACK request from `claude`.

Finding specific tasks

Use Beads to find tasks related to “graph”.

```
$ bd query title=Graph or title=graph or description=graph or description=Graph
```

qwen	4 issues (graph db with sqlite, graph query lang, schema, epic)
haiku	3 issues (graph schema, db options, epic)
sonnet	7 issues (schema, hybrid query, graph db with sqlite, KG extraction pipeline, epic, deps for FTS and sem search)
opus	15 issues (research: hybrid, extraction, SQL query, storage. deps, epic.

Select similar issue - titles

qwen	Research graph database options within SQLite
haiku	Research Graph Database Options
sonnet	Research graph representation options within SQLite
opus	Research graph storage patterns in SQLite

Select similar issue - descriptions

qwen Research graph database options that work within SQLite including recursive queries and graph-like structures

haiku Evaluate Neo4j, NetworkX (in-memory), SQLGraph, and other options for knowledge graph storage. Consider Free/Open Source requirements and embedding in sqlite3.

sonnet Evaluate approaches for storing a knowledge graph in SQLite: adjacency list table (nodes + edges tables), recursive CTEs for traversal, SQLite's built-in capabilities for graph queries (path finding, subgraph extraction). Compare with using NetworkX as an in-memory graph layer loaded from SQLite. Assess: query expressiveness, performance for typical knowledge graph queries (neighbors, shortest path, subgraph), Python library options (networkx, igraph, rustworkx). Output: recommended graph storage and query approach. Feeds into so-9l2.

opus Evaluate approaches for implementing a knowledge graph within SQLite3. Consider: adjacency list (edges table), closure table (for transitive queries), nested sets, materialized path, and recursive CTEs for traversal. Evaluate each for: query complexity, write performance, traversal depth support, and suitability for a knowledge graph with typed entities and relationships. Look at existing projects (e.g., simple-graph for SQLite). Output: recommended pattern with example schema and queries.

Aside observation: Opus “likes” research

Opus had 4 research issues related to the graph DB. Each with a different aspect.

- Research hybrid search and result merging strategies
- Research entity and relationship extraction
- Research graph traversal query patterns in SQL
- Research graph storage patterns in SQLite

Put Haiku and Sonnet to their respective research tasks

> *Perform the research as directed in issue ID-XYZ. Produce a report in markdown format and add it to the issue in the "Design" section of the issue.*

- I filled in the appropriate ID-XYZ issue ID for each case.
- See prior slides for the content of each LLM's issue.

Not apples-to-apples, each LLM consumes the prompt it genreated.

- Haiku task was simpler, focused on tech comparisons.
- Sonnet task was more conceptual and nuanced.

A better/future test would start them with identical prompts.

- Perhaps using tasks generated by Opus.

Observations on Haiku and Sonnet research task results

- Haiku is much faster than Sonnet.
  - I assume Opus would run even longer.
- Haiku result is unexpectedly more verbose (392 lines) than Sonnet (233 lines).
  - Both results are good so not sure longer or shorter is better.
- Haiku gives sources as links, Sonnet does not.
- Haiku report is bullets and tables. Sonnet report is prose, tables and code snippets.
- Both give options and recommendations.

Some conclusions on LLM planning performance

- To optimize time, cost and quality, a tiered approach is probably best.
  - Opus, Sonnet, Haiku obviously on a spectrum.
  - Consider the conceptual vs research vs implementation complexities of a task.
- I am forming a bias for:
  - Opus or Sonnet for initial plan-for-a-plan.
  - Opus or Sonnet for plan break down into issues.
  - Sonnet or Haiku for researching tasks.
  - Sonnet or Haiku for implementation.
- Qwen 3.5 is not adequate compared to even Haiku for software development.
  - Qwen seems still capable for tasks like summarizing.

Some conclusions on structured planning

Especially with Opus, can produce an explosion of plan and task issues.

- Requires a lot of human attention (but more than other approaches?)
- Some new ingredient to automate or simplify human effort more would be helpful.

Perhaps a change viewpoint is needed:

- Think of issues as “reservoir of possible ideas” instead of a strict plan to follow?
- Move more toward “agile” and away from “waterfall” model?

FIN

Backups

Qwen + llama.cpp + claude setup

```
$ uv init --build-backend setuptools --lib llmemory
$ cd qwen/llmemory/

$ bd dolt start
$ bd init --prefix lm
$ bd setup claude
$ bd doctor --fix

$ ~/opt/llama-cpp/bin/llama-server \
          -hf unsloth/Qwen3.5-122B-A10B-GGUF:Q4_K_M \
          --alias unsloth/Qwen3.5-122B-A10B \
          --ctx-size 131072 \
          --temp 0.6 \
          --top-p 0.95 \
          --top-k 20 \
          --min-p 0.00 \
          --host 127.0.0.1 --port 8181

$ ANTHROPIC_BASE_URL=http://localhost:8181 \
  ANTHROPIC_API_KEY="sk-1234" \
  claude

> /model unsloth/Qwen3.5-122B-A10B
```

API key is unused, put anything. Parameters from Unsloth guide, no optimization done but had to increase `--ctx-size` to get `claude` to start.

Initial prompt

We will develop a software application that provides a "memory" database with interfaces for LLMs and humans. It is called "llmemory". I will describe the development process and then give an initial description how "llmemory" will be implemented.
The development process will follow a hierarchically structured planning and implementation process (similar to "waterfall development process"). It will use Beads issues with dependencies and issue types to organize, track and drive the development. Actual planning and coding will be mostly performed by LLMs while the human developer will review and modify the issues and code.
We will create the following beads issue types that covers both planning and coding work.

- *'epic' - A high level software design consideration or a major aspect or concern of the implementation.*
- *'feature' - A specific feature of the implementation to provide.*
- *'task' - Specific research to perform to provide input to answer a software design question or specific coding work to implement a portion of a feature.*
- *'bug' - An issue describing something that must change in the planning or the code.*

Start by creating "epic" issues that cover software design considerations. There should be an epic for each major feature required and for software technology choices. For each "epic", define "task" issues that describe what research is needed to help answer the epic, include software technology options, any initial pros and cons. Each "task" will be executed by an LLM or a human and the result will be a "design" section added to the task, as well as comments and implementation tasks will be created in order to consider the planning task complete.
The initial philosophy and choices for this application are as follows:

- *This application and all its dependencies must be Free Software or at least Open Source Software.*
- *All data is held locally. Use a single sqlite3 database to store all "memory" wherever possible even if options exist that may have higher performance. If sqlite3 is wholly unsuitable, consider alternatives.*
- *Implementation is in Python, managed by "uv" with Pytest unit tests of good coverage. UIs included: CLI based on Click with well formatted output using TBD module, TUI based on TBD module, GUI based on Qt, web based on Flask and TBD web framework. MCP based on TBD module. A simple text and JSON output also available.*
- *Information retrieval methods include: RAG for semantic search. Graph database for knowledge graph. Full text search with BM25 ranking. Each of these require research to flesh out their features, determine implementation methods, datatabase schema and how they may interoperate.*

Given this introduction, produce beads epic issues covering design considerations and dependent tasks covering research and other work needed toward producing a design specification.