An Agentic Planner for Constraint-Aware Scheduling: Edge-Optimised Design and Evaluation

Michael Sigamani

March 2025

Contents

1 Introduction **1**

2 System Architecture **2**

3 Agent Workflow and Graph Logic **2**

4 Prompt Engineering Strategy **2**

5 LangSmith Traceability **2**

6 Backend Support and Edge Compatibility **3**

7 Benchmarking and Evaluation **3**

7.1 LangSmith Evaluation View 3

7.2 W&B Performance Metrics 3

8 Conclusion **3**

1 Introduction

This document presents the design and implementation of a lightweight, edge-compatible agentic planning system tailored for constraint-aware scheduling tasks such as meeting coordination. Developed in response to a time-boxed technical assessment, the project showcases a focused design that avoids fine-tuning in favour of a prompt-engineered agent loop. The key goal was to create a functional planner capable of parsing scheduling constraints, reasoning over temporal overlaps, and proposing valid meeting times—all while maintaining high responsiveness and local deployability.

The system is implemented using `LangGraph` for orchestration, `MLC-LLM` for efficient local inference, and `LangSmith` and `Weights & Biases (W&B)` for traceability and benchmarking. Models used include `LLaMA 3 8B Instruct`, hosted on a remote CUDA-enabled machine via `Vast.ai`, and `Mistral 7B Instruct` locally compiled on macOS. A testing backend using OpenAI GPT-4 is also available.

The primary constraint was to avoid model fine-tuning and instead rely entirely on high-quality prompt engineering and architectural composition to achieve competitive planning performance.

2 System Architecture

The system operates as a simple yet expressive **LangGraph** application, leveraging an agentic loop between a `planner` and `validator` node. Each run of the graph takes a structured task input, invokes a language model to generate a proposed meeting plan, and evaluates whether this plan satisfies all scheduling constraints. If not, the planner retries with feedback for up to five iterations.

Backends supported include `MLC-LLM` for fast CUDA and Metal-based inference, `llama.cpp` for local subprocess execution, and OpenAI for fallback validation. The graph orchestration and loop logic are entirely declarative and tracked via LangSmith for traceability.

3 Agent Workflow and Graph Logic

The planning agent is composed of two core nodes: `PlannerNode` and `ValidatorNode`. The `PlannerNode` takes in a task description (e.g., "schedule a meeting") along with structured constraints (e.g., individual calendars) and produces a candidate plan in a strict text format. This plan is passed to the `ValidatorNode`, which evaluates whether it conflicts with any participant's schedule. If validation fails or ambiguity is detected, the graph loops back to the planner with an incremented retry counter. This feedback loop ensures iterative improvement and is bounded to terminate after a fixed number of attempts.

The full orchestration is implemented using `LangGraph`'s conditional edge mechanism, where the output of the validator determines the next step. If `validation_passed=True`, the graph terminates. Otherwise, it re-invokes the planner.

4 Prompt Engineering Strategy

The agentic planner is designed using a prompt-first methodology that eliminates the need for parameter fine-tuning. The prompts are tailored to induce structured reasoning over scheduling constraints and to yield outputs in a deterministic format.

The planning prompt is engineered to frame the task as one of "plan synthesis"—instructing the model to select the earliest time slot satisfying all constraints and avoiding existing meetings. It also incorporates examples and formatting constraints to increase compliance. The planner output must follow a template like:

```
Here is the proposed time:  Monday, 09:30 - 11:00
```

This guarantees that the validator and evaluation layers can reliably parse and assess outputs.

The validator prompt enforces exact match criteria: it compares the model's prediction against a golden plan, checking for exact time and day alignment, and returns a JSON object indicating correctness and reason. This strict feedback loop improves consistency and helps debug errors.

5 LangSmith Traceability

LangSmith enables full observability for every planner run. All nodes, outputs, and retry logic are recorded with timestamped traces, enabling fine-grained debugging and performance analysis.

A public example trace is available here: LangSmith Run Viewer.

6 Backend Support and Edge Compatibility

The system supports multiple backends:

- `LLaMA 3 8B Instruct` via `MLC-LLM`, deployed on CUDA (Vast.ai)
- `Mistral 7B Instruct` via local `llama.cpp` subprocess (macOS Metal)
- OpenAI GPT-4 (for comparison/testing)

These backends are pluggable via a single flag. MLC-LLM enables inference on edge devices with sub-2s latency, supporting batch mode and multi-threaded generation.

7 Benchmarking and Evaluation

Benchmarks were run on the calendar scheduling dataset. The benchmark harness (`run_benchmarks.py`) loads a series of JSONL test cases, feeds them through the LangGraph planner loop, and records both LangSmith traces and W&B metrics. Accuracy is evaluated using an auto-judger that checks whether the predicted plan exactly matches the golden plan.

7.1 LangSmith Evaluation View

LangSmith's structured view shows planner and validator attempts, retry counts, and validation paths. This reveals convergence patterns and failure modes.

7.2 W&B Performance Metrics

We track the following:

- Tokens used per input/output
- Time-to-first-token (TTFT)
- Total latency per benchmark item
- Tokens-per-second throughput
- Memory and CPU utilisation (macOS, no GPU)

These metrics validate the feasibility of local inference and provide guidance on prompt cost and system load.

8 Conclusion

This project successfully delivers a fast, accurate, and observable agentic planner built without fine-tuning. The use of LangGraph enables modular reasoning, while MLC-LLM brings efficient edge deployment. LangSmith and W&B provide strong observability and evaluation support.

The planner performs well on calendar tasks, producing valid outputs within strict formats. Prompt engineering strategies—especially few-shot examples and structured output constraints—proved effective at increasing reliability and determinism.

With more time, the system could be extended to handle travel planning, integrate with real-time APIs, and use OpenAPI schemas for constraint propagation. The backend abstraction is already in place for such scaling.

Appendix A: Known Risks and Tradeoffs

Though the system is robust, it is not without tradeoffs. The validator currently operates at the string level and may incorrectly score functionally equivalent plans as incorrect. Ambiguous or under-specified inputs may also lead the planner to hallucinate time slots or ignore soft constraints. Additionally, MLC-LLM lacks native tracing support, limiting real-time observability without LangGraph instrumentation.

Appendix B: Future Work

With more time, the following areas would be explored:

- Trip planning with constraint-aware routing using real-world travel APIs.
- Incorporation of OpenAPI tools and schemas for structured constraint evaluation.
- Dynamic backend switching to adapt to runtime resource availability.
- UI-based debugging via Streamlit or Gradio.