A Natural Language Copilot for Interactive Plan Space Exploration

Paul Horn[1], **Daniel Gnad**[1,2]

[1]Heidelberg University, Germany
[2]Linköping University, Sweden
paul.horn@stud.uni-heidelberg.de, daniel.gnad@uni-heidelberg.de

Abstract

Classical planning systems can produce, count, and reason about sets of valid solutions, but the resulting interfaces are typically command-line driven and require expert knowledge of the underlying formalism. This is a substantial barrier when non-experts need to inspect alternative plans, justify choices, or impose constraints on the solution space. We present `PlanCopilot`, a chat-based front end for the interactive plan space exploration tool `PlanPilot` that uses a Large Language Model (LLM) as a translator between natural language and the underlying *facet*-reasoning commands. The LLM is deliberately not asked to plan, count, or reason about solutions itself; it only orchestrates calls to the planner and explains the returned data. The behavior is controlled by a structured system prompt that defines facet state semantics, a catalog of *planning analysis questions*, and execution rules that prevent common reasoning errors. We evaluate the resulting system in a user study with ten participants on 12 plan-space analysis tasks in Blocksworld. Users solved *all* tasks with the copilot, but only $9.5/12$ on average with the original tool, while completing tasks roughly $3\times$ faster (16:12 vs. 45:07 minutes on average). The study suggests that delegating only the natural language layer to an LLM, while keeping all planning reasoning in a sound symbolic backend, is a practical recipe for usable planning assistants.

Introduction

Classical planning aims at finding a sequence of actions that transforms an initial state into a state satisfying a goal condition. Many real-world applications, however, do not stop at finding a single, possibly optimal, plan: users need to compare alternatives, understand which actions are forced, and forbid or enforce specific decisions to align plans with operational constraints (Boddy et al. 2005; Sohrabi et al. 2018). Quantitative and qualitative reasoning over the plan space has therefore received increasing attention (Speck et al. 2025; Gnad et al. 2025), and *facet* reasoning—originally introduced for answer set programming (Fichte, Gaggl, and Rusovac 2022)—was recently ported to classical planning to enable a step-by-step navigation of large plan spaces (Speck et al. 2025; Gnad et al. 2025). The `PlanPilot` tool of Gnad et al. (2025) exposes this functionality to end users via a textual interactive shell.

While powerful, the `PlanPilot` interface is also representative of a wider problem in planning tools: the syntax is terse (e.g., `+∼occurs(action((stack,a,b)),4)`), the available commands and their arguments must be memorized, and the output consists of raw ASP-style facts that are difficult to digest for users without prior planning background. This is unfortunate, because the core operations `PlanPilot` supports—"which actions are forced", "which decisions are most restrictive", "can I still solve the task if I forbid this action"—are exactly the questions a non-expert user would want to ask in plain English.

Large Language Models are an obvious candidate to bridge this gap, but they should not be asked to do the actual planning. Recent work has shown that LLMs are unreliable planners and frequently fabricate seemingly correct but invalid solutions (Valmeekam et al. 2023b,a; Stechly, Marquez, and Kambhampati 2023). At the same time, LLMs are very good at language: turning a user request into a structured query, summarizing tabular data, and explaining the meaning of a metric. This suggests a clean division of labor in which the LLM handles *only* the natural language interface and a sound symbolic backend handles *all* planning reasoning, in the spirit of recent "LLM modulo" proposals (Gestrin, Kuhlmann, and Seipp 2024; Silver et al. 2024).

Contribution. We instantiate this idea in a tool we call `PlanCopilot`, which couples `PlanPilot` with an LLM acting as a copilot.

1. We describe the integration: a single tool call lets the LLM forward `PlanPilot` commands and observe their textual output, while a structured system prompt fixes the semantics the LLM is allowed to assume about facets, plan states, and metrics.
2. We identify a catalog of recurring *planning analysis questions* that arise repeatedly in plan space exploration, together with deterministic execution recipes that map each question to a sequence of `PlanPilot` commands.
3. We report a user study with ten participants on 12 Blocksworld plan-space analysis tasks comparing the original `PlanPilot` to `PlanCopilot`. The copilot improves task success ($12/12$ vs. $9.5/12$) and reduces the median completion time per task by a factor of three.

This work is based on the Bachelor thesis by Paul Horn, which was conducted at Heidelberg University (Horn 2026).

The study additionally exposes the failure modes of the approach: rare hallucinations on numerical data, language mismatch when the user switches the chat language, and a class of confidently-wrong plan comparisons that the symbolic backend cannot prevent.

Background

Classical planning. We use the standard SAS$^+$ formulation (Bäckström and Nebel 1995; Helmert 2006). A planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{V}$ is a finite set of state variables, $\mathcal{A}$ a finite set of actions $a = \langle pre_a, \mathit{eff}_a \rangle$, and $\mathcal{I}$, $\mathcal{G}$ describe the initial and goal condition. A plan $\pi = \langle a_1, \ldots, a_n \rangle$ is a sequence of applicable actions transforming $\mathcal{I}$ into a goal state. Deciding plan existence is PSPACE-complete (Bylander 1994). The set of all plans of length at most ℓ is denoted $\mathrm{Plans}_\ell(\Pi)$, and we sometimes refer to it as the *plan space*.

Facets and `PlanPilot`. Gnad et al. (2025) introduced `PlanPilot`, an interactive plan-space exploration tool building on the multi-shot ASP solver `clingo` and the facet reasoning of Speck et al. (2025). A *facet* is an action that occurs in some but not all plans of $\mathrm{Plans}_\ell(\Pi)$ There are inclusive facets $\mathcal{F}^+$, activating such a facet restricts the plan space to plans containing that action, and excluding facets $\mathcal{F}^-$, activating which forbids the action. `PlanPilot` supports queries such as enumerating plans (`! n`), listing facets (`?`), counting plans (`#!`), counting facets (`#?`), and listing facets with restrictiveness metrics `#??`) (Gnad et al. 2025). Because $\mathrm{Plans}_\ell(\Pi)$ can be astronomical, even for small instances of Blocksworld, the design of `PlanPilot` is deliberately oriented towards reasoning about plans rather than enumerating them. This is reflected in the complexity of the corresponding problems, while enumerating $\mathrm{Plans}_\ell(\Pi)$ is #P-complete, facet reasoning is only NP-complete, which makes is practically usable (Speck, Mattmüller, and Nebel 2020; Speck et al. 2025).

LLMs in planning. Direct planning with LLMs has consistently underperformed even on small Blocksworld benchmarks (Valmeekam et al. 2023b,a; Stechly, Marquez, and Kambhampati 2023; Olmo, Sreedharan, and Kambhampati 2021). A more successful pattern is to keep the LLM out of the solving loop and use it for problem elicitation (Gestrin, Kuhlmann, and Seipp 2024), design of heuristics for state-space search (Corrêa, Pereira, and Seipp 2025), or, as we do here, for natural language explanation and orchestration. Our copilot is in this spirit: it never proposes a plan and never counts plans by itself.

The PlanCopilot

`PlanCopilot` is a thin wrapper around `PlanPilot`, implemented as an interactive command-line based chat interface. Concretely, the tool starts `Fast Downward` (Helmert 2006) as a backend translator on a given PDDL instance, then launches `PlanPilot` on the resulting SAS$^+$ task with a user-chosen horizon H. An LLM session is then opened with a fixed system prompt and an orchestrator, which forwards a string to `PlanPilot`'s standard input and returns its standard output. We use ChatGPT-5.4 from OpenAI as the underlying model; no fine-tuning is performed, and no other tools, like web search or file access, are exposed. `PlanCopilot` is publicly available.

The architecture intentionally constrains what the LLM can *do*: every observation about the plan space must come from a real `PlanPilot` response. The LLM is therefore forced to ground its answers in the planner output rather than generating plans or counts directly. This is critical, given the well-documented tendency of LLMs to hallucinate plausible but incorrect plans (Valmeekam et al. 2023b; Stechly, Marquez, and Kambhampati 2023).

System Prompt

The behavior of the copilot is mostly determined by its system prompt, which we organize around three types of rules.

Facet semantics. `PlanPilot` distinguishes three facet states: *Enforced* (action must occur), *Forbidden* (action must not occur), and *Not selected* (neutral). The natural-language vocabulary around these states is ambiguous (e.g., "activate" vs. "enforce" vs. "use"), and a naive LLM frequently confuses removal of a facet activation with the introduction of a enforcement. We address this in a dedicated rule that pairs each user intent with the exact `PlanPilot` commands required and additionally enumerates the case distinctions on the current state of the facet (e.g., "if the user requests deactivation and the action is currently enforced, then send `-FACET_ID`. A separate rule enforces that facet identifiers must be copied verbatim from a preceding facet query response (e.g. `?`, `#??`), forbidding the LLM to invent facets.

Metric interpretation. `PlanPilot`'s reasoning commands return numeric fields such as the fraction of plans removed when a facet is activated or the number of facets that remain modifiable when the facet is activated. We define the meaning of each field once in the system prompt, including worked examples that distinguish fractions (0.34 means 34% of plans) from absolute counts. Without this clarification, the LLM repeatedly mixed the values up.

Planning analysis questions. We identified a catalog of recurring questions and encoded each as a structured recipe. Each recipe declares the *intent*, a list of *example user requests* (so the LLM can pattern-match the surface form), a sequence of *required `PlanPilot` calls*, and a presentation rule. We include the following recipes:

- **Current plan** (`! 1`).
- **Decisions that reduce the solution / facet space.** Calls `#!!` or `#??` and ranks facets by the reduction of plans / facts after facet activation.
- **Add preferred actions.** For an wishlist of actions, repeatedly query `#??`, select the facet with the highest number of remaining facets, activate it, and continue until either all preferred actions are inserted or no compatible facet remains. Crucially, an action that already occurs in

https://potassco.org/clingo/

https://github.com/paulhorn0/PlanningCopilot

the plan after a higher-priority activation is counted as fulfilled even without an explicit activation.

- **Apply constraint.** List facets via `?`, deactivate every facet that violates the constraint, and call `! 1` to display a witness or report infeasibility.
- **Swap detection.** A swap is a temporary action that does not contribute to the goal but only enables another action. The recipe formalizes a domain-aware definition (an object a is moved at t_1, another object is manipulated at t_2, and a is moved again at t_3 to an irrelevant position) and references the *Domain Understanding* rule for the meaning of objects and locations.

The recipes are deliberately deterministic. A user request that matches a recipe is answered through the same fixed sequence of `PlanPilot` calls every time, which means that the variability of the LLM affects only the wording of the answer, not the content. Open-ended user requests are still possible, and the copilot is allowed to compose recipes (e.g., to apply a constraint and then identify the most restrictive remaining facet), but the catalog covers the operations users in our pilot interviews described as the most common.

Walkthrough

To illustrate the copilot, consider a Blocksworld instance with five blocks. The user starts the session and types "Please show me the current plan." The copilot recognizes the *Current plan* recipe, sends `! 1` to `PlanPilot`, and returns the resulting 16-step plan in a numbered list. The user follows up with "What is the most restrictive facet in the problem?". The copilot now invokes the *Decisions that reduce the solution space* recipe, sending `#!!` and `#??`, ranks facets by their impact, and reports `occurs(action((stack,b,e)),8)` as the most restrictive option, noting that activating it removes 99.80% of facets and only one plan remains. The user proceeds with "Are there any swaps in this plan?". The copilot now applies the swap-detection recipe and identifies a candidate swap of block e between steps 7 and 8, suggesting that this temporary relocation has no direct goal benefit. Throughout the session, the user does not type a single `PlanPilot` command, and every claim made by the copilot is backed by a real planner response.

Evaluation

We evaluate `PlanCopilot` along two axes: (i) does the copilot let users solve plan-space analysis tasks they would not otherwise solve, and (ii) how does it change task completion time and user experience?

Setup. We recruited ten unpaid volunteers; the study logged only task times and anonymous questionnaire responses, so no ethics/IRB approval was required. None had used `PlanPilot` and only one had heard of planning, while self-rated computer proficiency spanned the full 1–10 range, i.e., the sample is mixed in computer experience but uniformly novice in planning. The participants were divided into two groups of five. Group 1 first used `PlanCopilot` on a fixed set of twelve Blocksworld plan-space analysis tasks, then repeated them with `PlanPilot`; Group 2 used the opposite order. The tasks (Table 1, Appendix) cover the analysis question catalog above. Both tools were operated through the *same* interactive terminal chat with identical output formatting—the only difference is whether the user types raw commands or natural language—so the measured usability gap reflects the language layer, not the rendering. Before the study, all participants received the same five-minute introduction to planning and `PlanPilot` syntax. There was no automated failure-recovery mechanism: when an output was unclear, copilot users could ask follow-up questions in plain English, whereas `PlanPilot` users had to retry the command; either way a participant could skip a task at any time, counted as a failure. We measured wall-clock time per task and a post-study questionnaire on difficulty and usability; the full setup and results are in the thesis (Horn 2026).

Task success and time. Across all participants and tasks, users with copilot solved $12/12$ tasks while they solved $9.5/12$ (79.2%) on average with `PlanPilot`. Mean total time per participant dropped from $45{:}07$ min with `PlanPilot` to $16{:}12$ min with `PlanCopilot`; the medians of $46{:}07$ and $16{:}13$ min show that the difference is not driven by outliers. Per-task averages are $4{:}44$ min vs. $1{:}21$ min, i.e., a roughly threefold speed-up. Variance over participants is also smaller for the copilot ($18{:}02$ min) than for the bare tool ($26{:}21$ min), indicating that the LLM-driven interface is not only faster on average but also more consistent across users.

Learning effect. Both groups were faster on their second tool. Group 1, which saw the copilot first, took $20{:}14$ min on the copilot and $38{:}21$ min on `PlanPilot`; Group 2, which saw the opposite order, took $51{:}52$ min on `PlanPilot` and $12{:}11$ min on the copilot. The learning effect is substantial in both directions but does not change the relative ordering: every individual participant solved the copilot tasks faster than the `PlanPilot` tasks.

Subjective experience. On a 1–10 scale (10 = extremely easy), the copilot received ratings between 7 and 10 from all participants, while `PlanPilot` received ratings between 1 and 5. Asked whether a person without prior planning knowledge could use the tool, the copilot received no rating below 7, while `PlanPilot` received none above 3. Free-text feedback attributed the gap primarily to two factors: (i) the ability to ask follow-up questions in plain English when an output was unclear, and (ii) the elimination of the need to remember `PlanPilot`'s command syntax. Two participants nevertheless preferred the bare tool because of its lower latency (no LLM round-trip) and its more compact output, and one tester noted that "the LLM could be wrong".

Failure types. Despite the architectural separation between language and reasoning, three classes of errors remained. First, the LLM occasionally hallucinates numerical values: in our trace logs we observed one instance where it reported a value for facet-count reduction that was inconsistent with the corresponding `#?` output. Second, the

copilot sometimes preferred a non-optimal facet in the "activate the facet that leaves the most solutions" task, suggesting that the LLM is using a coarse heuristic on the listed facets rather than a strict argmax over the returned metrics. Third, when one tester switched to German, the LLM auto-translated parts of its own system prompt and mixed up the removal of an activation with the introduction of a deactivation. These observations suggest that even in a pure orchestration setting, the symbolic backend cannot fully prevent the LLM from fabricating wrong-but-plausible intermediate facts; lightweight cross-checks (e.g., re-issuing a query and comparing) would be a natural mitigation. These are intermediate slips, not task failures: each was rare and recoverable via a follow-up question, which is why all twelve tasks were solved with the copilot. The $2.5/12$ unsolved tasks all occurred with `PlanPilot`, driven by syntax errors and tool crashes rather than the language layer.

Discussion

Our results, while based on a small user study, are consistent with a simple lesson: when the planning task is intrinsically symbolic, an LLM is most useful as a *translator* between user intent and a sound backend, rather than as a reasoner over the plan space itself. This echoes both negative results on direct LLM planning (Valmeekam et al. 2023b; Stechly, Marquez, and Kambhampati 2023) and recent positive results on LLM-driven plan elicitation (Gestrin, Kuhlmann, and Seipp 2024) and on generalized planning with LLM priors (Silver et al. 2024). In our setting, every non-trivial claim about plan counts, restrictiveness, or solvability is computed by `clingo` via the `plasp` encoding of `PlanPilot`, and the LLM only chooses which queries to issue and how to phrase the answer.

The structured system prompt, with its catalog of *planning analysis questions*, played a larger role than we initially expected. Without the recipes, the LLM still reached correct answers most of the time, but it tended to re-derive the same query strategy from scratch on every session, occasionally with subtle errors (e.g., reading a count rather than a fraction). With the recipes, the LLM behaved much more deterministically; in particular, given two phrasings of the same question, the copilot now issued the same sequence of `PlanPilot` commands. This is encouraging from a reproducibility perspective and suggests that, beyond chain-of-thought style prompting, *decision-tree* prompting that lists exact tool sequences per intent class is a useful design for planning copilots.

Limitations. Our evaluation is restricted to Blocksworld and to ten non-expert participants; we do not yet know how the copilot behaves on more complex planning problems, such as the Airbus Beluga logistics tasks (Gnad et al. 2025) where the action vocabulary is larger and the natural-language description of objects (jigs, racks, hangars) is itself non-trivial. The error analysis also points to scenarios in which a self-check would help: re-issuing critical queries, or constraining the LLM to literally quote `PlanPilot` fields into the answer, are concrete and cheap improvements we plan to evaluate next. Our only baseline is `PlanPilot` without an LLM, matching our usability claim but not quantifying the correctness gain of the backend; an LLM-only ablation—a model answering the same questions without `PlanPilot`—would measure rather than assert that gain, and is the most important addition for a full-length version. Finally, our setup uses a closed model (ChatGPT-5.4); replicating the experiment with an open-weight small model is a natural next step.

Conclusion

We presented `PlanCopilot`, a chat front end for the interactive plan-space exploration tool `PlanPilot`. The system follows a strict separation of concerns: an LLM handles natural language understanding, intent recognition, and answer generation, while all plan-space reasoning is done by the existing symbolic backend. The behavior of the LLM is controlled by a system prompt that fixes facet semantics, metric interpretation, and a catalog of deterministic recipes for the most common planning analysis questions. In a user study with ten non-expert participants on twelve Blocksworld tasks, the copilot improved task success from 79.2% to 100% and reduced average completion time per task by roughly a factor of three. We see this as evidence that, for planning tools that are otherwise too syntactically demanding for end users, an LLM front end—deliberately deprived of any planning autonomy—is a practical and reliable usability layer.

Future Work

A natural next step is to replace the hand-designed interaction between the symbolic reasoner and the LLM with the Model Context Protocol (MCP) (Anthropic 2024). In the current design, every `PlanPilot` command, every facet rule, and every analysis recipe is curated by hand in a static system prompt, and the orchestrator exposes only a single "send raw input" tool. MCP would let us instead publish `PlanPilot`'s capabilities as a structured server: each reasoning command (`! n`, `?`, `#!`, `#??`, facet (de)activation) becomes a typed tool with a machine-readable schema, and the catalog of planning analysis questions can be advertised as MCP prompts or resources that any MCP-aware client discovers at runtime. This would lift the copilot from a bespoke wrapper around one tool to a general interface: any planner that exposes plan-space queries through an MCP server could be driven by the same chat front end, and conversely any MCP-aware assistant could orchestrate `PlanPilot` without prompt engineering. We expect this to reduce the maintenance burden of the system prompt, make the failure modes related to invented facets and mis-typed commands harder to trigger, and open the door to multi-tool sessions in which the LLM combines a planner, a domain-modeling assistant, and a visualization service through a uniform protocol. Relatedly, packaging the orchestration layer and the analysis-question catalog as a reusable *skill* wrapping `PlanPilot`, rather than a standalone application, would let any agent invoke plan-space exploration as a callable capability without a separate chat front end.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

2023. *Proceedings of the Thirty-Seventh Annual Conference on Neural Information Processing Systems (NeurIPS 2023)*.

Anthropic. 2024. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol. Accessed: 2026-05-09.

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS^+ Planning. *Computational Intelligence*, 11(4): 625–655.

Boddy, M.; Gohde, J.; Haigh, T.; and Harp, S. 2005. Course of Action Generation for Cyber Security Using Classical Planning. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 12–21. AAAI Press.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69(1–2): 165–204.

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code. In *Proceedings of the Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS 2025)*.

Fichte, J. K.; Gaggl, S. A.; and Rusovac, D. 2022. Rushing and Strolling among Answer Sets - Navigation Made Easy. In Honavar, V.; and Spaan, M., eds., *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, 5651–5659. AAAI Press.

Gestrin, E.; Kuhlmann, M.; and Seipp, J. 2024. NL2Plan: Robust LLM-Driven Planning from Minimal Text Descriptions. In *ICAPS 2024 Workshop on Human-Aware and Explainable Planning (HAXP)*.

Gnad, D.; Hecher, M.; Gaggl, S.; Rusovac, D.; Speck, D.; and Fichte, J. K. 2025. Interactive Exploration of Plan Spaces. In Ortiz, M.; Wassermann, R.; and Schaub, T., eds., *Proceedings of the Twenty-Second International Conference on Principles of Knowledge Representation and Reasoning (KR 2025)*. IJCAI Organization.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Horn, P. 2026. LLMs for textual plan space explanation in PlanPilot. https://doi.org/10.5281/zenodo.20700859.

Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2021. GPT3-to-plan: Extracting plans from text using GPT-3. arXiv:2106.07131 [cs.CL].

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J.; Pack Kaelbling, L.; and Katz, M. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In Dy, J.; and Natarajan, S., eds., *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*, 20256–20264. AAAI Press.

Sohrabi, S.; Riabov, A. V.; Katz, M.; and Udrea, O. 2018. An AI Planning Solution to Scenario Generation for Enterprise Risk Management. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 160–167. AAAI Press.

Speck, D.; Hecher, M.; Gnad, D.; Fichte, J. K.; and Corrêa, A. B. 2025. Counting and Reasoning with Plans. In Shah, J.; and Kolter, Z., eds., *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2025)*, 26688–26696. AAAI Press.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic Top-k Planning. In Conitzer, V.; and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, 9967–9974. AAAI Press.

Stechly, K.; Marquez, M.; and Kambhampati, S. 2023. GPT-4 Doesn't Know It's Wrong: An Analysis of Iterative Prompting for Reasoning Problems. arXiv:2310.12397 [cs.AI].

Valmeekam, K.; Marquez, M.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2023a. PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change. In (neu 2023), 38975–38987.

Valmeekam, K.; Marquez, M.; Sreedharan, S.; and Kambhampati, S. 2023b. On the Planning Abilities of Large Language Models - A Critical Investigation. In (neu 2023), 75993–76005.

Appendix

Evaluation tasks. Table 1 lists the twelve tasks that every participant solved with both tools, in the order they were presented. All tasks operate on the `blocks 5-0` instance with horizon 14, exact encoding, and concrete time steps. The tasks were read out by the supervisor from a fixed sheet and progress in difficulty from loading the instance and displaying a plan to multi-step facet manipulation and constraint application, thereby exercising the full catalog of planning analysis questions of the main text.

System prompt. The complete system prompt is part of the public repository and is reproduced in full in the Bachelor thesis (Horn 2026). It is organized into five blocks: (i) a *command reference* that lists every `PlanPilot` command (`! n`, `?`, `#!`, `#?`, `#??`, `#!!`, facet (de)activation) with its exact syntax; (ii) *facet-state semantics* fixing the meaning of the *enforced*, *forbidden*, and *not selected* states and the case distinctions for switching between them; (iii) *metric definitions* for the numeric fields returned by the reasoning commands (the `reduction` and `remaining` fields over solutions and facets, each for the positive and negative case), with worked examples distinguishing fractions from counts; (iv) a *state-consistency rule* requiring facet identifiers to be copied verbatim from a preceding query and forbidding the LLM to invent plans, facets, or commands; and (v) the *planning analysis question* recipes of the main text, each pairing an intent with example phrasings, a fixed sequence of `PlanPilot` calls, and a presentation rule.

#	Task (as given to participants)	Analysis question
1	Load `blocks 5-0`, horizon 14, exact encoding, concrete time steps	Setup
2	Show the first plan	Current plan
3	Find swaps (actions only done to free space) in the plan	Swap detection
4	Show the second plan; what differs from the first?	Enumeration / comparison
5	Find alternatives to the current facet at time step 7	Alternative actions
6	Activate the facet in step 7 that leaves the most solutions	Most-permissive facet
7	How many facets remain modifiable now?	Facet count
8	Deactivate the facet activated in step 6	Facet deactivation
9	How many plans remain?	Plan count
10	Find a plan with a prioritized wishlist of five facets	Add preferred actions
11	Forbid `stack(e,c)` and `stack(c,e)` entirely	Apply constraint
12	How many plans, modifiable facets, and landmarks remain?	Combined metrics

Table 1: The twelve plan-space analysis tasks used in the user study.

https://github.com/paulhorn0/PlanningCopilot