The Curious Case of Planning for Unreliable Agents: Challenges and Opportunities in Orchestrating Generative AI Agents

Roya Daneshi[1], **Sunandita Patra**[2], **Kshama Dwarakanath**[2], **Sriram Gopalakrishnan**[2], **Daniel Borrajo**[2], **Sarath Sreedharan**[1]

[1]Colorado State University
[2]J.P. Morgan AI Research
roya.daneshi@colostate.edu, sunandita.patra@jpmchase.com, kshama.dwarakanath@jpmorgan.com, sriram.gopalakrishnan@jpmchase.com, daniel.borrajo@jpmorgan.com, sarath.sreedharan@colostate.edu

Abstract

This paper looks at the plausibility and challenges of applying planning techniques when orchestrating different LLM-based agents towards an overarching goal. We will look at existing planning community literature that we can already leverage towards this effort, and then consider novel challenges and opportunities raised by the setting. We finally end the paper with a demonstration leveraging an existing symbolic planner to perform travel planning. The demonstration is implemented within the Google Agent Development toolkit over LLM-agents implemented using the Gemma4-31B model.

Code — https://github.com/royadaneshi/llm-orchestration

Introduction

Large-language models (LLMs) are proving to be transformational, not just in different subfields of AI, but also in many industries. While state-of-the-art models remain limited in generating long-horizon plans, researchers are exploring ways to leverage them to improve planning. These include efforts at leveraging these models to acquire planning models (Guan et al. 2023) or to synthesize domain-specific heuristics (Tuisov, Vernik, and Shleyfman 2025). In this paper, we focus on one such opportunity that has received relatively less attention, namely, the use of planners to orchestrate the activities of smaller specialized LLM or Gen-AI agents towards a larger overall goal. In other words, how do we plan in domains where each action corresponds to a call to an LLM-based agent?

To illustrate the opportunity, let us look at a simple scenario where you are generating a travel plan using a PDDL planner to attend a conference. The PDDL plan may operate at a fairly high level of abstraction, relying on information about the current location, destination, required mode of transportation, and places to visit. Here, each action, like booking a flight or a hotel, could be passed to an LLM-based agent. Such an agent could search through booking websites to identify which flight/hotel to book. It may also work at a more detailed level of abstraction, and may have access to information that the original model did not, such as the total budget for the trip and information, such as the check-in time for the hotel. This information could have an impact on further actions, such as which flight it books, and the time the taxi should drop you off at the lobby.

Now, one might look at the previous example and think, "This is just another example of web service composition, something the planning community has studied in the past". While many of the discussions we present here share some common challenges with planning web service composition, or, for that matter, planning over any complex action formulations (including programs as actions), the use of LLMs does raise some unique opportunities and challenges. First off, *the ability to operate over partially specified action parameters*. As with the example, for a limited problem, an LLM could infer missing but required information, such as flight time and carrier, from context, previous information about the user, and, as we will see in our case study, future actions. Second, *they require minimal engineering effort to deploy and reuse*. As we will see in our case study, a single agent could be used for booking flights, taxis, and hotels by simply changing the input provided to it as text. Again, in many cases, one could use off-the-shelf LLMs and integrate them with various tools to develop surprisingly effective systems. Finally, *they are extremely adaptable*. An LLM-based agent might be relatively robust to changes in the API and even in the PDDL representation, thereby further improving its reusability. More interestingly, an LLM agent in the face of failure could suggest changes to the upstream planning models that could help avoid it in the future or recover from it. We believe this represents a closed-loop planning-execution paradigm that has not been investigated before.

In the rest of the paper, we will cover related planning/execution paradigms, point out important ideas we can leverage in this work, and highlight salient differences. Next, we will outline what we see as pressing challenges and opportunities for this class of problems. Finally, we will conclude the paper with a case study of our implementation of this planning paradigm in the travel planning domain.

Related Work and Terminology

This section defines our terminology, surveys relevant prior work, and highlights their key similarities and differences.

Terminology: We use the term **agent** in the sense it is usually used in standard AI textbooks (cf. (Russell, Norvig et al.

2016)), namely an entity that can perceive its environment through sensors and act upon that environment through actuators. The definition captures all existing AI systems, including LLMs. However, when we use the term **LLM-agent**, rather than an isolated language model, we will use it to refer to hybrid systems in which the LLM can call external tools and, by extension, execute complex operations (Schick et al. 2023). In practice, calling an LLM agent to carry out a task would involve some level of reasoning by the agent to identify which tools to call and in what order. As such, these agents aren't purely performing execution, but involve a certain level of reasoning or planning. However, we do not recommend using agents for long-horizon tasks; instead, we recommend using a **planner** to determine how to invoke different LLM agents in sequence to achieve the overall goal. Effectively, this would involve using a planning model that includes an action for each possible agent or a specific call to an agent. This operation has also been referred to as agent orchestration or simply orchestration (Wu et al. 2024). While we will primarily focus on various flavors of symbolic planners, this architecture could be adapted to other methods.

Hierarchical Task Networks One conceptually clear way to view LLM-agents is to see them as temporally abstracted actions that, for a given objective (usually encoded in the PDDL action definition and the rest of the plan), identify the sequence of more granular actions that achieve it. In current architectures, this would correspond to calls to various tools it can access (Yao et al. 2022). As such, a useful counterpart to existing planning literature would be a Hierarchical Task Networks (HTNs) (Erol, Hendler, and Nau 1998), and the problem of identifying the right refinement for a given subtask. However, our problem differs from HTNs in some key regards. First off, the central planner may not have complete control over the exact refinement the LLM-agent chooses. This partly comes from the fact that LLM-agents are inherently non-deterministic and noisy, and secondly, you wouldn't want the planner to enumerate all possible action sequences the LLM-agent can perform in its representation. Such an exhaustive representation may be infeasible, and, even when possible, might render the planner ineffective due to the complexity of the planning problem. Even in our simple example, enumerating all available carriers and their times, and potential layovers would result in a very large planning model. Secondly, in most cases, identifying the exact refinement for the LLM agent would involve execution, as it would rely on specific output from its tools to determine the next step. In our example, the LLM needs to query the API to receive the list of flights available before deciding which one to book. Such an execution-refinement loop may be anathema to existing HTN planners. However, the ability to perform some action, especially an information-gathering action, before deciding the rest of the actions could give the system substantial flexibility. However, as we will see in the next section, it also comes at the cost of making the planning problem more complex.

Planning-Execution-Replanning Loop Given this close interplay between planning and execution, one could look at existing paradigms for both. While a majority of planning literature does not concern itself with execution, planning+execution has been studied extensively in fields such as robotics (Martín et al. 2021) and space exploration (Chien et al. 2000), where researchers have examined assumptions about atomic actions, strict execution semantics, and perfect/partial observability, among others. The most common paradigm in these works is a three-layered architecture to execute, observe, monitor, and replan (Wertheim and Brafman 2025; Martín et al. 2021). Other approaches include integrated acting and planning algorithms (RAE+UPOM) (Patra et al. 2021), RMPL (Kim, Williams, and Abramson 2001), OPMAS (Turi, Bit-Monnot, and Ingrand 2023) and planning with behavior trees (Colledanchise, Almeida, and Ögren 2019).What distinguishes our approach is not the treatment of these well-studied aspects of planning and execution, but rather the idea of not predefining how actions are will be executed by some fixed code. Our framework covers the known aspects of planning and execution in a flexible way, one that does not require predefining how planning actions will be mapped to low-level execution.

Task and Motion Planning Another context in which planning operates at a higher level of abstraction than the final execution is in the context of task and motion planning for robotics (Mansouri, Pecora, and Schüller 2021). In a typical task and motion planning problem, while the planner operates on the level of grasping and placing objects, the low-level planner will need to account for details like grasp position and gripper angles. Similar to our case, these settings also do not rely on fully enumerating all possible refinements. Some works in this direction have also tried to formalize the planning level representation of the problem as a non-deterministic one (Srivastava, Russell, and Pinto 2016). However, most existing task-and-motion planning problems still assume that the refinement can be performed at the planning level. This requirement comes from a very fundamental fact related to planning with abstract actions, namely, one cannot identify the refinement of an abstract action in isolation. The choices made to achieve the outcomes of the current action not only depend on the past, but also on how future actions would build on it. For example, the angle by which an object is grasped and picked up might determine what the robot could do with it in future steps. However, the use of LLM agents allows us to circumvent this problem to a degree, as one could pass information about future actions and agents/tools used for those actions as inputs to the current agent. There is also the question of whether downward refinements are available for each plan (Knoblock 1994), wherein certain plans from the abstract model cannot be realized in terms of low-level actions. While the effectiveness of such a method depends on both the model capability and the complexity of the task, our initial results show that LLM-agents are capable of such context-driven refinements.

Webservice-Composition and Planning for Workflows A closely related problem space is that of planning for web service composition. Our example can, in theory, be captured as a series of calls to the web service APIs. While earlier works have looked at mapping such problems into planning problems, they have generally relied on carefully

crafted PDDL models that exactly capture the different API interfaces and how they relate to each other (Camacho et al. 2005; Hoffmann et al. 2007). In our framework, the LLMs should be capable of translating content between different representations used by different API services A closely related problem is also the use of planners to capture business process workflows and, in general, perform business process automation (Marrella 2019).Existing solutions to these problems again rely on handwritten planning models.

Planning With LLMs While a number of studies have shown LLMs' inability to plan effectively (Valmeekam et al. 2023), there is also a body of work examining the incorporation of LLMs into planning workflows. Firstly, there are works that have tried to use LLMs directly as planners. While still lagging behind existing planners, works have shown that the performance of these planners can be improved by fine-tuning the models (Verma et al. 2025; Pallagani et al. 2022). There are also planning works that propose using LLM-based planners alongside validators, with feedback from the validators used to further improve the performance of the LLM output (Kambhampati et al. 2024). Next, there are works that have tried to use LLMs as part of the model acquisition pipeline (Guan et al. 2023; Liu et al. 2023). In a related strain of work, LLMs have also been used to learn domain-dependent heuristics (Tuisov, Vernik, and Shleyfman 2025), and successor functions (Katz et al. 2024). Finally, there are works that have looked at the use of LLMs during execution and execution monitoring (Borrajo et al. 2025). However, in this paper, we are interested in settings where the LLM agent is free to identify the individual steps to achieve the specific task assigned to it by the planner.

Multi-agent Planning Finally, if we were to look at this setting from a multi-agent planning perspective, we would see that the proposed method would live somewhere between a purely centralized planning-based method, such as DEC-POMDP (Bernstein et al. 2002), and a purely decentralized game-theoretic formulation (Shoham and Leyton-Brown 2008). In DEC-POMDP, the policy for each agent is determined centrally, and the agents execute it independently. However, in our case, each agent independently identifies its policy, with the centralized planner imposing constraints on the kinds of behaviors each agent can exhibit. Here, the actions of the individual agents aren't fully determined by the central planner. As such, from a game-theoretic perspective, one could view the role of the planner as being one of mechanism design (Parkes 2004) to support coordination between a set of agents that may not be rational. Additionally, one could borrow ideas from multi-agent literature to further improve the performance of the overall system, such as imbuing each agent with the ability to communicate with other agents (Oguntola 2025), and allowing the agents to maintain models about other agents' capabilities. Finally, outside of these specific decision-theoretic and game-theoretic frameworks, multi-agent planning has also been looked at in the context of symbolic planning contexts, supported through languages like MA-STRIPS (Brafman and Domshlak 2008). These works also present valuable sources of insights that can be applied in the current setting, including notions like private actions and being able to communicate planning information.

Challenges and Opportunities

With connections to existing literature established, we revisit the challenges and opportunities opened by the new paradigm of integrating classical planning with LLM agents.

Planning for LLM-Agent Orchestration The first opportunity, hinted at earlier, is a hierarchical planning procedure that can identify refinements in isolation and on the fly. Such a procedure would require each agent to have access to the overall plan, the current action (assigned to it), and the execution history (or a Markovian state). It will also need access to the capabilities of the other agents that will carry out future actions, so it can choose compatible refinements. While in the case study, we demonstrate how existing LLM models can accomplish this for simple tasks, supporting such refinements for more complex tasks remains an open question. Another question is what information each agent needs about the overall task and the plan, and how to communicate it to the agents. Possibilities here range from causal links involving the current action to the complete plan model. Another question is how to communicate other agents' capabilities. While most existing LLM-agent orchestration frameworks require some mechanism to describe agent capabilities (in terms of inputs, outputs, and tools used), it remains an open question whether such descriptions are sufficient, especially when provided in ambiguous natural language.

Noisy Observation and State Estimation One central challenge in this paradigm is relying on agents to determine the current state of the world. Whether it's from the output of an agent that executed an action, or from a monitoring agent responsible for identifying the current state by performing different actions. However, all of these agents could make mistakes and give a wrong estimate of the state. A key component of any planning and replanning loop is the ability to identify the need to replan. While one could employ planning formalisms that support partial observability, such planners might be relatively inefficient and might not scale to the problems we are interested in solving with the proposed method. Because LLM responses are open-ended and exhibit semantic uncertainty rather than discrete probabilistic observations, this uncertainty cannot be readily captured by the fixed observation models assumed in traditional planning over stochastic actions. However, one could ask whether we could trade away some of the guarantees while still using efficient classical planning methods to handle such cases. This could involve developing compilations that generate plans robust to action failures. Which means even if some action failures are overlooked, the overall goal may still be achieved. We will also need to develop novel execution monitoring techniques that leverage relationships among action outcomes to identify potential failures.

Supporting Non-Determinism of Agent Behavior Another challenge would be supporting the non-determinism of agent behavior in the abstract planning representation. While one might think we could leverage relatively popular modeling techniques like FOND (Muise, McIlraith, and

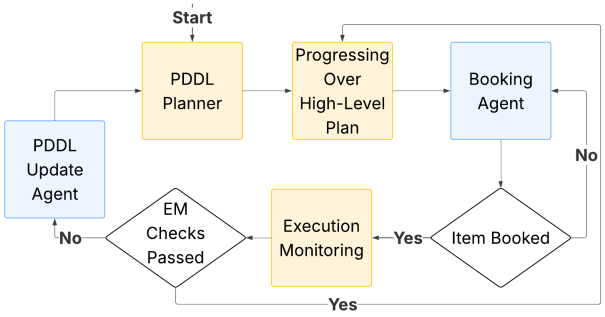

Figure 1: The architecture used for the travel booking case study. Blue boxes represent LLM-based agents, and yellow boxes represent Python modules.

Beck 2024) to capture this, there are some fundamental challenges here. There are two forms of non-determinism we have to deal with. One that comes from the probabilistic nature of the agents and from unexpected errors that might occur. The second one comes from the fact that the agent is free to choose what outcomes it can generate by selecting different refinements. The latter is usually referred to as angelic non-determinism (Srivastava, Russell, and Pinto 2016), and FOND-based planners cannot capture it directly.

Planning to Allow for Replanning One fact we can be quite confident about within this framework is that agents are going to eventually fail and even produce incorrect outcomes. This would be in addition to cases where the agents may fail to produce the correct outcome due to external factors (such as there being no flights available at the planned time). As such, when planning, we need to allow for the possibility that there will be a need to replan. Solutions here could include planning to avoid getting into dead-end states, and even pre-computing conditions during planning that will allow us to detect cases from where the plan can recover or replan for more effective paths (Borrajo and Veloso 2021).

Design and Refinement of the Planning Representation The choice of abstract planning representation is also central to the success of this paradigm. As discussed, it needs to be detailed enough to support efficient task decomposition between the agents, while still being sufficiently concise to be solved effectively. While this has always been a challenge in automated planning, using an LLM-based agent for plan execution provides us with a unique opportunity that has not been considered before. Namely, the fact that the agent, when executing its plan, can propose changes to the overall planning representation that can support more sufficient planning. LLM-based methods are already being used to design and extract PDDL models (Guan et al. 2023). Here, as part of the agent's actions, we could have the agent propose to the planner to model additional information, which it has access to, if it believes that this information is necessary for the successful completion of the plan.

Case-Study

In this section, we present our case study architecture (Figure 1) that integrates PDDL planning into the agents' execution loop to enable dynamic replanning and failure recovery.

System Architecture The orchestration process begins by progressing over a higher-level plan, which operates at a higher level of abstraction than the information handled by individual agents. For each booking action in the plan, we employ a general booking agent designed to execute the corresponding booking request, regardless of the action type or parameters. This modularity allows the system to scale to plans with an arbitrary number of components. Following each booking operation, an Execution Monitoring module validates the outcome to identify failures and initiate the appropriate recovery mechanisms as needed.

Plan-Level Action Execution The Booking Agent invokes an API to retrieve available items and their detailed attributes (e.g., flight routes, and hotel costs), which provide a more granular representation than the symbolic PDDL model. Given these details, causal links at the plan level, and the specific booking request, the agent has the discretion to select an item that it considers a better fit for the defined budget. As mentioned earlier, failures in LLM-based agents are inevitable; however, increasing the agent's degree of freedom may increase the likelihood of the error types discussed below. Our work focuses on identifying these failure types and applying appropriate, type-specific recovery strategies.

Failure Taxonomy We categorize agent failures into two types based on their impact on the orchestration state. Type I errors are execution-level failures and are resolved through simple retries. Type II errors are more critical plan inconsistencies. In these cases, an agent successfully executes an action that violates the plan requirements. These include Contradiction errors, which mainly arise from discrepancies between the abstract PDDL model and granular execution-time information (see Appendix A.1), and Temporal Infeasibility errors, which violate schedule constraints. A detailed description of these failures is provided in Appendix A.2.

Execution Monitoring and System Recovery To handle more complex Type II errors, we introduce an Execution Monitoring module that audits the system state after each booking agent execution to detect plan deviations and temporal conflicts. Upon detection of an error, the module triggers a replanning loop that first synchronizes the PDDL state using the PDDL Update Agent (see Appendix A.3) and then generates a corrected plan to reach the goal. Detailed recovery logic for each error subtype is provided in Appendix A.4.

Implementation details The system was implemented using the Gemma4-31B model and the Fast Downward planner. Hardware specifications and hyperparameter configurations are provided in Appendix A.5.

Conclusion

The goal of this paper was to highlight the challenges and opportunities related to a relatively understudied problem within the area of subarea of leveraging LLMs for planning problems. While the implementation demonstrates the feasibility of the approach, it does not begin to address any of the main challenges we have highlighted in the paper. We hope that the paper will help motivate more researchers to consider tackling the challenges posed by the problem setting.

Disclaimer

References

Bernstein, D. S.; Givan, R.; Immerman, N.; and Zilberstein, S. 2002. The complexity of decentralized control of Markov decision processes. *Mathematics of operations research*, 27(4): 819–840.

Borrajo, D.; Canonaco, G.; de la Rosa, T.; Garrachón, A.; Gopalakrishnan, S.; Kaur, S.; Morales, M.; Patra, S.; Pozanco, A.; Ramani, K.; et al. 2025. Genplanx. generation of plans and execution. *arXiv preprint arXiv:2506.10897*.

Borrajo, D.; and Veloso, M. 2021. Computing opportunities to augment plans for novel replanning during execution. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 51–55.

Brafman, R. I.; and Domshlak, C. 2008. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *ICAPS*, volume 8, 28–35.

Camacho, D.; Aler, R.; Borrajo, D.; and Molina, J. M. 2005. A Multi-Agent Architecture for Intelligent Gathering Systems. *AI Communications*, 18(1): 15–32.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; et al. 2000. ASPEN-Automating space mission operations using automated planning and scheduling. In *SpaceOps 2000*. AIAA Press.

Colledanchise, M.; Almeida, D.; and Ögren, P. 2019. Towards blended reactive planning and acting using behavior trees. In *2019 international conference on robotics and automation (ICRA)*, 8839–8845. IEEE.

Erol, K.; Hendler, J.; and Nau, D. S. 1998. Semantics for HTN planning.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36: 79081–79094.

Hoffmann, J.; Bertoli, P.; Pistore, M.; et al. 2007. Web service composition as planning, revisited: In between background theories and initial state uncertainty. In *AAAI*, volume 7, 1013–1018.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Bhambri, S.; Saldyt, L. P.; and Murthy, A. B. 2024. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In *Forty-first International Conference on Machine Learning*.

Katz, M.; Kokel, H.; Srinivas, K.; and Sohrabi, S. 2024. Thought of search: Planning with language models through the lens of efficiency. *Advances in Neural Information Processing Systems*, 37: 138491–138568.

Kim, P.; Williams, B. C.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *IJCAI*, 487–493. Citeseer.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial intelligence*, 68(2): 243–302.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*.

Mansouri, M.; Pecora, F.; and Schüller, P. 2021. Combining task and motion planning: Challenges and guidelines. *Frontiers in Robotics and AI*, 8: 637888.

Marrella, A. 2019. Automated planning for business process management. *Journal on data semantics*, 8(2): 79–98.

Martín, F.; Clavero, J. G.; Matellán, V.; and Rodríguez, F. J. 2021. Plansys2: A planning system framework for ros2. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9742–9749. IEEE.

Muise, C.; McIlraith, S. A.; and Beck, J. C. 2024. PRP rebooted: Advancing the state of the art in FOND planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20212–20221.

Oguntola, I. 2025. *Theory of Mind in Multi-Agent Systems*. Ph.D. thesis, Carnegie Mellon University.

Pallagani, V.; Muppasani, B.; Murugesan, K.; Rossi, F.; Horesh, L.; Srivastava, B.; Fabiano, F.; and Loreggia, A. 2022. Plansformer: Generating symbolic plans using transformers. *arXiv preprint arXiv:2212.08681*.

Parkes, D. C. 2004. On learnable mechanism design. In *Collectives and the design of complex systems*, 107–131. Springer.

Patra, S.; Mason, J.; Ghallab, M.; Nau, D.; and Traverso, P. 2021. Deliberative acting, planning and learning with hierarchical operational models. *Artificial Intelligence*, 299: 103523.

Russell, S. J.; Norvig, P.; et al. 2016. Artificial intelligence: a modern approach.

Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Hambro, E.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36: 68539–68551.

Shoham, Y.; and Leyton-Brown, K. 2008. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press.

Srivastava, S.; Russell, S.; and Pinto, A. 2016. Metaphysics of planning domain descriptions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Tuisov, A.; Vernik, Y.; and Shleyfman, A. 2025. LLM-Generated Heuristics for AI Planning: Do We Even Need Domain-Independence Anymore? *arXiv preprint arXiv:2501.18784*.

Turi, J.; Bit-Monnot, A.; and Ingrand, F. 2023. Enhancing operational deliberation in a refinement acting engine with continuous planning. In *Integrated Acting, Planning and Execution (IntEx), ICAPS'23 Workshop*.

Valmeekam, K.; Marquez, M.; Sreedharan, S.; and Kambhampati, S. 2023. On the planning abilities of large language models-a critical investigation. *Advances in neural information processing systems*, 36: 75993–76005.

Verma, P.; La, N.; Favier, A.; Mishra, S.; and Shah, J. A. 2025. Teaching llms to plan: Logical chain-of-thought instruction tuning for symbolic planning. *arXiv preprint arXiv:2509.13351*.

Wertheim, O.; and Brafman, R. I. 2025. Model-based AI planning and execution platforms for robotics. *AI Magazine*, 46(3): e70034.

Wu, Q.; Bansal, G.; Zhang, J.; Wu, Y.; Li, B.; Zhu, E.; Jiang, L.; Zhang, X.; Zhang, S.; Liu, J.; et al. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First conference on language modeling*.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

A Extended Case Study Details

A.1 Decision Making under External Information Discrepancy

The decision-making process of the booking agent is based on both external information that is not fully captured in the PDDL representation and the causal links at the PDDL level, introducing a discrepancy between the modeled state and execution-time reasoning. This discrepancy is one of the main causes of contradiction error subtype. In our case study, we provide the booking agent with additional information about the available budget, which is not represented in the PDDL model. At each step, the booking agent must determine whether booking the requested item is feasible given the current budget. If it is not feasible, the agent selects an alternative item of the same type that is within the budget constraints. This ensures that the booked items remain within the budget limits. In some cases, budget constraints cannot be satisfied even with the available alternatives, which are not addressed in our current implementation. A potential recovery strategy in such situations involves recursively backtracking over previous booking actions by undoing them to restore budget flexibility. The system would then update the PDDL problem model to reflect the revised constraints and trigger replanning. This mechanism addresses the violation of the downward refinement property. However, this extension remains a direction for future work.

A.2 Detailed Taxonomy of Agent Failures

Type I errors occur when the booking agent fails to book any item of the requested type. This may result from hallucinated bookings, exceeding chain-of-thought token limits, or failure to select an appropriate item from the available options. These errors are non-fatal and can typically be resolved by retrying the booking process. Type II errors comprise two subtypes. The first occurs when the booking agent deviates from the specified item in the plan and books a different item. We refer to this as a Contradiction error, as it reflects a mismatch between the planned item and the item actually booked. The second subtype arises when booked items are temporally incompatible, such as overlapping sequential actions or incorrect ordering (e.g., a taxi arrives at the hotel at 7 p.m., while check-in is scheduled for 3 p.m.), resulting in Temporal Infeasibility errors.

A.3 Dynamic PDDL State Synchronization

While replanning enables recovery from execution errors, its success relies on a continuously updated PDDL problem model that reflects all confirmed bookings. To maintain this model with newly booked items, we employ another LLM-based agent, referred to as the PDDL Update Agent, which updates the PDDL problem model. This agent identifies the predicates affected by each booking and outputs them with appropriate parameters, allowing us to update the problem model accordingly and prevent the planner from reconsidering booked items. While these predicates could theoretically be extracted from the PDDL model, a fixed mechanism would lack the flexibility to handle diverse modeling formulations. Leveraging an LLM-based agent instead allows for a more generalizable and robust inference of these dependencies.

A.4 Implementation Details of System Recovery Mechanisms

The Execution Monitoring module resolves Type II errors by applying specific recovery procedures based on the error subtype. This ensures that the replanning process is as surgical as possible, preserving successful bookings while correcting deviations. The replanning phase varies depending on the subtype of error. In the case of a Contradiction error, the system fixes the already booked items in the PDDL problem model and reruns the planner to generate a new plan for the remaining actions. This ensures that subsequent bookings remain consistent with the previously booked items. In the case of a Temporal Infeasibility error, the system removes the last booked item that caused the temporal conflict, retains the other valid bookings, and reruns the planner to generate a new feasible plan. This replanning mechanism enables the system to recover from potential dead-end states by dynamically adjusting its strategy. By generating a newly optimized path, the execution loop can overcome agent-induced errors while maintaining overall task integrity.

A.5 System Setup Details

The system was developed using the Google Agent Development toolkit (ADK) framework, within which we defined two distinct LLM-based agents. We employed Gemma4-31B as the backbone model, deployed via the Ollama service on a single NVIDIA H100 GPU. In this architecture, the agents orchestrate a set of APIs and navigate complex task environments in pursuit of their defined objectives. We used Fast Downward as the PDDL planner. To mitigate stochasticity and ensure reproducibility of agent behavior, we set the LLM temperature to 0.0. Based on our implementation experience, a context window constraint of 1000 tokens per inference turn provided sufficient "scratchpad" space for Chain-of-Thought (CoT) reasoning. In our implementation, any instance in which an agent exceeds this threshold is treated as a divergence failure, typically characterized by repetitive reasoning loops or an inability to converge on an API call. While such reasoning-loop failures may arise due to the stochastic and autoregressive nature of the agents, they are considered a secondary concern in our setting. Nevertheless, we calibrate the token limit to minimize their occurrence. We do not explore alternative mitigation strategies for these loops, as such interventions fall outside the scope of this study, and are considered a limitation of the current system.