# A Requirements Engineering-Driven Methodology for Planning Domain Generation via LLMs with Invariant-Based Refinement

**Angelo Casciani**[1], **Giuseppe De Giacomo**[1, 2], **Andrea Marrella**[1], **Christoph Weinhuber**[2]

[1]Sapienza University of Rome, Italy
[2]University of Oxford, United Kingdom
{casciani, marrella}@diag.uniroma1.it, {giuseppe.degiacomo, christoph.weinhuber}@cs.ox.ac.uk

**Abstract**

The creation of planning models remains a labor-intensive and error-prone bottleneck in the deployment of Automated Planning solutions in real-world environments. Recent approaches have explored the use of Large Language Models (LLMs) as modelers to generate PDDL specifications from natural language. However, most rely on single-shot generation or limited feedback loops, often failing to enforce semantic correctness and common-sense coherence. In this paper, we present a structured, multi-stage methodology inspired by principles from Requirements Engineering for generating PDDL models. Our approach leverages scenarios and user stories to elicit domain and initial state, which are incrementally refined via syntax validation, state progression through plan execution, and invariants analysis to detect and correct modeling flaws. Indeed, the method's novelty lies in the integration of automated invariant extraction and LLM-guided model refinement, which significantly improves the semantic robustness and realism of the generated domains. We assume a humanoid agent in the environment, enforcing real-world constraints such as limited hand capacity, and we evaluate our methodology on a synthetic dataset of room environments, using both expert judgment and an LLM-as-a-judge approach. Results show that the generated planning domains are syntactically correct, semantically sound, and actionable, enabling real-world deployment in physical agents.

## 1 Introduction

*Automated Planning* is a branch of Artificial Intelligence (AI) focused on generating a sequence of actions to achieve a desired goal. Specifically, *Classical Planning* assumes a deterministic and fully observable environment, with a known initial state, a finite set of possible actions with preconditions and effects, a clearly defined goal, and a single agent. The planner searches for a sequence of actions (i.e., a *plan*) that leads from the initial state to a goal state while adhering to the constraints imposed by the specification.

The de-facto standard language for this, the *Planning Domain Definition Language* (PDDL) (Ghallab et al. 1998), represents planning tasks using two files: a *domain* that defines the general rules and actions of an environment, and a *problem* that describes a specific instance with an initial state and a goal. Generating PDDL models remains an intensive and error-prone task, representing a recognized bottleneck for the adoption of planning technologies in real-world applications (McCluskey, Vaquero, and Vallati 2017).

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

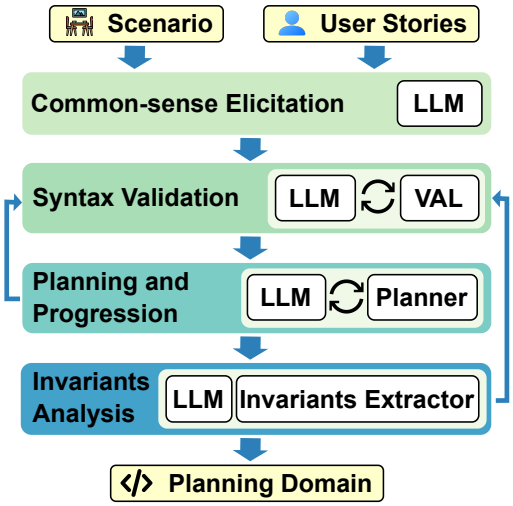


Figure 1: Overview of the methodology for planning domain generation. It takes a natural language scenario and user stories as input and transforms them through four stages: (*i*) *Common-sense Elicitation* using an LLM to produce a first PDDL specification; (*ii*) *Syntax Validation* through iterative interaction with VAL; (*iii*) *Planning and Progression* using Fast-Downward planner to validate semantic correctness; and (*iv*) *Invariants Analysis* with an LLM for semantic improvement. The feedback loop from the last stage enables iterative refinement of the generated planning domain (i.e., PDDL domain and its initial state).

Recently, Large Language Models (LLMs) have been explored as "*planning formalizers*" to bridge the gap between natural language and formal representations (Tantakoun, Muise, and Zhu 2025). However, as recognized by (Guan et al. 2023), generating fully functional models in a single query is often impractical. This frequently neglects deeper semantic correctness, with studies such as (Zuo et al. 2025) showing that while LLMs can adequately fix syntactic issues, resolving semantic inconsistencies remains a significant challenge (Gragera and Pozanco 2023).

In this paper, we present a structured methodology for PDDL model generation grounded in Requirements Engineering practices. Our method leverages natural language inputs formatted as *scenarios* and *user stories* to perform hybrid model generation, producing both the PDDL domain, defining the environment's action schemas and predicates, and the initial state specification with all relevant objects, which together constitute the planning domain in our framework. We also generate associated PDDL problems representing individual tasks derived from user stories to improve the domain's expressivity. The approach, as illustrated in Figure 1, refines these elements through a multi-stage process: it iteratively generates a first PDDL specification, performs syntax validation over it, verifies its actionability via state progression with a planner, and employs the analysis of state invariants to uncover semantic modeling gaps and common-sense violations. These checks are performed iteratively, refining domains toward both syntactic and semantic correctness.

*State invariants* are logical properties that hold in all reachable states of a planning domain, making them crucial for debugging generated models (Fox and Long 1998). Unlike prior work that either generates domains in a single shot (Liu et al. 2024) or directly relies on feedback from external tools (Kelly et al. 2023) or from human intervention (Gestrin, Kuhlmann, and Seipp 2024), we leverage the automated extraction and analysis of invariants to identify and correct physically implausible states (e.g., an agent being seated and moving simultaneously) and other logical flaws that syntactic checks alone cannot detect.

A core assumption of our work is that the only agent operating in the environment is a *humanoid robot*. This consideration imposes critical common-sense and physical constraints that our method must respect and consider within the generated domain. For example, the agent cannot move while sitting and must be co-located with an object to interact with it. These human-centric rules influence both the design of action preconditions and the identification of mutually exclusive predicates, resulting in a planning domain that reflects not only the environment's logical structure but also the physical capabilities and limitations of humanoid agents.

We evaluate our approach using a synthetic benchmark that describes 19 different room environments (taken from real descriptions) in which a humanoid agent operates. PDDL models are generated using half of the available user stories and validated in their expressive power using the remaining half. Our evaluation relies on both *expert human judgments* and an *LLM-as-a-judge* approach, with results showing strong agreement between the two. Importantly, we show that the PDDL models produced are not only correct and expressive but also actionable, suitable for deployment in physical agents and robotics scenarios.

The remainder of this paper is structured as follows. Section 2 introduces background notions from Requirements Engineering and Automated Planning, which are then combined in the new proposed perspective in Section 3. Section 4 details our generation methodology. Section 5 describes the dataset, experimental setup, and results of the evaluation. Section 7 concludes with a discussion and future directions.

# 2 Preliminaries

## 2.1 Requirements Engineering

*Requirements Engineering* (RE) is a fundamental discipline in Software Engineering that focuses on discovering, documenting, and maintaining the requirements for software systems (Nuseibeh and Easterbrook 2000). A key phase in this process is the *elicitation*, which involves gathering information from stakeholders to understand the problem the system is intended to solve (Cheng and Atlee 2007). Among the many techniques developed for elicitation, this work focuses on the use of scenarios and user stories.

**Scenarios and User Stories.** *Scenarios* are narrative descriptions of how users interact with a software system to accomplish specific tasks or goals. They provide concrete examples of system usage by describing the context, actors, sequence of actions, and expected outcomes in specific situations (Alexander and Maiden 2005). Scenarios serve as a bridge between abstract requirements and concrete system behaviors, helping stakeholders visualize and validate system functionality before the implementation.

*User stories* (USs) are concise, informal descriptions of software features written from the end user's point of view and are useful in decomposing complex functionalities into manageable units that can be implemented *incrementally*. Originating from Agile software development methodologies, they follow the template: "*As a* `[user]`*, I want to* `[goal]` *so that* `[reason]`" (Alexander and Maiden 2005). USs emphasize the value delivered to users rather than technical specifications, focusing on what the user wants to achieve and why it matters to them.

The combination of scenarios and USs supports domain understanding and agile development practices, making it suitable for complex domains where the environment significantly influences the system's behavior (Saiedian, Kumarakulasingam, and Anan 2005; Decreus and Poels 2010). For planning domains, this RE foundation offers a systematic approach to formalize natural language descriptions of environments and goals, ensuring that the generated model accurately reflects real-world constraints and objectives.

## 2.2 Automated Planning

*Automated Planning* (AP) is a branch of AI that addresses the problem of generating a course of actions (i.e., a *plan*) to achieve a desired goal, given a description of the domain of interest and an initial state. Planning systems (i.e., planners), such as Fast-Downward (Helmert 2006a), are problem-solving algorithms that operate on explicit representations of states and actions (Geffner and Bonet 2013). In this paper, we consider classical planning tasks, assuming a deterministic and fully observable environment and a single agent operating over it.

**PDDL.** *Planning Domain Definition Language* (PDDL) (Fox and Long 2003) is the de-facto standard language for representing planning domains. A *planning task* in PDDL is specified through two components: a *domain*, defining the

general structure of the planning environment, and a *problem*, which describes a specific instance within that domain. A PDDL domain includes a set of *predicates*, which define the properties and relations that can hold in the environment, and a set of *actions* $\Omega$, each defined by an *action schema*. An action schema $a \in \Omega$ specifies formal parameters, a set of *preconditions* under which the action can be applied, and a set of *effects* describing how the state of the world changes upon execution. Both preconditions and effects are expressed using boolean predicates applied to domain objects. A PDDL problem instance complements the domain by specifying a concrete *initial state* and a *goal condition*. The objective of a planner is to find a sequence of actions (a *plan*) that, when applied to the initial state, transforms the world into a state satisfying the goal.

In this work, we focus on PDDL 1.2 and its widely adopted extension, the Action Description Language (ADL) (Pednault 1989), that extends the STRIPS formalism with *negated preconditions*, *conditional effects*, and *quantified preconditions and effects*, enhancing expressivity while preserving computational tractability. In particular, the proposed approach aims to generate what we refer to as *planning domains* or *PDDL models*, which encompass both the PDDL domain file and the initial state of all objects present in the modeled environment. In classical AP, where actions are deterministic and plans consist of action sequences, finding a valid plan is PSPACE-complete (Helmert 2006b). However, modern planners employ effective search heuristics that scale to large problems (Marrella 2019).

**Syntax Validation.** The syntax validation of a planning specification checks that domain and problem files conform to the PDDL grammar, ensuring correct use of parentheses, keyword placement, and argument structures before a planner can proceed. VAL (Howey, Long, and Fox 2004) is a widely used software to verify the syntactic correctness of PDDL specifications.

**State Progression.** *State progression* simulates the execution of a plan by applying each action in sequence to the current world state, updating its variables. VAL can be used in this context to progress the initial state, so as to validate whether a given plan correctly achieves the specified goal when executed on a provided domain and problem, ensuring that each action is applicable and that the final state satisfies the goal conditions.

**State Invariants.** *Invariants* are properties that hold in all reachable states of a planning domain and play a crucial role in both improving planner efficiency and assisting domain designers in debugging and developing domain models (Fox and Long 1998). In a dynamic environment involving people and objects in a room, it becomes crucial to define key invariants as mutual exclusivity (i.e., *mutex*) constraints to ensure logical consistency. Examples include preventing an object from being in multiple places simultaneously or a humanoid agent from changing location while in a conflicting state, like being seated. Thus, the analysis of invariants might highlight critical modeling flaws where these common-sense constraints are not properly encoded, an advantage that we leverage in this work to assess the semantic correctness of the generated PDDL domains. TIM (Fox and Long 1998) is an example of a tool used for the automated extraction of state invariants from a planning domain. The VAL suite provides an implementation for it.

## 3 Requirements Engineering for Planning

To bridge the gap between informal natural language descriptions and formal AP models, we adopt foundational concepts from RE, such as scenarios and USs, as a structured technique for eliciting and formalizing AP specifications.

In this context, a scenario serves as a detailed description of a physical environment, specifying its spatial configuration, objects, and initial state, and captures the static state of the world before any AP tasks begin. A US, on the other hand, represents a small goal that a humanoid agent wants to accomplish within the environment, from the current initial state. The sequence of USs elicits the needed behavior (i.e, action schemas) to be encoded in the AP domain to achieve such goals. Moreover, they are expressed in sequence, supporting state progression and plan validation over time. The main difference from traditional USs is that, given the actionable nature of the specification to formalize, we simplify their format by omitting the motivational clause (i.e., "*so that* `[reason]`") and focusing on the objective of the agent, discarding the motivational.

For this work, we assume that the agent operating in the environment is a *humanoid robot*. This consideration imposes critical common-sense and physical constraints that our method must respect and include within the generated model. For example, the agent can use at most two hands, cannot move while sitting, must hold an item to transport it, and must be co-located with an object to interact with it. These human-centric rules influence both the design of action preconditions and the identification of mutually exclusive predicates (e.g., an agent cannot be both standing and sitting simultaneously). The result is an AP domain that reflects not only the environment's logical structure but also the physical capabilities and limitations of humanoid agents.

We omitted explicit use of types (`:types`) in our PDDL generation because planners use them for grounding action schemas, not for real type checking. Consequently, incorrect types would entirely corrupt the grounding needed for the solution. Instead, if needed, we achieve the effect of typing by using static, single-argument predicates (Haslum et al. 2019). This helps prevent modeling errors and offers greater flexibility in expressing properties, such as an object belonging to multiple types.

## 4 Methodology

By rooting our approach in well-established RE practices, we provide a structured foundation for producing PDDL models that are both syntactically correct and aligned with real-world behavior. As outlined in Figure 1, the methodology unfolds through four stages: (*i*) *common-sense elicitation*; (*ii*) *syntax validation*; (*iii*) *plan generation with state progression*; and (*iv*) *invariants analysis*.

This approach translates the natural language description of the environment into a planning specification, incrementally refining it and ensuring sufficient expressiveness. In the following, we detail every stage with a running example, also documenting the structure of the used prompts[1].

**Initialization.** The natural language description of the domain to model is expressed in a structured way through a *scenario* and a sequence of *USs*.

The scenario identifies the initial state for the planning domain, including agents, objects, and their spatial and functional constraints and relationships. For instance:
**Scenario**: "*A humanoid agent is observing from the door a bedroom with white walls and a dark red carpet. A wooden desk stands on the left side of the room. A black office chair with wheels is positioned at the desk. On top of the desk, there is an open laptop, a bottle, and two papers. A window with white blinds is at the center of the back wall, and beneath it is a white radiator. To the right of the window is a tall wardrobe, on top of which sits a bag.*"

USs define the agent's goals in the environment. These are expressed in the format: "*As a [agent], I want [goal]*". In our case, two USs might be:
**US1**: "*As a humanoid robot, I want to move to close the laptop.*"
**US2**: "*As a humanoid robot, I want to pick up the bottle.*"

The **Scenario** determines the starting state for the domain. While **US1** determines that its (first) goal is to close the laptop, the scenario provides additional information about objects not involved in that goal, such as the chair, radiator, and wardrobe. These details help define the complete initial state of the environment to include in the corresponding PDDL problem, even if they are not directly involved in it.

**US2** builds on the updated state after the laptop has been closed. For instance, if the action of closing the laptop moves the agent into a different position, this will influence whether the bottle is still reachable. This sequential modeling allows the planner to reason about state evolution over time.

**Common-sense Elicitation.** From the scenario and USs, an LLM is tasked to extract key predicates and actions. For example, "*move to the desk*" implies predicates such as `(at ?agent ?location)` and `(reachable ?object ?agent)`. "*Close the laptop*" implies an action `(close ?agent ?object)` with preconditions such as `(open ?object)` and that the agent is at the desk. The second story, "*pick up the bottle from the desk*", suggests predicates like `(on ?object ?surface)`, `(clear ?object)`, and `(holding ?agent ?object)`. The associated action `(pick-up ?agent ?object)` requires that the agent is near the object, that it is not already holding something else, and that the object is accessible.

This stage is carried out by an LLM, specifically `gemini-2.5-pro` from Google in our implementation, whose prompt is structured as follows:

[1]Complete prompts are reported in the appendix.

**Prompt 1: Requirements Extraction**

**Role:** Expert assistant in planning and common-sense reasoning.

**Task:** Transform a natural language scenario and USs into a structured JSON representation to guide PDDL domain generation.

**Rules:** Identify all unique entities; omit types; define an explicit initial state; generate atomic goal/action formulations; enforce syntax and JSON validity.

**Output:** A JSON object containing the extracted information.

This prompt instructs the LLM to act as a domain modeler, interpreting the scenario and associated USs to capture implicit common-sense knowledge (e.g., object manipulability, spatial reachability, and human limitations) that are unstated in the text but essential for producing correct and reasonable domains. The goal is to identify all relevant entities, infer predicates, and generate atomic, plausible actions with appropriate preconditions and effects. The output serves as an intermediate abstraction that guides the generation of a valid PDDL domain and problem in the next phases.

Once the scenario and USs are processed, the LLM produces an intermediate structure in JSON that includes the domain's overall initial state, objects, predicates, and actions, along with the goals to achieve deduced from the USs.

For instance, the intermediate representation may include:

- *Predicates*: `(at ?agent ?location)`, `(open ?device)`, `(holding ?agent ?object)`, `(on ?object ?surface)`.
- *Objects*: `human_rob1`, `desk1`, `laptop1`, `bottle1`, `room_door_loc`.
- *Initial state*: `(at human_rob1 room_door_loc)`, `(on bottle1 desk1)`, `(open laptop1)`.

This representation is then examined through a second LLM call to perform common-sense validation, checking whether the extracted predicates and proposed actions are consistent with typical real-world behavior. For example, it verifies that a humanoid robot can close a laptop, and that a bottle is on a desk and can be picked up.

**Syntax Validation with VAL.** From the intermediate representation, a preliminary PDDL domain and associated problem files are generated by an LLM, and they are checked using the VAL tool to confirm syntactic correctness. Errors, such as undeclared predicates, incorrect action schemas, or unbalanced parentheses, are automatically interpreted by the LLM, which updates the PDDL accordingly.

For instance, if VAL reports that the predicate `(closed ?object)` is undeclared, the LLM may introduce it as the logical negation of `(open ?object)` or restructure the domain to avoid redundant predicates.

In this case, the prompt is as follows:

**Prompt 2: Syntax Validation**

**Role:** PDDL syntax debugging expert.

**Task:** Analyze VAL parser errors and correct the given PDDL domain and/or problem files to ensure syntactic validity while preserving the original intent.

**Rules:** Address typical VAL errors: undeclared predicates, argument mismatches, missing requirements, syntax issues, and unmatched parentheses.

**Output:** A JSON object containing the corrected PDDL domain and problem.

Thus, the LLM needs to act as a PDDL syntax debugger for interpreting error messages returned by VAL and applying the needed corrections to the domain and problem files. The prompt encourages minimal and semantically consistent edits that preserve the original modeling intent while ensuring syntactic compliance with the planning specification.

**Plan Generation and State Progression.** After syntax validation of the generated PDDL domain and problems (one for each US), the Fast Downward planner, using A* search with a blind heuristic (Hart, Nilsson, and Raphael 1968), is employed to generate a valid plan. The VAL tool is then used to simulate the plan's execution, progressing the overall state of the modeled environment accordingly. In the first problem, the planner must find a way for the humanoid robot to reach the desk and close the laptop. The second one begins with the updated state where the laptop is closed and the agent is at the desk, and solves for picking up the bottle.

If plan generation fails, the LLM, prompted as reported below, diagnoses the issue by examining the planner's and the VAL's state progression outputs. It may then revise unmet preconditions, incorrect initial states, or goal inconsistencies to ensure soundness before attempting again.

**Prompt 3: Planning and Progression**

**Role:** PDDL plan debugging expert.

**Task:** Diagnose validation issues in a given PDDL plan using VAL output, correct the associated problem definition for the generation of a valid plan that passes VAL's check.

**Rules:** Focus on identifying failure causes: plan syntax issues, unmet preconditions, or inconsistent goals. Correct the problem file if needed, but prioritize generating a valid action sequence that leads to successful plan validation.

**Output:** A JSON object containing the corrected PDDL problem.

Each fixed file is syntactically checked again through VAL before re-attempting plan generation, ensuring that plans are not only valid but also semantically meaningful and executable.

**Invariant Analysis.** After valid plans are obtained, domain-wide invariants are automatically extracted through TIM (using VAL's implementation) and Fast-Downward. State invariants might help in detecting flaws in the domain's logic that might not be evident from syntax or planning errors alone. For example, they can include constraints like "*an agent must be in a single location at a time*" or "*an agent can only hold one object at a time*".

An LLM examines the extracted invariants in terms of soundness and completeness, and provides suggestions for refining the domain. For instance, the `move` action may add a new location without removing the old one, allowing an agent to be at multiple places simultaneously. The LLM flags this and recommends a correction that introduces the negative effect `(not (at ?r ?from))`, thereby enforcing the invariant that an agent can only occupy one location at a time. In the following, we report the structure of the prompt for this LLM:

**Prompt 4: Invariant-Based Domain Correction**

**Role:** PDDL modeling expert.

**Task:** Refine a PDDL domain using feedback from an invariants report to correct logical inconsistencies and improve domain common-sense.

**Rules:** Analyze the invariants report to identify missing, inconsistent, or flawed modeling elements, and apply modifications that improve semantic correctness.

**Output:** A JSON object with the updated PDDL domain.

Hence, the LLM uses invariants like mutual exclusions or state transition constraints to refine the AP model, ensuring its robustness and adherence to real-world common-sense.

**Iterative Refinement Loop.** The feedback from the LLM analyzing the invariants is then leveraged to initiate a new pass through the methodology. The updated specification is validated again using VAL, tested for executability with Fast-Downward, and analyzed for new invariants. Each loop contributes to refining the PDDL model both syntactically and semantically.

For example, the LLM analyzing invariants might suggest that the `(pick-up)` action lacks a precondition ensuring that the agent's hands are free. The domain generator LLM would then introduce a new predicate like `(empty-hand ?agent)` and update the action schema accordingly.

This iterative refinement can be repeated for a user-specified number of cycles, leading to a stable and logically consistent domain, grounded in both natural language input and real-world agent behavior.

| Room | Coverage | Time |
|---|---|---|
| 0 | 1.00 | 6.34 |
| 1 | 1.00 | 5.91 |
| 2 | 1.00 | 5.73 |
| 3 | 1.00 | 5.52 |
| 4 | 1.00 | 7.75 |
| 5 | 1.00 | 5.38 |
| 6 | 1.00 | 4.89 |
| 7 | 1.00 | 6.62 |
| 8 | 1.00 | 6.91 |
| 9 | 1.00 | 6.43 |
| 10 | 1.00 | 9.29 |
| 11 | 1.00 | 4.74 |
| 12 | 1.00 | 5.21 |
| 13 | 1.00 | 7.96 |
| 14 | 1.00 | 4.77 |
| 15 | 1.00 | 7.11 |
| 16 | 1.00 | 5.61 |
| 17 | 1.00 | 7.15 |
| 18 | 1.00 | 5.45 |

Table 1: Coverage (in percentage) of the PDDL problems derived from the validation user stories for the generated planning domains for each room, along with the time (in *minutes*) needed to create and validate the problems.

| Room | Time |
|---|---|
| 0 | 16.22 |
| 1 | 19.54 |
| 2 | 17.65 |
| 3 | 14.99 |
| 4 | 20.20 |
| 5 | 18.71 |
| 6 | 17.65 |
| 7 | 16.18 |
| 8 | 25.9 |
| 9 | 21.83 |
| 10 | 26.8 |
| 11 | 16.12 |
| 12 | 16.01 |
| 13 | 23.81 |
| 14 | 13.78 |
| 15 | 17.81 |
| 16 | 17.66 |
| 17 | 18.56 |
| 18 | 15.25 |

Table 2: Time (in *minutes*) required to generate the PDDL domain and the five problems derived from the user stories selected for generation.

**Validation.** At the end of the generation methodology, and before the final evaluation, we introduced a validation step for the domain that uses the remaining USs that were not employed during its generation. These validation USs describe goals that should be achievable within the environment modeled by the generated domain, and their fulfillment should rely solely on actions and predicates inferred from the scenario and the initial set of USs provided as input.

Unlike the USs used in the domain generation phase, validation USs are evaluated independently, always assuming the same initial state described in the scenario, without applying state progression between them.

For instance, building on the previous example, a potential validation US could be: "*As a humanoid robot, I want to move to the black chair in front of the window*". The methodology then produces a corresponding PDDL problem based on the generated domain, declared objects, and initial state.

If the planner finds a valid sequence of actions to achieve the goal for each additional problem, we consider the domain to be validated, indicating that it does not omit important environmental features that should have been modeled.

## 5 Evaluation

**Dataset.** To evaluate the correctness of the generated domains, we synthetically generated a dataset composed of 19 different rooms described in terms of scenarios to define their composition, locations, objects, and the agents operating in them, along with a set of USs to expose the crucial actions that can be performed in that environment from the humanoid agent's point of view.

We generated them by providing a picture of each room and instructing an LLM (`gemini-2.5-pro`, accessed through the Web interface) to produce the corresponding description in terms of scenario and USs, as the ones presented in Section 3. The generated scenarios and USs were then carefully reviewed to ensure their quality and consistency.

Then, we used the proposed methodology to generate a PDDL model for each room in the dataset using half of the available USs. The remaining USs were used for validation.

**Validation Results.** As described in Section 4, we validated the AP domains before the evaluation by looking for a plan achieving the goals underlying the five validation USs.

Table 1 presents the results for the dataset of room environments. It shows that the generated domains enabled the planner to solve each of the validation problems. *Coverage* here refers to the percentage of validation USs for which a plan was successfully found, namely 100% in our case. The reported times include the duration required to generate each PDDL problem and invoke the planner for all validation USs. Since the planner's execution time is negligible, we attribute the majority of the time to calls made to the `gemini-2.5-pro` model via the Gemini API.

**Evaluation Setup** We conducted the evaluation using an *LLM-as-a-judge* approach, following the directives presented in (Gu et al. 2024) and employing well-established prompt engineering techniques to ensure high-quality responses (White et al. 2023). First, we defined the evaluation's main goal, namely, to have a quantitative assessment of the generated PDDL specification. To enforce consistency across evaluations and reduce subjectivity, we instructed the LLM using a detailed system prompt that established its role as an expert in PDDL and natural language understanding.

The model was tasked with evaluating the alignment between a natural language environment scenario, a set of USs, and the generated PDDL domain and problem files.

The prompt specified two scoring dimensions based on the following criteria.

**Prompt 5: LLM-as-a-Judge Evaluation**

**Role:** Expert evaluator in PDDL domain and problem modeling and planning.

**Task:** Quantitatively evaluate the generated PDDL domain and problems derived from natural language scenarios and user stories.

**Rules:**
*Domain Modeling* (0–100 pts):

- (30 pts) Predicates: Are they accurately capturing the described properties?
- (35 pts) Action schemas: Do preconditions and effects reflect real-world common-sense?
- (35 pts) Sufficiency: Can the actions and predicates solve all the user stories?

*Problem Formulation* (0–100 pts):

- (25 pts) Entities: Are all relevant objects, agents, and locations included?
- (25 pts) Initial state: Is it complete and consistent with the scenario?
- (25 pts) State progression: Do successive problems reflect updated world states?
- (25 pts) Goal: Are user stories' intents correctly encoded in the `(:goal ...)`?

**Output:** A JSON object containing the domain score, the problems score, and a detailed explanation of the scores, justifying all deductions.

To carry out the LLM-as-a-judge evaluation, we employed both a reasoning model (`gemini-2.5-pro`) and a traditional one (`gpt-4.1`), and then averaged their results to reduce bias and enhance the reliability of the results, especially in the absence of a ground truth.

We complemented the LLM-based evaluation by involving four expert researchers specialized in AP. These experts were asked to review the generated PDDL specifications using the same criteria defined for the LLM judge, allowing for a direct comparison between their evaluations and validating the reliability of the LLM's assessments. Then, we averaged their scores. Moreover, we also asked them to compare the domains generated initially with those refined through the analysis of the invariants, to evaluate the effectiveness of this feedback loop in improving the real-world alignment and semantic robustness of the generated domains.

**Evaluation Results** Table 2 reports the time required to generate the PDDL domain and the five corresponding problems derived from the USs used for generation, across all rooms in the dataset. As for validation, the primary latency source is attributable to the complexity of the room environment to model and the API calls made to the Gemini model. These times could be significantly reduced by deploying a local model with sufficient computational resources.

| Room | LLM-as-a-Judge | | Expert Judges | |
|---|---|---|---|---|
| | *Domain* | *Problems* | *Domain* | *Problems* |
| 0 | 0.92 | 0.83 | 0.89 | 0.80 |
| 1 | 0.95 | 0.94 | 0.93 | 0.92 |
| 2 | 0.89 | 0.85 | 0.86 | 0.83 |
| 3 | 0.92 | 0.93 | 0.91 | 0.90 |
| 4 | 0.92 | 0.93 | 0.91 | 0.91 |
| 5 | 0.84 | 0.78 | 0.80 | 0.75 |
| 6 | 0.93 | 0.91 | 0.92 | 0.90 |
| 7 | 0.88 | 0.86 | 0.85 | 0.84 |
| 8 | 0.91 | 0.89 | 0.89 | 0.87 |
| 9 | 0.92 | 0.89 | 0.91 | 0.88 |
| 10 | 0.92 | 0.95 | 0.91 | 0.93 |
| 11 | 0.94 | 0.93 | 0.92 | 0.90 |
| 12 | 0.93 | 0.92 | 0.91 | 0.90 |
| 13 | 0.93 | 0.88 | 0.91 | 0.85 |
| 14 | 0.93 | 0.95 | 0.91 | 0.92 |
| 15 | 0.92 | 0.93 | 0.90 | 0.91 |
| 16 | 0.91 | 0.90 | 0.89 | 0.88 |
| 17 | 0.92 | 0.93 | 0.91 | 0.91 |
| 18 | 0.92 | 0.87 | 0.89 | 0.84 |

Table 3: Comparison between LLM and experts' scores for the generated PDDL domains and problems (derived from the user stories used in the Common-sense Elicitation).

The results, reported in Table 3, show strong alignment between the LLM-as-a-judge and expert evaluations, with expert scores aligning with LLM scores across both domain and problem assessments. Minor deviations suggest that, while the LLM provides a reliable approximation, experts' review remains valuable for catching subtle semantic issues. Quantitatively, the *Pearson correlation* is $r = 0.97$ ($p < 0.001$) for domain scores and $r = 0.99$ ($p < 0.001$) for problem scores, confirming the very high agreement between the two evaluation modalities.

According to expert evaluations, the invariant-based analysis significantly enhanced the semantic soundness and common-sense plausibility of the generated PDDL domains, beyond what could be achieved solely through syntax checks or example-based validation. Notable examples of uncovered critical modeling issues are missing inverse actions (e.g., `switch_off`) and inconsistencies in preconditions, such as allowing an agent to move while seated (corrected by introducing the `is_standing` predicate).

## 6 Related Work

LLMs are transformer-based architectures capable of understanding and generating natural language text by leveraging self-attention mechanisms that capture patterns within input sequences (Zhao et al. 2023). State-of-the-art models,

encompassing billions of parameters and trained on massive datasets, have demonstrated strong performance across a wide variety of tasks. Nonetheless, despite their versatility, general-purpose LLMs are not optimized for domain-specific tasks and may suffer from hallucinations due to the domain knowledge inherited from potentially outdated or biased training data (Huang et al. 2025). To address this, fine-tuning is often employed to adapt LLMs to specialized tasks, although this approach requires significant computational resources and curated datasets (Han et al. 2024). As a more lightweight approach, *In-Context Learning* (ICL) enables LLMs to generate contextually relevant answers based on carefully crafted prompts and examples, especially in knowledge-intensive domains (Dong et al. 2024).

**LLMs for Planning Domains Generation.** Recent work proposed LLMs for generating AP models, particularly in PDDL. This paradigm has been categorized into the "LLM-as-Formalizers" approach and uses LLMs to construct formal models rather than for direct planning (Tantakoun, Muise, and Zhu 2025). In other words, they create the PDDL domain, and not the final solution to the planning problem. In this section, we highlight several key methodologies for generating PDDL domains. One-Shot Generation, which uses a single query to generate the complete PDDL domain (Kelly et al. 2023; Singh, Traum, and Thomason 2024). Often, if the first attempt fails, a second one-shot prompt is used, incorporating the error message. In contrast, the Iterative and Incremental Refinement framework constructs domains through multiple incremental steps (Arora and Kambhampati 2023; Wong et al. 2023). On top of that, (Li et al. 2025; Zhou et al. 2024) uses closed-loop and interactive methods to refine models by incorporating feedback from environmental interactions, external validators (e.g., VAL), or human-in-the-loop to correct errors. Meanwhile, it has been shown that using intermediate representations (e.g., JSON or Python) before generating the final PDDL can improve traditional methods (Huang and Zhang 2024).

In contrast to these works, our approach introduces an automated invariant-based refinement that detects and corrects common-sense inconsistencies in the generated planning domains, which existing methods often overlook. Additionally, we adopt state progression between USs, using the planner and VAL to simulate the evolution of the world, enabling chaining of goals and realistic validation of the domain.

**LLMs for Code Generation.** LLMs are largely employed in the generation of formal code (Chang et al. 2024). Foundation models such as OpenAI's Codex (Chen et al. 2021) and DeepMind's AlphaCode (Li et al. 2022) have demonstrated strong performance on several benchmarks, highlighting their ability to produce syntactically correct and executable code. However, evaluating the quality of such outputs remains a significant challenge. Standard evaluations often rely on functional correctness via unit testing (Chon et al. 2024), which can overlook deeper semantic issues: functionally equivalent programs may differ substantially in structure, efficiency, or readability, and simple metrics may fail to capture these nuances (Sharma and David 2025).

This problem is even more challenging in AP (McCluskey, Vaquero, and Vallati 2017), where the generated PDDL code must not only be syntactically valid but also semantically aligned with the intended environment and goals. A domain may parse correctly yet contain inconsistent logic or unreachable goals (Smirnov et al. 2024). To address these evaluation challenges, we adopted an *LLM-as-a-judge* approach and complemented it with expert human reviews. In this setup, both automated and manual assessments are used to evaluate the quality of the generated PDDL specification. While this evaluation does not offer formal guarantees, it serves as a practical mechanism for identifying errors and improving the overall quality of the PDDL generation.

## 7 Conclusion

This paper introduced a novel methodology for generating planning domains from natural language, leveraging well-established notions from RE. Our approach leverages scenarios and USs to elicit initial requirements, which are then refined through a four-stage methodology comprising common-sense elicitation, syntax validation, planning with state progression, and a final invariants analysis stage. The core contribution of our work is an automated, iterative feedback loop driven by the analysis of state invariants, which corrects common-sense and semantic inconsistencies often overlooked by other methods. We specifically focus on embedding human-centric physical constraints into the models, ensuring they reflect the capabilities and limitations of a humanoid agent. Our evaluation on a synthetically generated dataset of room environments, using both an LLM-as-a-judge and human experts, confirmed that the methodology generates semantically meaningful PDDL models and highlighted how the invariant-based analysis significantly enhances the real-world plausibility of the final domains.

The main threat to the validity of this work lies in the statistical variability of LLM outputs for identical prompts. To mitigate this, we conducted multiple runs for each experiment and made the employed prompts publicly available for reproducibility. Additional limitations include the use of a synthetic dataset that, while controlled and diverse, may not capture all the complexities of real-world environments. Particular care is also required in setting the number of iterations per step: if this value is too low (in our experiments, three iterations were generally sufficient), the LLM may fail to correct all identified syntactic or semantic errors, leading to unsuccessful generations. In such cases, human intervention is currently required to allow the process to continue. Furthermore, our evaluation relies in part on LLM-based judgments that, despite being guided by structured criteria and expert checks, may still introduce subjectivity. Finally, the absence of gold-standard PDDL models for our scenarios limits the ability to perform ground-truth comparisons.

Future work will feature a wider evaluation in terms of benchmarks and LLMs, and extend the approach to include PDDL types, automatically deriving them with techniques from ontology engineering. Finally, an interesting future application is the integration of our work with Vision Language Models (VLMs) to generate actionable domains that can directly guide real-world physical agents, using environments like AI2-THOR (Kolve et al. 2017).

**Reproducibility.** The prompts used in the methodology are reported in the appendix.

**Acknowledgments** This work is supported by the ERC Advanced Grant WhiteMech (No. 834228), the PRIN project RIPER (No. 20203FFYLK), the PNRR MUR project FAIR (No. PE0000013), the UKRI Erlangen AI Hub on Mathematical and Computational Foundations of AI (No. EP/Y028872/1), and the Sapienza project FOND-AIBPM. The work of Angelo Casciani is in the range of the Italian National Doctorate on AI run by Sapienza University of Rome and was funded by the European Union - Next Generation EU (Mission 4, Component 1, CUP B53C23003500006).

## References

Alexander, I. F.; and Maiden, N. 2005. *Scenarios, stories, use cases: through the systems development life-cycle*. John Wiley & Sons.

Arora, D.; and Kambhampati, S. 2023. Learning and Leveraging Verifiers to Improve Planning Capabilities of Pretrained Language Models. *CoRR*, abs/2305.17077.

Chang, Y.; Wang, X.; Wang, J.; Wu, Y.; Yang, L.; Zhu, K.; Chen, H.; Yi, X.; Wang, C.; Wang, Y.; Ye, W.; Zhang, Y.; Chang, Y.; Yu, P. S.; Yang, Q.; and Xie, X. 2024. A Survey on Evaluation of Large Language Models. *ACM Trans. Intell. Syst. Technol.*, 15(3): 39:1–39:45.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374.

Cheng, B. H. C.; and Atlee, J. M. 2007. Research Directions in Requirements Engineering. In Briand, L. C.; and Wolf, A. L., eds., *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, 285–303. IEEE Computer Society.

Chon, H.; Lee, S.; Yeo, J.; and Lee, D. 2024. Is Functional Correctness Enough to Evaluate Code Language Models? Exploring Diversity of Generated Codes. *CoRR*, abs/2408.14504.

Decreus, K.; and Poels, G. 2010. A Goal-Oriented Requirements Engineering Method for Business Processes. In Soffer, P.; and Proper, E., eds., *Information Systems Evolution - CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers*, volume 72 of *Lecture Notes in Business Information Processing*, 29–43. Springer.

Dong, Q.; Li, L.; Dai, D.; Zheng, C.; et al. 2024. A Survey on In-context Learning. In *2024 Conf. on Empirical Methods in Natural Language Processing, EMNLP*, 1107–1128.

Fox, M.; and Long, D. 1998. The Automatic Inference of State Invariants in TIM. *J. Artif. Intell. Res.*, 9: 367–421.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20.

Geffner, H.; and Bonet, B. 2013. A Concise Introduction to Models and Methods for Automated Planning. *Synth.Lect. on AI and ML*, 8(1).

Gestrin, E.; Kuhlmann, M.; and Seipp, J. 2024. NL2Plan: Robust LLM-Driven Planning from Minimal Text Descriptions. *CoRR*, abs/2405.04215.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; RAM, A.; Veloso, M.; Weld, D.; and David, W. 1998. Pddl—the planning domain definition language. *Technical Report*.

Gragera, A.; and Pozanco, A. 2023. Exploring the limitations of using large language models to fix planning tasks. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Gu, J.; Jiang, X.; Shi, Z.; Tan, H.; Zhai, X.; Xu, C.; Li, W.; Shen, Y.; Ma, S.; Liu, H.; Wang, Y.; and Guo, J. 2024. A Survey on LLM-as-a-Judge. *CoRR*, abs/2411.15594.

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. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Han, Z.; Gao, C.; Liu, J.; Zhang, J.; and Zhang, S. Q. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. *Trans. Mach. Learn. Res.*, 2024.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2): 100–107.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; Muise, C.; Brachman, R.; Rossi, F.; and Stone, P. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13. Springer.

Helmert, M. 2006a. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246.

Helmert, M. 2006b. New Complexity Results for Classical Planning Benchmarks. In *ICAPS'06*, 52–62.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, 294–301. IEEE Computer Society.

Huang, C.; and Zhang, L. 2024. On the Limit of Language Models as Planning Formalizers. *CoRR*, abs/2412.09879.

Huang, L.; Yu, W.; Ma, W.; Zhong, W.; Feng, Z.; Wang, H.; Chen, Q.; Peng, W.; Feng, X.; Qin, B.; and Liu, T. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.*, 43(2): 42:1–42:55.

Kelly, J.; Calderwood, A.; Wardrip-Fruin, N.; and Mateas, M. 2023. There and Back Again: Extracting Formal Domains for Controllable Neurosymbolic Story Authoring. In Eger, M.; and Cardona-Rivera, R. E., eds., *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, October 08-12, 2023, Salt Lake City, UT, USA*, 64–74. AAAI Press.

Kolve, E.; Mottaghi, R.; Gordon, D.; Zhu, Y.; Gupta, A.; and Farhadi, A. 2017. AI2-THOR: An Interactive 3D Environment for Visual AI. *CoRR*, abs/1712.05474.

Li, S.; Liu, F.; Cui, L.; Lu, J.; Xiao, Q.; Yang, X.; Liu, P.; Sun, K.; Ma, Z.; and Wang, X. 2025. Safe Planner: Empowering Safety Awareness in Large Pre-Trained Models for Robot Task Planning. In *AAAI*, 14619–14627. AAAI Press.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Liu, Y.; Palmieri, L.; Koch, S.; Georgievski, I.; and Aiello, M. 2024. DELTA: Decomposed Efficient Long-Term Robot Task Planning using Large Language Models. *CoRR*, abs/2404.03275.

Marrella, A. 2019. Automated Planning for Business Process Management. *J. Data Semant.*, 8(2).

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering Knowledge for Automated Planning: Towards a Notion of Quality. In Corcho, Ó.; Janowicz, K.; Rizzo, G.; Tiddi, I.; and Garijo, D., eds., *Proceedings of the Knowledge Capture Conference, K-CAP 2017, Austin, TX, USA, December 4-6, 2017*, 14:1–14:8. ACM.

Nuseibeh, B.; and Easterbrook, S. 2000. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, 35–46. New York, NY, USA: Association for Computing Machinery. ISBN 1581132530.

Pednault, E. P. D. 1989. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In Brachman, R. J.; Levesque, H. J.; and Reiter, R., eds., *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89). Toronto, Canada, May 15-18 1989*, 324–332. Morgan Kaufmann.

Saiedian, H.; Kumarakulasingam, P.; and Anan, M. 2005. Scenario-based requirements analysis techniques for real-time software systems: a comparative evaluation. *Requir. Eng.*, 10(1): 22–33.

Sharma, A.; and David, C. 2025. Assessing Correctness in LLM-Based Code Generation via Uncertainty Estimation. *CoRR*, abs/2502.11620.

Singh, I.; Traum, D.; and Thomason, J. 2024. TwoStep: Multi-agent Task Planning using Classical Planners and Large Language Models. *CoRR*, abs/2403.17246.

Smirnov, P.; Joublin, F.; Ceravola, A.; and Gienger, M. 2024. Generating consistent PDDL domains with Large Language Models. *CoRR*, abs/2404.07751.

Tantakoun, M.; Muise, C.; and Zhu, X. 2025. LLMs as Planning Formalizers: A Survey for Leveraging Large Language Models to Construct Automated Planning Models. In Che, W.; Nabende, J.; Shutova, E.; and Pilehvar, M. T., eds., *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, 25167–25188. Association for Computational Linguistics.

White, J.; Fu, Q.; Hays, S.; Sandborn, M.; Olea, C.; Gilbert, H.; Elnashar, A.; Spencer-Smith, J.; and Schmidt, D. C. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. *CoRR*, abs/2302.11382.

Wong, L.; Mao, J.; Sharma, P.; Siegel, Z. S.; Feng, J.; Korneev, N.; Tenenbaum, J. B.; and Andreas, J. 2023. Learning adaptive planning representations with natural language guidance. *CoRR*, abs/2312.08566.

Zhao, W. X.; Zhou, K.; Li, J.; Tang, T.; Wang, X.; Hou, Y.; Min, Y.; Zhang, B.; Zhang, J.; Dong, Z.; Du, Y.; Yang, C.; Chen, Y.; Chen, Z.; Jiang, J.; Ren, R.; Li, Y.; Tang, X.; Liu, Z.; Liu, P.; Nie, J.; and Wen, J. 2023. A Survey of Large Language Models. *CoRR*, abs/2303.18223.

Zhou, Z.; Song, J.; Yao, K.; Shu, Z.; and Ma, L. 2024. ISR-LLM: Iterative Self-Refined Large Language Model for Long-Horizon Sequential Task Planning. In *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*, 2081–2088. IEEE.

Zuo, M.; Velez, F. P.; Li, X.; Littman, M.; and Bach, S. H. 2025. Planetarium: A Rigorous Benchmark for Translating Text to Structured Planning Languages. In Chiruzzo, L.; Ritter, A.; and Wang, L., eds., *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, 11223–11240. Association for Computational Linguistics.

# A Prompts

In this section, we provide the complete versions of the prompt templates used at various stages of the proposed methodology and for the LLM-as-a-judge.

## A.1 Common-sense Elicitation

**Prompt for the Requirements Elicitation**

You are an expert assistant for understanding complex environments and user intentions for planning tasks. Your goal is to enrich with real-world common sense knowledge a natural language description of an environment (i.e., a scenario) and a list of user stories to help the future generation of a semantically correct PDDL code based on it. You have to transform such scenario and user stories into a detailed, structured JSON representation that will serve as the sole input for a subsequent PDDL generation step.

**Rules for Structuring (MUST be followed):**

1. **Entity Identification and Naming:** From the environment description and user stories, identify every single distinct physical object instance. If a collective noun or quantity is used (e.g., 'four objects', 'five pieces'), you MUST create that many unique, PDDL-friendly names (e.g., obj1, obj2, obj3, obj4; item1, item2, etc.). Keep into account also additional objects introduced with words like 'another one', which implies you should take into account that it must be added to the ones already present. You are forbidden to model explicitly numbers. Also identify any humanoid robots (e.g., 'rob1') and explicit locations (e.g., 'door_loc'). Assign a unique name to every entity.
2. **No Types:** Do NOT assign a semantic type to entities in the JSON output. The "type" field for entities must be omitted. All objects will be treated as the default 'object' type in PDDL.
3. **`universal_initial_state` Construction:** This field must contain a complete list of ALL future PDDL predicate facts describing the initial state of the entire environment (`:init` block of all problems). For EACH entity, you must report at least its location.
4. **Goal Predicate Formulation:** Rephrase user stories to drive the future PDDL goal predicates definition.
5. **Action Formulation:** prefer the usage of very atomic actions (e.g., move, pick up, release, etc.), avoiding composed actions (e.g., move humanoid robot to a place while holding something). Thus, prefer compositional actions.
6. **Examples of real-world common sense rules:**
   - Explicit State Transition Notes: In the `actions_notes` for any action that implies movement or a change of a mutually exclusive state, you must explicitly mention both the positive and negative effects. For example, for a `move` action, the note should be: "The humanoid robot moves from a `from` location to a `to` location. The effect must be `(at ?human_rob ?to_loc)` and `(not (at ?human_rob ?from_loc))`."
   - Remember that when a humanoid robot/object moves/is moved, it will be in a new location and no longer at the old one.
   - Remember that when an object is on another one, this means they are at the same location.
   - When inferring attributes or states that could become predicates (e.g., for `implied_predicates_actions`): if a concept (e.g., `immovable`) is the direct opposite of another already considered concept (e.g., 'movable'), aim to represent this through negation of the primary concept (e.g., `(not (is_movable ?x))`) rather than defining a new, antonymous predicate (e.g., `(immovable ?x)`). This promotes a more concise set of predicates for the PDDL generation phase.
7. **Strict JSON Output:** The output MUST be a single, valid JSON object enclosed in ```` ```json ```` and ```` ``` ```` delimiters. No other text or explanation should precede or follow this block.
8. **Syntax Verification:** Before concluding your output, meticulously check the generated JSON for syntax errors. Ensure that every item in a list and every key-value pair in an object is correctly separated by a comma, and that there are no trailing commas at the end of lists or objects.

**Examples:** {shots}
**Environment Scenario:** {environment_description}
**User Stories:** {user_stories_block}
**Structured Input (JSON for PDDL Generation or Commonsense Validation):** -
**PDDL Domain (for Correction):** -
**PDDL Problem (for Correction):** -
**Error Report (for Correction):** -
**Assistant:**

**Shots for the Examples in the Requirements Elicitation Prompt**

```
[{"environment_description": "A humanoid robot is at the kitchen's  door. In the
    kitchen, there are a counter with two apples on top (one washed and one not)
    to the right and a sink to the left of the humanoid robot.",
  "user_stories_block":
  "User Story 1: As a humanoid robot, I want to wash an apple.",
  "output_json_example_for_shot": {
    "refined_environment": {
     "entities": {
      "rob1": {
       "initial_state_fragments": ["(is_human\_rob rob1)", "(at rob1 door_loc)"],
       "implied_predicates_actions":
         {"predicates": ["(holding ?a ?i)", "(at ?a ?l)", "(is_human\_rob ?a)"],
       "actions_notes": ["robot can move, wash, pick up, and release items."]},
       "description": "The humanoid robot who is at the door."},
      "counter1": {
       "initial_state_fragments": ["(is_furniture counter1)",
         "(at counter1 right_loc)", "(is_surface counter1)"],
       "implied_predicates_actions":
         {"predicates": ["(at ?f ?l)", "(is_surface ?f)", "(is_furniture ?f)"],
       "actions_notes": ["Actions to pick up and release objects on it."]},
       "description": "A counter where items can be put on and picked up."},
      "sink1": {
       "initial_state_fragments": ["(is_furniture sink1)","(at sink1 left_loc)",
          "(is_surface sink1)", "(is_washing_station sink1)"],
       "implied_predicates_actions":
          {"predicates": ["(at ?f ?l)", "(is_surface ?f)",
            "(is_washing_station ?f)", "(is_furniture ?f)"],
        "actions_notes": ["Action 'wash' might use the sink."]},
       "description": "A sink where objects are placed, picked up, and washed."},
      "apple1": {
       "initial_state_fragments": ["(is_item apple1)",
                   "(on apple1 counter1)", "(at apple1 right_loc)",
                   "(washed apple1)", "(is_movable apple1)"],
       "implied_predicates_actions": {"predicates":
                   ["(at ?i ?l)", "(on ?i ?f)", "(washed ?i)",
                   "(is_movable ?i)", "(is_item ?i)"],
       "actions_notes": ["Apple can be picked up, placed, and washed."]},
       "description": "A washed apple that is on the counter."},
      "apple2": { ... }},
     "locations": {"door_loc": {"description": "The kitchen door location."},
       "left_loc": {"description": "The location at the left of the door."},
       "right_loc": {"description": "The location at the right of the door."}},
     "universal_initial_state": ["(is_human\_rob rob1)", "(at rob1 door_loc)",
       "(is_furniture counter1)", "(at counter1 right_loc)",
       "(is_surface counter1)", "(is_furniture sink1)", "(at sink1 left_loc)",
       "(is_surface sink1)", "(is_washing_station sink1)", "(is_item apple1)",
       "(at apple1 right_loc)", "(on apple1 counter1)", "(washed apple1)",
       "(is_movable apple1)", "(is_item apple2)", "(at apple2 right_loc)",
       "(on apple2 counter1)", "(not (washed apple2))", "(is_movable apple2)"]},
     "refined_user_stories": [
       {"original_story": "rob1 wants to wash apple2 that is not washed.",
       "refined_goal_description": "The apple should be washed by the robot in
       the sink, which is the only source of water in the kitchen.",
     "pddl_goal_predicates": ["(washed apple2)"]},
       {"original_story": "rob1 wants to bring the two apples at the door.",
       "refined_goal_description": "The apple that is on the sink location
       should be picked up, then the robot should go to pick up the other
       apple on the counter and go to the door.",
       "pddl_goal_predicates": ["(at apple1 door_loc)","(at apple2 door_loc)"]}]}
}, ... ]
```

**Prompt for the Common-sense Validation**

You are a meticulous reviewer with strong real-world commonsense. Your input is a structured JSON object representing an environment and user stories, intended for producing PDDL files for planning. Your task is to review this JSON for any logical inconsistencies, implausible physical states or interactions, or violations of real-world common-sense that would make the scenario unrealistic or unplannable in the real world.

**Key Validation Rules:**

1. **Entity Identification and Counting:** Verify that EVERY distinct physical object instance mentioned in the original environment description is present in `'refined_environment.entities'` with a unique name.
2. **Constraints**: Every physical entity must have exactly one location fact defining its current location. There cannot be humanoid robots or objects that, in a particular moment (neither in the initial state, nor in the goal, nor in an intermediate state), are in two locations at the same time.
3. **Goal Feasibility and Correctness:** Are `'pddl_goal_predicates'` achievable and correctly formulated?
4. **Action Implication Review:** Are `'implied_predicates_actions'` notes sensible and do they correctly detail preconditions/effects for common real-world interactions?
5. **Examples of real-world common sense rules:**
   - Remember that when a humanoid robot/object moves/is moved, it will be in a new location and no longer at the old one.
   - Remember that when an object is on another one, this means they are at the same location.
6. **Output Format:** Crucially, your entire output MUST be a single JSON object, enclosed in ```` ```json ```` and ```` ``` ```` delimiters. This JSON object must be the corrected (or original if no changes) version of the input JSON, in the exact same format. No other text or explanation should precede or follow the ```` ```json ... ``` ```` block. Focus on semantic and commonsense errors based on these rules, not PDDL syntax. Ensure all original fields are present in your output.

**Examples:**
{shots}

**Environment Scenario:**
{environment_description}

**User Stories:**
{user_stories_block}

**Structured Input (JSON for PDDL Generation or Commonsense Validation):**
{structured_input_json}

**PDDL Domain (for Correction):** -

**PDDL Problem (for Correction):** -

**Error Report (for Correction):** -

**Assistant:**

**Prompt for the Initial PDDL Generation**

You are an expert PDDL generator. Your input is a structured JSON object containing a detailed `'refined_environment'` (with entities, a `'universal_initial_state'` for the entire environment, and notes on implied predicates/actions) and a list of `'refined_user_stories'` (each with PDDL goal predicates). Your task is to generate a single, valid PDDL domain and one PDDL problem file for each user story to insert as values respectively in the `'pddl_domain'` and `'pddl_problems'` keys of the JSON format of the provided example. Your entire output MUST be a single JSON object, enclosed in ```` ```json ```` and ```` ``` ```` delimiters. This JSON object must strictly adhere to the format shown in the examples. No other text or explanation should precede or follow the ```` ```json ... ``` ```` block.

**PDDL Domain Generation Rules:**

1. **No Typing:** Do NOT use the `'(:types ...)'` section. All objects are of the default 'object' type.
2. **Predicates:** Define predicates according to the `'implied_predicates_actions'` or those necessary for the actions. All parameters should be untyped (e.g., `'(at ?obj ?l)'`). Crucially, before defining a new predicate, check if the intended meaning can be accurately represented by an existing predicate, possibly using negation. Avoid creating semantically redundant predicates. For example, if a predicate `'(movable ?o)'` exists, represent an `'immovable'` state as `'(not (movable ?o))'` rather than introducing a new predicate such as `'(is_immovable ?o)'`. Furthermore, don't generate predicates that are not used in the action definitions.
3. **Actions:** Define actions based heavily on hints in `'implied_predicates_actions'`. Ensure all standard actions are present. Strictly use untyped variables in action parameters (e.g., `':parameters (?ag ?i ?furn ?loc)'`).
4. **Action Effect Invariants:** For any action that changes a state where an object can only be in one state at a time (e.g., location, temperature), the `':effect'` of that action MUST include both the addition of the new state and the explicit negation of the old state. This is a non-negotiable rule to ensure the domain is logically sound.
   - Example for Location: An action that moves `'?obj'` from `'?from'` to `'?to'` MUST contain the effect `'(and (at ?obj ?to) (not (at ?obj ?from)))'`.
   - Example for Properties: An action that makes `'?obj'` `'hot'` when it was previously `'cold'` MUST contain `'(and (is_hot ?obj) (not (is_cold ?obj)))'`.
5. **Consistency:** Ensure predicate usage is consistent. Avoid defining predicates that are not used in the actions' preconditions and effects.
6. **Requirements:** Use only the requirements supported by untyped PDDL (e.g., `':strips'`, `':negative-preconditions'`). Do NOT include `':typing'`.
7. **Don't use constants**: use only variables in the definitions of predicates and actions.

**PDDL Problem Generation Rules** (one problem per `'refined_user_story'`):

1. **Objects:** Declare all entities from `'refined_environment.entities'` in the `':objects'` section. Do NOT declare types.
2. **Initial State (`':init'`):** For EACH problem, the `':init'` section MUST consist SOLELY of the facts listed in the `'refined_environment.universal_initial_state'`.
3. **Goal (`':goal'`):** Use `'pddl_goal_predicates'` from the corresponding `'refined_user_story'`.

**JSON Output Formatting Rules:**

1. The new JSON must contain in a `'pddl_domain'` key as value the PDDL domain corresponding to the NL description formatted as a string as in the example below.
2. The new JSON must contain in a `'pddl_problems'` key as value a JSON array with the PDDL problems corresponding to the NL user stories formatted as single strings as in the example below.

**Examples:** {shots}
**Environment Scenario:** {environment_description}
**User Stories:** {user_stories_block}
**Structured Input (JSON for PDDL Generation or Commonsense Validation):** {structured_input_json}
**PDDL Domain (for Correction):** -
**PDDL Problem (for Correction):** -
**Error Report (for Correction):** -
**Assistant:**

**Shots for the Examples in the Initial PDDL Generation Prompt**

```
[{"refined_environment": {
   "entities": {
    "rob1": {"init_fragments": ["(is_human\_rob rob1)","(at rob1 door_loc)"],
     "implied_predicates_actions": {
      "predicates": ["(holding ?a ?i)", "(at ?a ?l)", "(is_human\_rob ?a)"],
      "actions_notes": ["rob can move, wash, pick up, and release items."]},
     "description": "The primary humanoid robot."},
    "counter1": {"init_fragments": ["(is_furniture counter1)",
                "(at counter1 right_loc)", "(is_surface counter1)"],
     "implied_predicates_actions": {
      "predicates": ["(at ?f ?l)", "(is_surface ?f)", "(is_furniture ?f)"],
      "actions_notes": ["A surface for items."]},
    "description": "A kitchen counter."},
    "sink1": {"init_fragments": ["(is_furniture sink1)","(at sink1 left_loc)",
                           "(is_surface sink1)", "(is_washing_station sink1)"],
     "implied_predicates_actions": {
      "predicates": ["(at ?f ?l)", "(is_surface ?f)", "(is_washing_station ?f)",
                           "(is_furniture ?f)"],
      "actions_notes": ["A place to wash items."]},
     "description": "A kitchen sink."},
    "apple1": {"init_fragments": ["(is_item apple1)","(on apple1 counter1)",
            "(at apple1 right_loc)", "(washed apple1)", "(is_movable apple1)"],
     "implied_predicates_actions": {
      "predicates": ["(at ?i ?l)", "(on ?i ?f)", "(washed ?i)",
                    "(is_movable ?i)", "(is_item ?i)"],
      "actions_notes": ["Can be picked up and washed."]},
     "description": "A washed apple."},
    "apple2": { ... },
   "locations": {
    "door_loc": { "description": "The door location." },
    "left_loc": { "description": "The location at the left of the door." },
    "right_loc": { "description": "The location at the right of the door." }},
   "universal_initial_state": [
    "(is_human\_rob rob1)", "(at rob1 door_loc)", "(is_furniture counter1)",
    "(at counter1 right_loc)", "(is_surface counter1)", "(is_furniture sink1)",
    "(at sink1 left_loc)", "(is_surface sink1)", "(is_washing_station sink1)",
    "(is_item apple1)", "(at apple1 right_loc)", "(on apple1 counter1)",
    "(washed apple1)", "(is_movable apple1)", "(is_item apple2)",
    "(at apple2 right_loc)", "(on apple2 counter1)", "(not (washed apple2))",
    "(is_movable apple2)"]},
  "refined_user_stories": [
    {"original_story": "rob1 wants to wash apple2.",
    "refined_goal_description": "The apple is washed by the robot in the sink.",
    "pddl_goal_predicates": ["(washed apple2)"]},
    {"original_story": "rob1 wants to bring the two apples at the door.",
    "refined_goal_description": "The apple that is on the sink location should
          be picked up, then the robot should go to pick up the other apple on
          the counter and go to the door.",
    "pddl_goal_predicates": ["(at apple1 door_loc)", "(at apple2 door_loc)"]}],
  "pddl_domain": "(define (domain kitchen) (:requirements :strips
    :negative-preconditions) (:predicates (at ?obj ?loc) (on ?item ?surface)...)
    (:action move_rob :parameters (?ag ?from ?to) :precondition (and
     (is_human\_rob ?ag) (is_location ?from) (is_location ?to) (at ?ag ?from))
     :effect (and (not (at ?ag ?from)) (at ?ag ?to))) ...)",
  "pddl_problems": [
    "(define (problem wash_apple_task) (:domain kitchen) (:objects rob1 apple1
    apple2 counter1 sink1 door_loc left_loc right_loc) (:init
    (is_human\_rob rob1) (at rob1 door_loc) (is_furniture counter1)
    (at counter1 right_loc) ...) (:goal (and (washed apple2))))",
    "(define (problem bring_apples_to_door_task)..." ] }, ... ]
```

## A.2 Syntax Validation

**Prompt for the Syntax Validation**

You are a PDDL debugging expert.
You will be given a PDDL domain, a PDDL problem, and an error report from the VAL PDDL parser. Your task is to analyze the VAL errors and correct the PDDL domain and/or problem to resolve these syntactic errors.
Always No Typing: Do NOT include a `(:types ...)` section. All objects are of the default `object` type.
When making corrections, especially if they involve modifying or adding predicates to the domain, aim for predicate parsimony.
If a concept is needed, first consider if it can be expressed using an existing predicate and negation before introducing a new predicate that might be semantically redundant.
For example, if a predicate `(movable ?o)` exists, represent 'immovability' as `(not (movable ?o))` instead of adding a new predicate like `(is_static ?o)` if `movable` appropriately captures the intended distinction.

Common VAL errors include:

- Undeclared predicates.
- Mismatched number of arguments in predicates or actions.
- Incorrect use of requirements (e.g., using negation without `:negative-preconditions`).
- Syntax errors in definitions, preconditions, effects, or goal statements.
- Missing parentheses at the end of the domain or problem files.

Carefully review the VAL error report. Identify the exact lines or constructs causing issues.
Modify the PDDL domain and/or problem content to fix these errors.
Ensure your corrections maintain the original intent of the domain and problem as much as possible while achieving syntactic validity.

Output the corrected (or the original one if no corrections are made) PDDL as a JSON object with two keys: - *'corrected_pddl_domain'*: The full string content of the corrected PDDL domain. - *'corrected_pddl_problem'*: The full string content of the corrected PDDL problem.
If you believe the domain is correct and only the problem needs changes, return the original domain content. Similarly, if the problem is correct and only the domain needs changes, return the original problem content. If both need changes, provide both corrected versions.

**Examples:**
{shots}

**Environment Scenario:**
{environment_description}

**User Stories:**
{user_stories_block}

**Structured Input (JSON for PDDL Generation or Commonsense Validation):**
{structured_input_json}

**PDDL Domain (for Correction):**
{pddl_domain_content}

**PDDL Problem (for Correction):**
{pddl_problem_content}

**Error Report (for Correction):**
{error_report}

**Assistant:**

**Shots for the Examples in Syntax Validation Prompt**

```
[{"input_pddl_domain":
   "(define (domain flawed_example) (:requirements :strips)
     (:predicates (at ?obj ?loc) (holding ?rob ?item) (on ?item))
     (:action pickup :parameters (?a ?i ?l)
      :precondition (and (at ?i ?l) (not (holding ?a ?i)))
      :effect (and (holding ?a ?i) (not (at ?i ?l)) (rob_at ?a ?l))))",
  "input_pddl_problem": "(define (problem flawed_task) (:domain flawed_example)
   (:objects apple bob loc desk1)
   (:init (at apple desk1) (rob_at bob loc) (on apple))
   (:goal (holding bob apple)))",
  "val_error_report": "VAL STDOUT: Error: Predicate 'on' in action 'pickup' has
   arity 2, but is used with 1 arguments in the :init of the problem.
   Error: Predicate 'rob_at' used in action effect 'pickup', but not
   declared in :predicates declaration! VAL STDERR: VAL Return Code: 1",
  "output_corrected_pddl_json": {
   "corrected_pddl_domain": "(define (domain corrected_example)
    (:requirements :strips :negative-preconditions)
    (:predicates (at ?o ?l) (holding ?a ?i) (on ?i ?s) (is_human\_rob ?a)
    (is_item ?i) (is_surface ?s) (is_location ?l))
    (:action move_rob :parameters (?ag ?from ?to)
    :precondition (and (is_human\_rob ?ag) (is_location ?from) (is_location ?to)
    (at ?ag ?from)) :effect (and (not (at ?ag ?from)) (at ?ag ?to)))
    (:action pickup_item_from_surface :parameters (?ag ?it ?surf ?l_loc)
    :precondition (and (is_human\_rob ?ag) (is_item ?it) (is_surface ?surf)
    (is_location ?l_loc) (at ?ag ?l_loc) (at ?it ?l_loc) (on ?it ?surf)
    (not (holding ?ag ?it))) :effect (and (holding ?ag ?it) (not (on ?it ?surf))
    (not (at ?it ?l_loc)))) (:action place_item_on_surface :parameters
    (?ag ?it ?surf ?l_loc) :precondition (and (is_human\_rob ?ag) (is_item ?it)
    (is_surface ?surf) (is_location ?l_loc) (at ?ag ?l_loc) (at ?surf ?l_loc)
    (holding ?ag ?it)) :effect (and (not (holding ?ag ?it)) (on ?it ?surf)
    (at ?it ?l_loc))))",
   "corrected_pddl_problem": "(define (problem corrected_task)
   (:domain corrected_example) (:objects apple1 bob1 loc desk1)
   :init (is_item apple1) (is_human\_rob bob1) (is_location loc)
   (is_surface desk1) (at desk1 loc) (on apple1 desk1) (at apple1 loc)
   (at bob1 loc)) (:goal (holding bob1 apple1)))"} }, ...]
```

### A.3 Planning and Progression

**Prompt for the Corrections After Unsuccessful Planning and Progression**

You are an expert in PDDL planning and debugging.
You will be given a PDDL domain, a PDDL problem, an existing PDDL plan (which might be flawed or incomplete), and the output from running `'val Validate -v domain.pddl problem.pddl plan.txt'` (referred to as `'Progressor Output'`).
If the Progressor Output is missing the string `'Plan Validation details'`, there are issues with the plan's execution or validation.
Always No Typing: Do NOT include a `'(:types ...)'` section. All objects are of the default `'object'` type.

Your task is to:

1. Analyze all inputs to understand why the plan might be failing during VAL's verbose validation, or why `'Plan Validation details'` is absent (e.g., plan is syntactically invalid, preconditions not met, unexpected effects, VAL tool issue hinted by output).
2. If the PDDL Problem seems to be the cause (e.g., the goal is inconsistent), provide a corrected PDDL Problem.
3. Primarily, generate a new, syntactically correct PDDL Plan that is likely to be valid for the given domain and (original or corrected) problem, and would result in VAL's verbose output containing 'Plan Validation details'. The plan should be a sequence of actions.
4. Focus on correcting the problem.

Output your response as a JSON object with the following keys:

- `'corrected_pddl_problem'`: The full string content of the corrected PDDL problem. If no changes are needed to the problem, return the original problem content.
- `'corrected_plan'`: The full string content of the new PDDL plan. If you cannot generate a new plan or believe the original plan is usable with problem corrections, this can be the original plan or empty (though a new plan is preferred if the original failed).

Ensure the plan adheres to typical PDDL plan formats (e.g., sequence of `'(action-name param1 param2)'`).

**Examples:**
{shots}

**Environment Scenario:**
{environment_description}

**User Stories:**
{user_stories_block}

**Structured Input (JSON for PDDL Generation or Commonsense Validation):**
{structured_input_json}

**PDDL Domain (for Correction):**
{pddl_domain_content}

**PDDL Problem (for Correction):**
{pddl_problem_content}

**Error Report (for Correction):**
{error_report}

**Assistant:**

**Shots for the Planning and Progression Corrections**

```
[{"input_pddl_domain_content": "(define (domain example-domain)
   (:requirements :strips)  (:predicates (at ?x) (connected ?x ?y))
   (:action move :parameters (?from ?to) :precondition (and (at ?from)
   (connected ?from ?to)) :effect (and (not (at ?from)) (at ?to))))",
  "input_pddl_problem_content": "(define (problem example-task)
   (:domain example-domain) (:objects a b c) (:init (at a)
   (connected a b) (connected b c)) (:goal (at c)))",
  "input_plan_content": "Plan for example-task(move a b)(move b d);
   error: object d does not exist",
  "input_progressor_output_content": "Checking plan: plan_example-task.txt
   Plan File: plan_example-task.txt
   Validating domain.pddl problem_example-task.pddl plan_example-task.txt...
   Instantiating... problem appears to be unsolvable -- no plan will be found!
   Plan failed to validate. Final value: 0.000
   Problem file contains no errors.
   Something is wrong with the plan/problem/domain combination.",
  "output_json_example_for_shot": {
    "corrected_pddl_problem": "(define (problem example-task)
    (:domain example-domain) (:objects a b c)
    (:init (at a) (connected a b) (connected b c)) (:goal (at c)))",
    "corrected_plan": "; Plan for example-task (move a b)(move b c)"}},
 {"input_pddl_domain_content": "(define (domain gripper) ...)",
  "input_pddl_problem_content": "(define (problem prob01) ...)",
  "input_plan_content": "...",
  "input_progressor_output_content": "...",
  "output_json_example_for_shot": {
    "corrected_pddl_problem": "(define (problem prob01) ...)",
    "corrected_plan": ... }}, ... ]
```

## A.4 Invariants Analysis

**Prompt for the Analysis of the Extracted Invariants**

You are an expert in PDDL generation and logic.
You will be given a PDDL domain, the original natural language description, the user stories, and a PDDL invariants report containing structural and semantic invariants.
Your task is to perform a deep analysis of the extracted invariants, using them as a measure for the quality of the produced PDDL, according to these two major questions:

- **Soundness**: Are the found invariants all correct?
- **Completeness**: Are there any invariants missing?

Assess the invariants based on the proper abstraction layer, similar to the model. Do not overcomplicate things. Then, use the insights you got from the analysis to suggest modifications to improve the generated PDDL domain to address any logical flaws, inconsistencies, or modeling inefficiencies revealed by the report.

1. **Analyze the Structural Invariants (from Fast Downward):**
   - *Mutex Groups:* This is critical. If `'mutex_groups'` is empty, it signifies a major flaw. The actions do not correctly model mutually exclusive states (e.g., an object cannot be in two places at once). You MUST revise the actions' effects to include `'(not (predicate ...))'` for the old state.
   - *Variables:* A high number of boolean variables for related concepts (e.g., `'at-a'`, `'at-b'`, `'at-c'`) instead of a single multi-valued variable indicates poor modeling. While you cannot change to SAS+, you should ensure the PDDL actions enforce the logic that would lead to a factored representation (i.e., ensure mutexes are explicit).
2. **Analyze the Semantic Invariants (from TIM):**
   - *Transition Rules:* These are the "laws of physics" of your domain.
   - *State Machine Discovery:* TIM might reveal state machines (e.g., `'on-surface' -> 'held' -> 'on-surface'`). If these transitions are illogical or incomplete, your action definitions are flawed.

**Correction Protocol:**

- *Prioritize Invariant Feedback:* Your corrections should directly address the issues highlighted by the invariants report.
- *Revise Actions:* Modify preconditions and effects to enforce logical consistency and generate correct invariants.
- *Refine Predicates:* Only add new predicates if absolutely necessary. Prefer using negation `'(not ...)'` to model opposing states.

Produce a new, corrected version of the PDDL domain between ```` ```pddl ```` and ```` ``` ````, following the output format reported in the "*Corrected PDDL Domain:*" part of the following example.

Example:
*Flawed PDDL Domain:*

```
```pddl (define (domain flawed_move) (:requirements :strips) (:predicates (at
?r ?l)) (:action move :parameters (?r ?from ?to) :effect (at ?r ?to) )) ```
```

*Corrected PDDL Domain:*

```
```pddl (define (domain corrected_move) (:requirements :strips
:negative-preconditions) (:predicates (at ?obj ?loc) (is_robot ?obj)
(is_location ?loc) ) (:action move :parameters (?r ?from ?to) :precondition
(and (is_robot ?r) (is_location ?from) (is_location ?to) (at ?r ?from)) :effect
(and (at ?r ?to) (not (at ?r ?from))) ))```
```

Please evaluate the following PDDL generation task, paying close attention to the Invariant Analysis report.

1. **Generated PDDL Domain:**
   ```` ```pddl {pddl_domain} ``` ````
2. **Invariant Analysis Report:**
   ```` ```json {invariants_report} ``` ````

**Your Evaluation:**

**Prompt for the PDDL Update based on the Invariants Analysis**

You are an expert PDDL modeler.
Your task is to correct a PDDL domain file based on an invariants report from an analysis tool. The report will highlight inconsistencies, logical flaws, or missing elements, and potential solutions.
Carefully apply the suggested changes to improve the domain's correctness and completeness.
Output only the corrected PDDL domain in a JSON object.

**Examples:**

```
{
  "input": {
    "pddl_domain_content": "(define (domain flawed-domain) ... )",
    "invariants_report": "The action 'move' is missing a precondition to check
                          if the destination is clear."
  },
  "output": {
    "domain": "(define (domain corrected-domain) ...
               (:action move ... :precondition (and (clear ?to))) ... )"
  }
}
```

**Environment Scenario:**
{environment_description}

**User Stories:**
{user_stories_block}

**Structured Input (JSON for PDDL Generation or Commonsense Validation):**
{structured_input_json}

**PDDL Domain (for Correction):**
{pddl_domain_content}

**PDDL Problem (for Correction):**
{pddl_problem_content}

**Error Report (for Correction):**
{error_report}

**Assistant:**

### A.5 LLM-as-a-judge

**Prompt for the Evaluation**

You are an expert in PDDL and natural language understanding, acting as a rigorous evaluator. Your task is to provide a quantitative assessment of how well a generated PDDL specification captures a natural language description and a set of user stories, which are representative but not exhaustive (several other user stories are not explicit). We are explicitly not modeling `:types` in the domain, so avoid judging based on that.
Carefully analyze the provided materials:

1. The Natural Language Environment Scenario Description.
2. The Natural Language User Stories.
3. The generated PDDL Domain file.
4. The generated PDDL Problem files.

Your entire output **MUST** be a string with the following structure:

- Domain Modeling Score: $<$`integer` from $0-100>$
- Problem Formulation Score: $<$`integer` from $0-100>$
- Rationale: $<$`string`$>$

Use the following rules to determine the scores:

1. **Domain Modeling Score (0-100 points)**: This score assesses how well the PDDL domain file captures the predicates needed for the environment and the actions from the NL scenario, as well as the common-sense implications.
   - (30 pts) Predicates: Are the predicates accurately representing the described properties?
   - (35 pts) Actions: Do the action schemas (preconditions and effects) make logical sense? Do they comply with common-sense? (e.g., a humanoid robot must be at a location to interact with an object there; picking something up means it's no longer at its old location).
   - (35 pts) Sufficiency: Are the predicates and the actions defined sufficient to solve the tasks described in the user stories? (e.g., if a user story is "wash an apple", is there a 'wash' action or a sequence of actions that can achieve this?)
2. **Problem Formulation Score (0-100 points):** This score assesses how well the PDDL problem files capture the specific scenarios and goals from the NL user stories.
   - (25 pts) Entities: Are all relevant entities (objects, humanoid robots) and locations from the NL scenario present in the PDDL?
   - (25 pts) Initial State: Does the initial state `` `:init` `` block of the first problem accurately and completely represent the initial state of the world as described in the NL?
   - (25 pts) State Progression: For subsequent problems, is their initial state the final state of the immediately previous problem?
   - (25 pts) Goal: How well does the `` `(:goal ...)` `` section of each problem file capture the intent of its corresponding NL user story?
3. **Rationale (string)**:
   - Provide a concise but detailed explanation for your scores.
   - Justify every point deduction.
   - Mention any key strengths that led to a high score.

Please evaluate the following PDDL generation task.

1. Natural Language Environment Scenario:
   ```` ```text {nl_description} ``` ````
2. Natural Language User Stories (Goals for which PDDL problems were successfully generated and provided below):
   ```` ```text {nl_user_stories_formatted} ``` ````
3. Generated PDDL Domain:
   ```` ```pddl {pddl_domain} ``` ````
4. Generated PDDL Problems (corresponding to the user stories above):
   {`pddl_problems_formatted`}

**Your Evaluation:**
   ````