CoSP-TL: Common Sense enhanced Planning under Temporal Logics constraints with LLMs*

Emanuele Musumeci[1,*,†], Alessio Saladino[1,†], Elena Umili[1], Luca Iocchi[1] and Daniele Nardi[1]

[1]Sapienza University of Rome

Abstract

In this paper, we present CoSP-TL, a two-level architecture that combines common-sense reasoning knowledge from Large Language Models with formal symbolic planning to address temporally constrained planning problems expressed in natural language. Our approach employs a hierarchical planner composed of a High-Level Planner, which iteratively refines a natural-language plan until it satisfies a Linear Temporal Logic formula encoding the problem constraints, and a Low-Level Planner, which translates the high-level plan into executable PDDL with extensive validation and error-recovery mechanisms. We evaluate our framework on the recent and challenging LexiCon benchmark across three IPC domains, Blocksworld, Logistics, and Sokoban considering an increasing number of temporal constraints over domain fluents. Results show that our approach significantly outperforms standard LLM-based planning, achieving strong performance on unconstrained problems and maintaining robust results as the number of constraints increases, highlighting the scalability of CoSP-TL in handling temporally constrained planning tasks at which standard LLM planners exhibits substantial performance degradation.

Keywords

Neuro-Symbolic Planning, LLM, Temporal Logics

1. Introduction

Classical planning methods rely on formal symbolic models to represent domain knowledge, including action rules, preconditions, and effects, typically expressed in declarative languages such as PDDL. These models enable reliable reasoning and execution in structured decision-making tasks but assume the availability of precise, formally specified task descriptions, whose construction is challenging in complex, compositional domains and often demands substantial domain expertise. Large Language Models (LLMs) offer a promising alternative by acting either as planners or as translators from natural language into formal symbolic representations, leveraging common-sense knowledge acquired during pretraining. This paradigm assumes that much of the knowledge required for planning is implicit and can be reconstructed by LLMs, yet such implicit reasoning creates a gap between the internal self-consistency of the generated planning domain and the groundability of the resulting plan in real-world robotic environments, where symbols must correspond to actual objects and executable actions. As a result, recent direct LLM-as-planner approaches, which bypass intermediate grounding, remain largely unsuitable for embodied agents: generated plans may appear coherent while still containing token-level hallucinations, such as misaligned symbols or action and argument names that do not correspond to the underlying domain specification. This challenge becomes even more pronounced when formal constraints are introduced, as recent benchmarks such as LexiCon show that performance of LLM-based planners degrades substantially with the number and complexity of temporal constraints. In this context, Linear Temporal Logic (LTL) [1] and its finite-trace variant LTLf [2] provide expressive

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.

*Corresponding author.

†These authors contributed equally.

musumeci@diag.uniroma1.it (E. Musumeci); saladino@diag.uniroma1.it (A. Saladino); umili@diag.uniroma1.it (E. Umili); iocchi@diag.uniroma1.it (L. Iocchi); nardi@diag.uniroma1.it (D. Nardi)

https://emanuelemusumeci.com (E. Musumeci); https://alessiosal98.github.io/AlessioSaladino/ (A. Saladino); https://sites.google.com/view/elenaumili/home (E. Umili)

0009-0004-2359-5032 (E. Musumeci); 0009-0009-1553-6127 (A. Saladino); 0000-0002-5639-6038 (E. Umili); 0000-0001-9057-8946 (L. Iocchi); 0000-0001-6606-200X (D. Nardi)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-03

CEUR-WS.org/Vol-4311/CARLA_regular_03.pdf

formalisms to specify temporal constraints over system executions, making them a natural choice for constrained planning in robotics. To address these challenges, we propose CoSP-TL (Common-Sense Enhanced Planning under Temporal Logic Constraints), a two-level hierarchical framework for temporally constrained planning from natural language instructions. At the high level, CoSP-TL exploits LTLf constraints to autoregressively construct plans while ensuring that the generated trace satisfies the temporal specification before a final plan is produced. The resulting high-level plan is then passed to a low-level planner, which translates each instruction into an executable symbolic plan, ensuring consistency and executability with respect to the ground-truth domain. We evaluate CoSP-TL on the LexiCon benchmark across three IPC domains, Blocksworld, Logistics, and Sokoban, under an increasing number of temporal constraints over domain fluents. Experimental results show that CoSP-TL significantly outperforms standard LLM-based planning approaches, achieving strong performance on unconstrained tasks while maintaining robust results as constraint complexity increases, highlighting the effectiveness and scalability of CoSP-TL for temporally constrained planning settings where standard LLM planners exhibit substantial performance degradation. The implementation is available at: https://github.com/EmanueleMusumeci/CoSP-TL.

2. Related Works

Natural language to formal planning models The challenge of translating natural language into formal PDDL has gained traction as a way to bridge informal requirements with classical planning. Recent work explores the use of LLMs to transform incomplete or ambiguous natural language specifications into formal planning representations, a process referred to as *model elicitation* [3] [4], where structural elements of a planning domain are inferred by extracting implicitly modeled knowledge from LLMs through textual prompts [5] [6]. This comes at the cost of limited control over the fine-grained precision of predicted tokens. While this token-level flexibility benefits many NLP tasks where multiple outputs can express the same concept [7], it negatively impacts requirements for strict structural and token-level precision [8]. Prior work shows that LLMs encode structural priors over symbolic representations. For instance, contextual PDDL action names can provide sufficient semantic cues to guide coherent action sequence generation [9]. Nevertheless, these associations may be biased toward the training distribution, reducing robustness in complex or out-of-distribution scenarios [10]. In this work, we address the limitations of model elicitation by proposing a hierarchical architecture with two abstraction levels, aiming to improve control over token generation and produce executable plans.

LLM-based neuro-symbolic planning Recent approaches attempt to bypass the explicit construction of symbolic planning domains. frequently sacrificing semantic validity and executability of produced plans, which may appear reasonable and executable, while containing token-level hallucinations like misnamed symbols, actions or arguments. These methods typically begin with a model elicitation stage that produces a formal intermediate representation of the planning domain, then used to derive a grounded plan. NL2Plan [11] employs a 6-step extraction pipeline to synthesise full domains and problems from minimal text. LLM+P [12] assumes a provided ground-truth domain and translates natural language instructions into PDDL problem files. ISR-LLM [13] refines PDDL problems against ground-truth domains using iterative prompting guided by formal validation. Planning in the Dark [14] bypasses expert intervention through a schema library and semantic validation. Other frameworks rely on intermediate representations: TIC [15] translates natural language first to Answer Set Programming (ASP) before compiling to PDDL. Multi-agent architectures have also been explored, e.g. La Malfa et al. [16], who use iterative validator feedback. Tools like PDDL-GenAI [17] and recent surveys [18] reflect a growing ecosystem of LLM-based support in domain modeling. As Vallati et al.[19] note, formal validators remain critical for robust deployment. To improve reliability, several works incorporate external validation mechanisms into the generation process. LLM-Modulo [20] formalizes this paradigm by combining LLMs with symbolic or learned validators to iteratively refine outputs. Similarly, ISR-LLM [13] integrates validation feedback directly into the prompting loop. These approaches emphasize

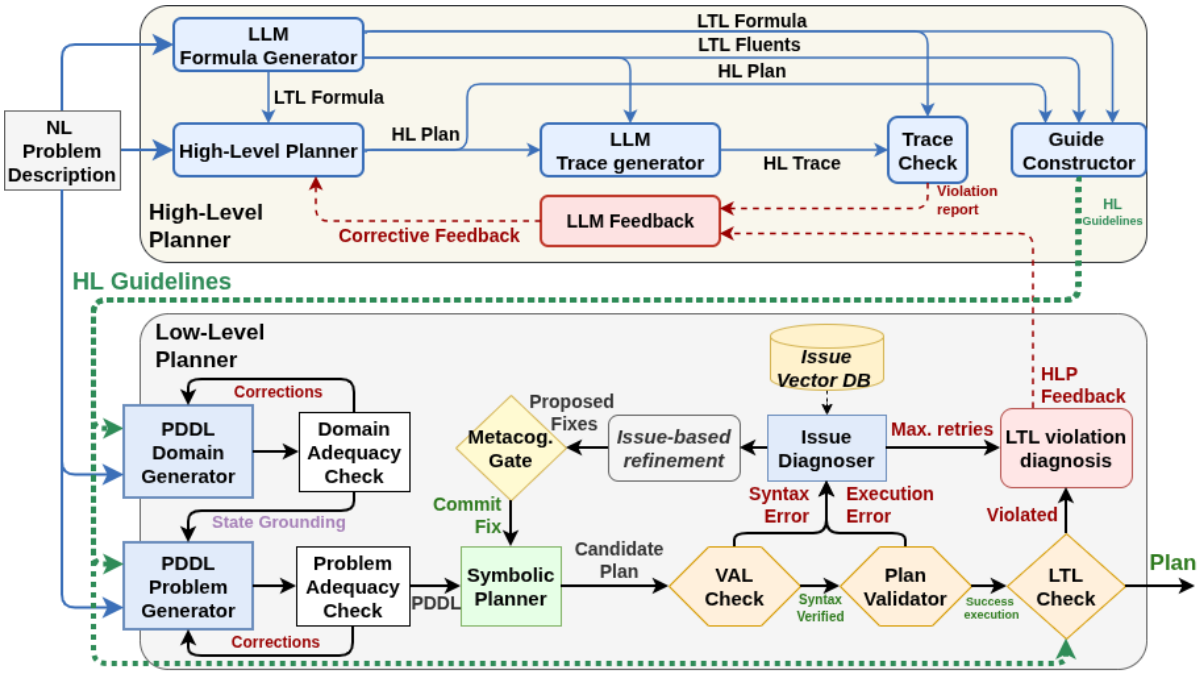


Figure 1: Overall CoSP-TL architecture. The High-Level Planner (HLP) receives the input problem/domain/constraints description and produces an ordered list of natural language subgoals. The Low-Level Planner (LLP) solves the subgoals one by one, using the input description to elicit symbolic PDDL. In case of failure in the LLP, an additional feedback channel, represented by the dashed arrow, triggers the HLP to fix the failing subgoal and regenerate the following ones, which are then processed again at the low level.

the importance of external verification for ensuring semantic correctness in LLM-generated planning artifacts. Planning under temporal logic constraints, such as Linear Temporal Logic on finite traces (LTLf), extends classical planning by requiring the execution traces to satisfy additional temporal constraints in addition to the goal condition. In this context, LexiCon [21] offers a benchmark for evaluation compliance with such temporal specifications. Prior works has explored neuro-symbolic approaches that combine learning-based components with symbolic reasoning [22], [23], [24]. In this work, we address constrained plan generation with a hierarchical architecture that translates problem constraints into an LTLf formula, which guides both high-level planning and low-level plan refinement.

3. Methodology

The problem of eliciting at the same time a temporally constrained high-level plan and a symbolic model to compute a solution plan is an extremely challenging task. Temporally extended tasks require pipelines that exhibit agentic capabilities, particularly from the point of view of working memory management. Moreover, although LLMs exhibit in general good capabilities as high-level planners [25], the token-level ambiguity constitutes an obstacle for formal model elicitation tasks. We propose a hierarchical architecture, shown in Figure 1 and described in Alg. 1, that exploits both high-level planning grounding capabilities [26] of LLMs. The system is composed of a High-Level Planner (HLP) and a Low-Level Planner (LLP): the former interprets domain/problem/constraints in natural language, producing a sequence of high-level natural language subgoals, while the latter translates these instructions into low-level, executable actions. Together they form a hierarchical refinement loop. The HLP regenerates the high-level plan whenever the LLP diagnoses a failure. For each subgoal, the LLP runs a refinement loop to refine the symbolic domain and problem using feedback from a symbolic validator.

Algorithm 1 CoSP-TL planning pipeline. Colours correspond to Figures 1-3: HLP, LLP, symbolic planner, validator/simulator, state grounding, init-diff gate, cross-level feedback.

Require: Natural-language problem P with objects, initial state, goal, and constraints
Ensure: Executable plan π satisfying the goal and the LTLf formula

$\Phi_{NL} \leftarrow$ NLInterpreter(P) ▷ paraphrase + predicates
$\varphi_{LTLf} \leftarrow$ FormulaGenerator(Φ_{NL})
$\mathcal{C}^* \leftarrow$ ConstraintOrderSelector(φ_{LTLf}) ▷ Imm/Excl/Dep/Cond/Glob
fb $\leftarrow \emptyset$
for $t = 1 \dots T_{HLP}$ **do** ▷ outer HLP attempts
  $\Pi \leftarrow$ HLPlanner($\Phi_{NL}, \mathcal{C}^*$, fb) ▷ subgoals $g_1 \dots g_n$
  $\tau \leftarrow$ TraceGenerator(Π)
  ok $\leftarrow$ TraceCheck(τ, φ_{LTLf})
  if not ok **then**
    fb $\leftarrow$ TraceCorrector(τ, φ_{LTLf}) ; **continue**
  $\Sigma \leftarrow$ PDDLDomainGen(Φ_{NL})
  for $k = 1 \dots n$ **do** ▷ per-subgoal LLP
    $s_k \leftarrow$ StateGrounding(P, sim$_{k>1}$) ▷ NL at $k = 1$, sim at $k > 1$
    $\mathcal{P}_k \leftarrow$ PDDLProblemGen(Σ, s_k, g_k)
    for $r = 1 \dots R$ **do** ▷ outer refinement
      $\pi_k \leftarrow$ Plan($\Sigma, \mathcal{P}_k$)
      if $\pi_k \neq \emptyset \wedge$ Simulate(π_k) ok **then break**
      $\mathcal{I} \leftarrow$ IssueDiagnoser($\Sigma, \mathcal{P}_k$, errors)
      for $i \in \mathcal{I}$ **do** ▷ per-issue inner loop
        $(\mathcal{P}_k', \Sigma') \leftarrow$ Corrector($i, \Sigma, \mathcal{P}_k$)
        accept $\leftarrow$ InitDiffGate($\mathcal{P}_k, \mathcal{P}_k'$)
        if accept **then** commit $(\mathcal{P}_k', \Sigma')$
      temp $\leftarrow$ TempLadder(group($\mathcal{I}$), r)
    if $\pi_k = \emptyset$ **then**
      fb $\leftarrow$ TranslateFailure($\mathcal{I}$) ; **break**
  if all π_k succeed and GTExec($\pi_1 \dots \pi_n$) ok **then**
    return $\pi \leftarrow \pi_1 \cdot \pi_2 \cdots \pi_n$
return $\emptyset$

World Objects: Blocks brown_block_1, grey_block_1, brown_block_2, blue_block_1, red_block_1.

Original World State: brown_block_1 is on the table; grey_block_1 is on top of brown_block_1; brown_block_2 is on top of grey_block_1; blue_block_1 is on the table; red_block_1 is on top of brown_block_2; no block on top of blue_block_1; there is no block on top of red_block_1; you are not holding any block.

Goal and Constraints: The task is to bring about the following situation: grey_block_1 is on the table.
A valid plan must satisfy: the following expression must hold in at least one state, "you are holding brown_block_2"; if that expression holds in some state s, then "you are holding blue_block_1" must hold at s or after s; the following expression must hold in at least one state, "brown_block_1 is not on top of grey_block_1"

Running example: system input We provide an example problem from the blocksworld domain, further denoted as P_{ref} in the remainder of this manuscript. We will show the contribution of each component of the architecture and the evolution of the input from a problem expressed in natural language to a low-level executable plan. Our system starts taking as input a description in natural language of the constrained problem to be solved. The latter is shown below and includes: (I) a description of the domain and its rules, along with its actions pre-conditions and post-conditions, (II) a list of the objects present in the problem (III) a description of the problem initial state, (IV) a description of the goal, or desired final state, (V) a list of constraints that must be satisfied during the plan execution.

3.1. High Level Planner

The High-Level Planner (HLP) receives a natural language input, from which it builds and iteratively refines a high-level plan, that satisfies all the given natural language constraints. The HLP output consists in a sequence of natural language instructions, used as subgoals in the LLP. These instructions have low granularity, meaning that in some cases a single instruction of the HLP can be decomposed in multiple steps by the LLP in order to become executable. The high-level system of our framework is composed by the a NL interpreter, a formula generator, a HL Planner module, a trace generator and a trace corrector, described below. We can summarize the general pipeline of the HLP as follows:

1. The NL Interpreter analyzes the Domain/Problem to detect relevant objects and predicates;
2. From the extracted formalism, the Formula Generator encodes the NL constraints to LTLf;
3. The HLP generates a plan as a sequence of NL instructions, which is supposed to both reach the goal from the initial state and satisfy the constraints at the same time;
4. The Trace Generator generates a trace representing the plan trajectory, which is checked with the formula; if the trace satisfies the formula, the plan is forwarded to the low-level planner, otherwise the LLM Feedback module formulates a feedback about the reason of failure the formula used by the HL planner to correct the wrong steps.

In the following paragraphs we provide additional implementation details high-level modules.

NL Interpreter The NL Interpreter extracts the relevant semantics from the input Domain/Problem/-Constraints. Although the domain description is provided in NL, object names or state descriptions still have a structure very similar to the symbolic one used at the low level. To reduce the risk of hallucinations related to overly rigorous syntax, the original information is reinterpreted in a more discursive manner (e.g., the state description *"red_block_1 is on the table"* becomes *"the first red block is located on the table"*). Specifically, the extracted information is divided into: the set of *actions* allowed by the domain; an extensive description of the *initial and goal states*; the set of problem *constraints*; the set of *objects* that appear in the problem; the set of *predicates* that can be associated with the objects according to the domain, expressed in FOL syntax (i.e. onTop(X,Y), holding(X) etc.); a natural language description of *predicates semantics* that explains their semantics. This division of information enriches context provided to subsequent modules, mitigating hallucinations of objects instances and predicates.

Formula Generator The Formula Generator is an LLM that receives in input the Domain/Problem/-Constraints natural language interpretation from the NL interpreter and generates a LTLf formula that represents the constraints of the problem. To enforce the model capability to understand LTLf syntasx, we have injected the description of the main LTLf operators (F, G, X, U, $\wedge$, $\vee$, $\neg$, $\Longrightarrow$) into the prompt. The generated formula will be used by subsequent modules to verify that the plan meets all constraints. The goal is not included in the formula, but is managed separately. In the case of P_{ref}, the NL constraints shown in "Running example" are represented by the following formula:

$$F(holding(brown_block_2)) \wedge$$
$$G(holding(brown_block_2) \Rightarrow F(holding(blue_block_1))) \wedge$$
$$F(\neg on_top(brown_block_1, grey_block_1))$$

HL Planner Module This module, responsible for generating a plan in natural language, is the core of the high-level architecture. Initially, a separate LLM works as a heuristic to decide in which order the constraints must be satisfied inside the plan. This prevents that a constraint may be satisfied at some point in the plan but become later unsatified, when the planner is trying to solve another constraint. We ask the LLM to order the constraints solution according on the factors of *Immediacy*, *Exclusivity*, and *Dependency*. "Immediate" constraints can be solved in one or a few moves and, once solved, they can then be ignored (i.e. the F(...) constraints). Exclusive constraints involve objects that appear only within that constraint. These two types of constraints are assigned a higher priority and the planner

will try to solve them first. Constraints involving Dependencies (e.g. the same object appearing in multiple constraints) have a lower priority. In P_{ref}, constraints are ordered by the LLM as follows:

1. $F(holding(brown_block_2))$
2. $F(!on_top(brown_block_1, grey_block_1))$
3. $G(holding(brown_block_2) \Rightarrow F(holding(blue_block_1)))$

Notice how the Immediate constraints (F operator) were selected first by the LLM heuristic, with the most elementary one involving only black_block_2 being selected first. The ordered constraints are managed by the Planner, which constructs the plan auto-regressively: each LLM call generates the subsequent action/state pair from the previously calculated states. The planner starts with a description of the initial state S_0 extracted from the NL Interpreter module. For each constraint C_i, it defines a subsequence A_i of actions to solve the constraint and builds the description of the new state after the execution of the A_i actions. The new natural language state S_i is given as input with the next constraint C_{i+1} and the plan is iteratively built until all constraints are processed. The goal C_g is eventually processed like the other constraints. Below we show the HLP generated for P_{ref}.

1. **Instruction:** brown_block_2 is held by the robotic arm. ⇒ **Outcome:** constr. 1 satisfied, precondition of constr. 3 activated;
2. **Instruction:** grey_block_1 is no longer on top of brown_block_1 by removing grey_block_1 from it. ⇒ **Outcome:** constr. 2 satisfied;
3. **Instruction:** Robotic arm holds blue_block_1. ⇒ **Outcome:** post-condition of constr. 3 satisfied;
4. **Instruction:** Ensure grey_block_1 is on the table. ⇒ **Outcome:** Goal is satisfied.

Trace Generator and Trace Corrector The Trace Generator generates an intermediate state trace of the plan by using the object and predicate notation provided by the NL Interpreter and the natural language plan. This trace does not correspond to a real execution trace (which is provided by the low-level planner), but is rather a prediction of the trace provided by an LLM according to the sequence of instructions. This trace is compared state-by-state by the Trace Corrector with the formula to verify that the constraints are satisfied. If satisfied, the plan is sent unchanged to the low-level. If one or more constraints are violated, the Trace Corrector will describe where the formula has been violated and provide a suggestion to the planner to reformulate the plan correctly.

3.2. Low-Level Planner

Once the HLP has generated a high-level plan that is expected to satisfy the constraints while achieving the goal, the role of the Low-Level Planner (LLP) is to translate the received set of high-level natural language instructions into a sequence of primitive actions actually executable within the domain.

To accomplish this, the LLP performs the following operations, detailed in the following paragraphs:

1. A PDDL domain $\mathscr{D}$ is generated from the initial NL input, and is reused across all subgoals.
2. For each high-level NL subgoal g_i, a problem is generated, grounded to the current world state $\mathscr{P}_i = \langle \mathscr{D}, s_i, g_i \rangle$, given the initial state s_i;
3. Planning is attempted. A *symbolic validator* simulates the plan in the generated domain and plan consistency is evaluated.
4. Issues highlighted by the validator are classified in a structured taxonomy of issue types (*structural* or *reasoning* issues). Each issue is then repaired in an order determined by an associated priority, using a group-specific temperature schedule. A vector-database ensures that past fixes are reused. *Metacognitive gating* judges any refinement modifying the initial problem state, to prevent hallucinations. Problem is refined a fixed amount of times until a valid plan can be computed.
5. Any plan π_i reaching this point is self-consistent. In case LTLf constraints are not satisfied, a NL diagnostic of the failure is emitted, triggering a HLP iteration to repair the failing subgoal.
6. Evaluation is conducted over a ground truth simulator to verify real-world executability.

STRUCTURAL T : 0.05→0.15 **(step** 0.02**)**

P0_SYNTAX
corrective rewrite, retains XML wrapping

P2_GOAL_DRIFT
rewrite goal block from subgoal NL, preserve `:init`

Tight low-temperature ladder, minimal prompt deviation.

REASONING T : 0.20→0.70 **(step** 0.10**)**

P1_TYPE - vector-DB past-fix reference
P1_PREDICATE - predicate-vocabulary anchor
P2_ACTION - domain re-diagnose; init-diff gate
P2_HALLUCINATION - init-diff reject; one-shot regen
P3_OTHER - default-temp rewrite

Figure 2: Taxonomy used by the per-issue refinement loop, organised in two groups, with a corresponding *temperature ladder* (lower for the *structural* group, steeper for the *reasoning* group, which additionally consults the vector database of past fix patterns). Each issue type has an associated priority `P*`.

PDDL generation. The PDDL domain $\mathscr{D}$ is initially generated from the NL input, encoding basic action schemas, types, and predicates. As in [11], type definitions, predicates, and constants from the generated domain are provided to the problem generation step, ensuring lexical consistency. The problem generator only accesses the initial state description at the first subgoal. Ground truth is never read by the problem generator at subsequent subgoals (see Section 4.1). Only the current subgoal is provided to the LLM prompt, to prevent the LLM from improving goal reachability by inserting goal-state predicates directly into the domain definition. For instance, on *blocksworld*, the generator was observed introducing an auxiliary predicate `(goal-stack ?x ?y)` into `:predicates` and adding an action `achieve-goal` with empty preconditions and effects `(goal-stack ?x ?y)`, so that the planner trivially reached the formal goal in a single step while the underlying stacking semantics were never enforced. Restricting visibility on the goal description removes this short-circuit forcing the LLM to reach the goal through a more reasonable action trajectory. A vocabulary of object instances and constants is built by normalizing names (e.g. "a purple block" is normalized into "purple_block_1") and used for all subgoals. Symbolic generation is preceded and followed by two distinct chain-of-thought checks over the *semantic adequacy* of the generated symbolic domain. The *domain adequacy check*, taking place after domain generation, proposes improvements on action schema coverage over predicates observed at the initial state of the current subgoal. Previous generation issues that break symbolic validation are reverted. The *problem adequacy check*, matches objects in the generated problem against the normalized object vocabulary. Missing symbols trigger an error signal, carrying suggestions for alternative symbols and the full normalized vocabulary back into the next refinement prompt.

Domain $\mathscr{D}$

```
(:predicates (on ?x ?y) (on-table ?x) (clear
?x) (holding ?x) (hand-empty))
(:action pickup :parameters (?b - block)
 :precondition (and (clear ?b) (on-table ?b)
(hand-empty))
 :effect (and (holding ?b) (not (on-table
?b))))
```

Problem $\mathscr{P}_0$: Init

```
(:init (on-table brown_block_1) (on
grey_block_1 brown_block_1) (on brown_block_2
grey_block_1) (clear red_block_1) (hand-
empty))
```

Goal

```
(:goal (holding brown_block_2))
```

PDDL refinement. Errors flagged by the symbolic planner and validator drive an iterative refinement loop. Each failure is parsed into a list of issues, each assigned a category (syntax, typing, predicate mismatch, action precondition/effect, hallucination, goal drift, other) and grouped, following the taxonomy in Fig. 2, as *structural* or *reasoning* issue. Within each group, issue types carry a priority (P0 before P1, etc.): for instance, syntactic faults are repaired before semantic ones. The LLM fixes one issue at a time, up to a certain maximum number of retries. Each category has an associated LLM *temperature schedule*: structural issues use a low-temperature ladder ($0.05 \rightarrow 0.15$) to preserve formatting, while reasoning issues ramp higher ($0.2 \rightarrow 0.7$) to encourage broader non-determinism. The refinement loop is nested: an outer loop refines the full domain/problem pair under a global temperature ladder that *escapes* to raise non-determinism in case of repeated generation issues, while an inner loop applies per-issue corrections to the problematic PDDL slice, gated by the symbolic validator after each fix, and *converges*

to a error-free solution by tightening determinism. Resolved issues (those that stop reappearing) are embedded and stored as issue-fix pairs in a vector database. If the embedding of a new issue has a (cosine) similarity against a stored entry exceeding the threshold of 0.8, the corresponding fix is injected into the repair prompt as a few-shot reference. If every iteration fails, a fallback domain/problem regeneration at high temperature attempts to leave the unrecoverable error condition. To prevent the LLM from achieving goal reachability in a generated PDDL problem by artificially creating an easier initial state (e.g. teleporting an object to clear a precondition failure), the problem refinement prompt locks the `:init` block: any proposed change is examined by a metacognitive gate and reverted to the locked initial state if rejected. The resulting domain and problem are initially checked for PDDL syntactic correctness via standard tools, then a sequential simulator replays the plan against the generated domain, checking goal reachability. Both stages operate entirely in the generated world and are aimed at producing a feedback signal for iterative refinement and correction.

Example: vector-DB-assisted issue repair

New issue (`P0_SYNTAX`, subgoal 0, refinement 0): the domain declares `:universal-effects` in `(:requirements ...)` but no action uses a `forall` effect, and the Unified Planning parser only supports `:strips` and `:typing`; parsing aborts at line 2.

Nearest stored match (similarity 0.83, `P0_SYNTAX`): a prior logistics run declared `:conditional-effects` while no action used when-effects; repaired by removing the unsupported requirement.

Injected reference fix: "Drop every entry from `:requirements` that does not appear in the body of any action; keep only `:strips :typing`."

If an issue does not reappear in the following refinement iteration, it is marked as solved, its description is embedded and used to store the issue-fix pair in a vector database. This knowledge base of resolved issues is used for Retrieval Augmented Generation, injecting few-shot example reference solutions of previously solved issues into the repair prompt. To use the proper level of non-determinism in the generation of different solutions for recurring issues, each issue category has an associated *temperature scaling strategy*. If the same issue reappears, the *temperature* hyperparameter used in generating the corrected PDDL problem is updated with the scale of that specific scaling strategy. Structural errors use a gentle, low-temperature progression to ensure strict formatting, while reasoning errors ramp up to higher temperatures to encourage creative problem-solving. Resolution rates per category are logged for post-hoc analysis. If all refinement iterations fail, the system performs one last regeneration of domain and problem at a higher temperature, as a last attempt at exiting from the local refinement basin, useful when the iterations have converged on an unrecoverable attractor. The refinement structure features

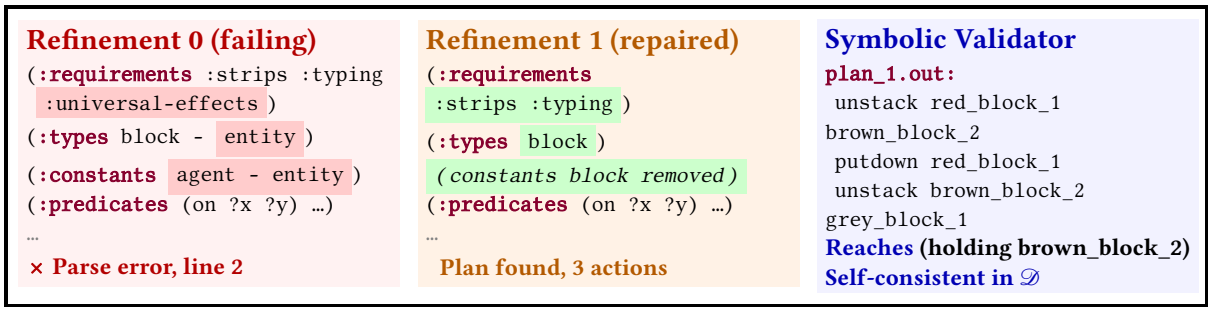


Figure 3: Issue-based refinement, on P_{ref} subgoal 0. *Refinement 0* is the initial LLM-generated PDDL: the `:universal-effects` requirement, the orphan `entity` supertype, and the unused `agent` constant (highlighted in red) each cause a parser error. *Refinement 1* reflects the three issue-level fixes. The right panel shows that the resulting plan reaches the subgoal.

an outer loop, refining the full domain and problem pair simultaneously, with a global temperature ladder, and a inner loop, which instead performs issue-based corrections on the specific problematic PDDL slice. Progress in the inner loop is gated by the symbolic validator after each fix. The two ladders therefore play different roles: the outer ladder escapes "attraction basins" where the LLM may get stuck

without progress in the refinement, by increasing non-determinism in the next-token prediction, while the inner ladder instead converges within a basin by tightening determinism. An example of refinement over the running example is shown in Figure 3, while Figure 5 shows the valid plan.

HLP Feedback. If refinement iterations are exhausted or the check against the generated LTLf formula fails, the low-level planner returns a comprehensive diagnosis of the reason of failure, an example of which is shown in Figure 4. The HLP is expected to utilize this feedback to correct the high-level plan, establishing a robust feedback loop between the planning layers.

WHAT WENT WRONG. The arm never actually holds brown_block_2 (though listed as a step), and after brown_block_1 leaves grey_block_1 the plan never places red_block_1 on grey_block_1 or holds it before terminating.

HOW TO FIX. (1) Ensure brown_block_2 is clear before step 1. (2) Make the holding-brown_block_2 step genuinely reachable. (3) Place red_block_1 on grey_block_1 *before* the plan ends, immediately after grey_block_1 reaches the table.

Figure 4: *NL feedback for HLP at the end of LLP first attempt.* A translator converts the raw LTLf trace checker output into a NL feedback, used in the HLP repair.

Prompt engineering. Each LLM call is templated with the agent task followed by its required input. The LLM is instructed to wrap all generative output in explicit XML or JSON tags (e.g. `<DOMAIN>`, `<PROBLEM>`, structured JSON for issue lists). At each low-level step, the problem generation prompt isolates the current subgoal as the one to solve while exposing the still unsolved future subgoals as read-only context, giving the model foresight over the remaining schedule.

4. Experimental Validation

We evaluate the low-level planner end-to-end on three planning domains drawn from the LexiCon benchmark suite [21]. The goal of this section is twofold: we first want to motivate the hierarchical structure of this planner against a simple LLM-as-planner baseline (while comparison with other LLM-planning architectures is left to future work). The head-to-head numbers backing this claim are reported in Table 2. We then want to show how performance scales by deterministically increasing the complexity of the set of temporal constraints. We therefore address two research questions:
RQ1*: Given that LLMs are known to struggle with highly constrained planning [21], what is the impact of our hierarchical architecture, decomposing the LTLf constraints into subgoals and solving each with an improved LLM-planning loop?*
RQ2*: Does our system improve executability of computed plans and satisfaction of the LTLf formula in the ground-truth domain compared to the LLM-as-planner baseline?*

Dataset. We test the system on the *blocksworld*, *logistics*, and *sokoban* International Planning Competition domains. Each natural language problem contains the domain description (action signatures with preconditions and effects), an object list, an initial state description, a goal, and an LTLf constraint over the trace. We experiment with 0, 1, and 2 LTLf temporal constraints, each with ten problems per (domain, batch). The natural language descriptions are generated with the LexiCon system [21].

Experimental setup. We use Claude Sonnet 4.6. as a LLM backend. The symbolic backend is powered the FastDownward instance from the Unified Planning library. Plan validation relies on VAL, while semantic validation is performed using a sequential PDDL simulation component. We allow at most three outer high-level planner attempts and four low-level refinement iterations per subgoal, plus one final global fallback regeneration attempt at a high temperature. The temperature ladders for structural and reasoning issues (Figure 2) are 0.05→0.15 (with a step of 0.02) and 0.2→0.7 (with a step of 0.1)

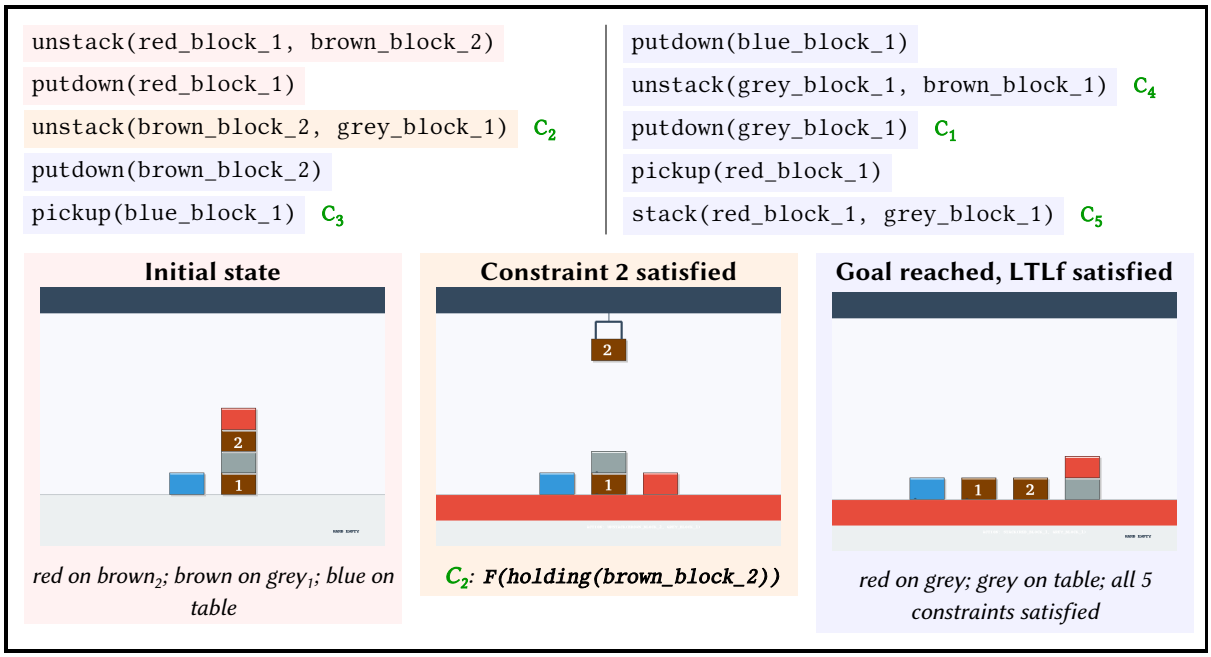


Figure 5: *Final grounded plan. Top*: plan π; *Bottom*: initial, intermediate and final state.

Table 1
Comparison of LTLf Consistency (Tier 2) and Ground-Truth LTLf Consistency (Tier 4) between the LLM-as-planner baseline and CoSP-TL. Values represent averages across difficulty batches (left) and domains (right).

	Baseline		CoSP-TL	
N° Constr.	LTLf%	GT%	LTLf%	GT%
0	100	80	100	100
1	70	40	93	83
2	57	23	93	83

	Baseline		CoSP-TL	
Domain	LTLf%	GT%	LTLf%	GT%
Blocksworld	87	80	93	93
Logistics	50	20	100	93
Sokoban	90	43	93	80

respectively; the two ladders are separated because syntactic failures amplify at higher temperature while reasoning-class failures benefit from broader sampling. The refinement vector database is enabled by default and reset for every new problem.

Evaluation metrics. Success rate is evaluated over 4 success condition tiers, each more restricting than the previous one. The first two tiers are computed using the *generated domain* that CoSP-TL constructs from the natural language input and evaluate syntactical and semantic self-consistency of the generated PDDL. *Tier 1* ("*Goal Success*") is satisfied when the plan reaches the goal in the *generated* PDDL domain; *Tier 2* ("*LTLf Consistency*") additionally requires that the trace satisfies the LTLf constraint generated in the High-Level planner, via the trace checker (see Alg. 1). The remaining two tiers evaluate the same plan against the *ground-truth (GT)* domain bundled with the benchmark (unavailable to the planner at runtime). *Tier 3* ("*GT Runnability*") requires that the same plan executes successfully against the *ground-truth*, evaluating the "*semantic validation gap*: a plan can be self-consistent in the generated world (Tier 1+2) and still mismatch the ground truth semantics. *Tier 4* ("*GT LTLf Consistency*") requires that the LTLf constraint holds along the plan replayed in a simulator initialized from the ground-truth domain rather than the induced one, where the problem is obtained through a homomorphism of the generated predicates and syntax to the ground-truth vocabulary.

Table 2
Detailed results for CoSP-TL for problems with **0**, **1** and **2** LTLf constraints respectively, over Blocksworld (**BW**), Logistics (**LO**), Sokoban (**SO**). **Goal Success** represents Tier 1, **LTLf Consistency** is Tier 2, **GT Runnability** is Tier 3 and **GT LTLf Consistency** is Tier 4. **HLP Attmpts.** represents the mean number of feedbacks to the high-level planner. **LLP Ref.** is the mean low-level planner refinement iterations. **LLM Calls** and **Total Tokens (k)** are per-problem means. **Wall Time (s)** is mean (total) planning time in seconds.

		Success Metrics (%)				Efficiency Metrics (Mean)				
Dom.	N° Constr.	Goal Success	LTLf Consistency	GT Runnability	GT LTLf Consistency	HLP Attmpts.	LLP Refnmts.	LLM Calls	Total Tokens (k)	Wall Time (s)
BW	0	100	100	100	100	1.00	2.7	15.1	19.1	92
	1	100	100	100	100	1.30	9.5	65.9	80.4	436
	2	100	80	100	80	2.20	20.0	144.8	174.7	994
LO	0	100	100	100	100	1.00	1.0	9.0	11.9	60
	1	90	100	90	90	1.00	5.1	40.1	60.1	269
	2	90	100	90	90	1.00	7.7	56.1	85.8	397
SO	0	100	100	100	100	1.00	1.0	9.0	24.4	105
	1	60	80	60	60	1.10	8.3	54.1	161.8	576
	2	80	100	80	80	1.00	6.4	47.6	133.4	499

4.1. Results

Table 1 compares CoSP-TL against an LLM-as-planner baseline, where an LLM is asked to directly produce a runnable plan from the NL input [21]. Results are aggregated by number of constraints (left) and by domain (right). As the number of constraints increases, baseline LTLf consistency drops monotonically while CoSP-TL keeps a stable performance, so the gap widens with constraint count. **This answers RQ1:** the hierarchical approach of decomposing the LTLf specification into subgoals and iteratively refining each mitigates the performance drops over highly constrained problems, where direct LLM-as-planner is known to fail [21]. Notably, in *Logistics* (baseline GT-LTLf at 20%), LLMs tended to omit the "airport" typing, so airplanes could "fly" between arbitrary waypoints. CoSP-TL tackles this issue by injecting domain predicates in the problem generation prompt, surfacing such precondition violations during refinement. **This answers RQ2:** against the ground-truth domain, the baseline collapses as the problem becomes more constrained, while CoSP-TL, especially in domains where tacit semantics really help (e.g. *Logistics*). This improvement effectively transfers from self-consistency to ground truth LTLf satisfaction, confirming that the hierarchical design improves both plan consistency and runnability. Table 2 reports CoSP-TL across the four success tiers. Unconstrained cells saturate at 100% on all tiers, isolating the LTLf constraints as the main source of difficulty. Goal success % (Tier 1) holds at 100% in only five of nine cells: saturates on *blocksworld* but degrades on *logistics* (90%) and *sokoban* (60-80%). The relation between Tier 2 and Tier 3 is domain-specific: on *logistics* and *sokoban* Tier 2 exceeds Tier 3 (a fraction of plans self-consistent in the induced domain fail to execute in the GT domain), whereas *blocksworld* at 2 constraints reverses this relation: Tier 2 80% vs. Tier 3 100%, every plan is goal-reaching and GT-runnable, but 20% genuinely violate the constraint. Tier 4 never exceeds Tier 2 and equals Tier 3 except on *blocksworld*-2c (100% vs. 80%): once a plan is GT-runnable it almost always also satisfies the constraint there, consistently with the analysis in Section 4.1.

Fairness. The low-level planner receives its world description in natural language. The generated simulator state is translated in NL before injection in the problem generation prompt, ensuring that compatibility with robotic applications based on VLM perception backbone. The only feedback from the ground-truth simulator to the HLP contains action rejection signals from the diagnosis step, similar to feedback coming from any low-level controller on a real robot.

4.2. Discussion

On *blocksworld* the generated schema is always isomorphic to the ground truth domain up to a simple renaming (e.g. `on-table`/`ontable`), so the gap closes by symbol remapping alone; the only issue is at 2 constraints, where both LTLf consistency tiers achieve a 80% against a 100% of ground truth runnability: the failure is due to LTLf constraint violation, not runnability in itself. *Logistics* and *sokoban* show an opposite trend: driven by tacit domain semantics (e.g. the fact that *airplanes only land at airports*): LTLf consistency over the generated domain exceeds GT-consistency: the induced model lets some plans pass the generated domain LTLf check but fails on GT, while every plan successful on GT also satisfies LTLf in the generated world. The metacognitive initial state gate prevents the LLM from drifting the initial state towards the goal and away from the simulator state. For instance, during refinement on a blocksworld problem, the LLM proposed a selective deletion of facts from `(:init)` to escape a precondition failure. The gate classified the diff against the hallucination pattern, reverted it, and the subsequent refinement iteration converged on a legitimate fix. Similarly, the LLM temperature ladder differentiation by issue category (structural $0.05 \rightarrow 0.15$, reasoning $0.2 \rightarrow 0.7$) meets a critical generation issue: structural refinements degrade under higher temperature, while reasoning refinements (type/predicate/action/hallucination class) benefit from broader sampling. The two ladders turn these regimes into explicit annealing schedules. CoSP-TL performance can be evaluated also in terms of the gap between success rates over the *induced* schema and over the *ground-truth* schema. Deviations that widen this gap (lower GT performance compared to the self-consistency in the generated domain) can be split in two classes: *notational* (e.g. predicate naming mismatch between generated and GT domain, like `on-table` vs. `ontable`), preserve semantics and are recoverable by a simple deterministic homomorphism, or *semantic* (e.g. relaxing the `airport` typing on `flyairplane`, letting airplanes reach non-airport waypoints), constituting real planning errors a homomorphism cannot fix. To rigorously verify this gap, the homomorphism is applied only post-hoc and reconciles 100% of vocabulary mismatches, canceling notational deviations. The residual ground truth runnability gap measures the semantic drift, which increases with the number of constraints.

Limitations. The current evaluation uses a single LLM (Claude Sonnet 4.6) and only the LexiCon benchmark. LexiCon's NL descriptions are generated by a compositional translator from PDDL atoms and therefore carry an inherent hinting level. Moreover, although planning is possible from natural language descriptions, no ablation study was performed over various styles and levels of verbosity of the natural language input. Finally, a major limitation is the high token cost, due to the frequent intervention of the LLM throughout the planning process.

5. Conclusion

This paper presented CoSP-TL, a hierarchical planner for problems expressed in natural language featuring temporal constraints. Our architecture combines common-sense knowledge embedded in LLMs with symbolic planning and Linear Temporal Logic specifications. An auto-regressive High-Level Planner builds a set of ungrounded instructions, which are grounded and translated into executable PDDL actions by a Low-Level Planner. Experimental evaluation on the LexiCon benchmark across three IPC domains shows that CoSP-TL substantially improves over standard LLM-based planning approaches, mitigating performance loss under temporal constraints and maintaining robust constraint satisfaction up to 2 constraints. As future work, we aim to expand the set of domains used to evaluate our architecture and to optimize the pipeline to increase the cost-efficiency of the planning process.

Acknowledgments

This work has been carried out while Emanuele Musumeci and Alessio Saladino were enrolled in the Italian National PhD in Artificial Intelligence, at Sapienza University of Rome. CUP B53C24003450004. Alessio Saladino is funded by Procter&Gamble.

6. Declaration on Generative AI

During the preparation of this work, the author(s) used Claude Sonnet 4.6 for layout optimization. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[2] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI '13, AAAI Press, 2013, p. 854–860.

[3] R. E. Wray, J. R. Kirk, J. E. Laird, Eliciting problem specifications via large language models, 2024. URL: https://arxiv.org/abs/2405.12147. arXiv:2405.12147.

[4] B. Li, A. Tamkin, N. Goodman, J. Andreas, Eliciting human preferences with language models, in: Y. Yue, A. Garg, N. Peng, F. Sha, R. Yu (Eds.), International Conference on Learning Representations, volume 2025, 2025, pp. 80984–81013. URL: https://proceedings.iclr.cc/paper_files/paper/2025/file/c9867d5a22653ce98b02595061e40f12-Paper-Conference.pdf.

[5] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, et al., Emergent abilities of large language models, Transactions on Machine Learning Research (2022).

[6] Z. Xu, M. Shen, J. Dineen, Z. Li, X. Ye, S. Lu, A. Rrv, C. Baral, B. Zhou, Tow: Thoughts of words improve reasoning in large language models, in: Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), 2025, pp. 3057–3075.

[7] G. Kamath, S. Schuster, S. Vajjala, S. Reddy, Scope ambiguities in large language models, Transactions of the Association for Computational Linguistics 12 (2024) 738–754. URL: http://dx.doi.org/10.1162/tacl_a_00670. doi:10.1162/tacl_a_00670.

[8] A. Keluskar, A. Bhattacharjee, H. Liu, Do llms understand ambiguity in text? a case study in open-world question answering, in: 2024 IEEE International Conference on Big Data (BigData), 2024, pp. 7485–7490. doi:10.1109/BigData62323.2024.10825265.

[9] S. Huang, N. Lipovetzky, T. Cohn, Planning in the dark: Llm-symbolic planning pipeline without experts, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 39, 2025, pp. 26542–26550. URL: https://ojs.aaai.org/index.php/AAAI/article/view/34855.

[10] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, Q. V. Le, Finetuned language models are zero-shot learners, in: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022, OpenReview.net, 2022. URL: https://openreview.net/forum?id=gEZrGCozdqR.

[11] E. Gestrin, V. Singhal, G. Hoffman, S. Chamberland, NL2Plan: Robust LLM-driven planning from minimal text, in: AAAI, 2025.

[12] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, P. Stone, LLM+P: Empowering large language models with optimal planning proficiency, in: IROS, 2023.

[13] Z. Zhu, R. Jiao, N. Peng, ISR-LLM: Iterative self-refined large language model for long-horizon sequential task planning, in: IEEE International Conference on Robotics and Automation (ICRA), 2024. VAL-guided iterative refinement for problem generation.

[14] S. Huang, N. Lipovetzky, T. Cohn, Planning in the dark: LLM-symbolic planning pipeline without experts, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2025. ArXiv:2409.15915; Schema library + semantic validation and ranking without expert intervention.

[15] S. Agarwal, A. Sreepathy, Tic: Translate-infer-compile for accurate "text to plan" using llms and

logical representations, in: International Conference on Neural-Symbolic Learning and Reasoning, Springer, 2024, pp. 222–244.

[16] E. La Malfa, et al., An end-to-end planning framework with agentic LLMs and PDDL, arXiv preprint arXiv:2512.09629 (2025). Multi-agent architecture with VAL + uVAL iterative feedback.

[17] J. Wang, R. Peng, T. Chen, C. Zeng, W. Wang, S. Huang, N. Lipovetzky, PDDL-GenAI: A plugin for LLM-based support in planning domain modeling, in: ICAPS Demo Track, 2025. NL-to-PDDL plugin for editor.planning.domains; supports ChatGPT, Deepseek, ZhipuAI.

[18] M. Tantakoun, C. Muise, X. Zhu, LLMs as planning formalizers: A survey for leveraging large language models to construct automated planning models, in: Findings of the Association for Computational Linguistics: ACL, 2025, pp. 25167–25188. Survey on NL-to-PDDL methods; discusses data contamination in IPC domains.

[19] M. Vallati, R. Barták, L. Chrpa, T. L. McCluskey, R. P. A. Petrick, Knowledge engineering for planning and scheduling in the LLM era, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), volume 35, 2025, pp. 391–395. LLMs can assist in knowledge acquisition but human expertise and symbolic validators remain critical.

[20] S. Kambhampati, K. Valmeekam, L. Guan, M. Verma, K. Stechly, S. Bhambri, L. Saldyt, A. Murthy, Position: Llms can't plan, but can help planning in llm-modulo frameworks, in: Proceedings of the 41st International Conference on Machine Learning, volume 235 of *Proceedings of Machine Learning Research*, PMLR, 2024, pp. 22895–22907. URL: https://proceedings.mlr.press/v235/kambhampati24a.html.

[21] P. Mantenoglou, R. Hazra, P. Zuidberg Dos Martires, L. De Raedt, Lexicon: a benchmark for planning under temporal constraints in natural language, Advances in Neural Information Processing Systems 38 (2026).

[22] P. Vaezipoor, A. C. Li, R. A. T. Icarte, S. A. Mcilraith, Ltl2action: Generalizing ltl instructions for multi-task rl, in: M. Meila, T. Zhang (Eds.), Proceedings of the 38th International Conference on Machine Learning, volume 139 of *Proceedings of Machine Learning Research*, PMLR, 2021, pp. 10497–10508. URL: https://proceedings.mlr.press/v139/vaezipoor21a.html.

[23] C. Voloshin, H. Le, S. Chaudhuri, Y. Yue, Policy optimization with linear temporal logic constraints, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Advances in Neural Information Processing Systems, volume 35, Curran Associates, Inc., 2022, pp. 17690–17702. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/70b8505ac79e3e131756f793cd80eb8d-Paper-Conference.pdf.

[24] M. Jackermeier, A. Abate, Deepltl: Learning to efficiently satisfy complex ltl specifications for multi-task rl, in: Y. Yue, A. Garg, N. Peng, F. Sha, R. Yu (Eds.), International Conference on Learning Representations, volume 2025, 2025, pp. 14000–14028. URL: https://proceedings.iclr.cc/paper_files/paper/2025/file/24c523085d10743633f9964e0623dbe0-Paper-Conference.pdf.

[25] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, Y. Su, Llm-planner: Few-shot grounded planning for embodied agents with large language models, in: Proceedings of the IEEE/CVF international conference on computer vision, 2023, pp. 2998–3009.

[26] J. Yang, X. Chen, S. Qian, N. Madaan, M. Iyengar, D. F. Fouhey, J. Chai, Llm-grounder: Open-vocabulary 3d visual grounding with large language model as an agent, in: 2024 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2024, pp. 7694–7701.