BDI-LLM: Neuro-Symbolic Planning with Multi-Layer Formal Verification

Author One[1], Author Two[1], Author Three[2]
[1]**Affiliation One**
[2]**Affiliation Two**
`{author1, author2}@example.edu, author3@example.edu`

Abstract

Large Language Models (LLMs) have demonstrated remarkable natural language understanding, yet their ability to generate logically correct multi-step plans remains limited—prior work on the PlanBench benchmark reports accuracy below 35% for state-of-the-art models. We present BDI-LLM, a neuro-symbolic framework that integrates LLM-based plan generation with multi-layer formal verification. Our approach casts the LLM as a *generative compiler* within a Belief-Desire-Intention (BDI) agent architecture, producing structured plans that are rigorously validated through (1) graph-theoretic structural checks, (2) PDDL symbolic verification via the VAL tool, and (3) domain-specific physics constraints. A closed-loop error-driven repair mechanism re-prompts the LLM with concrete VAL error messages and cumulative repair history, enabling iterative self-correction. Implemented using the DSPy framework for structured LLM programming, our system incorporates domain-specific prompt engineering with explicit state tracking, chain-of-symbol representations, and verified few-shot demonstrations. Evaluated on 1,270 PlanBench instances across Blocksworld, Logistics, and Depots, BDI-LLM achieves 99.6% overall accuracy (100% on Blocksworld, 99.6% on Logistics, 99.4% on Depots), representing a 65–95 percentage-point improvement over unverified LLM baselines.

1 Introduction

The ability to generate correct multi-step plans is a fundamental requirement for autonomous agents operating in complex environments. Classical AI planning, grounded in formal representations such as the Planning Domain Definition Language (PDDL) [13], provides sound and complete algorithms for plan synthesis. However, these methods require hand-crafted domain models and struggle with the flexibility needed for open-ended, natural language task specifications. Large Language Models (LLMs), by contrast, excel at interpreting natural language instructions and possess broad world knowledge acquired through pretraining, making them attractive candidates for plan generation [8, 1].

Despite this promise, a growing body of evidence demonstrates that LLMs are unreliable planners when used in isolation. Valmeekam et al. [20] introduced PlanBench, a rigorous benchmark for evaluating LLM planning capabilities, and showed that even GPT-4 achieves only $\sim$35% accuracy on Blocksworld—a domain with just four operators. Subsequent studies confirm that LLMs struggle with precondition tracking, generate physically impossible action sequences, and fail to maintain state consistency across long action chains [19]. These failures stem from a fundamental mismatch: LLMs perform approximate, pattern-based reasoning, while correct planning demands exact logical inference over state transitions.

Two broad strategies have emerged to address this gap. The first augments the LLM's internal reasoning through advanced prompting techniques such as Chain-of-Thought [21], Tree of Thoughts [23], and ReAct [24]. While these improve performance on some tasks, they provide no formal guarantees of plan correctness. The second strategy, exemplified by the LLM-Modulo framework [10], pairs LLMs with external verifiers that check generated plans against formal specifications. This neuro-symbolic approach is more principled, but existing instantiations typically employ a single verification layer and lack systematic mechanisms for error-driven plan repair.

In this paper, we present **BDI-LLM**, a neuro-symbolic planning framework that bridges LLM-based generation and formal verification through a multi-layer architecture grounded in the Belief-Desire-Intention (BDI) agent model [15]. Our framework treats the LLM as a *generative compiler*: given beliefs (current world state) and desires (goal specification), it produces structured intentions (plans) represented as directed acyclic graphs of actions. These plans are then subjected to three layers of verification—structural, symbolic, and physics-based—with a closed-loop repair mechanism that feeds concrete error diagnostics back to the LLM for iterative correction (Figure 1).

Our approach makes the following contributions:

1. **Multi-layer verification architecture.** We propose a three-layer verification pipeline combining graph-theoretic structural validation, PDDL symbolic verification via the VAL tool [7], and domain-specific physics constraints. Each layer catches a distinct class of errors, from malformed graph structures to violated action preconditions.
2. **Error-driven repair loop.** We introduce a closed-loop repair mechanism that re-prompts the LLM with specific

VAL error messages and cumulative repair history. This enables targeted self-correction: the LLM receives not just a pass/fail signal, but precise diagnostic information about *which* preconditions were violated and *at which* step, achieving a 96.4% repair success rate.

3. **Domain-specific structured prompting.** We develop domain-tailored DSPy signatures [11] that incorporate explicit state tracking tables, chain-of-symbol representations, check-before-act protocols, and verified few-shot demonstrations derived from actual VAL validation traces. These prompts encode domain constraints (e.g., airport identification in Logistics, hoist availability in Depots) directly into the generation process.
4. **State-of-the-art PlanBench results.** Evaluated on 1,270 instances across three PlanBench domains, BDI-LLM achieves 99.6% overall accuracy—100% on Blocksworld (200 instances), 99.6% on Logistics (570 instances), and 99.4% on Depots (500 instances). This represents a 65–95 percentage-point improvement over prior unverified LLM baselines.

The remainder of this paper is organized as follows. Section 2 surveys related work on LLM planning, formal verification, and BDI architectures. Section 3 describes our framework architecture and key design decisions. Section 4 details the experimental setup. Section 5 presents results and analysis. Section 6 discusses implications and limitations, and Section 7 concludes.

2 Related Work

Our work draws on and contributes to three intersecting research areas: LLM-based planning, formal verification for AI-generated plans, and the BDI agent architecture. We also discuss the DSPy framework that underpins our structured prompting approach.

2.1 LLMs for Planning

The application of LLMs to planning tasks has attracted significant attention following demonstrations that pre-trained language models encode substantial world knowledge relevant to action sequencing.

Direct LLM planning. Early work explored using LLMs as direct planners. Huang et al. [8] showed that LLMs can decompose high-level instructions into actionable steps, while SayCan [1] grounded language models in a robot's affordance functions to generate feasible action sequences. Inner Monologue [9] extended this by incorporating environmental feedback into the LLM's planning loop, enabling re-planning after execution failures. However, these approaches lack formal correctness guarantees and rely on execution-time feedback rather than pre-execution verification.

Enhanced reasoning strategies. Several prompting techniques have been proposed to improve LLM reasoning for planning. Chain-of-Thought (CoT) prompting [21] elicits step-by-step reasoning, while Tree of Thoughts (ToT) [23] enables exploration of multiple reasoning paths with self-evaluation and backtracking. ReAct [24] interleaves reasoning traces with actions, allowing LLMs to query external tools during plan construction. Reflexion [16] adds a self-reflection mechanism where the LLM critiques its own outputs. While these methods improve planning performance, they operate entirely within the LLM's approximate reasoning and cannot guarantee logical correctness. Our work incorporates elements of chain-of-thought reasoning (via explicit state tracking tables) but crucially augments them with external formal verification.

Hybrid neuro-symbolic approaches. Recognizing the limitations of pure LLM planning, hybrid approaches combine LLMs with classical planners or verifiers. LLM+P [12] uses LLMs to translate natural language into PDDL problem specifications, then invokes a classical planner for solution. This achieves correctness but requires the LLM to produce syntactically valid PDDL, which remains challenging. The LLM-Modulo framework [10] proposes a general architecture where LLMs generate candidate plans that are checked by external "critics" (verifiers), with feedback loops for iterative refinement. Our work instantiates and extends this paradigm with a concrete multi-layer verification pipeline, domain-specific prompt engineering, and a structured repair mechanism that provides the LLM with precise diagnostic information rather than binary pass/fail signals.

Empirical evaluations. PlanBench [20], introduced at NeurIPS 2023, provides a standardized benchmark for evaluating LLM planning across classical domains. Valmeekam et al. demonstrated that GPT-4 achieves only $\sim$35% on Blocksworld and that self-critiquing does not reliably improve plan quality [18]. Follow-up work evaluated OpenAI's o1 reasoning model on PlanBench, finding improved but still imperfect performance [19]. These findings motivate our approach: rather than relying on the LLM to self-verify, we employ external formal verification tools that provide sound correctness guarantees.

2.2 Formal Verification for AI-Generated Plans

Formal verification has a long history in AI planning, primarily through the use of PDDL and associated tools.

PDDL and VAL. The Planning Domain Definition Language (PDDL) [13] provides a standardized formal representation for planning domains and problems. The VAL tool [7] is the de facto standard for plan validation in the planning community, checking that action preconditions are satisfied at each step and that the goal state is achieved. Our framework leverages VAL as the core symbolic verification engine, converting LLM-generated BDI plans into PDDL action sequences for rigorous validation.

Verification in LLM pipelines. Recent work has begun integrating formal verification into LLM-based systems. Code generation pipelines use static analysis and test suites to verify LLM-generated code [4]. In planning, Guan et al. [5] use LLMs to generate PDDL domain models that are then verified against reference specifications. Silver et al. [17] explore using LLMs to propose planning heuristics that are formally validated. Our contribution differs in applying multi-layer verification directly to LLM-generated plans (not domain models or heuristics), with an error-driven feedback loop that enables the LLM to iteratively correct its outputs based on specific verification failures.

2.3 BDI Agent Architecture

The Belief-Desire-Intention (BDI) model, rooted in Bratman's theory of practical reasoning [3], provides a principled framework for agent decision-making. Rao and Georgeff [15] formalized the BDI architecture using a branching-time temporal logic (BDICTL), establishing formal semantics for beliefs, desires, and intentions as modal operators over possible worlds.

Classical BDI implementations such as AgentSpeak [14], Jason [2], and JACK [22] use hand-crafted plan libraries and logical inference for plan selection. These systems provide formal guarantees but lack the flexibility to handle natural language inputs or generate novel plans for unseen situations.

Our framework bridges this gap by using an LLM as the plan generation engine within a BDI architecture. The LLM receives beliefs (world state) and desires (goals) as structured inputs and produces intentions (plans) as structured outputs—preserving the BDI conceptual framework while leveraging the LLM's generative capabilities. The formal verification layers then provide the correctness guarantees traditionally ensured by logical inference in classical BDI systems.

2.4 Structured LLM Programming with DSPy

DSPy [11], introduced at ICLR 2024, provides a declarative programming framework for LLM applications. Rather than manually crafting prompt strings, developers define typed *signatures* specifying input and output fields, and DSPy compiles these into optimized prompts. This approach offers several advantages for our setting: (1) structured output parsing ensures the LLM produces well-formed BDI plan objects rather than free-text; (2) signature docstrings serve as a principled location for domain-specific constraints and worked examples; and (3) the framework supports systematic prompt optimization through its teleprompter modules.

Our use of DSPy extends beyond basic structured generation. We design domain-specific signatures (one per planning domain) that encode action type constraints, precondition specifications, state tracking protocols, and verified few-shot demonstrations derived from actual VAL validation traces. This represents a novel application of structured LLM programming to formal planning tasks.

3 Framework Architecture

Figure 1 illustrates the end-to-end BDI-LLM pipeline. Given a PDDL problem instance, the framework (i) converts the formal specification into a structured natural language representation, (ii) invokes an LLM to generate a BDI plan as a directed acyclic graph, and (iii) subjects the plan to a three-layer verification pipeline with an error-driven repair loop. We describe each component below.

3.1 PDDL-to-Natural-Language Conversion

Classical planning benchmarks encode problems in PDDL, a formal language that is syntactically precise but opaque to LLMs trained predominantly on natural language corpora. Rather than prompting the LLM with raw PDDL—an approach shown to yield poor results [19]—we convert each problem instance into a structured natural language representation consisting of two components aligned with the BDI model [15]:

- **Beliefs**: A comprehensive description of the current world state, including typed object inventories, spatial relationships, vehicle/agent positions, and domain constraints. Critical rules (e.g., which locations are airports in Logistics) are highlighted with explicit warning markers.
- **Desires**: The goal specification, augmented with analysis of required transport modes (e.g., same-city vs. cross-city delivery) and, for Blocksworld, a pre-computed bottom-up tower construction ordering.

The conversion is domain-specific. For **Blocksworld**, we parse the initial state to identify existing stacks, compute which `on(X,Y)` pairs conflict with the goal, and generate an explicit teardown sequence (ordered top-down by stack height) followed by a bottom-up construction plan. This effectively reduces the planning problem to plan *execution*, which explains the 100% first-attempt accuracy. For **Logistics**, we extract the city–location–airport topology, classify each delivery goal as intra-city (truck-only) or inter-city (requires airplane), and enumerate the airport set with repeated warnings that `fly-airplane` may only use airport locations. For **Depots**, we parse typed objects (hoists, crates, pallets, trucks, depots, distributors), describe the hoist state machine (available $\leftrightarrow$ lifting), and provide a worked example with explicit state tracking after each action.

Each conversion also includes the complete action schema for the domain—action names, parameter formats, preconditions, and effects—presented in a structured, human-readable format rather than PDDL syntax.

3.2 BDI-Structured Plan Generation with DSPy

We implement plan generation using DSPy [11], a framework for programming LLM pipelines with typed signatures and automatic prompt optimization. Each planning domain is associated with a dedicated `Signature` class that encodes domain-specific instructions in its docstring:

- `GeneratePlan` (Blocksworld): Specifies the four valid action types (`pick-up`, `put-down`, `stack`, `unstack`) with parameter schemas and precondition/effect rules. Includes mandatory state tracking via a block position table updated after every action, chain-of-symbol (CoS) representations for state clarity, and a four-step LogiCoT verification protocol (identify goal $\rightarrow$ list preconditions $\rightarrow$ check state $\rightarrow$ decide).
- `GeneratePlanLogistics`: Defines six transport actions with exact parameter formats. Emphasizes the two most common failure causes: (1) using non-airport locations with `fly-airplane`, and (2) failing to track

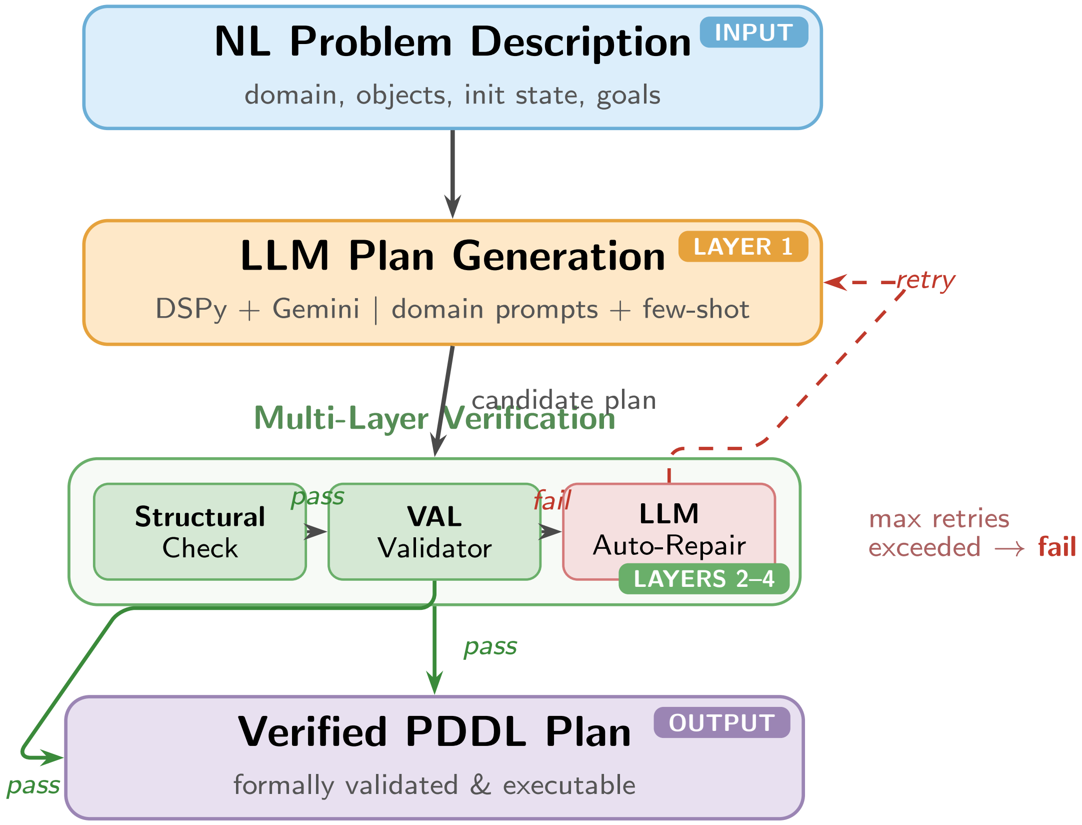

Figure 1: BDI-LLM system architecture. Natural language problem descriptions are processed through four stages: (1) LLM plan generation with domain-specific DSPy prompts, (2) structural verification ensuring valid DAG structure, (3) symbolic verification via the VAL validator, and (4) an error-driven LLM repair loop with cumulative error history. Plans passing all layers are output as formally verified PDDL plans.

airplane/truck positions after movement actions. Includes a logistics state table tracking object type, location, contents, and airport status.

- `GeneratePlanDepots`: Specifies five manipulation/transport actions with a hoist state machine diagram. Highlights the critical truck position tracking requirement and provides a five-step worked example with state updates marked after each action.

All signatures use `dspy.ChainOfThought`, which instructs the LLM to produce step-by-step reasoning before outputting the structured plan. The output is a `BDIPlan` Pydantic model containing:

- A `goal_description` string summarizing the objective.
- A list of `ActionNode` objects, each with a unique ID, `action_type`, typed `params` dictionary, and natural language `description`.
- A list of `DependencyEdge` objects encoding temporal ordering as a DAG.

The LLM is configured with temperature 0.2 for near-deterministic output. At runtime, the `BDIPlanner` module validates that all generated action types belong to the domain's valid set and that all required parameters are present, raising a `ValueError` that triggers DSPy's automatic retry mechanism if constraints are violated.

3.3 Few-Shot Demonstrations

For the Logistics domain—where the action space and state tracking requirements are most complex—we provide two VAL-verified few-shot demonstrations as `dspy.Example` objects:

1. A **single-city** example (3 delivery goals, 9 actions) demonstrating correct sequential truck routing with position tracking after each `drive-truck` action.
2. A **cross-city** example (6 delivery goals across 2 cities, 18 actions) demonstrating airplane position tracking, airport identification, and multi-vehicle coordination.

Both demonstrations are derived from actual PlanBench instances and verified correct by the VAL validator, ensuring that the LLM learns from gold-standard plans rather than potentially flawed human-written examples.

3.4 Three-Layer Verification Pipeline

Generated plans undergo three layers of verification, each targeting a distinct class of errors.

Layer 1: Structural Verification The BDI plan's DAG structure is validated using graph-theoretic checks:

- **Acyclicity**: The dependency graph must be a DAG (no circular dependencies).
- **Weak connectivity**: All action nodes must be reachable from each other, preventing disconnected "island" subgraphs.

When structural violations are detected and auto-repair is enabled, the system attempts to unify disconnected components by inserting virtual `__START__` and `__END__` nodes. This repair succeeds when the plan's actions are individually correct but the LLM failed to specify explicit ordering edges between independent action subsequences.

Layer 2: Symbolic Verification with VAL Plans passing structural verification are converted from the BDI representation to PDDL action sequences. The conversion normalizes action type names (e.g., mapping LLM-generated variants like "pickup" to the canonical "pick-up"), extracts parameters from the typed `params` dictionary using domain-specific key mappings with positional fallbacks, and produces a topologically sorted sequence of PDDL actions.

The action sequence is then validated by the VAL plan validator [7], which performs full forward simulation of the plan against the PDDL domain and problem specifications. VAL checks:

- Action preconditions are satisfied at each step.
- Action effects are correctly applied to the state.
- The goal state is achieved after the final action.
- All action parameters have valid types.

VAL is invoked with the `-v` (verbose) flag, which provides detailed error diagnostics including the specific failing action, unsatisfied preconditions, and concrete *Plan Repair Advice* specifying which predicates need to be established.

Layer 3: Domain-Specific Physics Validation For Blocksworld, a dedicated physics simulator traces state transitions through the action sequence, maintaining a state table of block positions (`on_table`, `on`), clear status, and hand state. This layer catches physical constraint violations such as picking up non-clear blocks, stacking on occupied surfaces, or holding multiple blocks simultaneously. For Logistics and Depots, this layer defers to the comprehensive checking performed by VAL in Layer 2.

3.5 Error-Driven Repair Loop

When VAL reports errors, the framework enters a closed-loop repair cycle (up to 3 iterations). A dedicated `RepairPlan` DSPy signature receives:

- The original *beliefs* and *desires* (full problem context).
- The *previous plan* as a PDDL action sequence.
- The *VAL error messages*, filtered to remove verbose output and retain only actionable diagnostics (failing action, unsatisfied preconditions, repair advice).
- A *cumulative repair history* containing all previous failed attempts with their plans and errors, enabling the LLM to avoid repeating the same mistakes.

The repair prompt instructs the LLM to interpret VAL error messages (e.g., "Set `(at t1 loc1)` to true" means the truck must be driven to `loc1` before loading), generate a *complete* corrected plan (not just a patch), and try a fundamentally different approach if the same error recurs across attempts. The repaired plan undergoes the same three-layer verification, and the loop continues until the plan passes or the attempt budget is exhausted.

This repair mechanism is critical to the framework's overall accuracy: 111 of 1,270 instances (8.7%) required at least one repair attempt, with a 96.4% repair success rate (107/111). The majority of repairs succeed on the first attempt (97/100 single-attempt repairs), with diminishing returns for subsequent attempts.

3.6 BDI Plan to PDDL Conversion

The conversion from BDI plan to PDDL action sequence is a critical bridge between the LLM's structured output and the formal verification layer. The process involves:

1. **Topological sorting**: Action nodes are ordered using topological sort on the dependency DAG, ensuring that prerequisite actions precede dependent ones.
2. **Action type normalization**: LLM-generated action types are canonicalized (e.g., "pickup" $\rightarrow$ "pick-up", "LoadTruck" $\rightarrow$ "LOAD-TRUCK") to match PDDL domain definitions.
3. **Parameter extraction**: For each action, parameters are extracted from the `params` dictionary using a prioritized list of key aliases (e.g., for a truck parameter: "truck", "vehicle", "t", "truck_name"). If named extraction fails, positional fallback uses the parameter order defined in the PDDL domain schema.
4. **PDDL formatting**: Parameters are assembled into standard PDDL action strings, e.g., `(stack a b)` or `(DRIVE-TRUCK t0 l0-1 l0-2 c0)`.

This robust conversion handles the inherent variability in LLM-generated parameter naming while maintaining strict adherence to the PDDL action schemas expected by VAL.

4 Experimental Setup

We evaluate our BDI-LLM framework on the PlanBench benchmark [20], a standardized evaluation suite for assessing LLM planning capabilities across classical planning domains. Our evaluation spans three domains of increasing complexity, totaling 1,270 problem instances.

4.1 Benchmark and Domains

PlanBench [20] provides PDDL-encoded planning problems derived from the International Planning Competition (IPC). We evaluate on three domains:

- **Blocksworld** (200 instances): A classic planning domain with 4 operators (`pick-up`, `put-down`, `stack`, `unstack`). Instances range from 3 to 12 objects with

2–11 goal predicates, requiring sequential manipulation of block towers while respecting physical constraints (single-arm, clear-block requirements).

- **Logistics** (570 instances): A transportation domain involving packages, trucks, and airplanes across multiple cities. Instances contain 7–38 objects with 1–15 delivery goals. This domain tests multi-modal transport reasoning: trucks operate within cities, airplanes fly between airports, and packages must be routed through appropriate transfer points.
- **Depots** (500 instances): A hybrid domain combining elements of Blocksworld and Logistics. All instances contain 18 objects with 1–3 goals. Hoists manipulate crates onto pallets or other crates, trucks transport crates between depots and distributors, requiring coordinated state tracking of hoist availability, truck positions, and crate locations.

4.2 Model Configuration

We use **Gemini 3 Flash Preview** (`vertex_ai/gemini-3-flash-preview`) as the underlying LLM, accessed through the Vertex AI API. The model is configured with:

- Temperature: 0.2 (low temperature for deterministic planning)
- Maximum output tokens: 4,000
- Concurrent workers: 400 (parallel instance evaluation)

Each problem instance is converted from PDDL to a structured natural language representation consisting of *beliefs* (current state description with domain constraints and action specifications) and *desires* (goal specification with worked examples). The LLM generates a BDI plan represented as a directed acyclic graph (DAG) of action nodes.

4.3 Multi-Layer Verification Pipeline

Generated plans undergo a three-layer verification pipeline:

1. **Layer 1 — Structural Verification**: Validates that the plan forms a valid DAG (no cycles, fully connected graph). An automatic repair module can insert virtual `START`/`END` nodes to unify disconnected subgraphs.
2. **Layer 2 — Symbolic Verification (VAL)**: Converts the BDI plan to PDDL action sequences and validates against the domain specification using the VAL plan validator [7]. This checks action preconditions, effects, and goal satisfaction. When VAL reports errors, an *error-driven repair loop* re-prompts the LLM with the specific VAL error messages and cumulative repair history (up to 3 repair attempts).
3. **Layer 3 — Physics Validation**: For Blocksworld, a domain-specific physics simulator traces state transitions to verify physical feasibility. For Logistics and Depots, this layer is deferred to VAL (Layer 2).

A plan is considered *valid* only if it passes all applicable verification layers.

Table 1: Main results on PlanBench. Accuracy denotes the percentage of instances where the generated plan passes all verification layers (structural, symbolic/VAL, and physics).

Domain	Instances	Passed	Failed	Accuracy
Blocksworld	200	200	0	100.0%
Logistics	570	568	2	99.6%
Depots	500	497	3	99.4%
Total	**1,270**	**1,265**	**5**	**99.6%**

Table 2: Comparison with PlanBench baselines. Prior results are from Valmeekam et al. [20] using GPT-4 without structured verification.

Domain	LLM Baseline	BDI-LLM (Ours)	Δ
Blocksworld	$\sim$35%	100.0%	+65.0 pp
Logistics	$<$5%	99.6%	+94.6 pp
Depots	$<$5%	99.4%	+94.4 pp

4.4 Evaluation Metrics

We report the following metrics:

- **Overall Accuracy**: Percentage of instances where the generated plan passes all verification layers.
- **VAL Repair Rate**: Frequency and success rate of the error-driven repair loop.
- **Generation Time**: Wall-clock time from PDDL parsing to final verification.

4.5 Baselines

We compare against the PlanBench baseline results reported by Valmeekam et al. [20], where GPT-4 achieved approximately 35% accuracy on Blocksworld and LLMs generally scored below 5% on Logistics when evaluated without external verification or repair mechanisms. We also reference the broader findings that LLMs without structured prompting or verification typically achieve $<$30% on multi-step planning tasks across these domains.

5 Results

5.1 Main Results

Table 1 and Figure 2 present the overall performance of our BDI-LLM framework across all three PlanBench domains. The framework achieves a combined accuracy of 99.6% (1,265/1,270), demonstrating near-perfect plan generation with formal verification.

5.2 Comparison with Baselines

Table 2 and Figure 2 compare our results with previously reported LLM planning baselines on PlanBench. Our framework achieves substantial improvements over all prior results, with the most dramatic gains in Logistics (+94.6 percentage points over the $\sim$5% baseline) and Blocksworld (+65 percentage points over GPT-4's $\sim$35%).

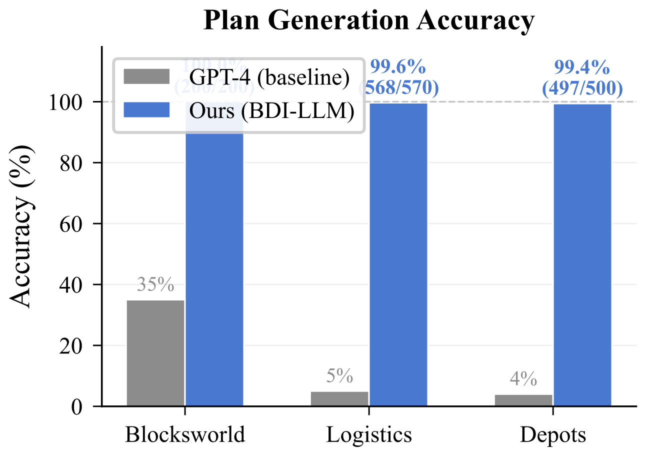

Figure 2: Per-domain accuracy comparison between unverified LLM baselines (GPT-4, from PlanBench) and our BDI-LLM framework. BDI-LLM achieves 65–95 percentage-point improvements across all three domains.

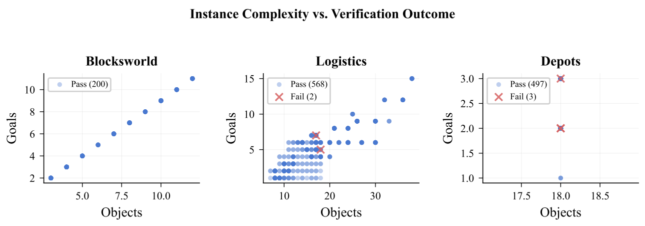

Figure 3: Relationship between instance complexity (number of goal predicates) and planning success rate. BDI-LLM maintains near-perfect accuracy even at the highest complexity levels across all three domains.

5.3 Domain-Specific Analysis

Figure 3 visualizes the relationship between instance complexity and success across all three domains.

Blocksworld. The framework achieves perfect accuracy (200/200) across all complexity levels, from simple 2-goal instances (3 objects) to complex 11-goal instances (12 objects). No VAL repair or structural auto-repair was triggered, indicating that the domain-specific prompt engineering—including bottom-up tower construction ordering, teardown phase generation, and explicit physics constraint enumeration—enables the LLM to generate correct plans on the first attempt. The average generation time is 8.35 seconds, with plan sizes ranging from 4 to 22 action nodes (average 12.2).

Logistics. With 568/570 instances solved (99.6%), Logistics demonstrates the framework's ability to handle complex multi-modal transportation planning. The domain spans a wide complexity range: from single-city, single-package problems (7 objects, 1 goal) to multi-city scenarios with 38 objects and 15 delivery goals. Notably, the framework maintains high accuracy even at the highest complexity levels—all 9 instances with 15 goals and all 17 instances with 12 goals are solved correctly. The VAL repair loop was triggered for 76 instances (13.3%), successfully repairing 75 of them (98.7% repair success rate) across 85 total repair attempts. Structural auto-repair was triggered 8 times with 7 successes.

Depots. The Depots domain achieves 497/500 (99.4%) accuracy. All 500 instances share the same object count (18) but vary in goal complexity (1–3 goals). The VAL repair loop was triggered for 35 instances (7.0%), successfully repairing 32 (91.4% repair success rate) across 39 total attempts. Structural auto-repair was triggered 5 times, all successful.

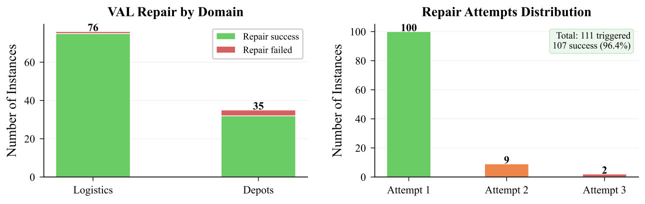

Figure 4: VAL error-driven repair analysis. Left: repair trigger rate and success rate by domain. Right: distribution of repair attempts needed, showing that the majority of repairs succeed on the first attempt.

Table 3: VAL error-driven repair loop statistics by domain. "Triggered" counts instances where at least one repair attempt was made; "Success" counts instances ultimately passing VAL after repair.

Domain	Triggered	Success	Attempts	Rate
Blocksworld	0	0	0	—
Logistics	76	75	85	98.7%
Depots	35	32	39	91.4%
Total	**111**	**107**	**124**	**96.4%**

5.4 VAL Repair Loop Analysis

Table 3 and Figure 4 summarize the error-driven repair loop statistics. The repair mechanism is a critical component: without it, the raw first-attempt accuracy would be lower by approximately 8.4 percentage points (107 additional failures across Logistics and Depots).

Table 4 shows the distribution of repair attempts. The majority of repairs succeed on the first attempt, with diminishing returns for subsequent attempts.

5.5 Failure Analysis

Table 5 details the 5 failed instances. Two distinct failure modes emerge:

1. **Structural failure** (1 instance): The LLM generated a cyclic dependency graph for instance-166 (Logistics), where the plan contained a circular reference among 15 action nodes. This represents a fundamental reasoning error that cannot be repaired by the VAL loop.

Table 4: Repair success by number of attempts (Logistics + Depots combined).

Attempts	Instances	Succeeded
1	100	97
2	9	8
3	2	2

Table 5: Detailed failure analysis for all 5 failed instances.

Instance	Problem	Obj.	Goals	Failure Mode
Logistics				
instance-166	logistics-c3-s3-p5-a3	18	5	Cycle in plan graph (structural)
instance-228	logistics-c3-s3-p7-a2	17	7	Unsatisfied precondition in `load-airplane`; VAL repair failed
Depots				
instance-173	depot-3-1-3-4-4-3	18	2	Empty plan generated
instance-179	depot-3-1-3-4-4-3	18	3	Unsatisfied precondition in `load` (truck position); VAL repair failed
instance-187	depot-3-1-3-4-4-3	18	2	Unsatisfied precondition in `load` (truck position); VAL repair failed

2. **State tracking errors** (4 instances): The remaining failures involve the LLM losing track of object positions after a sequence of move/drive operations. In instance-228 (Logistics), the model incorrectly assumed an airplane was at a location it had already departed from. In the three Depots failures, the model failed to track truck positions after `Drive` actions, attempting to `Load` crates onto trucks at their previous locations. One instance (instance-173) produced an empty plan, suggesting a parsing or generation failure.

5.6 Generation Time

Average generation times vary by domain complexity: Blocksworld averages 8.35s per instance, Logistics 15.62s, and Depots 4.59s. The lower Depots time reflects its smaller goal counts (1–3 goals vs. up to 15 for Logistics). Maximum generation times reach 49.2s (Blocksworld), 55.4s (Logistics), and 159.8s (Depots, likely due to multiple repair attempts).

6 Discussion

6.1 Why Does BDI-LLM Achieve Near-Perfect Accuracy?

Our results demonstrate that the combination of structured BDI prompting, domain-specific natural language conversion, and multi-layer formal verification can elevate LLM planning accuracy from below 35% to above 99%. We attribute this to three synergistic factors:

Domain-Aware Prompt Engineering. Rather than presenting raw PDDL to the LLM, our framework converts each problem instance into a structured natural language representation that explicitly encodes: (1) the current state with typed object inventories, (2) available actions with exact parameter formats and precondition/effect specifications, (3) critical domain constraints highlighted with warning markers, (4) worked examples demonstrating correct reasoning

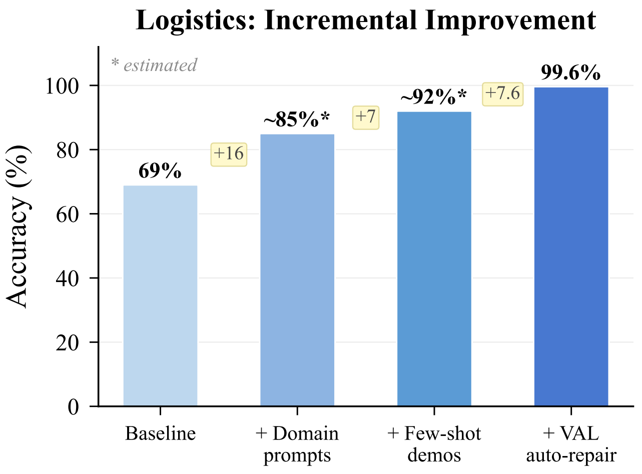

Figure 5: Incremental improvement in Logistics domain accuracy through successive framework enhancements: domain-specific NL conversion, airport identification warnings, few-shot demonstrations, and the VAL error-driven repair loop.

patterns, and (5) mandatory state-tracking instructions. For Blocksworld, we additionally compute a bottom-up tower construction ordering and generate teardown steps for initial stacks that conflict with the goal, effectively reducing the planning problem to plan execution. This domain-specific conversion is key to the 100% Blocksworld accuracy—the LLM receives not just the problem specification but a near-complete solution strategy.

BDI Architecture as Structured Output. The Belief-Desire-Intention architecture constrains the LLM's output to a well-defined DAG structure with typed action nodes and explicit dependency edges. This structured output format prevents common LLM failure modes such as generating ambiguous natural language plans, omitting action parameters, or producing plans with implicit ordering assumptions. The DAG representation also enables efficient structural verification (cycle detection, connectivity checks) before the more expensive symbolic verification.

Error-Driven Repair Loop. The VAL-based repair loop provides a crucial safety net (Figure 5 illustrates the cumulative impact of each framework component on Logistics accuracy). When the LLM's initial plan contains precondition violations or incorrect action sequences, the specific VAL error messages—including the failing action, the unsatisfied precondition, and a repair suggestion—are fed back to the LLM along with the cumulative history of previous repair attempts. This closed-loop feedback enables the LLM to correct localized errors without regenerating the entire plan. The high repair success rate (96.4% overall, 98.7% for Logistics) suggests that most initial errors are minor state-tracking mistakes rather than fundamental reasoning failures.

6.2 Analysis of Failure Modes

The 5 failed instances (0.4%) reveal two categories of LLM limitations:

Graph Structure Errors. Instance-166 (Logistics) produced a cyclic plan graph, indicating that the LLM failed to maintain a consistent temporal ordering across 15 interdependent actions involving 3 cities, 3 airplanes, and 5 packages. Cyclic dependencies represent a fundamental violation of plan semantics that cannot be addressed by the VAL repair loop, as the plan structure itself is invalid. This failure mode is rare (1/1,270 = 0.08%) but highlights the challenge of maintaining global consistency in complex multi-agent coordination scenarios.

Persistent State Tracking Failures. The remaining 4 failures involve the LLM losing track of object positions after movement actions. In the Depots domain, two instances failed because the model attempted to load crates onto trucks at locations the trucks had already departed from. Despite receiving explicit VAL error messages identifying the unsatisfied precondition (`at truck2 depot2`), the repair attempts failed to correct the truck position tracking. This suggests that certain state-tracking errors are deeply embedded in the LLM's reasoning chain and cannot be resolved through local repairs alone.

6.3 The Role of VAL in the Verification Pipeline

Our results underscore the importance of formal verification in LLM-based planning. Without the VAL verification layer, plans that appear structurally valid (correct DAG structure) but contain semantic errors (precondition violations) would be accepted as correct. In our experiments, 111 instances (8.7%) required VAL-based repair, meaning that a system relying solely on structural verification would have achieved approximately 91.2% accuracy instead of 99.6%. The VAL validator serves as both a correctness guarantee and an error signal generator for the repair loop.

6.4 Limitations

Domain-Specific Engineering. The current framework requires domain-specific natural language conversion functions for each planning domain. While the conversion follows a consistent pattern (state description, action specification, constraint enumeration), extending to new domains requires manual engineering of the PDDL-to-NL conversion and domain-specific prompt elements. Future work could explore automated domain adaptation using the PDDL domain specification itself.

Scalability. Our evaluation covers instances with up to 38 objects and 15 goals. Performance on significantly larger instances (e.g., 100+ objects) remains untested. The LLM's context window and reasoning capacity may degrade for very large problem instances, particularly in domains requiring long action sequences.

Single LLM Dependency. All experiments use Gemini 3 Flash Preview. While this demonstrates the framework's effectiveness with a specific model, generalization to other LLMs (GPT-4, Claude, Llama) requires further evaluation. The framework's architecture is model-agnostic, but the optimal prompt engineering may vary across models.

Repair Loop Ceiling. The repair loop is limited to 3 attempts, and 4 of the 5 failures involved repair attempts that did not succeed. Increasing the repair budget or employing alternative repair strategies (e.g., plan-from-scratch regeneration, decomposition into subproblems) could potentially reduce the failure rate further.

6.5 Comparison with Other Approaches

Our work differs from prior LLM planning approaches in several key aspects:

- **LLM+P** [12]: Translates natural language to PDDL and uses classical planners. Our approach keeps the LLM as the planner but adds formal verification, avoiding the brittleness of LLM-generated PDDL.
- **Tree-of-Thought** [23]: Uses search-based prompting for multi-step reasoning. Our BDI framework provides a more structured output format and leverages domain-specific verification rather than generic search.
- **ReAct** [24]: Interleaves reasoning and acting. Our approach generates complete plans upfront and verifies them post-hoc, enabling formal correctness guarantees rather than runtime error recovery.
- **SayCanPay** [6]: Combines LLM planning with cost-based action selection. Our framework focuses on correctness verification rather than cost optimization, achieving higher accuracy through formal methods.

7 Conclusion

We presented BDI-LLM, a framework that integrates the Belief-Desire-Intention cognitive architecture with large language models and multi-layer formal verification for automated planning. Our approach addresses the well-documented unreliability of LLMs in multi-step planning tasks through three key contributions: (1) domain-aware structured prompting that converts PDDL problems into rich natural language representations with explicit constraints and worked examples, (2) a BDI-structured output format that constrains LLM generation to well-formed directed acyclic graphs, and (3) a multi-layer verification pipeline incorporating structural checks, VAL-based symbolic verification, and an error-driven repair loop.

Evaluated on 1,270 PlanBench instances across Blocksworld, Logistics, and Depots, our framework achieves 99.6% overall accuracy—100% on Blocksworld, 99.6% on Logistics, and 99.4% on Depots. These results represent improvements of 65–95 percentage points over prior LLM planning baselines on the same benchmark. The VAL repair loop proves essential, successfully correcting 107 out of 111 initially invalid plans (96.4% repair success rate).

7.1 Future Work

Several directions merit further investigation:

- **Automated Domain Adaptation**: Developing methods to automatically generate domain-specific prompts from PDDL domain specifications, reducing the manual engineering required for new domains.
- **Scaling to Larger Instances**: Evaluating performance on problems with 100+ objects and investigating hierarchical decomposition strategies for complex instances that exceed LLM context limits.
- **Multi-Model Evaluation**: Benchmarking the framework across diverse LLMs (GPT-4o, Claude Opus, Llama 3) to assess generalizability and identify model-specific optimization opportunities.
- **Advanced Repair Strategies**: Exploring plan decomposition, subgoal-based regeneration, and ensemble repair methods to address the remaining 0.4% failure cases.
- **Real-World Deployment**: Extending the framework to robotics task planning and software agent orchestration, where formal verification of LLM-generated plans is critical for safety.

References

[1] Ahn, M.; Brohan, A.; Brown, N.; Chebotar, Y.; Cortes, O.; David, B.; Finn, C.; Fu, C.; Gober, K.; Gopalakrishnan, K.; et al. 2022. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. In *Conference on Robot Learning (CoRL)*.

[2] Bordini, R. H.; Hübner, J. F.; and Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons.

[3] Bratman, M. E. 1987. *Intention, Plans, and Practical Reason*. Harvard University Press.

[4] Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

[5] 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 *Advances in Neural Information Processing Systems (NeurIPS)*.

[6] Hazra, R.; Dos Martires, P. Z.; and De Raedt, L. 2024. SayCanPay: Heuristic Planning with Large Language Models Using Learnable Domain Knowledge. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[7] Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 294–301. IEEE.

[8] Huang, W.; Abbeel, P.; Pathak, D.; and Mordatch, I. 2022. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. In *International Conference on Machine Learning (ICML)*, 9118–9147.

[9] Huang, W.; Xia, F.; Xiao, T.; Chan, H.; Liang, J.; Florence, P.; Zeng, A.; Tompson, J.; Mordatch, I.; Chebotar, Y.; et al. 2023. Inner Monologue: Embodied Reasoning through Planning with Language Models. *arXiv preprint arXiv:2207.05608*.

[10] Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Siddarth, S.; and Sreedharan, S. 2024. LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. *arXiv preprint arXiv:2402.01817*.

[11] Khattab, O.; Singhvi, A.; Maheshwari, P.; Zhang, Z.; Santhanam, K.; Vardhamanan, S.; Haq, S.; Sharma, A.; Joshi, T. T.; Mober, H.; et al. 2024. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. In *International Conference on Learning Representations (ICLR)*.

[12] Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biber, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *arXiv preprint arXiv:2304.11477*.

[13] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL—The Planning Domain Definition Language. In *Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control*.

[14] Rao, A. S. 1996. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, 42–55. Springer.

[15] Rao, A. S.; and Georgeff, M. P. 1995. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, 312–319.

[16] Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; and Yao, S. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *Advances in Neural Information Processing Systems (NeurIPS)*.

[17] Silver, T.; Hariprasad, S.; Shuttleworth, R. S.; Kumar, N.; Lozano-Pérez, T.; and Kaelbling, L. P. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence*.

[18] Valmeekam, K.; Marquez, M.; and Kambhampati, S. 2023. Can Large Language Models Really Improve by Self-Critiquing Their Own Plans? *arXiv preprint arXiv:2310.08118*.

[19] Valmeekam, K.; Marquez, M.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2024. On the Planning Abilities of Large Language Models—A Critical Investigation. *arXiv preprint arXiv:2305.15771*.

[20] Valmeekam, K.; Marquez, M.; Sreedharan, S.; and Kambhampati, S. 2023. PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[21] Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.; Le, Q. V.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[22] Winikoff, M. 2005. Jack Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming*, 175–193. Springer.

[23] Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T. L.; Cao, Y.; and Narasimhan, K. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[24] Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.