LiTS: A Modular Framework for LLM Tree Search

Xinzhe Li
RMIT University
xinzhe.li@rmit.edu.au

Yaguang Tao
RMIT University
yaguang.tao@rmit.edu.au

Abstract

LiTS is a modular Python framework for LLM reasoning via tree search. It decomposes tree search into three reusable components—Policy, Transition, and RewardModel—that plug into algorithms like MCTS and BFS. A decorator-based registry enables domain experts to extend to new domains by registering components, and algorithmic researchers to implement custom search algorithms. We demonstrate composability on MATH500 (language reasoning), Crosswords (environment planning), and MapEval (tool use), showing that components and algorithms are orthogonal: components are reusable across algorithms within each task type, and algorithms work across all components and domains. We also report a mode-collapse finding: in infinite action spaces, LLM policy diversity—not reward quality—is the bottleneck for effective tree search. A demonstration video is available at https://youtu.be/nRGX43YrR3I. The package is released under the Apache 2.0 license at https://github.com/xinzhel/lits-llm, including installation instructions and runnable examples that enable users to reproduce the demonstrated workflows.

1 Introduction

Tree search methods for LLM inference—such as Tree-of-Thoughts (Yao et al., 2023), RAP (Hao et al., 2023), and ReST-MCTS (Zhang et al., 2024)—have demonstrated strong performance on complex reasoning tasks. However, existing implementations are largely task-specific, requiring substantial reimplementation effort when adapting to new domains or comparing across methods.

We present **LiTS** (Language Inference via Tree Search), a modular Python framework designed to let users *modify what they need and reuse the rest*. LiTS targets two user groups with complementary workflows:

- **AI/NLP researchers** devise novel search algorithms for direct evaluation on downstream tasks, or design new reasoning structures (e.g., sub-question decomposition (Hao et al., 2023))— changes to step structure cascade only to dependent abstractions (detailed in §2), reusing search algorithms.
- **Domain experts** inject domain-specific logic (prompts, tools, environment dynamics) without touching search internals.

LiTS achieves this by isolating inference algorithms from domain-specific implementations, enabling bidirectional reusability (Table 1): domain experts can compare different methods (MCTS, BFS, CoT) with identical domain components for fair algorithmic comparison; researchers can evaluate the same method across domains by swapping only domain-specific components.

Correspondingly, LiTS also addresses three key challenges in developing LLM reasoning agents.

1. **Reusability**: General data structures (State, Action, Step) decouple inference algorithms from domain implementations, enabling bidirectional sharing (Table 1). Domain-specific logic is further decomposed into three fine-grained components—Policy, Transition, and RewardModel (Li, 2024). As a result, domain experts reuse search algorithms across domains by swapping only domain-specific components; researchers evaluate the same algorithm across domains with identical domain logic. The package design would be further demonstrated in §2.
2. **Extensibility**: Components and search modules can be extended via external import and decorator-based registration, without modifying the core package. Table 2 exemplifies extension patterns which would further demonstrated in §3.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 47–56
July 2-7, 2026 ©2026 Association for Computational Linguistics

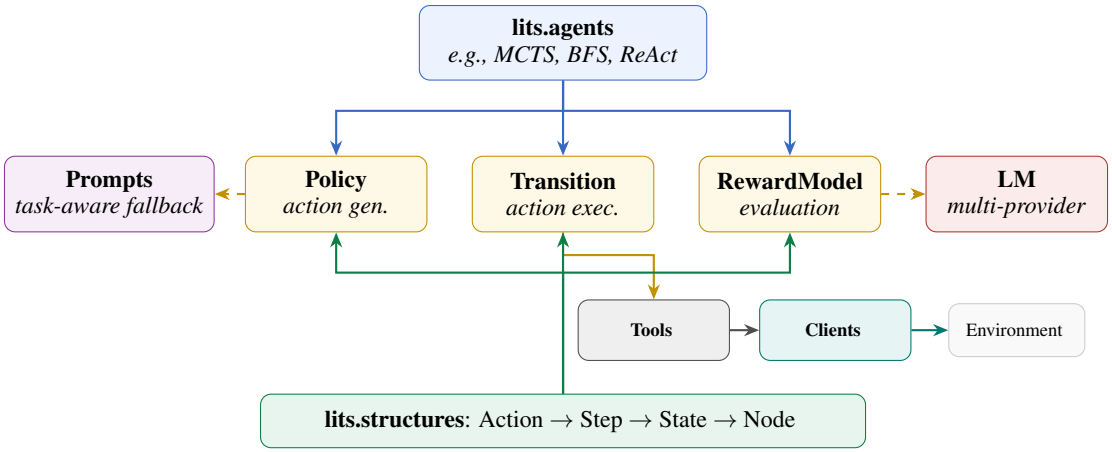

Figure 1: LiTS architecture. Agents orchestrate Components (Policy, Transition, RewardModel). Components use Prompts and LM interfaces. Transition can invoke Tools via Clients for tool-use tasks. All layers communicate through shared Data Structures, decoupling search algorithms from domain logic.

3. **Observability**: Built-in `InferenceLogger` tracks token usage and latency at component, instance, and search-phase granularities, plus incremental checkpointing for post-training.

The accompanying video demonstrates both modular configuration workflows and a brief live execution illustrating real tree-search behavior.

User Group	Reusability Direction	Benefit
Domain experts	Same components → different methods	Fair algorithmic comparison
AI/NLP researchers	Same method → different domains	Generalization testing
AI/NLP researchers	New formulation → existing search	Novel reasoning paradigms

Table 1: Bidirectional reusability enabled by LiTS's modular design.

2 LiTS: A Unified Grammar

LiTS abstracts three task types— **Environment Grounded** (e.g., BlocksWorld), **Language Grounded** (e.g., Math QA), and **Tool Use** (e.g., MapEval)—to unify diverse LLM reasoning paradigms.

Figure 1 presents the architecture. Each module is a separately deployable unit with dependencies on adjacent modules.

2.1 lits.structures: General Data Structures for Decoupling

The framework relies on a hierarchical type system to separate inference algorithms from domain-specific implementations: Action → Step → State → Node. Each level defines: (1) a **task-agnostic interface** that enables `lits.agents` to operate without knowing underlying implementations, and

- **Action** (general interface): The atomic unit generated by Policy.
- **Step** (general interface): Encapsulates an action with its execution results (if any).
- **State** (general interface): Accumulation of steps over time, with unified rendering methods.
- **Node** (general interface): Search-oriented wrapper managing parent pointers, children, rewards, and visit counts for search algorithms.

(2) *polymorphic subclasses* for message passing between task-specific components (see Appendix Table 9 for concrete subclasses across task types).

This design decouples search algorithms from task-specific logic. Researchers implement new reasoning domains by defining task-specific Action, Step, and State subclasses, while reusing the same search pipelines (BFS, MCTS, etc.).

2.2 Modular LLM Components (lits.components)

LiTS decomposes LLM-based reasoning into three modular components, defined by a **general interface** with task-specific subclasses: **Policy** generates candidate actions from states . **RewardModel** evaluates action quality for search guidance . **Transition** executes actions and produces new states.

Task Type	Minimum Required Injection	Extension Mechanism	Framework Knowledge
Language-grounded	Prompt, Dataset	`@register_user_prompt`, `@register_dataset`	None
Env-grounded	Transition class	`@register_transition` (1 decorator)	Minimal[‡]
Tool-use	Tool definitions	`BaseTool` subclass	Minimal[*]

Table 2: Domain extension patterns by task type. Component reuse varies: tool-use tasks share Policy and Transition across domains; env-grounded tasks share Policy and RewardModel; language-grounded tasks may require task-specific prompts. [‡]Requires understanding Transition interface (`goal_check`, `generate_actions`, `_step`). [*]`BaseTool` requires `name`, `description`, `args_schema`, and `_run()`—compatible with LangChain's Tool interface.

Provided Implementations. Most components are designed for a general task type (as are the prompts described below). They declare a `TASK_TYPE` class constant (e.g., `'language_grounded'`, `'tool_use'`, `'env_grounded'`). For example, both math reasoning and commonsense reasoning can use the same components, along with default prompts.

However, component design is not limited to the predefined types. For task instances requiring specialized behavior, developers can create task-instance-specific components with `TASK_TYPE = None`. Our package provides `BlocksWorldTransition` as an example, which implements BlocksWorld-specific state parsing and goal checking. Subclass implementations focus solely on task-specific logic; infrastructure concerns such as inference logging, prompt registry lookup, and search-phase context are handled by base classes.

Decorator-Based Component Extension. LiTS uses a `ComponentRegistry` with decorator-based registration so that users extend the framework without modifying core code. For environment-grounded tasks, users register a `Transition` subclass (implementing `_step`, `goal_check`, and `generate_actions`) via `@register_transition("name")` alongside a dataset loader via `@register_dataset("name")`; by convention both share the same key, so the factory resolves the transition from the dataset name (e.g., `-dataset crosswords`; see §4.1). For language-grounded tasks, a *search framework* (e.g., `-search_framework rap`) bundles a matching Policy, Transition, and RewardModel from the registry, while users may also define custom Step subclasses for novel reasoning formulations (see §4.3). For tool-use tasks, users define tools via `BaseTool` subclasses; the generic `ToolUsePolicy` and `ToolUseTransition` handle orchestration (see §4.2). In all cases, users can independently specify `-policy`, `-transition`, and `-reward` on the CLI to compose custom component configurations without writing new code—effectively assembling their own search formulations from registered building blocks (see §4.3 for a concrete example). This differentiates LiTS from LLM Reasoners (Hao et al., 2024): rather than requiring task-specific component implementations for each search method, LiTS provides a plug-and-play extension mechanism where domain experts focus solely on domain logic.

2.3 lits.prompts: Task-Instance-Prioritized Fallback.

LiTS uses a centralized `PromptRegistry` to manage prompts for all LLM-based components, enabling prompt reuse and customization. Components use two prompt types: `task_prompt_spec` (system instructions) and `usr_prompt_spec` (user message templates).

The registry supports a fallback lookup priority: explicit parameter → `task_name` → `TASK_TYPE` → default. This allows users to: (1) reuse general task-type prompts for new task instances (e.g., a new math dataset can use `language_grounded` prompts), or (2) register task-instance-specific prompts when needed. Task-instance-specific components should set `TASK_TYPE = None` to skip the `TASK_TYPE` fallback, avoiding potential mismatch between prompts and parsing logic.

2.4 Agents

LiTS provides chain agents (`ReActChat`, `EnvChain`) for sequential reasoning and tree search agents (`MCTS`, `BFS`) for branching exploration. Chain and tree methods share the

same Policy and Transition—tree search adds a RewardModel for path selection. This component sharing enables fair comparison: identical domain logic evaluated under different inference strategies.

Extending Search Algorithms. All tree search algorithms inherit from a `BaseTreeSearch` abstract base class that handles peripheral concerns—node ID management, root creation, checkpoint I/O, runtime limits, terminal node collection, and binding search-specific context (e.g., tree depth, iteration index) to LLM call records for fine-grained cost analysis—so that subclasses implement only the `search()` method containing pure algorithm logic. New algorithms are registered via `@register_search("name")`, which wraps the class into a callable that `main_search.py` invokes through a unified interface. Custom algorithms (e.g., beam search variants, Monte Carlo variants) automatically work with all registered components and task types, because `lits.structures` standardizes the interface between search algorithms and components.

2.5 Tool Use

LiTS adopts a LangChain-compatible tool protocol: each tool is a `BaseTool` subclass declaring `name`, `description`, `args_schema`, and a `_run()` method. This ensures heterogeneous actions (SQL queries, API calls, geospatial lookups) share a consistent callable interface, so the same `ToolUsePolicy` and `ToolUseTransition` invoke them interchangeably. Tools are instantiated with a backend `Client` that encapsulates I/O logic, decoupling reasoning from infrastructure. At runtime, a list of `Tool` instances is passed to the agent; the policy selects tools and the transition executes them, grounding abstract reasoning into concrete world interactions. For benchmarks whose tools carry per-example mutable state (e.g., a knowledge-graph variable tracker that differs per question), the resource registry supports an optional `prepare_example` callback that resets tool internals before each example—stateful tools are natively supported by `lits-chain` and `lits-search` without benchmark-specific CLI modifications.

3 Demonstrations and Results

To demonstrate LiTS's generality and extensibility, we evaluate on three task categories, each showcasing different extension mechanisms. Our goal is to validate component reusability rather than benchmark performance; see Appendix B for full settings and Appendix Table 10 for component configurations. Environment-grounded and tool-use experiments use Claude 3.5 Sonnet via AWS Bedrock API, so we report cost; language-grounded experiments use self-deployed Llama3-8B, so we report wall-clock time.

LiTS provides a *CLI-first* workflow where all run artifacts (configs, logs, checkpoints, evaluation outputs) are written to a single `save_dir`, enabling post-hoc evaluation via `eval -save_dir` without re-specifying configuration (Appendix A). Search artifacts are saved incrementally—terminal nodes, paths, and intermediate states—so evaluation can be reproduced without re-running search. We report accuracy and efficiency metrics captured by LiTS's `InferenceLogger`.

3.1 Environment-Grounded Tasks: Adding New Domains

BlocksWorld is a domain tested in RAP (MCTS) (Hao et al., 2023), re-implemented under LiTS. To add Crosswords, a domain with different state representations, users register three components sharing the key `"crosswords"`: a Transition subclass, domain-specific prompts, and a dataset loader. The same `EnvGroundedPolicy` and `EnvGroundedPRM` are reused.

```
@register_transition("crosswords",
    task_type="env_grounded")
class CrosswordsTransition(
    EnvGroundedTransition):
    @staticmethod
    def goal_check(goals, env_state):
    ...
    @staticmethod
    def generate_actions(env_state): ...
    def _step(self, state, action, goals
    , **kwargs): ...

@register_user_prompt('policy', '
    env_grounded', 'crosswords')
def crosswords_policy_prompt():
    return PromptTemplate(template="..."
    , ...)

@register_dataset("crosswords",
    task_type="env_grounded")
def load_crosswords(data_file=None, **
    kwargs): ...
```

Switching from BlocksWorld to Crosswords requires only `-dataset crosswords` on the CLI—the same agent works automatically.

For search, we use 10 iterations with branching factor 3 for BlocksWorld (max depth 6, early termination on first solution) and 30 iterations for Crosswords (max depth 10, no early termination),

with rollout depth matching max search depth in both cases. All these can be set in the `lits-search` CLI command.

Task	Method	Out Tok	Cost	Acc
BlocksWorld	Chain	17K	$1.48	26.7%
	MCTS	488K	$21.99	66.7%
Crosswords	Chain	2.5K	$0.28	6.67%/10.33%*
	MCTS	14K	$2.42	0%22.67%*

Table 3: Environment-grounded results (30 examples). Same components across domains. *Crosswords accuracy: exact match (all 10 clues correct) / partial match (average clue accuracy).

Mode Collapse in Infinite Action Spaces. Despite temperature escalation (0.8→1.2) upon duplicate detection, we observe an 81.1% duplicate rate across 127 LLM calls on Crosswords (Table 4), with nearly identical rates among incorrect outputs. Even with oracle rewards from ground-truth answers, tree search fails—confirming the bottleneck is action diversity, not reward quality. While our measurements are on a single environment, the cause is structural: in open-ended text action spaces, the policy is an LLM with no calibrated stochastic policy head, and temperature-based sampling explores at the token level rather than the action level. Two LLM calls with the same prompt therefore tend to produce semantically duplicated actions even at high temperature, regardless of domain. In contrast, finite action spaces (e.g., BlocksWorld) guarantee branching diversity via deterministic fallback to unselected valid actions.

Metric	Value
Unique states visited	16
Avg. policy calls per state	7.9
Duplicate rate (all)	81.1%
Duplicate rate (incorrect)	81.0%
Correct outputs	17.3%

Table 4: LLM action diversity on Crosswords with temperature escalation (T: 0.8→1.2). Near-identical duplicate rates for all outputs vs. incorrect-only outputs indicate temperature scaling fails to improve exploration.

3.2 Tool-Use Tasks: Resource Registration

For tool-use tasks, users register two functions sharing the same key: a dataset loader (via `@register_dataset`) and a resource loader (via `@register_resource`) that returns the tools and context the agent can use during search.

```
@register_dataset("mapeval-sql",
    task_type="tool_use")
def load_mapeval_sql(**kwargs):
    return [{"question": ..., "answer":
    ...}, ...]

@register_resource("mapeval-sql")
def load_mapeval_sql_resource(**kwargs):
    return {
        "tools": [QuerySQLTool(db),
    ...],
        "tool_context": "...",
    }
```

Tools follow a LangChain-compatible `BaseTool` protocol (`name`, `description`, `args_schema`, `_run()`); users can use built-in tools or define their own subclasses. The generic `ToolUsePolicy` and `ToolUseTransition` handle orchestration—no component flags are needed on the CLI.

Out Tok	Inv	Cost	Acc
10.6K	62	$0.57	40%

Table 5: MapEval-SQL ReAct results (10 examples). **Inv** = total LLM invocations.

On MapEval-SQL (10 examples), ReAct achieves 40% (Table 5). We ran MCTS on a 3-example subset following the LATS reward design (Zhou et al., 2024); at $18.40 for 3 examples ($6.13/example vs. $0.05/example for ReAct), it achieved 0%. Self-preference bias (Chen et al., 2025) in the LLM-as-judge reward model leads MCTS to favor verbose but incorrect queries. This contrasts with §3.1, where ground-truth environment rewards enable effective tree search—highlighting that reward model quality is the bottleneck for tool-use tree search.

3.3 Language-Grounded Tasks: Composability and Extensibility

This subsection demonstrates three levels of extensibility on MATH500.

Component Reuse Across Algorithms. ReST-MCTS* (Zhang et al., 2024) uses built-in components—ConcatPolicy, ConcatTransition, and GenerativePRM—requiring no user code. Since the original PRM is unreleased, we substitute the publicly available PRM from Xiong et al. (2024), illustrating how decoupled components enable reproduction with missing pieces.

Custom Formulations via Registered Components. RAP (Hao et al., 2023) requires a different reasoning structure: sub-question decomposition instead of step-by-step concatenation. Users register custom components that redefine the Step structure and the Policy/Transition logic:

```
@dataclass
class SubQAStep(Step):
    sub_question: str = ""
    sub_answer: str = ""

@register_policy("rap")
class RAPPolicy(Policy):
    """Generates candidate sub-questions
    ."""
    def get_actions(self, state, **kw):
    ...

@register_transition("rap")
class RAPTransition(Transition):
    """Answers sub-questions, updates
    state."""
    def _step(self, state, action, **kw)
    : ...
```

The MCTS algorithm itself needs no code changes—only parameter tuning. This follows the same decorator pattern as §3.1, but registers Policy, Transition, and RewardModel instead of just a Transition.

Custom Search Algorithms. BFS (Yao et al., 2023) is implemented as a registered search algorithm via `@register_search`, inheriting peripheral concerns from `BaseTreeSearch` (§3.4):

```
@register_search("bfs", config_class=
    BFSConfig)
class BFSSearch(BaseTreeSearch):
    def search(self, query, query_idx):
        # Pure algorithm logic: depth-
    bucketed
        # frontier loop with beam
    pruning
        ...
```

BFS automatically works with all registered components—the same ConcatPolicy, ConcatTransition, and GenerativePRM used by ReST-MCTS* require no adaptation.

Setup. We evaluate on the first 100 MATH500 examples with numerical answers using Llama3-8B. RAP requires a completion model (Llama3-8B base); ReST and ToT use the instruct variant (Llama3-8B-Instruct). All tree search methods use 10 iterations, branching factor 3, and temperature 0.7–0.8 (see Appendix B for full settings).

Analysis. ReST (MCTS) and ToT (BFS) share identical built-in components, isolating the effect of the search algorithm: BFS achieves comparable accuracy (39% vs. 37%) in roughly half the wall-clock time, demonstrating that a user-registered algorithm immediately benefits from existing components. RAP, despite using MCTS with the same iteration budget, achieves only 18%—its sub-question decomposition formulation is less effective on MATH500 than step-by-step reasoning, illustrating how component choice dominates algorithm choice. All three tree search methods improve over CoT (17%), confirming that branching exploration helps even with a small model.

Method	Acc	Out Tok	Inv	Time
CoT	17%	12.9K	100	0.6h
RAP (MCTS)†	18%	4.47M	3.6K	8.0h
ReST (MCTS)	37%	2.24M	4.0K	26.0h
ToT (BFS)‡	39%	1.53M	2.8K	14.7h

Table 6: MATH500 results (100 examples, Llama3-8B). †User-registered components; ‡user-registered search algorithm; unmarked rows use only built-in components and algorithms. **Inv** = total LLM invocations; **Time** = total wall-clock time including all LLM calls.

Feature	LiTS	LLM-R	LG
Task-agnostic search	✓	✓	✗
Component sharing	✓	✗	✗
Tool-use tree search	✓	✗	✗
Prompt registry	✓	✗	✗
Decorator extension	✓	✗	✗
Inference logging	✓	✗	✗
Checkpointing	✓	✗	~

Table 7: Feature comparison. LLM-R = LLM Reasoners; LG = LangGraph; ~ = partial.

4 Related Work

We compare LiTS with **LLM Reasoners** (Hao et al., 2024), the only other framework providing complete tree search implementations, and **LangGraph** (LangChain Inc, 2025), the most widely adopted LLM orchestration framework, which practitioners may consider as an alternative despite lacking native tree search support. Table 7 summarizes key differences.

LLM Reasoners bundles task-specific logic into monolithic configuration classes, requiring users to re-implement full task logic for each search method. LiTS factors out reusable components—users implement domain-specific functions once. LangGraph requires custom state types, expand/score/prune functions, and manual graph wiring for tree search; LiTS provides pre-

implemented algorithms where the same components work across tasks with only prompt registration.

5 Conclusion

We presented LITS, a modular framework that decomposes LLM tree search into reusable components—Policy, Transition, and RewardModel—shared across search algorithms and task types. A decorator-based registry enables extension without modifying core code: users register domain-specific components, datasets, tools, or search algorithms, and the CLI automatically resolves them by registered name. Our demonstrations show that the same built-in components work across MCTS and BFS, that adding a new domain requires only a Transition subclass and dataset loader, and that tool-use benchmarks integrate via resource registration. We also identify that in infinite action spaces, LLM policy diversity—not reward quality—is the primary bottleneck for effective tree search.

Extensibility in practice. The modular design has already supported follow-up research without modifying the core: a Branching-Necessity (BN) evaluator that drives the Chain-in-Tree continuation phase (Li, 2025) integrates as a new component type alongside Policy/Transition/RewardModel; a context-augmentation module pluggable at the Policy layer is currently under active development for cross-trajectory memory and self-reflection. Both extensions reuse the same data structures and search loop without changes.

Reproducibility Statement

Every run writes a single `config.json` containing all resolved CLI flags, component selections, and per-component arguments (including defaults) to its `save_dir`, alongside seeded sampling and saved checkpoints (§3, Appendix A). Evaluation can then be reproduced from saved terminal nodes alone via `lits-eval -save_dir`, which auto-loads the config from `save_dir/config.json` without re-running search. Because the saved `config.json` captures every parameter, any reported run can also be re-executed by passing the same flags back to `lits-search`.

Limitations and Future Work

LITS's scope is intentionally narrow at present, with several directions left for follow-up work:

- **Empirical scale.** Our evaluation is demonstration-focused; rigorous benchmarking across larger datasets and a wider range of model sizes is left for downstream studies that build on LITS.
- **Mode-collapse generality.** The finding is based on a single environment (Crosswords). Systematic study across decoding strategies (nucleus sampling variants, temperature schedules, structured generation constraints) and additional open-ended domains is needed to establish generality.
- **Tool-use reward calibration.** The negative result reflects a known limitation of LLM-as-judge reward models (Chen et al., 2025) rather than a framework limitation, but motivates integrating calibrated verifiers and trained PRMs as future work.
- **Algorithm coverage.** Only MCTS and BFS ship as built-ins; A* and beam-search variants are natural extensions via `@register_search` and are planned.
- **Throughput.** Tree search at scale requires concurrent and batched LLM calls; this is an ongoing engineering direction.
- **Tool ecosystem.** The current `BaseTool` interface (compatible with LangChain) requires each tool to be implemented as a Python class and registered before search begins via `@register_resource`. Adding a new tool therefore requires editing and reloading the user's Python module. We plan to integrate the Model Context Protocol (MCP), under which tools are exposed by separate MCP servers and an agent connects via an MCP client to enumerate available tools and invoke them through a standard JSON-RPC interface. With this, users can attach a third-party MCP server (e.g., for databases or file systems) directly from the CLI, and tool updates propagate without code changes in LITS.

References

Zhi-Yuan Chen, Hao Wang, Xinyu Zhang, Enrui Hu, and Yankai Lin. 2025. Beyond the surface: Measuring self-preference in LLM judgments. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1672,

Suzhou, China. Association for Computational Linguistics.

Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, Zhen Wang, and Zhiting Hu. 2024. LLM reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173, Singapore. Association for Computational Linguistics.

LangChain Inc. 2025. Langgraph. `https://github.com/langchain-ai/langgraph`.

Xinzhe Li. 2024. A review of prominent paradigms for llm-based agents: Tool use (including rag), planning, and feedback learning. *arXiv preprint arXiv:2406.05804*.

Xinzhe Li. 2025. Chain-in-tree: Back to sequential reasoning in llm tree search. *arXiv preprint arXiv:2509.25835*.

Wei Xiong, Hanning Zhang, Nan Jiang, and Tong Zhang. 2024. An implementation of generative prm. `https://github.com/RLHFlow/RLHF-Reward-Modeling`.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Preprint*, arXiv:2305.10601.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. ReST-MCTS*: LLM self-training via process reward guided tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning acting and planning in language models.

A Run Artifacts and Directory Structure

LiTS writes all run artifacts to a single `save_dir`, enabling post-hoc evaluation via `eval -save_dir` without re-specifying configuration. Table 8 summarizes the artifact layout for search and chaining runs.

Polymorphic Serialization. A central `TYPE_REGISTRY` resolves polymorphism across `Action`, `Step`, and `State` subclasses. The `state.save()` method serializes trajectories into JSON with `__type__` metadata, ensuring that heterogeneous traces—such as a ReAct trajectory containing interleaved thoughts, tool calls, and environment observations—are preserved with full fidelity. `state.load()` utilizes the registry to dynamically reconstruct the exact Python dataclasses from these JSON payloads.

B Experimental Setup

Below are the setups for the three types of tasks: (1) *language-grounded reasoning* using MATH500 (100 examples), (2) *environment-grounded planning* using BlocksWorld and Crosswords (30 examples each), and (3) *tool-use tasks* using MapEval-SQL (10 examples).

As a demonstration paper, our goal is to validate component reusability across task types rather than benchmark state-of-the-art performance. The sample sizes are sufficient to demonstrate that the same components work across algorithms (MCTS vs. BFS) and domains (BlocksWorld vs. Crosswords), which is the core claim of this work. Tree search methods are computationally expensive—each search iteration requires multiple LLM calls for policy sampling and reward evaluation—so we use representative subsets that balance demonstration coverage with practical cost.

Environment-grounded (§3.1) and tool-use (§3.2) experiments use Claude 3.5 Sonnet via AWS Bedrock. Language-grounded experiments (§3.3) use self-deployed Llama3-8B: the base model for RAP (which requires a completion model) and Llama3-8B-Instruct for ReST-MCTS* and ToT-BFS. All tree search methods use $n\_actions = 3$; see per-task settings in §3.1, §3.2, and §3.3.

C Additional Tables and Interfaces

BaseTool Interface. Each tool extends `BaseTool` with three attributes and one method:

```
class QuerySQLTool(BaseTool):
    name = "query_sql"
    description = "Execute SQL query"
    args_schema = QuerySQLInput  # Pydantic model
    def _run(self, query: str) -> str:
    ...
```

Search (`lits_search`)	**`blocksworld_rap/run_0.2.5/`**
`checkpoints/`	`{query_idx}_{iter}.json`: intermediate tree states per iteration; `{query_idx}_result.json`: final result with full tree path.
`terminal_nodes/`	`terminal_nodes_{query_idx}.json`: all terminal nodes found.
`config.json`	Search configuration (many params).
`execution.log`	Execution log.
`eval.log`	Evaluation log.
`inferencelogger.log`	LLM inference log (token/latency).
`treetojsonl.jsonl`	Selected paths for visualization.
Chaining (`lits_chain`)	**`blocksworld_chain/run_0.2.5/`**
`checkpoints/`	`{query_idx}.json`: single trajectory per query.
`config.json`	Simple config (fewer params).
`execution.log`	Execution log.
`eval.log`	Evaluation log.
`eval_results.json`	Aggregated evaluation results.
`inferencelogger.log`	LLM inference log (token/latency).

Table 8: Per-run artifact layout for search and chaining in LiTS. Each run writes all outputs under a single `save_dir`, enabling post-hoc inspection and evaluation via `eval -save_dir`.

Task Type	**Action**	**Step**	**State**
Language-Grounded	`StringAction`	`ThoughtStep` / `SubQAStep`	`TrajectoryState`
(e.g., Math500, GSM8K)	Free-form text generation	Reasoning thought or sub-QA pair	Sequence of reasoning steps
Tool-Use	`ToolUseAction`	`ToolUseStep`	`ToolUseState`
(e.g., MapEval)	Structured tool call (JSON)	Thought, Action, Observation	Trajectory with tool history
Environment-Grounded	`EnvAction`	`EnvStep`	`EnvState`
(e.g., BlocksWorld)	Valid environment command	Action and resulting state	Trajectory + Env Snapshot

Table 9: Structure subclasses across task types. Each task type defines polymorphic Action, Step, and State subclasses that carry task-specific data while exposing a uniform interface to search algorithms.

Task Type	**Execution**	**Method**	**Policy**	**Transition**	**Reward Model**
Environment-Grounded	Chain	EnvChain	EnvGroundedPolicy	BlocksWorldTransition†	–
	Tree	MCTS / BFS	EnvGroundedPolicy	BlocksWorldTransition†	EnvGroundedPRM
	Tree	MCTS / BFS	EnvGroundedPolicy	CrosswordsTransition†	EnvGroundedPRM
Tool-Use	Chain	ReActChat	ToolUsePolicy	ToolUseTransition	–
	Tree	MCTS / BFS	ToolUsePolicy	ToolUseTransition	ToolUsePRM
Language-Grounded	Tree	MCTS	RAPPolicy†	RAPTransition†	RapPRM†
	Tree	MCTS / BFS	ConcatPolicy	ConcatTransition	GenerativePRM

Table 10: Component configurations used in our demonstrations. Unmarked components are built-in (shipped with LiTS); †components are user-registered via `@register_*` decorators. The same built-in components are reused across methods within a task type; only the RewardModel is added for tree search.