L.E.A.R.N.: A Hybrid Architecture For Language-Guided Induction of Hierarchical Task Networks

Glen Smith GLENSMITH@GATECH.EDU
Christopher J. MacLellan CMACLELL@GATECH.EDU
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA

Abstract

Hierarchical Task Networks (HTNs) provide an interpretable, structured framework for problem solving but rely on a fixed set of predefined operators. Further, learning them can require numerous examples to develop new procedures. In contrast, Large Language Models (LLMs) offer generative flexibility but lack the reliability and transparency required for robust cognitive systems. This paper introduces L.E.A.R.N. (Learning by Example Authoring and Reasoning Network), a hybrid cognitive architecture that integrates the strengths of both approaches to learn task hierarchies. Unlike other learning systems, L.E.A.R.N. can use its LLM module to acquire new primitive operators. We evaluate our system in the intelligent tutoring system domain to create expert models for tutor problems. Our evaluation shows that L.E.A.R.N. acquires expert problem-solving skills, such as solving quadratic equations, faster and with fewer demonstrations than an HTN-only baseline, while still providing the explainability and reliability that purely generative models lack. The architecture represents a step toward more adaptive and flexible cognitive systems.

1. Introduction

A central feature of human cognition is the ability to decompose complex problems into manageable sub-tasks. When solving a novel problem, people rarely reason monolithically; instead, they break a goal into smaller steps that can be planned, executed, and refined independently. This hierarchical organization of thought supports modularity and reuse, allowing individuals to reason at multiple levels of abstraction – from overarching goals down to concrete actions. Such structuring is one of the defining hallmarks of human problem-solving and has inspired numerous computational models that emulate it.

Among these, Hierarchical Task Networks (HTNs) (Nau et al., 2003; Nejati et al., 2006) have proven especially effective for representing procedural knowledge in a structured and interpretable way. HTNs model problem-solving as a hierarchy of tasks, methods, and primitive operators, mirroring how people decompose actions into constituent steps. An accompanying planner or interpreter reasons over this structure to decompose tasks into subtasks and select appropriate operators for execution. However, while such systems capture the process of human reasoning, it does not fully capture its *flexibility*. Humans can readily extend their behavioral repertoire by learning new actions from experience and integrating them into existing hierarchies, whereas many symbolic systems are constrained by a fixed library of predefined operators. When faced with a situation that requires an unmodeled action, such systems cannot proceed without expert intervention.

In addition to executing tasks, HTN-based agents may also learn them. Learning an HTN from examples involves inferring both the *structure* of the task hierarchy – how high-level tasks decompose into subtasks – and the *preconditions* that determine when each method is applicable. During this process, the learner searches over its space of operators to explain the demonstrations it receives, seeking sequences of known skills that could reproduce the observed outcomes. As the number of available operators grows, this search becomes increasingly intractable, and the quality of what is learned depends strongly on the examples provided by humans. If demonstrations are sparse or unrepresentative, the resulting methods may fail to generalize beyond the specific contexts they were shown. This dependence on exhaustive and well-curated examples makes traditional HTN learning difficult to scale, particularly in domains like intelligent tutor authoring, where agents must acquire new skills across a wide variety of domains.

By contrast, large language models (LLMs) exhibit strong generative flexibility. Trained on vast corpora of natural language and code, they can generate HTNs, plans, explanations, and even candidate operators from only a few examples, demonstrating a form of one-shot or few-shot generalization. However, while their outputs often appear plausible and contextually appropriate, LLMs lack the transparency, consistency, and grounding that characterize symbolic systems. An LLM may produce a valid-looking plan without a verifiable reasoning process, leading to errors or "hallucinations" that undermine their use in contexts that demand reliability. These issues limit their use as standalone cognitive agents.

To address these limitations, we introduce **L.E.A.R.N.** (**Learning by Example Authoring and Reasoning Network**), a human-in-the-loop, hybrid cognitive agent architecture that integrates the generative power of LLMs with an HTN learning model to interactively learn task hierarchies for procedural problems. L.E.A.R.N. employs the *authoring-by-tutoring* paradigm used to create expert models for intelligent tutors where agents learn from problem demonstrations and feedback received from users (Matsuda et al., 2008; Maclellan et al., 2016; Weitekamp et al., 2020; Smith et al., 2024). However, instead of relying on the user to provide all demonstrations, L.E.A.R.N. uses the LLM to generate complete problem examples (solution traces) that substitute for or augment human demonstrations. These LLM-generated demonstrations are designed to be diverse and representative of the problem space, thereby reducing the dependence on users to consider and provide "representative enough" training examples that help the expert model generalize. When the existing operator set is insufficient, the LLM can also propose and implement new primitive operators in real time, enabling the agent to extend its operator library dynamically. Moreover, because the LLM proposes specific operators and inputs during solution-trace generation, the need for the learning model to perform operator search is greatly reduced. Together, these capabilities allow L.E.A.R.N. to generalize to a correct model from fewer examples, and reduce the authoring burden on humans while maintaining the transparency and reliability of symbolic reasoning systems. We demonstrate these capabilities in the domain of intelligent tutor authoring, where expert models must capture a diverse range of valid problem-solving strategies with minimal human input.

```
Head:
  (multiply, ?x, ?y, ?res)
Preconditions:
  (and
    (is_number, ?x)
    (is_number, ?y))
Effects:
  (and
    (value, ?res, (* ?x ?y)))
```

Figure 1: An example operator definition to multiply two numbers. Here, effects indicate either derived information supplied back to the planner (a) or actions to take in the environment (b). *?res* parameters are variables bound as the result of the execution of an operator and can be passed to subsequent operators as parameters.

2. Background

2.1 Hierarchical Task Networks

2.1.1 Definition

Hierarchical Task Networks (HTNs) represent procedural knowledge in terms of tasks, methods, and operators organized as an AND-OR hierarchy (Nau et al., 2003; Langley, 2025) . Each task may be associated with multiple methods, which are alternative strategies for accomplishing that task. Selecting one of these methods corresponds to the OR choice. Each method, in turn, specifies a set of subtasks that must all be completed successfully for the method to succeed, forming the AND structure. Subtasks can then be recursively decomposed further until they eventually resolve into primitive operators, which are the basic actions a planning agent can execute.

More formally, an HTN domain $\mathcal{H}$ is defined as a tuple:

$$\mathcal{H} = (\mathcal{T}, \mathcal{O}, \mathcal{M}, \mathcal{A}),$$

where $\mathcal{T}$ is the set of tasks, $\mathcal{O}$ is the set of primitive operators, $\mathcal{M}$ is the set of decomposition methods, and $\mathcal{A}$ is a set of axioms or logical constraints that define domain facts and inference rules.

Each *operator* $o \in \mathcal{O}$ represents a primitive action and is specified as

$$o = (\text{head}(o), \text{pre}(o), \text{eff}(o)),$$

where $\text{head}(o)$ defines the task and its input arguments, $\text{pre}(o)$ defines the conditions under which the operator applies, and $\text{eff}(o)$ specifies the operator's effects. In classical planning, these effects typically include both additive and subtractive state updates to a symbolic world model. In contrast, the L.E.A.R.N. architecture employs *reactive, or online, planning* (Georgeff & Lansky, 1987; Langley et al., 1994), in which operators are executed directly in the external environment and the resulting state is updated by querying the world itself rather than applying a predefined model. Thus, operator effects in this formulation are either actions taken in the environment or

information derived and passed to the planner to inform future decisions. This distinction allows the planner to act and adapt in real time without maintaining a complete internal action model of the world.

A *method* $m \in \mathcal{M}$ defines how to decompose a compound task into a partially ordered set of subtasks. Each method is expressed as

$$m = (\text{head}(m), \text{pre}(m), \text{sub}(m)),$$

where $\text{head}(m)$ denotes the compound task to be achieved and its input arguments, $\text{pre}(m)$ specifies the conditions under which the method applies, and $\text{sub}(m) = \langle t_1, t_2, \ldots, t_n \rangle$ represents an ordered or partially ordered set of subtasks to accomplish $\text{head}(m)$. Planning proceeds by recursive task decomposition until only primitive actions remain.

2.1.2 Performance Component

The HTN learning architecture includes a *performance component*, which functions as a reactive planner (or interpreter) that searches the space of possible decompositions $T_0 \Rightarrow^* \pi$ while executing in the real world.

Formally, a *planning problem* is defined as the tuple $P = (\mathcal{H}, T_0, s_0)$, where $\mathcal{H}$ is the domain, T_0 is the initial set of tasks, and s_0 is the initial world state. The objective of the performance component is to produce a sequence of primitive actions

$$\pi = \langle a_1, a_2, \ldots, a_k \rangle$$

such that each action a_i is executed in the environment, yielding an updated observed state s_{i+1} derived from the world after a_i completes. The overall process reflects a continuous cycle of planning, acting, and perceiving rather than an offline search over static symbolic states.

Task decomposition proceeds according to the relation $\Rightarrow$, which defines how a task $t \in T$ can be refined using a method m, resulting in a new set of tasks T':

$$t \Rightarrow_m t',$$

and

$$T' = \text{sub}(m) \cup (T - \{t\}),$$

where $\text{pre}(m)$ holds in the currently observed state s. This expresses that decomposing task t replaces it with its subtasks from method m, which must be solved before completing the remaining tasks. Planning continues recursively, alternating between task decomposition and operator execution. If an operator fails or a method becomes inapplicable due to a state change, the system backtracks to search for an alternative method that satisfies the current state. If no applicable method is found, the system terminates.

2.1.3 Learning Algorithm

The *learning component* is responsible for acquiring and refining the elements of $\mathcal{M}$ and $\mathcal{O}$ from data, demonstrations, and user feedback. Learning may infer new task decompositions that explain

observed behavior or *precondition refinement*, in which the applicability of existing methods is adjusted based on success or failure outcomes.

Let $\mathcal{L}$ denote the learning function that updates the HTN given data D and performance feedback f:

$$\mathcal{H}' = \mathcal{L}(\mathcal{H}, D, f).$$

When applied iteratively, $\mathcal{L}$ yields increasingly complete and general task hierarchies.

Initially, an HTN may contain only primitive operators without corresponding methods for decomposing higher-level tasks. When the performance component encounters an unsolvable task, it requests a demonstration or an example of the correct solution. The learning algorithm then attempts to *explain* the demonstrated behavior by identifying a sequence of existing operators that reproduce the observed outcome from the current state s_i. If a valid explanation is found, the sequence is encoded as a new method, with its operators forming the ordered subtasks and the initial state s_i serving as the method's preconditions. If no explanation can be constructed, the system creates a specialized "memorized" method that replays the demonstrated behavior only in the specific state in which it was observed. This ensures progress even in sparse or novel domains.

Over multiple demonstrations, the learning component generalizes its acquired knowledge. When two or more methods achieve the same task using the same sequence of subtasks but under slightly different conditions, the system merges them by generalizing their preconditions. This allows the agent to extend its applicability to broader contexts.

The system also incorporates user feedback into the learning loop. During learning, the agent may query the user to verify whether a selected method or action is appropriate. Positive feedback reinforces the method's preconditions as valid, whereas negative feedback causes the learning component to refine (or specialize) them to exclude the current context. In cases where no alternative methods apply, the system requests a new demonstration, which results in a new method for that individual behavior. This combination of demonstration and corrective feedback enables the HTN learning architecture to incrementally acquire robust procedural knowledge and refine method applicability over time.

2.2 Large Language Models

Large Language Models (LLMs) have emerged as powerful, general-purpose reasoning and generation systems. Trained on vast corpora of natural language and code, they learn distributed representations that capture statistical and semantic regularities across domains (Zhao et al., 2023). When prompted with natural-language instructions or examples, an LLM can generate complete HTN-like structures, plans, and explanations that resemble human reasoning, often without any task-specific programming. Yet despite their expressive capacity, LLMs remain limited by their unreliability. They can produce plausible but incorrect outputs ("hallucinations") and lack mechanisms for verifying whether a proposed plan or explanation is correct or executable in a given environment (Valmeekam et al., 2022).

These limitations are especially crucial in educational authoring contexts. When constructing intelligent tutors, for instance, generated examples or explanations must adhere to pedagogical constraints and domain logic. An unverifiable problem solution can mislead both instructors and learn-

ers which would undermine the effectiveness of the system. Purely generative approaches therefore require complementary structures that can interpret, validate, and constrain model outputs within a formal reasoning framework. This ensures that generated content remains aligned with instructional intent.

These properties suggest that LLMs are most effective when embedded as *components* within larger cognitive or learning architectures rather than as standalone agents. Their generative capabilities make them valuable for proposing solution strategies, instructional examples, or candidate operators – especially when symbolic knowledge is incomplete. Conversely, symbolic frameworks such as HTNs provide the structure, consistency, and interpretability that LLMs lack. Together, they enable the construction of cognitive agents that reason fluidly yet remain reliable and verifiable.

2.3 Expert Model Authoring for Intelligent Tutors

While expert models traditionally needed to be hand-authored by programmers, many state-of-the-art expert model authoring systems have used teachable AI (TAI) approaches to frame authoring as a process of *teaching* rather than programming. In systems such as SimStudent (Matsuda et al., 2008), Apprentice Learner (AL) (Maclellan et al., 2016), and their successors AL+ and AI2T (Weitekamp et al., 2024), authors train an artificial agent by demonstrating steps and giving yes/no feedback, mirroring the familiar pedagogical act of instructing a human student. As the teacher provides examples, the system induces procedural rules, forming an expert model capable of solving similar problems autonomously. This framework directly addresses the knowledge-engineering bottleneck of traditional authoring by allowing tutors to learn from the same instructional interactions teachers already perform.

Despite their promise, existing teachable AI-based systems face practical limitations that have constrained their adoption beyond research settings. For example, many systems remain dependent on the quality, variety, and representativeness of the demonstrations they receive. When users provide too few or too narrow examples, the resulting expert models often fail to generalize to a correct model. Next, these systems typically depend on the human teacher to supply every example of a valid problem-solving path, which can make the authoring process slow and labor-intensive. Finally, teachers often lack visibility into what the system has actually learned or when the model has converged, leading to uncertainty about its correctness and completeness (Smith et al., 2024; Weitekamp et al., 2020). Consequently, while these frameworks reduce the technical barrier to expert model creation, they do not yet provide the appropriate level of user scaffolding required to efficiently author these models.

3. The L.E.A.R.N. Agent

L.E.A.R.N. (**L**earning by **E**xample **A**uthoring and **R**easoning **N**etwork) is a hybrid cognitive agent that combines a hierarchical task network (HTN) learning and reasoning system with a large language model (LLM) to enable the efficient creation of expert models for intelligent tutors. The agent follows the *authoring-by-tutoring* paradigm (Gupta & Maclellan, 2021; Matsuda et al., 2008), in which users teach the system by demonstrating problem-solving steps and providing corrective feedback – mirroring how an instructor might guide a human student. Through these interactions,

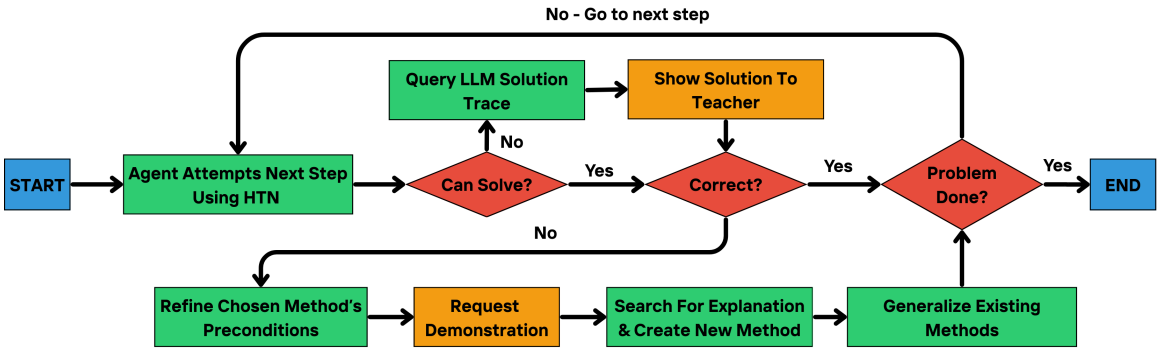


Figure 2: The L.E.A.R.N. learning algorithm.

L.E.A.R.N. incrementally acquires procedural and hierarchical problem-solving knowledge, gradually constructing an expert model that generalizes across related problems.

3.1 Core Algorithm

Figure 2 illustrates the complete expert-model induction process carried out by the L.E.A.R.N. agent. The cycle begins when the user initializes the environment by supplying the initial tutor state. When the HTN cannot be used to decompose or solve the current problem step, the agent encodes the tutor state and its existing operator library into a structured prompt for the LLM. The LLM responds with a proposed *solution trace* – a stepwise sequence of predicted predicted problem step values, the fields whose values serve as inputs to each output step, and the corresponding operator sequences that map inputs to outputs.

The agent processes this trace one step at a time. For each step, it first attempts to use its HTN to generate the correct value; if unsuccessful, it substitutes the LLM-proposed output into the tutor interface and asks the user to verify it. The user can confirm or modify any part of the proposed step, including the inputs, operators, or resulting value. When both the operators and values are correct, the step is accepted; the input fields, output value, and operator sequence are used to create a method for that step and the observed tutor state is recorded as a *positive example* for that method. If the user determines the value is correct but the input fields or operator sequence are incorrect, the user can adjust them. When the value itself is wrong, the agent requests a demonstration from the user and invokes the HTN learner to search for a sequence of known operators that can reproduce the demonstrated value and construct a new method.

When a user-corrected demonstration occurs, the remainder of the LLM's proposed trace is re-evaluated for consistency. Because each step in the trace contains explicit inputs and outputs, the agent can simulate the solution from problem start and identify any mismatches. If inconsistencies are detected, the LLM is prompted to regenerate the remaining steps beginning from the point of failure, ensuring that the trace remains coherent with the previously verified steps of the tutor.

If, during trace generation, the LLM cannot identify an appropriate operator from the library, it proposes a new one. Each proposed operator includes a name and arguments (defining its head), a textual description, and a set of preconditions. Because operators in L.E.A.R.N. are realized as

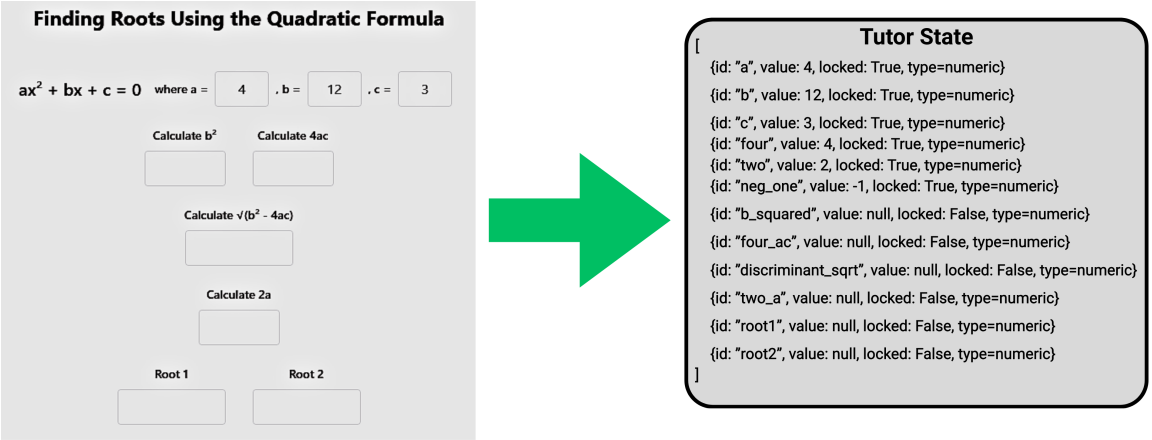


Figure 3: The Quadratic Formula tutor converted into a state representation for planning. This is the same state observed by the LLM when it is queried for a solution trace.

transformation functions, the LLM also produces an executable Python function implementing the operation. This function is tested on an unbiased, LLM-generated set of ten input–output pairs. Operators that pass all tests are accepted and permanently added to the HTN's library. Those that fail trigger a retry in which the LLM is prompted to generate a more general function. If the second attempt also fails, the agent reverts to the HTN learner's native explanation mechanism.

The interactive cycle continues until the entire problem is solved. Users can then initiate a new problem for the L.E.A.R.N. agent to solve until it has converged to a correct model.

4. Experimental Evaluation

4.1 Purpose and Scope

The experimental evaluation examines both the technical performance and authoring effort of the L.E.A.R.N. architecture. Specifically, it evaluates whether the system can (1) learn accurate and generalized models from limited user demonstrations and (2) reduce the authoring effort required by users to construct an expert model. To assess these goals, we conducted experiments across multiple model configurations and task conditions. The L.E.A.R.N. agent was compared against HTN-only and LLM-only baselines, along with a variant that removed operator induction. We employed a *simulated user* to serve as a domain expert capable of providing demonstrations and binary feedback on problem step correctness. This setup enabled a controlled evaluation of how different system configurations influence learning efficiency and overall authoring cost.

4.2 Task 1: Quadratic Formula Expert Model

The first evaluation task examined the agent's ability to learn and generalize procedural reasoning within a structured mathematical domain. The task involved constructing an expert model for solving problems that require applying the quadratic formula to find the roots of a polynomial equation.

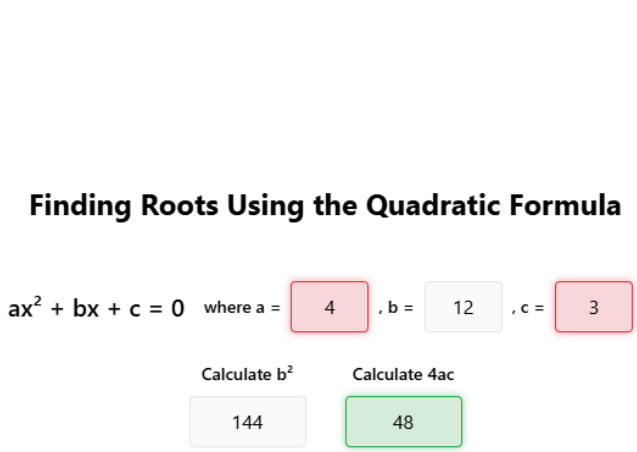
```
Head:
    (calculate_4ac,)
Preconditions:
    (and
      (value, a, ?v0)
      (value, c, ?v1)
      (value, four_constant, 4))
Subtasks:
    (Ordered
      (multiply ?v0 ?v0 ?res0)
      (multiply ?res0 4 ?res1)
      (input_value ?res1 field_4ac))
```

Figure 4: An HTN method for the `calculate_4ac` step of the quadratic formula tutor. This method has no arguments; instead, the values of `a` and `b` are bound during precondition matching. Note that this representation includes a "result" variable in some operator. This variable is bound as the result of the operator applied to all arguments prior to result term and then can be passed as an argument to subsequent subtasks in the sequence.

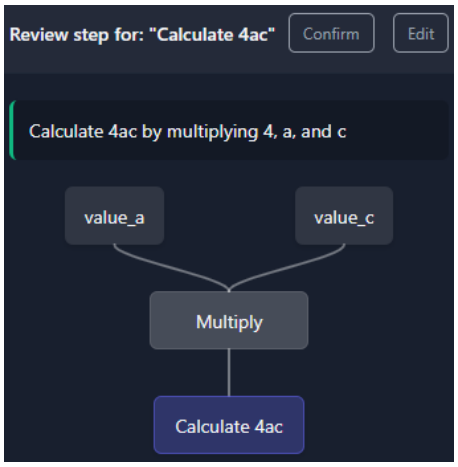
This domain was selected because it offers a clear decomposition of subtasks – such as computing the discriminant, taking the square root, and performing division – that can each be represented as HTN methods. It therefore provides a well-defined setting for assessing how the agent acquires, verifies, and reuses operators to form generalized solution methods.

4.2.1 Methodology

To assess the trade-offs between architectures, we evaluated six distinct agent conditions. The first three conditions consisted of HTN-based agents, each employing a different algorithm for learning method preconditions from demonstration and feedback. These included a *decision-tree learner* that induces rules from features and relations, a *version-space split–merge learner* (Hong & Tseng, 1999) that maintains and refines sets of candidate conjunctive rules, and a *unification-based learner* that generalizes and specializes preconditions through (anti-)unification (Plotkin, 1970; Bulychev et al., 2010; Jung et al., 2020). The fourth condition was the hybrid *L.E.A.R.N.* architecture. For this task, operator induction was disabled to ensure comparability with other models; the LLM's role was limited to proposing solution traces and identifying appropriate inputs for the existing operator set. Finally, two LLM-only baselines were evaluated: (1) a *zero-shot* model that generated a complete solution for each problem independently, and (2) an *in-context* model that incrementally augmented its prompt with solved examples after each training epoch, allowing it to learn through context accumulation. These baselines followed the evaluation setup used in the TutorGym framework (Weitekamp et al., 2025). To automate evaluation, we implemented a "simulated user," capable of solving the quadratic formula and providing demonstrations or binary correctness feedback when queried.



(a) The second step of the Quadratic Formula tutor (solving for $4ac$) being attempted by L.E.A.R.N. The green highlight shows the next step; the red shows the determined input.



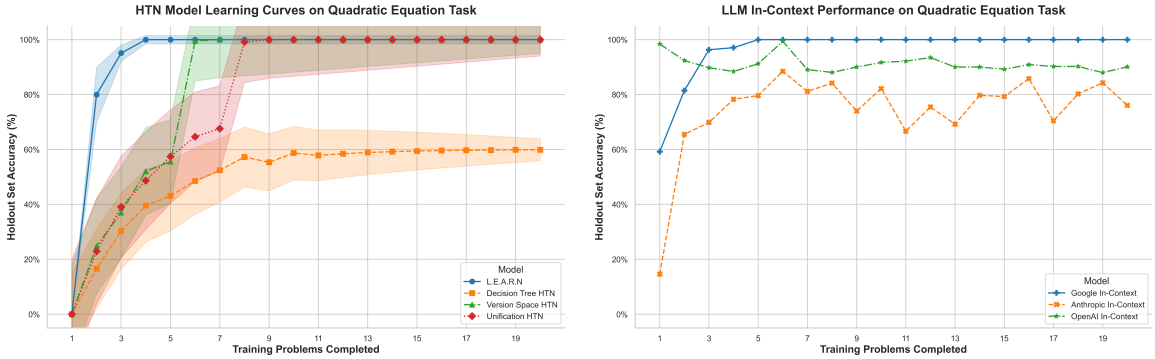
(b) The confirmation screen showing the input/operator/output relationship generated by L.E.A.R.N's LLM module for the "Calculate $4ac$" step.

Figure 5: L.E.A.R.N.'s authoring interaction for the "Calculate $4ac$" step: (a) the agent attempting to solve a step and (b) solution confirmation screen. Here, the two multiplication steps are condensed into one for display purposes.

To measure performance, we employed a set of metrics tailored to each model type. For the LLM zero-shot baseline, since each training epoch is independent and the model does not "learn", we computed a per-problem completion rate, defined as the number of correct steps divided by the total number of steps in the problem for 20 total problems. All other systems were trained on 20 problems and evaluated after every training instance on a 20-problem holdout set to produce accuracy learning curves. For all conditions except the zero-shot LLM, accuracy was measured as the number of evaluation set problems completed successfully – i.e., problems solved without any incorrect intermediate steps. For the LLM-only baselines, we evaluated three models – Google's *Gemini 2.5 Pro* (temperature = 1, top_p = 0.95), OpenAI's *GPT-5* (version 2025-08-07), and Anthropic's *Claude Sonnet 4* (temperature = 1). The L.E.A.R.N. agent used *Claude Sonnet 4* exclusively as its generative component. Each model received identical problem and operator definitions in their prompts.

4.2.2 Results

The L.E.A.R.N. agent was able to achieve faster convergence than both the HTN-only and LLM-only baselines. As shown in Figure 6a, the L.E.A.R.N. agent reached 100% accuracy on the holdout set after just four training problems, outperforming all three HTN-only baselines, which required six to eight problems to achieve comparable performance. Among the HTN-only conditions, the version-space and unification learners showed comparable performance, while the decision-tree learner performed poorly due to its inability to generalize to problem states not observed during



(a) Learning curves for HTN-based models on the quadratic equation solving task. The shaded regions represent the 95% confidence interval.

(b) Performance of in-context LLM models on the quadratic equation solving task.

Figure 6: Comparison of (a) HTN-based model learning dynamics and (b) LLM-based performance on the quadratic equation task.

training. For example, when learning the method for computing b^2, the decision-tree learner can fail to generalize if it encounters negative examples during training that cause it to form overly specific branching conditions so that during evaluation, even slightly different states fall outside the learned ranges and the method is incorrectly deemed inapplicable.

A key factor contributing to L.E.A.R.N.'s performance is its ability to bypass the costly operator search process that constrains HTN-only learners. In L.E.A.R.N., the LLM directly proposes the relevant operators for each demonstrated step, selecting them based on name, description, and arity. This allows the agent to immediately associate each step with an appropriate operator sequence rather than exploring a large combinatorial search space. In contrast, HTN-only agents must identify the correct operator sequence through search, which can lead to errors when multiple operators appear syntactically valid for different problems. For instance, when solving the b^2 step, the search procedure might return either $(multiply, ?v0, ?v0)$ – the correct operation – or $(add, ?v0, ?v0)$ in special cases where (b = 0) or (b = 2). This issue can be exacerbated for longer operator sequences. This ambiguity can increase the overall number of methods in the HTN and contribute to slow generalization, whereas L.E.A.R.N.'s direct operator matching enables faster convergence.

The LLM-only baselines varied across models and training conditions (Figure **??**). The Claude and ChatGPT in-context models improved marginally with additional examples but remained inconsistent, with accuracy oscillating throughout training. Google's *Gemini 2.5 Pro* achieved the highest overall performance, reaching 100% accuracy after about 5 prompt examples. While Gemini's results are promising and comparable to L.E.A.R.N.'s, it is unclear whether it can achieve similar performance on more complex tasks.

Overall, the results support our central claim that combining LLM-generated solutions with HTN-based learning yields a system that learns accurate and generalizable expert models from fewer examples. This combination enables L.E.A.R.N. to outperform both symbolic and generative systems alone, offering a data-efficient approach to expert-model authoring.

4.3 Task 2: Quadratic Formula Without Predefined Operators

A central claim of the L.E.A.R.N. architecture is that it can overcome the brittleness of fixed symbolic systems by creating new primitive operators when existing knowledge is insufficient. Whereas Task 1 demonstrated that the agent can efficiently acquire procedural knowledge when provided with a complete operator library, this experiment evaluates its capacity to learn the same domain *from scratch*. All arithmetic primitives were removed, leaving the agent with an empty operator set and requiring it to induce every operator through its language-model module. This task tests whether the architecture's operator-generation and verification cycle can create a functional symbolic model in the absence of required prior knowledge.

4.3.1 Methodology

The experimental procedure mirrored that of Task 1 in problem distribution, simulated-expert behavior, and evaluation setup. However, during each problem the LLM served as the exclusive source of operators. Whenever the agent encountered an unsolvable step, it prompted the LLM with the current task description, argument signatures, and partial operator list. The LLM then proposed a new operator defined by (1) a symbolic head (name and arity), (2) a natural-language description, and (3) an executable function. Each candidate was verified on an independently generated test set of 10 input-output pairs. Only operators that achieved 100% pass rate were accepted and permanently added to the library; failed candidates triggered a re-prompt for regeneration. Each accepted operator was immediately applied to complete the pending method, and its behavior was subsequently validated through user verification just as in Task 1.

To capture the cost and quality of induction, four quantitative metrics were computed:

1. **(1) New Operators Proposed**: of unique operator definitions attempted per run;
2. **(2) Operator Acceptance Rate**: fraction of operators passing all tests;
3. **(3) Retries per Operator**: average of LLM regenerations required for acceptance; and
4. **(4) Model Accuracy**: the model accuracy as defined in Task 1.

4.3.2 Results

Figure 7 compares L.E.A.R.N.'s learning curves in the operator-induction condition and the predefined-operator condition (averaged over five runs with 95% confidence intervals). The agent successfully acquired all skills required to solve the quadratic-formula task entirely through self-generated operators. Across runs it produced 8 ± 1 unique operators, including standard arithmetic functions (add, subtract, divide) and several variants of multiplication and negation. For example, to compute the $4ac$ term, the agent typically learned a two-operator sequence [$(multiply, 4,\ ?a,\ ?res_0)$, $(multiply, ?res_0,\ ?c,\ ?res_1)$], where "?res" variables are the results of applying the operator to its inputs. However, in several runs the LLM generalized this pattern into a single three-parameter operator, $(multiply, 4, ?a, ?c, ?res)$, which collapsed both multiplications into one function. On another occasion, it produced a specialized function multiply_by_neg_one, which multiplies a value by 1 instead of calling the generic **multiply** operator with a constant argument (i.e., $(multiply, ?b,\ -1, ?res)$).

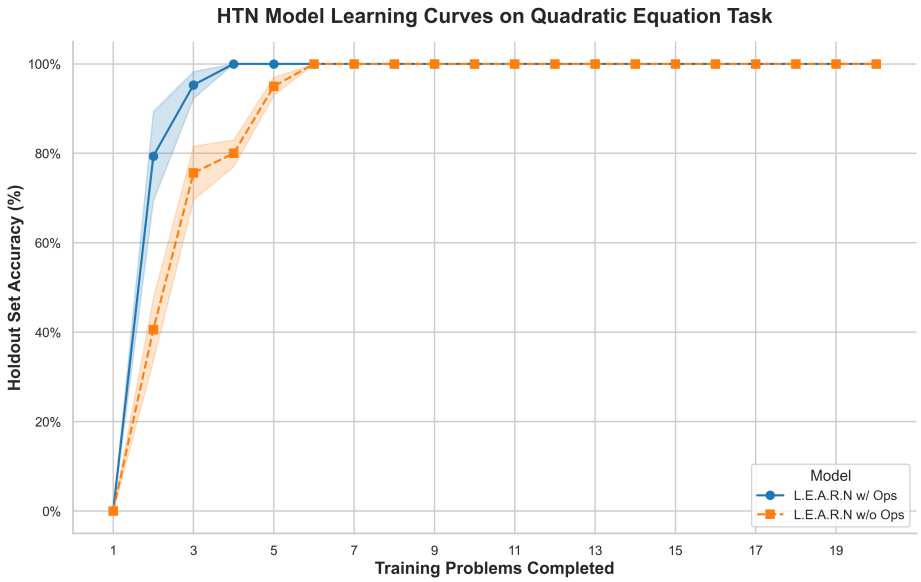


Figure 7: Performance of L.E.A.R.N. on Quadratic Formula task with predefined operators and without predefined operators (operator induction).

Although functionally redundant, this variant proved useful for expressing sign inversion as an explicit conceptual step within the HTN.

All operators eventually passed their verification tests (acceptance rate = 1.00), with a mean of 0.33 retries per operator. Most failures occurred when the LLM initially attempted to generate an overly broad function that combined multiple arithmetic operations (e.g., computing the entire square-root term in one step). These did not pass the initial verification tests, but were corrected on subsequent prompts.

Learning converged more slowly than in Task 1, reaching 100% accuracy after six training problems (+ 2 relative to baseline). This delay resulted from run-to-run variability in the sequence of induced operators, which led to inconsistent method subtask sequences and temporarily hindered precondition generalization. Nevertheless, once the operator set stabilized, the HTN quickly generalized across problems and matched the final accuracy and variance of the predefined-operator agent.

5. Discussion

Our results show that L.E.A.R.N. represents a practical step toward scalable expert-model authoring for intelligent tutoring systems. The verification dialogue – confirming or correcting candidate steps – mirrors the natural pedagogical act of reviewing a student's work without having to produce content oneself , which could lower the time barrier of other approaches (Maclellan et al., 2016; Weitekamp et al., 2024). Additionally, the agent's ability to converge to accurate expert models

from only a handful of demonstrations suggests that this approach can substantially reduce the technical burden of tutor development that requires users to think like programmers and AI experts.

Beyond tutor authoring, the experiments illustrate how real-time operator generation can extend the boundaries of symbolic reasoning. In traditional planning systems, the operator library defines the limits of what the agent can learn or execute; unmodeled actions halt progress until a human programmer intervenes. L.E.A.R.N. overcomes this limitation by allowing its language-model component to hypothesize, implement, and verify new operators on demand. Each newly induced operator expands the agent's representational vocabulary, enabling it to generalize to unseen domains where no predefined primitives exist. In principle, this mechanism supports transfer learning across problem types. Trivially, operators such as $(multiply, ?v0, v1)$ can apply across a variety of math-based domains, but other more complex operators, for instance an operator that implements an A* search over a grid environment, can be applied in various scenarios. This ultimately allows the architecture to incrementally build a cross-domain skill library.

Another aspect of the operator induction capability we might note is the idea of **operator abstraction**. During learning, L.E.A.R.N. occasionally merged multi-step procedures into higher-level functions (e.g., collapsing two multiplication steps into a single three-argument operator). Such behavior parallels human cognitive compression, where repeated patterns are reified into reusable concepts. This dynamic abstraction calls into note the balance between granularity and efficiency. Fine-grained operators are more flexible but can greatly expand the combinatorial search space in scenarios where search is required Coarse-grained operators can accelerate reasoning but the agent risks losing expressive power to construct complex skills. While L.E.A.R.N. is not currently designed to address this, future work might explore this area further.

A broader implication of this work is that hybrid neuro-symbolic systems like L.E.A.R.N. help advance the idea of adaptive cognitive architectures that learn not just what to do, but how to represent and refine their own knowledge structures over time. By tightly coupling generative models with interpretable reasoning frameworks, such systems move closer to the flexibility and self-extensibility characteristic of human cognition, i.e. the ability to invent new primitives when existing ones fail, reuse them across contexts, and progressively abstract recurring patterns into higher-order concepts. In the context of intelligent tutoring, this means authoring tools that grow alongside their users – learning and adapting in tandem with the educators who teach them.

5.1 Limitations and Opportunities

While the present work establishes the feasibility of real-time operator induction for expert-model authoring, several limitations should be acknowledged. First, the evaluation focused on a single, well-structured mathematical domain where the mapping between inputs, operators, and outputs is deterministic. Although this setting provides clear ground truth for assessing the system, it does not capture the variability and ambiguity present in more open-ended instructional domains. Extending L.E.A.R.N. to settings such as scientific reasoning, procedural writing, or clinical decision-making will require mechanisms for handling non-deterministic operators, partial observability, and noisy verification signals.

Second, the study employed a simulated expert to standardize feedback and ensure reproducibility. While this approach isolates system performance, it omits the complexities of human interac-

tion, including variability and errors in teacher feedback and the cognitive load of sustained verification. A important next step is to conduct human-in-the-loop studies that examine how teachers perceive, trust, and guide the system's reasoning process. Such studies would also help identify where additional scaffolding, visualization, or transparency mechanisms are needed to make the verification dialogue efficient and intuitive.

Third, the operator generation process, though reliable in constrained domains, remains dependent on functional testing that may not scale to operators with rich semantic structure. As the complexity of induced functions grows, more sophisticated verification pipelines will be needed to maintain reliability. Similarly, while the architecture occasionally exhibits useful operator abstraction, it lacks an explicit mechanism for regulating the representational granularity of the operators. Future work should explore adaptive strategies for deciding when to generalize, merge, or specialize operators to balance expressive power with efficiency.

Addressing these limitations will be essential for scaling L.E.A.R.N. beyond controlled evaluations toward robust, interactive cognitive systems that co-evolve with their human teachers.

6. Related Works

The L.E.A.R.N. architecture is situated at the intersection of several research areas, including Interactive Task Learning, Teachable AI, Hierarchical Task Network learning, and the use of Large Language Models as components in cognitive systems. Our approach builds upon the rich history of symbolic systems for task learning while leveraging recent advances in generative AI to overcome the brittleness and authoring challenges inherent in purely symbolic architectures.

A primary goal of Interactive Task Learning (Laird et al., 2017) is to enable agents to learn new tasks through natural interaction with a human instructor, rather than through traditional programming. The aim is to create systems that can acquire new skills and knowledge through dialogue, demonstration, and other intuitive forms of communication. Systems like PLOW (Allen et al. (2007)) and SUGILITE (Li et al. (2017)) are examples of this approach, learning procedural knowledge by observing and interacting with a user in a graphical user interface. However, a significant limitation of these systems is that they are ultimately constrained by their underlying architecture and pre-existing capabilities. While they can learn new procedures, these procedures must be composed of primitive actions and concepts that are already known to the system. If a task requires a capability that is not part of the agent's initial set of skills, the agent cannot learn the task without expert intervention to extend its core functionalities, creating a significant knowledge engineering bottleneck.

To address this limitation, recent work has focused on hybrid systems that integrate symbolic planners with Large Language Models. One area of research uses LLMs as a source of commonsense or procedural knowledge to fill in gaps in an agent's knowledge base. For instance, some researchers explore the use of LLMs as "zero-shot planners" (Huang et al., 2022) demonstrating that LLMs can decompose high-level tasks into plausible plans without any task-specific training. However, they also note that plans generated by LLMs are often not directly "executable" by an agent because they may not map to the agent's available actions or may lack necessary commonsense steps (e.g., opening a fridge before grabbing the milk). This necessitates a translation or

grounding step to make the LLM's output actionable. VAL (Verbal Apprentice Learner) (Lawley & Maclellan, 2024) contains GPT modules that leverage an LLM's linguistic capabilities to parse natural language instructions from a user into symbolic predicate-argument structures and can learn HTN structures from these instructions. In contrast to systems like (Huang et al., 2022), VAL uses it GPT modules to map user provided actions to known system actions, making them ultimately executable. However, the system is limited to known operators and may not always be able to express user intentions with its pre-authored capabilities.

Several systems have been developed to perform grounding within a planning or learning context. ChatHTN (Munoz-Avila et al. (2025)) is a hybrid HTN planner that queries an LLM for a plausible task decomposition when it lacks a method to decompose a given task. The LLM-generated plan is then verified for soundness, i.e. ChatHTN ensures that any plan it generates is correct with respect to the domain model. The STARS (Search Tree, Analyze, Repair, and Selection) framework (Kirk et al., 2024) shares important conceptual ground with L.E.A.R.N.. Like L.E.A.R.N., STARS embeds an LLM inside a symbolic agent that evaluates, constrains, and repairs the model's outputs to ensure grounding, interpretability, and contextual relevance. Whereas STARS focuses on extracting viable *goal knowledge* for embodied agents through internal analysis and targeted re-prompting, L.E.A.R.N. extends this principle to *prior knowledge knowledge acquisition* by inducing, testing, and abstracting new operators in real time.

Conceptually, L.E.A.R.N. might be seen as complementary to systems such as ChatHTN and STARS. They ensure that generated decompositions are viable and context-sensitive before execution, while L.E.A.R.N. ensures that generated actions are executable and generalizable within the task structures. Together, these frameworks illustrate a growing trend toward hybrid architectures where the generative capacity of LLMs is embedded within interpretive, self-correcting cognitive systems that maintain symbolic coherence and human-aligned reasoning.

7. Conclusion and Future Work

This paper introduced L.E.A.R.N, a hybrid cognitive architecture that integrates Hierarchical Task Networks planning with the generative flexibility of Large Language Models. Our experimental evaluation demonstrates that this approach addresses some of the key weaknesses of each model in isolation. The L.E.A.R.N. agent learned the procedure for solving quadratic equations faster and with significantly fewer demonstrations than HTN-only baselines, achieving high accuracy after just four training problems. This performance is partly due to the LLM introducing varied but valid solution strategies, allowing the HTN to generalize more quickly.

L.E.A.R.N. overcomes the inherent limitations of fixed symbolic systems by dynamically inducing new primitive operators when its existing knowledge is insufficient. Our second experiment showed the system could learn the quadratic formula task from scratch, generating all necessary arithmetic operators on the fly. This capability, combined with the user-verification framework, ensures that the system remains reliable and transparent and avoids the "hallucinations" that limit purely generative models. By shifting the human's role from a demonstrator to a verifier, L.E.A.R.N. can reduce the authoring burden, representing a meaningful step toward more flexible and accessible cognitive systems, particularly for authoring tools in domains like intelligent tutoring.

References

Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., & Taysom, W. (2007). Plow: A collaborative task learning agent.

Bulychev, P. E., Kostylev, E. V., & Zakharov, V. A. (2010). Anti-unification algorithms and their applications in program analysis. *Perspectives of Systems Informatics* (pp. 413–423). Berlin, Heidelberg: Springer Berlin Heidelberg.

Georgeff, M. P., & Lansky, A. L. (1987). Reactive reasoning and planning. *AAAI* (pp. 677–682).

Gupta, A., & Maclellan, C. J. (2021). Designing Teachable Systems for Intelligent Tutor Authoring. *AAAI2021 Spring Symposium on Artificial Intelligence for K-12 Education*.

Hong, T.-P., & Tseng, S.-S. (1999). Splitting and merging version spaces to beam disjunctive concepts. *IEEE Transactions on Knowledge and Data Engineering*, *11*, 813–815.

Huang, W., Abbeel, P., Pathak, D., & Mordatch, I. (2022). Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *International conference on machine learning* (pp. 9118–9147). PMLR.

Jung, J. C., Lutz, C., & Wolter, F. (2020). Least general generalizations in description logic: Verification and existence. *Proceedings of the AAAI Conference on Artificial Intelligence*, *34*, 2854–2861. From `https://ojs.aaai.org/index.php/AAAI/article/view/5675`.

Kirk, J. R., Wray, R. E., Lindes, P., & Laird, J. E. (2024). Improving knowledge extraction from llms for task learning through agent analysis. *Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 18390–18398).

Laird, J. E., et al. (2017). Interactive task learning. *IEEE Intelligent Systems*, *32*, 6–21.

Langley, P. (2025). Learning hierarchical task knowledge for planning. *Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 28652–28656).

Langley, P., Iba, W., & Shrager, J. (1994). Reactive and automatic behavior in plan execution. *AIPS* (pp. 299–304).

Lawley, L., & Maclellan, C. (2024). Val: Interactive task learning with gpt dialog parsing. *Proceedings of the CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery. From `https://doi.org/10.1145/3613904.3641915`.

Li, T. J.-J., Azaria, A., & Myers, B. A. (2017). Sugilite: creating multimodal smartphone automation by demonstration. *Proceedings of the 2017 CHI conference on human factors in computing systems* (pp. 6038–6049).

Maclellan, C. J., Harpstead, E., Patel, R., & Koedinger, K. R. (2016). The apprentice learner architecture: Closing the loop between learning theory and educational data. *International Educational Data Mining Society*.

Matsuda, N., Cohen, W. W., Sewall, J., Lacerda, G., & Koedinger, K. (2008). Simstudent : Building an intelligent tutoring system by tutoring a synthetic student. *NA*. From `https://api.semanticscholar.org/CorpusID:2331037`.

Munoz-Avila, H., Aha, D. W., & Rizzo, P. (2025). Chathtn: Interleaving approximate (llm) and symbolic htn planning. From https://arxiv.org/abs/2505.11814.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). Shop2: An htn planning system. *Journal of artificial intelligence research*, *20*, 379–404.

Nejati, N., Langley, P., & Konik, T. (2006). Learning hierarchical task networks by observation. *Proceedings of the 23rd international conference on Machine learning* (pp. 665–672).

Plotkin, G. D. (1970). A note on inductive generalization. *Machine intelligence*, *5*, 153–163.

Smith, G., Gupta, A., & MacLellan, C. (2024). Apprentice tutor builder: A platform for users to create and personalize intelligent tutors. *arXiv preprint arXiv:2404.07883*.

Valmeekam, K., Olmo, A., Sreedharan, S., & Kambhampati, S. (2022). Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Weitekamp, D., Harpstead, E., & Koedinger, K. (2024). Ai2t: Building trustable ai tutors by interactively teaching a self-aware learning agent. *arXiv preprint arXiv:2411.17924*.

Weitekamp, D., Harpstead, E., & Koedinger, K. R. (2020). An Interaction Design for Machine Teaching to Develop AI Tutors. *Conference on Human Factors in Computing Systems - Proceedings*, *July*, 1–11.

Weitekamp, D., Siddiqui, M. N., & MacLellan, C. J. (2025). Tutorgym: A testbed for evaluating ai agents as tutors and students. From https://arxiv.org/abs/2505.01563.

Zhao, W. X., et al. (2023). A survey of large language models. *arXiv preprint arXiv:2303.18223*, *1*.