A Self-Aware BDI Agent for LLM-Driven Reasoning: Architecture, Design, and Preliminary Evaluation

Christopher Lawton, David Sacharny and Thomas C. Henderson

UUCS-26-001

Kahlert School of Computing
University of Utah
Salt Lake City, UT 84112 USA

15 May 2026

Abstract

Large language models (LLMs) are capable of sophisticated reasoning over individual steps but exhibit irrationality across multi-step tasks: they contradict prior statements, forget commitments, and accumulate errors in proportion to task length. Classical Belief-Desire-Intention (BDI) agent theory offers a principled framework for structured, persistent, verifiable reasoning. This paper describes the first phase of an ongoing project to build a full BDI agent architecture powered by LLMs: the design and implementation of the core deliberation loop and its evaluation harness. We make three contributions. First, we introduce *architectural self-awareness*: a design principle in which each loop phase is structured so the LLM understands how its outputs will be mechanically consumed downstream, yielding schema beliefs tuned for NLI plan matching, machine-verifiable satisfaction conditions, and a three-tier independent verification system. Second, unlike prior BDI systems that rely on hand-authored plans, our agent generates *parameterized natural language plans* at runtime using role-based parameter slots, enabling each plan template to generalize across structurally equivalent questions without human authorship. Third, we introduce a *simulation-seeding workflow*: the agent refines its plan library across multiple rounds using environmental feedback (wrong answers as failure signals), and the resulting library seeds new independent agent instances, enabling structured cross-agent knowledge transfer without retraining. We evaluate the agent on 53 GAIA level-1 questions using Qwen3-8B via vLLM. Simulation mode achieves 13.2% (any round), while seeded and non-seeded single-pass runs each achieve 9.4%. Qualitative trace analysis shows that plan seeding improves reasoning structure even when raw accuracy is unchanged, and identifies web search reliability and effective tool use as the primary bottleneck.

A Self-Aware BDI Agent for LLM-Driven Reasoning: Architecture, Design, and Preliminary Evaluation

Christopher Lawton, David Sacharny and Tom Henderson
University of Utah
chris.ian.lawton@gmail.com

Abstract

Large language models (LLMs) are capable of sophisticated reasoning over individual steps but exhibit irrationality across multi-step tasks: they contradict prior statements, forget commitments, and accumulate errors in proportion to task length. Classical Belief-Desire-Intention (BDI) agent theory offers a principled framework for structured, persistent, verifiable reasoning. This paper describes the first phase of an ongoing project to build a full BDI agent architecture powered by LLMs: the design and implementation of the core deliberation loop and its evaluation harness. We make three contributions. First, we introduce *architectural self-awareness*: a design principle in which each loop phase is structured so the LLM understands how its outputs will be mechanically consumed downstream, yielding schema beliefs tuned for NLI plan matching, machine-verifiable satisfaction conditions, and a three-tier independent verification system. Second, unlike prior BDI systems that rely on hand-authored plans [5], our agent generates *parameterized natural language plans* at runtime using role-based parameter slots, enabling each plan template to generalize across structurally equivalent questions without human authorship. Third, we introduce a *simulation-seeding workflow*: the agent refines its plan library across multiple rounds using environmental feedback (wrong answers as failure signals), and the resulting library seeds new independent agent instances, enabling structured cross-agent knowledge transfer without retraining. We evaluate the agent on 53 GAIA level-1 questions using Qwen3-8B via vLLM. Simulation mode achieves 13.2% (any round), while seeded and non-seeded single-pass runs each achieve 9.4%. Qualitative trace analysis shows that plan seeding improves reasoning structure even when raw accuracy is unchanged, and identifies web search reliability and effective tool use as the primary bottleneck.

1. Introduction

Despite remarkable progress in language model capability, LLMs face a structural limitation when deployed as agents for long-horizon tasks: they fail to satisfy the rationality criteria that motivate BDI agent design. The BDI tradition holds that rational agents must maintain consistent beliefs, honor commitments once made, and act in accordance with their stated goals [4]. LLMs routinely violate all three: they contradict statements made earlier in the same context, abandon commitments across turns without explanation, and accumulate errors at a rate that grows exponentially with task length. A model that executes each individual reasoning step with 90% reliability achieves only $(0.9)^{30} \approx 4\%$ reliability over a 30-step task as errors compound without external structure to catch and correct them.

This *rationality problem* is not an artifact of insufficient scale. It is a consequence of architecture: autoregressive models reason from context windows that function as bounded working memory, without persistent, addressable, verifiable state. As models grow larger, this structural limitation becomes more consequential as they are, in practice, deployed on tasks with longer horizons and higher stakes. Structured agent architectures are therefore essential as LLMs scale and are applied in highly regulated and high-consequence domains such as autonomous vehicles and clinical applications.

Belief-Desire-Intention (BDI) agent theory, introduced by Rao and Georgeff [4] and implemented in systems such as PRS [2] and AgentSpeak [3], provides a formal account of how rational agents maintain consistent beliefs, commit to goals, and execute plans while remaining responsive to new information. The deliberation loop governing these states provides an explicit structure for multi-step reasoning that is absent from standard LLM inference.

This paper covers the first phase of a larger project whose proposal [1] outlines a complete multi-module system including a versioned persistent knowledge base, a policy layer for consistency enforcement, an RL-based intention filter, and human-in-the-loop approval gates. Here we cover a prerequisite layer: the core deliberation loop and the evaluation harness needed to measure it. Future modules are designed to slot into this foundation; none can be meaningfully evaluated without a working loop to run them on.

Contributions. This paper makes three contributions to the design of LLM-powered BDI agents:

1. **Architectural self-awareness**: a design principle in which each loop phase explicitly instructs the LLM how its outputs will be consumed downstream, yielding structured beliefs, desires, and intentions, along with verifiable plan postconditions.
2. **Parameterized, LLM-generated plans**: unlike

prior BDI systems that require human-authored, task-specific plan rules [5], our agent generates parameterized plan templates at runtime using abstract, role-based parameter slots, which in theory should enable plan reuse across structurally equivalent questions without human authorship.

3. **Simulation-seeding for cross-agent plan transfer**: a workflow in which the agent refines its plan library across multiple rounds using environmental feedback, and exports the resulting structured library to seed new independent agent instances, a form of experience transfer not achievable with bounded episodic memory systems such as Reflexion [7].

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the high-level architecture. Section 4, the core of the paper, describes each of the seven BDI loop phases in detail. Section 5 discusses self-awareness as a cross-cutting design principle. Section 6 describes the evaluation harness. Section 7 presents preliminary benchmark results. Section 8 discusses scope, lessons, and limitations. Sections 9 and 10 address future work and conclude.

2. Background and Related Work

2.1. The Rationality Problem in LLMs

An autoregressive model generates each token conditioned only on a finite context window that functions as bounded working memory. The model has no privileged access to beliefs expressed earlier in the conversation; it must re-derive them from text, making consistency across turns fragile. The result is systematic failure of BDI rationality criteria: belief inconsistency, commitment failures, and error accumulation.

2.2. BDI Agent Theory

BDI agent theory [4] models rational agents in terms of three mental states: *beliefs* (the agent's world representation), *desires* (goals to achieve), and *intentions* (desires the agent has committed to pursue). The deliberation cycle governs how these states are updated: new perceptions revise beliefs, revised beliefs generate candidate desires, candidate desires are filtered into committed intentions, and intentions drive action selection.

NatBDI [5] extends BDI to natural language environments using natural language inference (NLI) to match plans written in natural language against current beliefs, with a fallback RL policy when no applicable plan exists. Even 13 hand-crafted plans substantially improved performance over pure RL on ScienceWorld tasks (up to 98% on certain subtasks).

Two limitations of NatBDI directly motivate this work. First, plans are *human-authored*: NatBDI requires a developer to write plan rules in controlled natural language before deployment; the authors explicitly identify "learning plan-rules from data" as future work. Second, plans are *unparameterized*: each plan is bound to the specific task description it was written for (e.g., `IF your task is to get the metal pot...`), and cannot generalize to structurally equivalent tasks with different entities. Our system addresses both limitations: plans are generated at runtime by the LLM and use abstract, role-based parameter slots (e.g. `numerator`, `denominator`) that generalize across task instances of the same structural type.

2.3. LLM-Based Agents

ReAct [6] interleaves reasoning traces with external actions in a single LLM output stream, making the agent's reasoning inspectable and allowing corrections to propagate. ReAct achieves substantial improvements on AlfWorld (34%) and WebShop (10%) compared to existing imitation and RL methods. Our step execution phase is in the spirit of ReAct: the LLM drives its own inner tool-use loop.

Reflexion [7] introduces verbal reinforcement learning: agents generate self-reflections on task outcomes stored in a bounded episodic memory (typically 1–3 recent experiences), improving future attempts without weight updates. Reflexion achieves 91% pass@1 accuracy on HumanEval through iterative self-reflection.

Our plan review phase (Phase 7) shares Reflexion's core intuition of learning from failure without gradient descent, but differs structurally in three ways. First, our memory is a *structured plan library* with quantitative statistics (`useCount`, `successCount`, `failureCount`) rather than free-form textual summaries; this enables aggregate evaluation across many task instances, not just recent episodes. Second, the library is *unbounded*: plans accumulate indefinitely and are pruned by the review phase rather than truncated by a context window. Third, the library is *transferable*: the structured plan representation can be exported and used to seed entirely new, independent agent instances; a form of cross-agent knowledge transfer that Reflexion's per-agent episodic memory cannot support.

2.4. Neuro-Symbolic Verification

Logic-LM [8] proposes a three-stage hybrid approach: LLMs translate natural language problems into symbolic form, a deterministic solver reasons over the symbolic representation, and the LLM maps results back to natural language, achieving a 39.2% improvement over pure LLM on logical reasoning benchmarks.

Explanation-Refiner [9] integrates theorem provers with LLMs to verify and iteratively refine natural language explanations. GPT-4 improves from 36% to 84% valid explanations on e-SNLI through iterative refinement. Our tiered verification system is an instance of this LLM-verifier pat-

tern: the LLM generates step outputs, an independent verifier checks them, and failures trigger replanning.

2.5. Structured LLM Output and Harness Architectures

Kambhampati et al. [10] argue that LLMs should not plan autonomously but instead operate inside a *LLM-Modulo* loop: the LLM proposes candidates and an external critic verifies them before they are accepted. Our three-tier verification system is an instance of this pattern applied per-step within a BDI deliberation loop, the harness never trusts a step's self-reported completion and always applies an independent check.

DSPy [11] treats LLM calls as typed modules with explicit input/output signatures, and compiles pipelines of such modules into optimized prompt-structure pairs. Our system applies the same principle at the prompt level: each phase specifies the schema its LLM call must produce, and that schema is the input contract for the next phase. The key distinction is mechanism, DSPy enforces structure by optimizing prompts automatically; our system enforces it by explicitly telling the LLM *why* its output must take a particular form (e.g., that schema beliefs will be used as NLI queries, so they must be abstract and domain-agnostic).

3. Architecture Overview

The system is built around a plugin interface that separates the agent's environment (terminal UI, tool execution, benchmark runner) from the agent backend (the BDI deliberation loop). All agents implement the same interface:

```
interface AgentPlugin {
  initialize(env: Environment): Promise<void>;
  handle(perception: Perception):
    AsyncGenerator<Action>;
  reset(): Promise<void>;
}
```

This interface enables swappable agent implementations, clean benchmarking, and simulation (routing the same question to the same agent multiple times with an accumulating plan library).

BDI State. The agent maintains: `schemaBeliefs` (abstract, for plan matching); `groundBeliefs` (concrete key-value facts); `desires` (natural language goals); `currentIntention` (the typed commitment being pursued); `planLibrary` (accumulated plans with statistics); and `intentionHistory` (records used for plan review).

LLM Providers The loop is backend-agnostic. Three providers are implemented: Anthropic (via SDK), Ollama (local), and vLLM. All expose:

```
interface LLMProvider {
  complete(messages: LLMMessage[]): Promise<string>;
  runAgentic(messages, tools, executor,
    channel, signal): Promise<string>;
}
```

`complete` is used for single-turn phases; `runAgentic` for step execution where the LLM drives its own inner tool-use loop.

Plan Persistence and Seeding. The plan library is persisted to a `plans.json` file which can be loaded on initialization. Plans generated in one session survive across sessions and can seed a new agent run, enabling a *cross-agent knowledge transfer* approach.

Run Modes:

- *Standard*: one pass per query using an initially unpopulated plan library.
- *Simulation*: multiple rounds per query; plan library accumulates across rounds. If query is answered incorrectly an LLM is used as judge to refine executed plan for next round.
- *Seeded*: standard mode but starting with a prepopulated plan library generated from another agent's run which used either Standard or Simulation mode.

4. The BDI Loop: Design and Rationale

The deliberation loop consists of seven phases. Algorithm 1 presents the high-level structure; each phase is described below with design rationale, an example for the running Kipchoge question[1], and influence on adjacent phases.

Algorithm 1: BDI Deliberation Loop

Input : user message m, plan library $\mathcal{L}$, max cycles C
beliefs, *desire* ← `Perceive(`m`)`
intention ← `Formalize(`*desire*, *beliefs*`)`
repeat
  plan ← `SearchLibrary(`*intention*, *beliefs*, $\mathcal{L}$`)`
  if *plan* $= \emptyset$ **then** *plan* ← `GeneratePlan(`*intention*, *beliefs*`)`
  foreach *step* ∈ `TopoSort(`*plan.dag*`)` **do**
    result ← `Execute(`*step*, `Snapshot(`*beliefs*`))`
    ok ← `Verify(`*step.postcondition*, *result*, *beliefs*`)`
    if ***not*** *ok* **then** log failure; **break** to replan
    beliefs ← `Merge(`*beliefs*, *result.newBeliefs*`)`
  sat ← `CheckSat(`*intention*, *beliefs*`)`
  if *sat* **then** emit answer; **break**
until C *cycles exceeded*
every N intentions: `Review(`$\mathcal{L}$, *intention_history*`)`

4.1. Phase 1: Perceive - Belief and Desire Extraction

The LLM receives the raw user message and produces three outputs simultaneously:

[1] "If Eliud Kipchoge could maintain his record-making marathon pace indefinitely, how many thousand hours would it take him to run the distance between the Earth and the Moon at its closest approach?"

1. **Schema beliefs:** abstract, structural facts about the *type* of problem. No concrete values or domain-specific names. Examples: *"task requires computing a ratio"*, *"task requires retrieving a value from a specified external source"*.
2. **Ground beliefs:** concrete key-value pairs extracted from the message. Names are role-based and domain-agnostic: `numerator` not `distance_km`; `target_entity` not `moon`.
3. **Desire:** a natural language statement of the goal that becomes the input to Phase 2.

Example output for the Kipchoge question:

```
schemaBeliefs:
  - "task requires computing a ratio"
  - "task requires retrieving a value
     from a specified external source"
  - "task requires rounding to nearest 1000"
groundBeliefs:
  - name: target_entity
    value: earth_moon_distance
    unit: km
    source: user
  - name: target_entity
    value: eliud_kipchoge_speed
    unit: km/h
    source: user
  - name: rounding_unit
    value: "1000"
    unit: hours
    source: user
desire: "calculate the time required to run
  the Earth-Moon distance at Kipchoge's pace"
```

Why abstract schema beliefs? A question about how long Kipchoge takes to run to the Moon and a question about how many days a snail takes to cross the Sahara share the same abstract structure: retrieve two values, compute a ratio, round the result. If schema beliefs are query-specific, they cannot match across task instances. If abstract, a plan built for the Kipchoge question can match the snail question at library search time. Abstraction is the mechanism by which plans generalize. Schema beliefs are the NLI query for plan library search in Phase 3.

Why role-based ground belief names? Plans are parameterized, a plan's step can reference beliefs by name to use the value of an existing belief or to create a new belief as the output from said step. For example, a step can state "fetch the value of `numerator`" rather than "fetch Kipchoge's marathon pace." Role-based names make the plan library reusable; concrete names make it question-specific. Ground beliefs populate plan parameter slots in Phase 4.

4.2. Phase 2: Formalize - Intention creation and satisfiability conditions

The desire and schema beliefs are converted into a typed `Intention`:

```
type: "compute_ratio_retrieval_rounding"
goal: "calculate the time required..."
satisfactionCondition:
  "ground_belief(final_answer) is known"
openParameters:
  [earth_moon_distance, kipchoge_pace,
   rounded_result]
```

Here `type` is a string filter when selecting plans, each plan will contain an `intentionType` which must match `type`. If a plan does not exist with the generated `type`, a new plan will be created with the provided `intentionType`. This matching accompanies the matching of Schema beliefs to context conditions which is further described in Phase 3. `goal` is the promoted desire from the from previous phase which describes the intention in natural language. `satisfactionCondition` defines the success criteria of the intention once executed. The satisfaction condition determines which verification tier is invoked in Phase 6. `openParameters` declares upfront what ground beliefs the plan needs to populate before the synthesize step is reached in the plan.

Why a machine-checkable satisfaction condition? If the agent expresses "done" as natural language, deciding whether the task is complete requires an LLM judgment call which is expensive and hallucination-prone. The predicate `ground_belief(X) is known` is a deterministic check: is `X` a ground belief? This costs zero tokens and cannot be hallucinated. The formalization prompt presents the three-tier verification hierarchy and asks the LLM to prefer Tier 1-checkable forms. This is the first instance of architectural self-awareness, the LLM produces output shaped by knowledge of how it will be consumed.

4.3. Phase 3: Deliberate - Plan Search and Generation

3a: Plan Library Search

Plan matching is completed via a single prompt to the LLM. The prompt includes the generated intention, schema beliefs, ground belief names, and entire plan library. The prompt requests the LLM to match the plan based off matching plan intention type to the provided intention, the plan's context conditions to the schema beliefs, and the plan's parameters to the ground beliefs. A match increments the plan's `useCount`, feeding the review phase's statistics.

3b: Plan Generation

When no match exists, the LLM generates a new plan. This stands in direct contrast to NatBDI, where plans must be authored by a developer before deployment. Here the LLM generates top-level meta information about that plan that is used for BDI state matching and a *parameterized DAG* at runtime. Each step in the DAG references abstract parameter slots rather than concrete values, so the same plan template applies to any question of the same structural type. For example, a full plan follows:

```
id: "plan-1776450034091-3vor7s"
intentionType: "compute_ratio_retrieval_rounding"
contextConditions:
  "task requires computing a ratio"
  "task requires retrieving a value from a specified
      external source"
  "task requires rounding to nearest 1000"
Steps
  S1 [fetch, deps={}]
    "Retrieve Kipchoge marathon pace in km/h"
    post: ground_belief(denominator) is known
  S2 [fetch, deps={}]
    "Retrieve Earth-Moon min perigee (Wikipedia)"
    post: ground_belief(numerator) is known
  S3 [compute, deps={S1,S2}]
    "Compute time: numerator / denominator"
    post: ground_belief(raw_result) is known
  S4 [compute, deps={S3}]
    "Round raw_result to nearest 1000 hours"
    post: ground_belief(rounded_result) is known
  S5 [synthesize, deps={S4}]
    "Emit FINAL ANSWER: <rounded_result>"
    post: ground_belief(final_answer) is known
```

S1 and S2 have no dependencies and execute in parallel; S3 waits for both. The DAG structure makes dependencies explicit and enables concurrent execution. Each step requires a post condition to be generated so the system can verify the work done during DAG execution. Currently, plan generation prefers the simpler Tier 1 verification method for a given step, but all tiers are available for step verification. Plan generation is another instance of architectural self-awareness as the plan must be generalizable, matchable to the current BDI state, and processable by downstream verification mechanisms. Plan quality determines what beliefs get accumulated and whether the intention can be satisfied. A bad plan (e.g., in this case fetching speed of light instead of marathon pace) propagates failure through the system.

4.4. Phase 4: Execute - Running the DAG

All steps with satisfied dependencies are dispatched concurrently. Each step receives: schema beliefs, a *snapshot* of the ground beliefs at dispatch time (preventing race conditions between parallel steps), the current intention, and the step's description and postcondition.

The LLM drives its own inner agentic loop with three tools: `web_search` (DuckDuckGo/Brave snippets), `bash` (shell arithmetic and data manipulation), and `read_file` (local files, PDFs, spreadsheets). The loop runs until the LLM signals completion or a turn limit is reached. An example output for S1 above follows:

```
postconditionMet: true
newGroundBeliefs:
  - name: denominator
    value: "20.9"
    unit: km/h
    source: web_search
summary: "Retrieved Kipchoge world record
  pace of 20.9 km/h"
```

New beliefs are merged into the ground belief store (upsert by name); subsequent steps see the updated store. These accumulated beliefs are the inputs to Phases 5 and 6 with a `final_answer` belief triggering Phase 6.

4.5. Phase 5: Verify - Independent Postcondition Checking

After each step, the harness independently verifies the step's postcondition. It does not trust the step's self-reported `postconditionMet` flag. Verification proceeds through three tiers, stopping at the first applicable:

Tier 1 - Pattern matching Regex matching against known patterns: `ground_belief(X) is known` checks whether belief `X` exists in the store. Approximately 90% of postconditions match Tier 1 because the plan generation and formalization prompts explicitly elicit this form.

Tier 2 - Symbolic reasoning via tau-prolog The LLM translates the postcondition and ground beliefs into a Prolog [15] program (facts, rules, query). A fresh tau-prolog session evaluates the query.

Tier 3 - LLM fallback For qualitative or semantic predicates that neither pattern-match nor translate to Prolog, we prompt an LLM that takes in the existing BDI state and the postcondition and ask it to return True or False depending on if the condition is satisfied.

A failed verification logs the failure mode such as `bad_retrieval`, `incorrect_computation`, etc., and triggers a replan attempt starting back at the top of Phase 3 with the partially-populated belief store.

4.6. Phase 6: Satisfy - Goal Satisfaction Check

After all DAG steps complete, the harness applies the same three-tier system to the intention-level `satisfactionCondition`. If any step output contains `FINAL ANSWER: <value>`, the harness extracts the value and exits the loop.

If the condition is not met and `maxCycles` has not been exceeded, control returns to Phase 3 with the richer, partially-populated belief store. The agent replans with more information than it started with.

4.7. Phase 7: Review - Plan Refinement

Every N completed intentions (configurable; default $N = 3$), the agent reviews its plan library. The LLM receives the N recent `IntentionRecord` objects which contain information that happened during the previous deliberation loops: what happened, which beliefs were accumulated, whether the intention was satisfied, which plan was used, and the current plan library with `useCount`, `successCount`, `failureCount` per plan.

The LLM outputs per-plan decisions: `keep`, `refine` (update context conditions, step descriptions, or parameters), or `remove`. Refined plans are persisted to a configurable `plans.json` file and survive across sessions.

Like Reflexion, the agent learns from failure without gradient descent. But the mechanism differs in a way that enables contribution 3: rather than maintaining a bounded, unstructured episodic memory within a single agent's lifetime, our review phase maintains a *structured, persistent, transferable plan library*. This library can be exported and used to initialize an entirely new agent instance so that experience accumulated in one run directly benefits future independent runs.

Long-horizon influence. Plans that succeed frequently are retained and refined; failing plans are removed. Over sessions, the library converges toward plans that reliably handle the task distribution. The seeding mechanism is what makes this accumulated knowledge portable: structured parameterized plans (contribution 2) are the unit of transfer, and architectural self-awareness (contribution 1) is what makes them reliably executable when reused.

4.8. Cross-Phase Influences

Table 1 summarizes the I/O for each phase. Three feedback loops influence behavior:

1. **Verify → Replan**: failed step verification returns control to Phase 3 with a richer belief store.
2. **Satisfy → Loop control**: an unsatisfied intention re-enters Phase 3 up to `maxCycles` times.
3. **Review → Plan Library**: long-horizon refinement; plans from one run seed the next.

Table 1: I/O of BDI loop phases

Phase	Reads from	Writes to
Perceive	user message	schema/ground beliefs, desires
Formalize	desire, schemaBeliefs	currentIntention
Deliberate	beliefs, intention, plan lib	currentPlans
Execute	beliefs, intention	groundBeliefs (new)
Verify	postcondition, beliefs	step pass/fail
Satisfy	satisfactionCond	loop control
Review	intentionHistory, plan lib	planLibrary, plans.json

5. Self-Awareness as a Design Principle

Architectural self-awareness refers to a design principle applied across several phases in the BDI loop. These phases are structured so that the LLM receives explicit information about how and why its output will be consumed downstream, and produces a response shaped by that knowledge. This section focuses on contribution 1, but the three contributions are interdependent: parameterized plans only generalize because the LLM is told to, via self-awareness, to use role-based parameter names rather than concrete values. And simulation-seeding only works because the plans being transferred are structured and reliably executable, a property that self-awareness, in theory, builds in at generation time.

The following are notable points of self-awareness with in the BDI loop:

Perceive The extraction prompt tells the LLM that schema beliefs are "upstream" of plan matching via NLI, and that ground belief names must be stable, role-based identifiers to function as plan parameter slots.

Formalize The formalization prompt presents the three-tier verification hierarchy and asks for a verifiable satisfaction condition. The LLM writes `ground_belief(final_answer) is known` rather than "the task has been completed" because it understands the former will be verified mechanically.

Plan Generation The plan generation prompt asks for step postconditions which adhere to one of the verification tiers. The LLM structures each step as establishing a named ground belief because it understands that this pattern enables verification.

Review The review prompt provides success/failure statistics and intention records. The LLM evaluates its own prior plans as a critic, with explicit numerical evidence of what has worked and what has not. The Tier system is also described to the reviewer so that insufficient postconditions can be updated.

5.1. Contrast with ReAct / chain-of-thought

ReAct interleaves reasoning traces with actions, producing interpretable output for human readers. The traces are not structured for downstream processing, are not parsed, verified, or used as keys in a lookup. Our system structures every phase's output as an input contract to the next phase.

5.2. Contrast with LLM-Modulo and DSPy

LLM-Modulo [10] places the LLM inside a generate-then-verify loop with an external critic; our three-tier verifier is an instance of this applied per step. DSPy [11] enforces output schemas by optimizing prompts automatically; our system enforces them by explicitly telling the LLM how each output will be consumed downstream, so the LLM can reason about structure rather than just conform to it.

6. The Evaluation Harness

All agent plugins implement the same interface; the evaluation harness does not know which agent it is running, enabling direct comparison between the BDI agent and other agent architectures without implementation-specific scaffolding.

GAIA Benchmark. We evaluate on the GAIA benchmark [12] (level 1), which consists of questions requiring multi-step reasoning, tool use, web search, and numerical calculation. Human performance on the level 1 GAIA question set is approximately 95% with early GPT-4-based agent systems with plugins achieved approximately 30% on level 1.

Answer Extraction. The harness extracts an answer using a priority chain: (1) `FINAL ANSWER: <value>`, the GAIA convention; last match wins; (2) `Answer: <value>`; (3) last non-empty line of the output.

Trace Persistence. Every run is saved to a timestamped JSON file containing the full per-question trace: question text, expected answer, agent's answer, score, tool calls, and BDI state at completion. This enables post-hoc failure analysis without re-running the agent.

7. Preliminary Results

Three benchmark runs were conducted on GAIA level 1 (53 questions) using Qwen3-8B served via vLLM on the University of Utah CHPC cluster. Table 2 summarizes the runs.

Table 2: GAIA level-1 results across three run configurations (Qwen3-8B, vLLM, CHPC).

Run	Mode	Start Plans	Accuracy
Simulation	3 rounds, 8× parallel	0	13.2% (7/53)
Seeded	Single pass	52	9.4% (5/53)
Non-seeded	Single pass	0	9.4% (5/53)

7.1. Simulation Run

The simulation run allowed three attempts per question with 8-way parallelism, meaning 8 questions are answered concurrently and their plans generated are merged into a single plan library which is available for subsequent rounds to use. Round-by-round accuracy: Round 1: 6/53 (11.3%); Round 2: 1 additional correct (cumulative 13.2%); Round 3: 0 additional. The multi-round structure demonstrates that iterative self-correction is effective: one question answered incorrectly in Round 1 was corrected in Round 2 after the belief store was enriched by the first attempt. The run produced a plan library of 52 plans spanning the task types encountered.

7.2. Seeded vs. Non-Seeded Comparison

Despite starting with 52 plans, the seeded run achieved the same raw accuracy as the non-seeded run. However, qualitative trace analysis reveals that the plan library improved the *quality of reasoning* even when the final answer was wrong. The Kipchoge question illustrates this most clearly:

Non-seeded: with no applicable plan, the agent generated one from scratch. The first fetch step searched for "speed of light in km per hour" instead of marathon pace, eventually producing a final answer of 0.0003561723 hours, off by five orders of magnitude from the expected answer of 17.

Seeded: the plan from the simulation run guided the agent to fetch Kipchoge's marathon pace and the Moon's minimum perigee distance. The agent computed 16.95 hours, a near miss with a rounding precision issue preventing exact-match scoring.

These two outcomes received identical scores under exact-match evaluation, but represent qualitatively different performances: the seeded run executed correct reasoning and was defeated by a rounding issue; the non-seeded run failed at retrieval and produced a nonsensical result. This distinction is invisible to the accuracy metric but meaningful for understanding the plan library's contribution.

7.3. Failure Mode Analysis

Trace analysis across all three runs yields a taxonomy of failure modes:

- **bad retrieval** (most common): web search returns incorrect or irrelevant information or fails to fetch entirely. Errors cascade through the DAG, a wrong fetch value produces a wrong computation, which produces a wrong final answer.
- **Avoiding file read tool use**: a large portion of the GAIA questions rely on reading local files and performing some computation over the data. A persistent observed failure mode is that the agent seems to fallback to web searching even for questions that require reading a file. The data required to answer to questions is not accessible via web search.
- **incorrect computation**: arithmetic errors in bash commands (e.g., integer truncation), or application of the wrong formula.
- **verification false positive**: the step self-reports `postconditionMet: true`, but the Tier 1 verifier finds no belief with the expected name; typically from inconsistent naming between the postcondition and output JSON.
- **dag deadlock**: no steps are ready (all have unmet dependencies) but the plan is incomplete, typically caused by a step writing a belief under an unexpected name, breaking the dependency chain.
- **answer formatting**: the synthesize step emits a FINAL ANSWER with a format that does not match the expected answer (e.g., `16.95` vs. `17`). These cases represent near misses in which the final answer requires one additional step of formatting, rounding, etc.

8. Discussion

8.1. Contributions in Context

The three contributions of this paper are summarized here alongside their relationship to prior work:

Architectural self-awareness (contribution 1) is novel as a named, structured design principle in BDI+LLM systems. Neither NatBDI nor Reflexion discuss designing each phase so the LLM's outputs are shaped by knowledge of how they will be mechanically consumed. The practical payoff is reduced integration fragility. Incorrect outputs can be caught at phase boundaries rather than silently propagated.

Parameterized, LLM-generated plans (contribution 2) directly addresses the two acknowledged limitations of NatBDI: the need for human plan authorship and the inability of plans to generalize across task instances. By using role-based parameter slots, a single plan template handles all questions of the same structural type. NatBDI's authors identified learning plan-rules from data as future work; this system does it at runtime.

Simulation-seeding (contribution 3) enables cross-agent plan transfer that neither Reflexion nor NatBDI supports. Reflexion's episodic memory is bounded, unstructured, and per-agent. NatBDI's plan library is static. Our structured, persistent, exportable plan library can seed independent agent instances with accumulated experience, functioning as a lightweight form of few-shot knowledge transfer grounded in structured plan representations.

8.2. Scope of This Work

The original proposal outlines a complete multi-module system: a versioned persistent knowledge base with belief provenance and conflict resolution; a policy layer enforcing consistency checks before intention commitment; an RL-based intention filter; and human-in-the-loop approval gates. This paper describes the first phase: the core loop, its three contributions, and the evaluation harness needed to measure them.

This sequencing reflects a principled dependency. None of the proposed modules can be meaningfully evaluated in isolation: each module's effect is observable only through its impact on loop behavior. The versioned KB's effect is measured by whether accumulated beliefs improve accuracy across sessions. The policy layer's effect is measured by whether consistency enforcement reduces failed intentions. The RL filter's effect is measured by whether learned plan selection outperforms NLI matching. All require a working loop and harness to measure against. The loop and harness are the prerequisite for everything else.

8.3. Lessons Learned

Self-awareness reduces integration fragility and improves component utilization. Prompting the LLM to structure its outputs for downstream mechanical processing reduces parsing errors and silent hallucinations. When a step writes a ground belief under a name that doesn't match the postcondition pattern, Tier 1 verification catches it immediately. Beyond fragility reduction, self-awareness improves how the agent uses its own components: a plan library populated with role-based belief names and Tier 1 postconditions is more useful than one populated with ad-hoc names and narrative descriptions.

DAG plans need finer decomposition. Individual steps with broad scope such as a fetch step that must call web search multiple times, reconcile conflicting results, and write several beliefs, are difficult to execute reliably and contain multiple failure modes. A more robust architecture would decompose plans into single-concern units, each with a narrowly scoped postcondition (e.g., "retrieve one specific value from one specific source"). Rather than executing one large DAG, the agent would execute several smaller focused plans in sequence. This would reduce the failure mode potential per plan, make verification more reliable, and make plan reuse more granular.

8.4. Limitations

Ineffective tool use The 9.4% single-pass accuracy is primarily limited by web search reliability and choosing the right tool for the question at hand. As mentioned above the agent frequently would default to web searching over using the read file tool. Fetching the correct information is essential to answer a questions adequately. We cannot asses the performance of the BDI model without robust and effective tool use. Thus, the current accuracy is not an accurate measure of BDI as a cognitive model.

The plan library has not yet demonstrated accuracy gains. At 53 questions with one pass each, the seeded run had insufficient overlap between library plans and question types to show a statistically meaningful improvement. A longer simulation is likely needed to refine plans further. Additionally, decomposition of plans into single-concern plans and further generalization of steps may lead to an increase in plan reuse.

Symbolic verification is underutilized. The tau-prolog Tier 2 is rarely invoked because plan generation successfully elicits Tier 1-form postconditions. When Tier 2 is invoked, the LLM often produces Prolog programs that tau-prolog cannot evaluate (arithmetic expressions, undefined predicates). A tighter Prolog generation prompt and refinement loop would improve Tier 2 utilization, and as demonstrated in other neuro-symbolic studies, could greatly improve the consistency and quality in responses.

9. Future Work

Near-term improvements: Web search reliability can be improved by prompting step execution with more specific query formulation guidance. Read file tool use must be refined to ensure the agent does not fall back to web searching for data which is local to the agent. Step postcondition prompting can be refined to reduce Tier 1 failures and increase the use of Tier 2 verification. The DAG decomposition issue (Section 8.2) motivates restructuring plans as sequences of narrow single-concern plans.

Remaining proposal modules: The versioned persistent KB will augment the transient `groundBeliefs` store with versioned, provenance-tracked beliefs persisting across sessions. The policy layer will enforce consistency checks at intention commitment. The RL-based intention filter will augment NLI matching with a learned reward signal.

Evaluation: Three targeted ablations would isolate each contribution. To evaluate contribution 1 (self-awareness): compare the BDI loop with structured prompts against the same loop with free-form prompts, measuring parsing failure rates and Tier 1 hit rates. To evaluate contribution 2 (parameterized plans): compare LLM-generated parameterized plans against hand-authored task-specific plans (NatBDI-style) on the same question set, measuring plan reuse rates and accuracy per question type. To evaluate contribution 3 (simulation-seeding): run longer simulations (more questions with > 3 rounds) and measure accuracy improvements as the plan library grows; compare seeded vs. non-seeded runs at multiple library sizes. Multi-model comparison (Qwen3-8B vs. larger models) would additionally evaluate whether the architecture's benefits persist across model scales. Additionally, testing the performance of a small model on a seeded run using plans generated from a larger model can test the potential for cross model experience sharing.

10. Conclusion

LLMs are capable but structurally irrational for long-horizon tasks. BDI agent theory provides a principled framework for structured, persistent, verifiable reasoning. This paper describes the design and implementation of a BDI deliberation loop powered by LLM inference, with three contributions relative to prior work.

Architectural self-awareness structures each loop phase so the LLM understands how its outputs will be mechanically consumed, reducing integration fragility and enabling cheap postcondition verification. *Parameterized, LLM-generated plans* eliminate the need for human-authored task-specific plans by using abstract role-based parameter slots that generalize across structurally equivalent questions. *Simulation-seeding* enables cross-agent plan transfer: accumulated plan experience from simulation rounds seeds independent agent instances, a capability not achievable with bounded episodic memory systems.

The evaluation harness enables reproducible, agent-agnostic benchmarking. Three runs on GAIA level 1 (simulation 13.2%, seeded/non-seeded 9.4%) establish a baseline and reveal actionable failure modes. The loop and its three contributions are the foundation that the remaining modules from the original proposal can be built upon.

References

[1] D. Sacharny and T. C. Henderson. Dynamic and Explainable BDI Agents with Persistent Knowledge. Independent Study Proposal, University of Utah, 2026.

[2] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, 1987.

[3] A. S. Rao. 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-96)*, Lecture Notes in Artificial Intelligence, vol. 1038, pages 42–55. Springer, 1996.

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

[5] A. Y. Ichida, F. Meneguzzi, and R. C. Cardoso. BDI Agents in Natural Language Environments. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2024.

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

[7] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv preprint arXiv:2303.11366*, 2023.

[8] L. Pan, A. Albalak, X. Wang, and W. Y. Wang. Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning. In *Proceedings of EMNLP 2023 Findings*, 2023.

[9] X. Quan, Y. Shen, and B. Gao. Verification and Refinement of Natural Language Explanations through

LLM-Symbolic Theorem Proving. In *Proceedings of EMNLP 2024*, 2024.

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

[11] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[12] G. Mialon, C. Dessì, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Rozière, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz, E. Grave, Y. LeCun, and T. Scialom. GAIA: A Benchmark for General AI Assistants. *arXiv preprint arXiv:2311.12983*, 2023.

[13] A. Farjami et al. Logic-Parametric Neuro-Symbolic NLI: Controlling Logical Formalisms for Verifiable LLM Reasoning. *arXiv preprint*, 2024.

[14] H. Chen. Large Knowledge Model: Perspectives and Challenges. *arXiv preprint arXiv:2407.00674*, 2024.

[15] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.