Jarvis: LLM-Assisted Plan and Belief Generation via Artifact Introspection

Lucas C. Raupp[1], **Gustavo Paulo**[1], **Ana Laura S. Chagas**[1]

[1]Department of Automation and Systems Engineering
Federal University of Santa Catarina (UFSC), Florianópolis (SC), Brazil

lucascoelhoraupp@gmail.com, gustavopaulo-guga@hotmail.com
anasoethechagas@gmail.com,

Abstract. *BDI agents traditionally rely on plan libraries authored at design time, which limits their ability to operate in environments containing artifacts the programmer did not anticipate. We present Jarvis, a JaCaMo-native gateway that lets agents acquire plans and beliefs from a large language model at runtime, grounded on reflective introspection of the CArtAgO artifacts present in the workspace. The LLM receives the discovered operation signatures of each artifact and produces an AgentSpeak plan library; a validator filters the response against the discovered operations before two internal actions insert the surviving plans and beliefs into the agent's mental state. We illustrate the approach with a three-agent case study in which an agent learns to operate an air-conditioner artifact entirely from LLM-generated knowledge and serves comfort requests issued by peer agents.*

1. Introduction

BDI agents [Bratman 1987, Rao and Georgeff 1995] rely on plan libraries authored by the programmer at design time. When the environment contains artifacts whose existence or behavior was not foreseen, the agent has no plan to invoke. Conventionally it has two options: a programmer supplies it, or a peer agent that already knows the task shares it. Recent advances in large language models (LLMs) suggest a third option: synthesizing the missing procedural knowledge at runtime.

Recent work in the multi-agent community has begun to explore this possibility. [Ciatto et al. 2025] embed generative AI in the AgentSpeak reasoning cycle to generate plans on demand, after plan failure, or proactively, targeting the JaKtA framework. In a complementary direction, ChatBDI [Gatti et al. 2025] uses an LLM as the natural-language interface of a BDI agent, not as a plan generator. [Aguzzi et al. 2025] bring these two directions together under the name PlanchBDI and point to artifact-based infrastructures as a promising setting in which LLMs can reason over the artifacts available to an agent.

This paper presents Jarvis, a JaCaMo [Boissier et al. 2013] gateway that takes a complementary step in that direction. Jarvis grounds LLM-based plan generation in the artifacts that the agent already shares with its environment. Instead of asking the programmer to list the actions available to the LLM, the gateway introspects the CArtAgO [Ricci et al. 2011] artifacts in the workspace at runtime, serializes their operation signatures into the prompt, validates the generated AgentSpeak source against the

discovered operations and arities, and injects the accepted plans and beliefs into the agent through dedicated Jason internal actions [Bordini et al. 2007].

The paper makes four contributions: Jarvis itself, a CArtAgO artifact that mediates the agent-LLM dialogue and publishes usage metrics as observable properties available to peers; a reflective introspection mechanism that extracts the operations and observable properties of arbitrary CArtAgO artifacts at runtime, without binding them to a live workspace; a validation pass that rejects generated plans referencing operations or arities absent from the introspected artifact; and a runtime injection path through dedicated Jason internal actions that preserves the origin of LLM-authored knowledge.

2. Related Work

Recent work on LLMs and BDI agents follows two main integration patterns. In the first, the LLM acts as a natural-language interface for an otherwise traditional BDI agent. ChatBDI [Gatti et al. 2025], for example, extends Jason agents with a mediator and two CArtAgO artifacts that translate between KQML and natural language, letting humans converse with a Jason MAS, but the LLM does not generate plans. In the second pattern, the LLM participates in the reasoning loop by generating procedural knowledge. [Ciatto et al. 2025] propose a Plan Generation Procedure that can be plugged into an AgentSpeak(L) agent in three modes: on demand, reactively after plan failure, and proactively. Their procedure validates the LLM's hybrid YAML+Prolog output against the agent's statically declared action set and adds the surviving plans as retractable partial plans. [Aguzzi et al. 2025] explicitly discuss the combination of these two directions, naming the proposal PlanchBDI and identifying artifact-based infrastructures as a promising future path.

Outside the BDI tradition, MetaGPT [Hong et al. 2024] illustrates the LLM-as-agent pattern, in which prompt-defined personas such as Product Manager, Engineer, and QA execute encoded SOPs in a pipeline without an explicit mental state, plan library, or reasoning cycle. [La Malfa et al. 2025] criticize this style because it leaves aside core MAS properties such as autonomy, structured environments, and asynchronous coordination. This critique motivates the stance adopted here: the LLM should serve a BDI agent within its reasoning cycle rather than replace that agent.

Table 1 summarizes the main differences between Jarvis and the closest systems in the literature.

Table 1. Positioning of Jarvis relative to recent LLM + MAS integrations.

System	LLM role	Target	Action source	Validation
ChatBDI	NL interface	Jason/JaCaMo	no plan generation	best-effort
Ciatto et al.	plan generator	JaKtA	statically declared	syntax + actions
MetaGPT	agent role-play	ad-hoc	tools in prompt	not specified
Jarvis	plan generator	Jason/JaCaMo	introspected at runtime	syntax + arity vs. spec

The closest work to ours is the Plan Generation Procedure of [Ciatto et al. 2025]; the key difference is where the action set comes from. In Jarvis, the actions shown to the

LLM and later used for validation are discovered at runtime from CArtAgO artifacts in the workspace, whereas their approach relies on actions declared statically in the agent program. Their plan lifecycle management (invalidation, retraction of failing plans) is more mature; we return to this in Section 5.

3. Architecture

Figure 1 shows the overall arrangement of agents, artifacts, and the external LLM endpoint sharing the CArtAgO workspace.

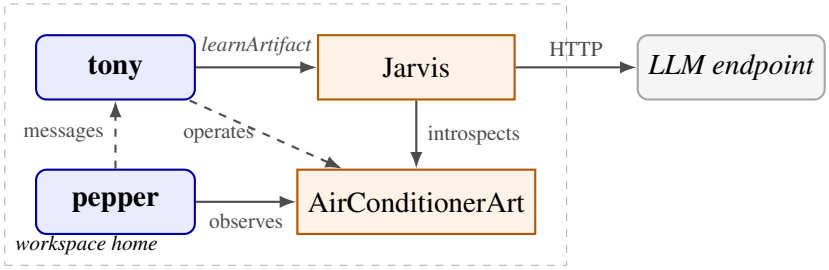

Figure 1. Overall architecture. Agents (blue) and CArtAgO artifacts (orange) share a workspace; solid arrows are operation invocations, dashed arrows secondary operations and inter-agent messages. The LLM endpoint is contacted exclusively through Jarvis.

Jarvis is realized as a CArtAgO artifact that mediates between Jason BDI agents and an external LLM endpoint. It offers two operations to agents: `listOtherArtifacts`, which queries the workspace's artifact registry and returns the name and Java class of every deployed artifact (filtering out Jarvis itself and CArtAgO's infrastructure), letting an agent discover artifacts it has no design-time knowledge of; and `learnArtifact`, whose only input is the target's `ArtifactId`. The agent never specifies which operations or properties it wants to operate: the full specification is recovered by introspection (Section 3.1). It also publishes observable properties that report its own usage, including counts of accepted and rejected plans, the cumulative token consumption, and a flag indicating whether a request is currently being processed. Centralizing the LLM dialogue in a single artifact keeps the API credential in one place, makes the cost of each generation visible to peers through ordinary observable properties, and lets agents request plans through standard CArtAgO operation invocations, with no specialized library on the agent side.

3.1. Reflective Artifact Introspection

The gateway extracts a specification of the target artifact via Java reflection. The specification records the class name, the operations and their parameter types, the observable properties and their types, and an optional natural-language description. Operations are discovered by scanning the artifact's declared methods for the `@OPERATION` annotation. Observable properties are harder to obtain: they are not Java fields, but are created dynamically inside the artifact's initialization method through calls to the property-declaration API. To capture them without binding the artifact to a live workspace, the gateway allocates a stub workspace object through the JDK's Unsafe API, populates the minimum internal state the initialization expects (a workspace handle, an artifact identifier, a property

map), invokes the initialization, and reads back the populated map. A reserved observable property named *description* is treated as a free-text manual and forwarded verbatim into the prompt.

The resulting specification grounds the LLM call in the current workspace: the model sees the operations the agent can actually invoke, rather than a list the programmer happened to mention in the agent program.

3.2. Prompt Construction and Structured Response

When an agent invokes the learning operation with a target artifact identifier, Jarvis introspects the target, assembles the prompt, and dispatches the request to the configured OpenAI-compatible endpoint. The user prompt lists the operation signatures and observable properties drawn from the specification. The system prompt asks the model to return AgentSpeak plans in two layers. The first is a primitive plan for each `@OPERATION`. The second is an external-request plan for each observable property that other agents may want to influence. The prompt also asks for an initial set of beliefs encoding ranges, modes, and constants relevant to the artifact. The response is constrained to a JSON Schema through the endpoint's structured-output parameter, eliminating the common failure mode in which the model mixes plain text with code. A small set of few-shot examples on unrelated artifacts (a dimmable lamp, a temperature sensor) accompanies the system prompt; Section 4.2 shows the concrete prompt assembled for the case-study artifact.

3.3. Validation against the Discovered Surface

Before any plan reaches the agent, the gateway walks through the body of each generated plan and applies four checks. The plan must parse under Jason's AgentSpeak parser. Every artifact action invoked in the body must correspond to an operation in the specification. The number of arguments at each call must match the operation's arity. Finally, every sub-goal must be resolved by a trigger generated in the same response. Standard Jason internal actions (such as printing or waiting) are recognized and skipped during the semantic check. Plans that fail any check are rejected with a structured reason, persisted to a per-request log alongside the prompt and the raw response. The key detail is that the validation oracle is the same specification used to build the prompt: hallucinated operations or wrong arities are caught because they were never present in the artifact.

3.4. Runtime Injection

After validation, the learning operation returns the surviving AgentSpeak source as two lists of strings, one for plans and one for beliefs. Two Jason internal actions, `jlm.add_plans_from` and `jlm.add_beliefs_from`, parse each string and update the agent's mental state. Plans are added to the plan library tagged with the source label *llm*, preserving the origin of LLM-authored knowledge for later inspection; beliefs are added to the belief base through the standard Jason API. Parsing failures are logged as warnings rather than raised as exceptions, so a single malformed item does not disrupt the rest of the learning sequence.

Figure 2 summarizes the end-to-end pipeline and makes explicit the dual role of the artifact specification, as both the grounding context fed to the LLM and the oracle against which the generated plans are validated.

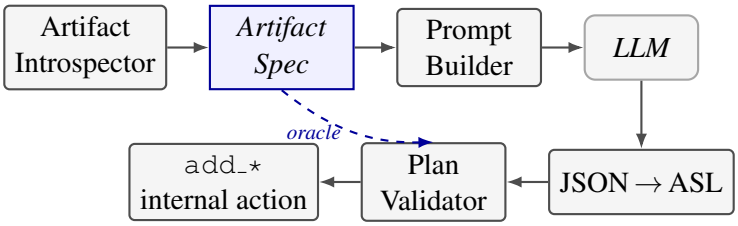

Figure 2. Plan generation pipeline. The specification produced by introspection feeds both the prompt builder and, via the dashed back-edge, the validator, guaranteeing that the validation oracle matches the grounding context shown to the LLM.

4. Case Study

We illustrate the system with a three-agent JaCaMo MAS deployed against an AirConditioner artifact in a workspace named *home*. The artifact exposes three operations (*setPower*, *setTemperature*, *setMode*) and three observable properties (*power*, *currentTemp*, *mode*). It also carries a free-text description listing the supported modes and the valid temperature range. The LLM is OpenAI's gpt-4.1-mini at temperature 0.2.

4.1. Agents and Roles

tony acts as the learner-broker: he discovers and learns every artifact in the workspace (Section 4.2), then announces his readiness and waits for goal-achievement messages from peers.

pepper acts as the comfort sensor, with a happiness range of 20–26 °C and a preferred temperature of 23 °C. Once tony signals readiness, she focuses on the air conditioner and reacts to every update of the current temperature: outside her comfort range she sends tony an achievement goal to drive it back to her preferred value; inside it she simply tells tony she is happy.

loki acts as the disruptor. After the system stabilizes, he focuses on the air conditioner and arbitrarily sets the temperature to 30 °C, taking pepper outside her comfort zone and provoking a fresh request.

4.2. Learning Walkthrough

Figure 3 (left) shows tony's complete program as authored by the programmer, before any LLM-generated knowledge is injected (`.print` calls and the plan acknowledging pepper's `happy` message are elided). It contains no reference to the air conditioner: no AC operation call, no temperature-related plan, no belief about modes or ranges.

The program makes the interaction explicit. Discovery: on startup, tony focuses on Jarvis and invokes `listOtherArtifacts`, which returns the name/class pairs of the artifacts present in the workspace (here, only `ac_room1`). Learning: for each pair, tony resolves the instance with the standard `lookupArtifact` and invokes `learnArtifact` as a plain CArtAgO operation, passing only the resolved `ArtifactId`; he never lists the operations or properties he wants to operate. The feedback parameters `Plans` and `Beliefs` return the validated AgentSpeak source strings, which the internal actions `jlm.add_plans_from` and `jlm.add_beliefs_from` parse into his plan library and belief base. Jarvis assembles the user prompt (Figure 3,

```
!start.

+!start
 <- joinWorkspace("home", _);
    lookupArtifact("jarvis", J);
    focus(J);
    !discover_and_learn.

+!discover_and_learn
 <- listOtherArtifacts(L);
    !learn_all(L);
    .broadcast(tell, tony_ready).

+!learn_all([]).
+!learn_all([[Name, Cls] | T])
 <- lookupArtifact(Name, Id);
    learnArtifact(Id, Plans, Beliefs);
    jlm.add_plans_from(Plans);
    jlm.add_beliefs_from(Beliefs);
    focus(Id);
    !learn_all(T).
```

```
Artifact to learn:
  class: AirConditionerArt
  workspace identifier: ac_room1

operations:
  - setPower(boolean)
  - setTemperature(int)
  - setMode(String)

observable_properties:
  - power: Boolean
  - currentTemp: Integer
  - mode: String

manual:
  "Split air conditioner.
  setMode accepts: 'cool',
  'heat', 'fan'. [...]
  setPower(false) resets
  mode internally."
```

Figure 3. Left: tony's program before injection; no artifact-specific knowledge appears anywhere. Right: the user prompt for `ac_room1`, generated entirely by introspection (manual abbreviated; full text in Section 4.3).

right) verbatim from the introspected specification: property types are inferred from initialization values, and the reserved *description* property becomes the manual. The fixed system prompt requests the two plan layers of Section 3.2 and states the constraints later enforced by the validator, notably that inventing an operation invalidates the plan and that sub-goals may only reference triggers generated in the same response.

4.3. Generated Knowledge

For the AirConditioner artifact, the LLM returns roughly half a dozen plans and a handful of beliefs. Figure 4 shows a representative excerpt of the accepted plans after validation.

```
+!setPower(P) : true <- setPower(P).
+!setTemperature(T) : true <- setTemperature(T).
+!setMode(M) : true <- setMode(M).
+!target_currentTemp(T) : true
   <- setPower(true);
      setMode(cool);
      setTemperature(T).
```

Figure 4. Representative LLM-generated AgentSpeak plans accepted by the validator for the AirConditioner artifact.

The first three plans expose each operation as a callable primitive goal. The fourth plan responds to pepper's external request to adjust the temperature, and it captures a precondition the agent was never explicitly told: adjusting the temperature requires the unit to be powered on and switched to a cooling mode. The LLM recovered this constraint from the artifact's natural-language manual, which reads "*setTemperature accepts integers from 16 to 30 degrees when mode='cool' or 'heat'; ignored when mode='fan'. setPower(false) resets mode internally*".

4.4. Execution Trace

In Phase 1, tony learns the AC. Pepper observes the current temperature at 18 °C, which is below her comfort range, and sends tony the request to drive the temperature to her

preferred value. Tony executes the LLM-generated plan: he powers the unit on, switches it to cooling mode, and only then applies the target temperature. Pepper then observes the new reading, finds it inside her range, and tells tony she is happy.

In Phase 2, loki sets the temperature to 30 °C. Pepper's belief revision fires the same uncomfortable branch, sends a new request, and tony reapplies his generated plan. The system recovers without any human-authored AC-adjustment plan anywhere in the codebase.

5. Discussion and Limitations

The case study provides initial evidence that artifact introspection can supply enough structural and descriptive context for an LLM to generate executable plans without artifact-specific code in the learner. However, successful runs in a single domain do not establish generality or semantic safety. We therefore distinguish the prototype's observed behavior from claims not yet supported and discuss artifacts as a grounding boundary between the BDI agent, its environment, and the LLM.

5.1. What Works

Across repeated runs with gpt-4.1-mini at temperature 0.2, the model produces a syntactically valid plan library on the first call, with every AirConditioner plan accepted by validation. The introspected specification also supports the correct multi-step sequence—power on, select the mode, then set the temperature—by combining operation signatures with the natural-language manual. One learning episode consumes approximately 1.5 K input and 200 output tokens.

5.2. What We Do Not Claim

The prototype is deliberately minimal in three respects. First, generated plans remain in the library until termination, without the failure-driven retraction provided by [Ciatto et al. 2025]. Second, generation is only on demand; reactive and proactive modes were not exercised. Third, validation covers syntax, operation existence, arity, and subgoal resolution, but not semantic domain invariants such as mode preconditions.

5.3. The Introspectable-Artifact Stance

The contribution is not runtime plan generation alone, where [Ciatto et al. 2025] is more mature, but the use of artifacts as the grounding substrate. Artifacts already represent environmental affordances through operation signatures, observable properties, and often a natural-language description. Planning over artifacts discovered live in the workspace therefore aligns the LLM's action model with CArtAgO's environmental model [Ricci et al. 2011], without a design-time action list. In response to the concerns of [La Malfa et al. 2025], Jarvis lets the LLM contribute within JaCaMo's structured environment and asynchronous coordination rather than replace them.

6. Conclusion and Future Work

Jarvis grounds runtime generation of AgentSpeak plans and beliefs in reflectively introspected CArtAgO artifacts and validates the result before injection. It complements the static action set of [Ciatto et al. 2025] and the conversational focus of

ChatBDI [Gatti et al. 2025], contributing toward PlanchBDI [Aguzzi et al. 2025]. Future work includes failure-triggered generation, plan retraction, broader artifact and prompt evaluations, and local open-weight models.

References

Aguzzi, G., Ciatto, G., Ferrando, A., Gatti, A., and Mascardi, V. (2025). LLMs as agents, LLMs at the service of agents, or agents at the service of LLMs? Extended abstract, 26th Workshop "From Objects to Agents" (WOA 2025). Introduces the PlanchBDI proposal unifying chattification and LLM-based plan generation under BDI control.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761.

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

Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA.

Ciatto, G., Aguzzi, G., Battistini, R., Baiardi, M., Burattini, S., Ricci, A., et al. (2025). Exploiting GenAI for plan generation in BDI agents. *Frontiers in Artificial Intelligence and Applications*, 413:3495–3502. Proceedings of the 28th European Conference on Artificial Intelligence (ECAI 2025).

Gatti, A., Mascardi, V., and Ferrando, A. (2025). ChatBDI: think BDI, talk LLM. In *Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 2541–2543.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Yau, S., Lin, Z., Zhou, L., et al. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. In *12th International Conference on Learning Representations (ICLR)*.

La Malfa, E., La Malfa, G., Marro, S., Zhang, J., Black, E., Luck, M., Torr, P., and Wooldridge, M. (2025). Large language models miss the multi-agent mark. In *Advances in Neural Information Processing Systems (NeurIPS), Position Paper Track*, volume 38.

Rao, A. S. and Georgeff, M. P. (1995). BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319.

Ricci, A., Piunti, M., and Viroli, M. (2011). Environment programming in multi-agent systems: An artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192.