MagicArtifact: A Language-Agnostic Approach for External Artifact Development in the Agents and Artifacts Model

Eric Pinheiro, Nilson Lazarin, Bruno Freitas, Carlos Pantoja

Federal Center for Technological Education Celso Suckow da Fonseca (Cefet/RJ)
Nova Friburgo, RJ – Brazil

chon@grupo.cefet-rj.br

Abstract. *JaCaMo integrates agent reasoning, environment programming, and organizational modeling through Jason, CArtAgO, and Moise. Despite its modularity, Java-based CArtAgO artifacts hinder reuse across software ecosystems. This paper presents MagicArtifact, a generic CArtAgO artifact that connects to an external runtime through a JSON-based WebSocket protocol, obtains an Artifact Interface Description, dynamically registers operations, and maps remote results to CArtAgO signals and observable properties. The approach remains language-agnostic across compatible runtimes. A TypeScript proof of concept hosted social-media collection and sentiment-analysis artifacts. Using real services, it classified seventeen unique replies: five positive, ten negative, and two neutral, preserving JaCaMo's interaction model.*

1. Introduction

Multi-agent systems provide abstractions for building applications composed of autonomous entities capable of perceiving their environment, making decisions, and interacting to achieve individual or collective goals [Wooldridge 2009]. Developing these systems involves not only programming the behavior of each agent, but also representing the shared environment and the organizational relationships among system participants. In this context, JaCaMo provides an integrated approach to multi-agent system development by combining different abstractions within the same platform.

JaCaMo combines three complementary dimensions: Jason for agent reasoning and behavior, CArtAgO for programming the shared environment, and Moise for organizational specification [Boissier et al. 2013]. In CArtAgO, environments are organized as workspaces containing artifacts, which expose operations and generate observable events and properties that agents can perceive [Ricci et al. 2007, Ricci et al. 2009]. The implementation used by JaCaMo is based on Java, and new artifact types are commonly defined as Java classes, preserving the separation between agent reasoning and environment logic while potentially introducing an additional technological layer.

Despite the benefits of this separation, the concrete implementation of the environment dimension may require knowledge of additional technologies, particularly when functionalities already available in other software ecosystems must be exposed as CArtAgO artifacts. This work investigates this practical barrier and proposes *MagicArtifact*, a generic bridge between JaCaMo agents and artifact implementations executed in external runtimes. The next section presents the foundations of environment programming in JaCaMo and discusses related approaches for integrating agent programming with different languages and software ecosystems.

2. Background and Related Work

Building on the overview presented in the Introduction, Jason provides an AgentSpeak-based language for representing agent beliefs, goals, and plans, whereas Moise supports the definition of organizational structures, roles, missions, and constraints [Bordini and Hübner 2006, Hübner et al. 2007]. The remainder of this section focuses on the environment dimension and on approaches that connect agent-oriented programming with different host-language ecosystems.

The environment dimension is provided by CArtAgO, which follows the Agents and Artifacts model. In this model, the environment is structured as a set of workspaces containing agents and artifacts. A workspace defines a logical space in which agents can locate, use, and observe artifacts. Artifacts represent resources or tools that provide functionalities through operations and may generate observable events and properties during their execution [Ricci et al. 2007, Ricci et al. 2009].

In the CArtAgO implementation, new artifact types are commonly defined as Java classes extending the base artifact class provided by the platform. Operations are implemented as methods, while observable events and properties are produced through the CArtAgO API [Ricci et al. 2009]. Although this model preserves the separation between agent reasoning and environment logic, its implementation may require developers without previous experience in the Java ecosystem to learn an additional technological layer when integrating agents with external services and systems.

The relationship between agent-oriented programming and general-purpose programming ecosystems has also been addressed by different approaches. JS-son provides BDI abstractions as a lightweight JavaScript library, allowing agents and their environments to be embedded into applications and toolchains already used in web development [Kampik and Nieves 2020]. Similarly, JaKtA provides an internal domain-specific language for BDI agent programming in Kotlin. Agents, environments, and actions can be defined through Kotlin abstractions, enabling the use of the libraries, development tools, and interoperability mechanisms available in the host language [Baiardi et al. 2023].

Other approaches implement agent models or languages on different platforms. MASPY provides a framework for developing BDI multi-agent systems in Python, combining abstractions for agents, environments, and communication [Mellado et al. 2025]. eJason, in turn, implements a subset of Jason in Erlang, aiming to combine rational-agent programming with the concurrency and distribution mechanisms offered by the Erlang ecosystem [Fernández Díaz et al. 2013]. In the version presented by its authors, however, support for agents acting on an environment was still identified as future work.

Table 1 summarizes how these approaches distribute agent and environment programming across different languages and software ecosystems. These approaches illustrate different strategies for bringing agent-oriented programming closer to widely used languages and tools. JS-son, JaKtA, MASPY, and eJason bring part of the agent programming model into their respective host ecosystems. *MagicArtifact* follows a different strategy: it preserves Jason, JaCaMo, and the CArtAgO artifact interface, while allowing only the concrete artifact logic to execute in another programming environment. Therefore, the proposal does not introduce a new agent language or reimplement the agent reasoning cycle, but provides a bridge for connecting external implementations to the

Table 1. Related approaches for agent and environment programming.

Approach	Agent Programming	Environment Programming	Ecosystem	Integration Strategy
JaCaMo [Boissier et al. 2013]	Jason / AgentSpeak	CArtAgO artifacts implemented through the Java API	Java / JVM	Integrates agent, environment, and organization through separate abstractions
JS-son [Kampik and Nieves 2020]	JavaScript BDI abstractions	JavaScript environment abstractions	Node.js	Implements agent and environment abstractions directly in the host language
JaKtA [Baiardi et al. 2023]	AgentSpeak-inspired Kotlin DSL	Kotlin environment and action abstractions	Kotlin / JVM	Embeds BDI programming into the host language and its tooling
MASPY [Mellado et al. 2025]	AgentSpeak-like BDI abstractions in Python	Python environment classes and environment plans	Python	Implements agents, environments, and communication in a Python framework
eJason [Fernández Díaz et al. 2013]	Subset of Jason translated into Erlang	Environment interaction identified as future work	Erlang	Reimplements part of the Jason execution model on the Erlang runtime
MagicArtifact	Jason / AgentSpeak	CArtAgO interface with artifact logic executed externally	JaCaMo + external runtimes	Preserves JaCaMo and externalizes artifact logic

existing JaCaMo model.

3. Proposal

MagicArtifact is a generic CArtAgO artifact designed to decouple the interface used by agents from the concrete implementation of the artifact logic. The approach preserves the JaCaMo interaction model: Jason agents continue to invoke operations and perceive observable events or properties through the CArtAgO environment [Boissier et al. 2013, Ricci et al. 2009]. However, instead of containing integration-specific logic, *MagicArtifact* acts as a bridge to an artifact hosted by an external runtime.

During initialization, each *MagicArtifact* instance receives the address of the runtime and the identifier of the remote artifact it should represent. After establishing a WebSocket connection, it obtains a description of the remote artifact interface and dynamically registers the corresponding operations in CArtAgO. When an agent invokes one of these operations, *MagicArtifact* forwards the request to the runtime, which dispatches it to the selected external implementation. Therefore, different artifacts can be exposed through instances of the same Java class without requiring a new CArtAgO artifact implementation for each integration. Figure 1 presents the architecture of the proposed approach.

3.1. Integration and Usage

Integrating an external artifact through *MagicArtifact* requires configuration on two sides. On the JaCaMo side, the reusable *MagicArtifact* package is declared and one generic artifact instance is created for each remote artifact that should be available in a workspace. On the external side, a compatible runtime provides the corresponding interface descriptions, dispatches incoming operation requests, and translates implementation results into protocol messages. Therefore, no application-specific Java artifact class is required, although an interface description and protocol-facing dispatch logic must still be provided for each external artifact.

The generic bridge `com.github.chon-group:MagicArtifact` is distributed as a package[1]. A JaCaMo application imports it through the JCM `uses`

[1]Available at: `https://github.com/chon-group/MagicArtifact`

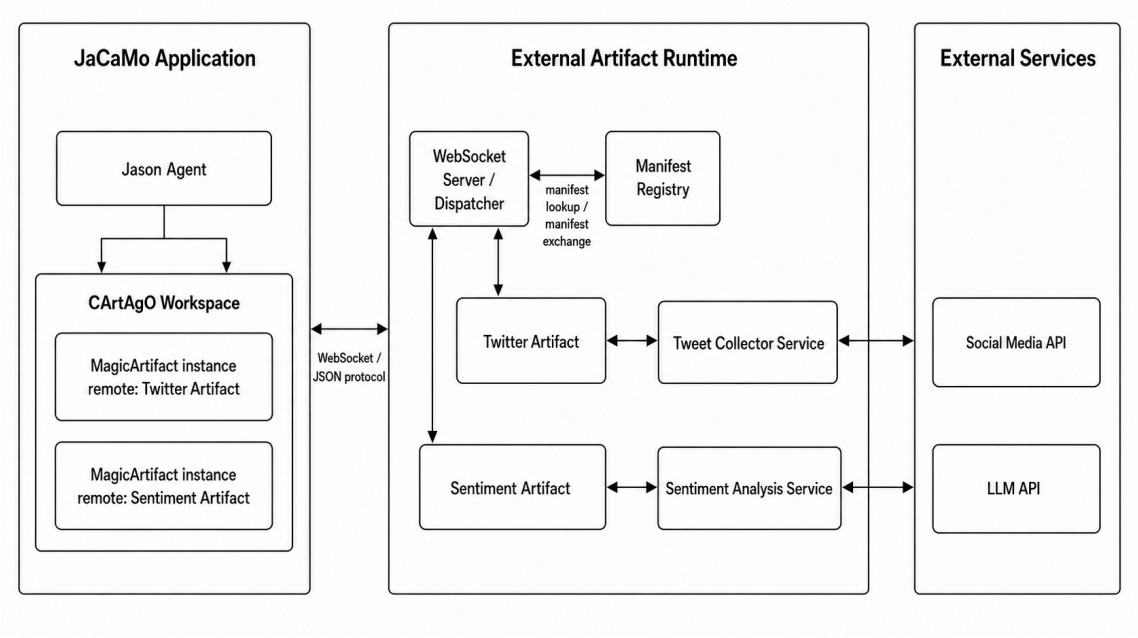

Figure 1. General architecture of the proposed approach. Instances of *MagicArtifact* expose artifacts hosted by an external runtime through an Artifact Interface Description and a JSON-based WebSocket protocol.

`package` declaration and represents each external artifact with an instance of `chon.MagicArtifact`. Code 1 illustrates this configuration using the two artifacts later employed in the proof of concept.

```
mas multiAgentSystemProject {
    workspace main {
        artifact twitter: chon.MagicArtifact("localhost", 8080, "TwitterArtifact")
        artifact sentiment: chon.MagicArtifact("localhost", 8080, "SentimentArtifact")
    }
    uses package: magicArtifact "com.github.chon-group:MagicArtifact:+"
}
```

Code 1. JaCaMo configuration declaring two *MagicArtifact* instances connected to the same external runtime.

In Code 1, the `uses package` declaration makes the reusable *MagicArtifact* implementation available to the JaCaMo application. Each `artifact` declaration creates an instance of `chon.MagicArtifact`. The first two constructor arguments specify the host and port of the external runtime, while the third identifies the remote artifact represented by that instance. The local artifact names `twitter` and `sentiment` identify the instances within the workspace, whereas `TwitterArtifact` and `SentimentArtifact` identify the corresponding implementations in the external runtime. Although both instances use the same Java class and runtime endpoint, they may expose different artifact interfaces. The external runtime must be reachable when the workspace initializes these instances so that their interfaces can be obtained.

After initialization, a Jason agent joins the workspace, focuses on a declared instance, and invokes its dynamically registered operations as ordinary CArtAgO operations. Signals and observable properties produced by the external implementation are perceived through the same agent-level constructs used with locally implemented arti-

facts. Consequently, the agent does not handle WebSocket connections or JSON messages directly, even though the concrete artifact logic is executed externally.

Interoperability between the JaCaMo application and the external runtime relies on two complementary elements: an Artifact Interface Description and the MagicArtifact Runtime Protocol. The Artifact Interface Description provides a machine-readable representation of the interface exposed by an artifact independently of its concrete implementation, while the protocol defines how this interface is discovered and used by the generic CArtAgO artifact. A runtime implemented in another programming language is compatible with *MagicArtifact* as long as it can provide Artifact Interface Descriptions and exchange messages according to the protocol.

On the external side, the runtime associates each remote artifact identifier with a concrete implementation. This implementation may delegate its functionality to existing services or libraries in the host-language ecosystem and is not required to reproduce the CArtAgO Java API. To be compatible with *MagicArtifact*, the runtime must accept WebSocket connections, provide the interface requested during initialization, dispatch incoming operation requests to the identified implementation, and produce responses according to the MagicArtifact Runtime Protocol.

The Artifact Interface Description identifies the external artifact and declares the operations, signals, and observable properties exposed to JaCaMo. In the current implementation, it is exchanged as a JSON `artifact_manifest` message. Operations specify their names and input arguments, whereas signals and observable properties specify their names and argument types. Code 2 presents a simplified description containing one operation, one signal, and one observable property.

```
{
  "type": "artifact_interface",
  "artifact": "ExampleArtifact",
  "operations": [{"name": "process", "args": ["name": "input", "type": "string"}]  }],
  "signals": [{"name": "processing_done", "args": [] }],
  "observableProperties": [{"name": "result", "args": ["string"]}]
}
```

Code 2. Simplified Artifact Interface Description transported as an `artifact_manifest` message.

The `artifact` value in Code 2 must match the remote artifact identifier supplied as the third constructor argument in the corresponding JCM declaration. In the current implementation, the `operations` array is used to dynamically register the corresponding CArtAgO operations. The `signals` and `observableProperties` arrays describe the results that the external artifact may expose through the CArtAgO interaction model, although these declarations are not yet validated against messages received from the runtime.

The MagicArtifact Runtime Protocol defines how the Artifact Interface Description is obtained and how subsequent interactions are performed. After establishing the WebSocket connection, *MagicArtifact* sends a `runtime_hello` message containing the protocol version and the requested artifact identifier. The runtime responds with the corresponding `artifact_manifest` message, from which *MagicArtifact* registers the declared operations. Subsequent operation invocations are represented by `operation_request` messages containing a `callId`, the artifact and operation iden-

tifiers, and their arguments. The `callId` associates protocol messages produced during the same remote execution.

The runtime may respond with `signal` or `observable_property` messages, which *MagicArtifact* translates into the corresponding CArtAgO constructs. Observable properties may also be removed through `clear_observable_properties`. Messages of type `done` and `error` are translated into the `remote_done` and `remote_error` signals, respectively. These messages report the asynchronous completion or failure of the remote execution; they do not provide native blocking completion or failure semantics for the original CArtAgO operation invocation.

A *MagicArtifact*-compatible runtime must therefore perform three main responsibilities: provide Artifact Interface Descriptions, dispatch incoming `operation_request` messages to the corresponding artifact implementations, and produce responses according to the MagicArtifact Runtime Protocol. The concrete functionality may remain in existing services or libraries of the host language, while the generic Java bridge remains unchanged across artifact implementations. The next section applies this integration procedure to the TypeScript proof of concept.

4. Proof of Concept

This section presents the proof of concept developed to assess the functional feasibility of *MagicArtifact* in a social media monitoring scenario. The source code and supporting material are publicly available in the project repository[2].

4.1. Scenario and Implementation

The application considered in the proof of concept was originally implemented in TypeScript and combines two main functionalities: collecting posts and user replies from a social media service and classifying the sentiment expressed in the collected texts. To introduce cognitive-agent reasoning into this scenario, a Jason agent was created to coordinate the collection and analysis activities while the existing application services remained in the TypeScript ecosystem.

The external artifact runtime was implemented with Node.js and TypeScript and hosts two artifact implementations. `TwitterArtifact` exposes the `collectTweets` operation and delegates data acquisition to `TweetCollectorService`, which communicates with the social media API. The artifact returns the selected post and its replies as signals that can be perceived by the Jason agent. `SentimentArtifact` provides operations for clearing previously stored data, adding collected replies, and requesting their analysis. The artifact stores the received texts and delegates their classification to `SentimentAnalysisService`, which communicates with an external LLM API. The current implementation processes up to twenty replies in each execution and exposes the number of analyzed texts and their individual classifications as observable properties.

Within the JaCaMo application, the agent and both *MagicArtifact* instances are created in the same CArtAgO workspace. One instance represents `TwitterArtifact`, while the other represents `SentimentArtifact`. Both connect to the same external

[2]Available at: `https://github.com/LabRedesCefetRJ/magicArtifactProof`

runtime endpoint, but use different remote artifact identifiers. The Jason agent focuses on these instances and accesses their functionalities through the standard CArtAgO interface, while the concrete collection and analysis logic remains in the TypeScript runtime.

4.2. Prototype Interaction Flow

Both *MagicArtifact* instances in the proof of concept follow the initialization and interaction mechanisms described in Section 3.1. During initialization, each instance establishes a WebSocket connection with the external runtime and obtains the Artifact Interface Description associated with its remote artifact identifier. Based on this description, *MagicArtifact* dynamically registers the corresponding operations in CArtAgO, making the remote interface available to the Jason agent through the standard artifact interaction model.

When the agent invokes a registered operation, the runtime dispatches the request to the selected artifact implementation. Depending on the requested functionality, the remote artifact may invoke an application service or communicate with an external API. Results are returned through the MagicArtifact Runtime Protocol and mapped by *MagicArtifact* to the corresponding CArtAgO constructs.

Figure 2 presents the interaction sequence used by the proof-of-concept implementation. The *Remote Artifact* participant corresponds to either *Twitter Artifact* or *Sentiment Artifact*. The *External Service* participant abstracts the downstream integration used by each implementation: the social media collection service and API in the first case, and the LLM-based sentiment analysis service and API in the second. Although the two artifacts expose different operations and behaviors, both reuse the same initialization, dispatch, and response-handling sequence.

4.3. Execution and Results

The proof of concept was executed using real social media and sentiment analysis services. The external TypeScript runtime established two independent WebSocket connections with the JaCaMo application, one for each *MagicArtifact* instance. During initialization, the runtime returned the corresponding Artifact Interface Descriptions, allowing the first instance to register the four operations exposed by `SentimentArtifact` and the second to register the `collectTweets` operation exposed by `TwitterArtifact`.

After joining the workspace and focusing on both artifact instances, the Jason agent cleared previously stored data and requested the collection of a social media post and its associated replies. Each collected reply was forwarded to `SentimentArtifact` through the `addReply` operation. Once collection was completed, the agent invoked `analyze`. The external artifact used the Gemini 2.5 Flash model to classify the stored texts and returned the analysis count, the individual `sentiment_result` observable properties, and the `analysis_done` signal.

For the reply-level result reported in this study, seventeen replies were considered. Among them, five were classified as positive, ten as negative, and two as neutral. The individual classifications were received through the CArtAgO environment, demonstrating that results produced by the external TypeScript implementation became available to the Jason agent through the standard artifact interaction model.

The execution confirmed that two distinct TypeScript artifact implementations could be exposed through instances of the same generic Java class. It also showed that op-

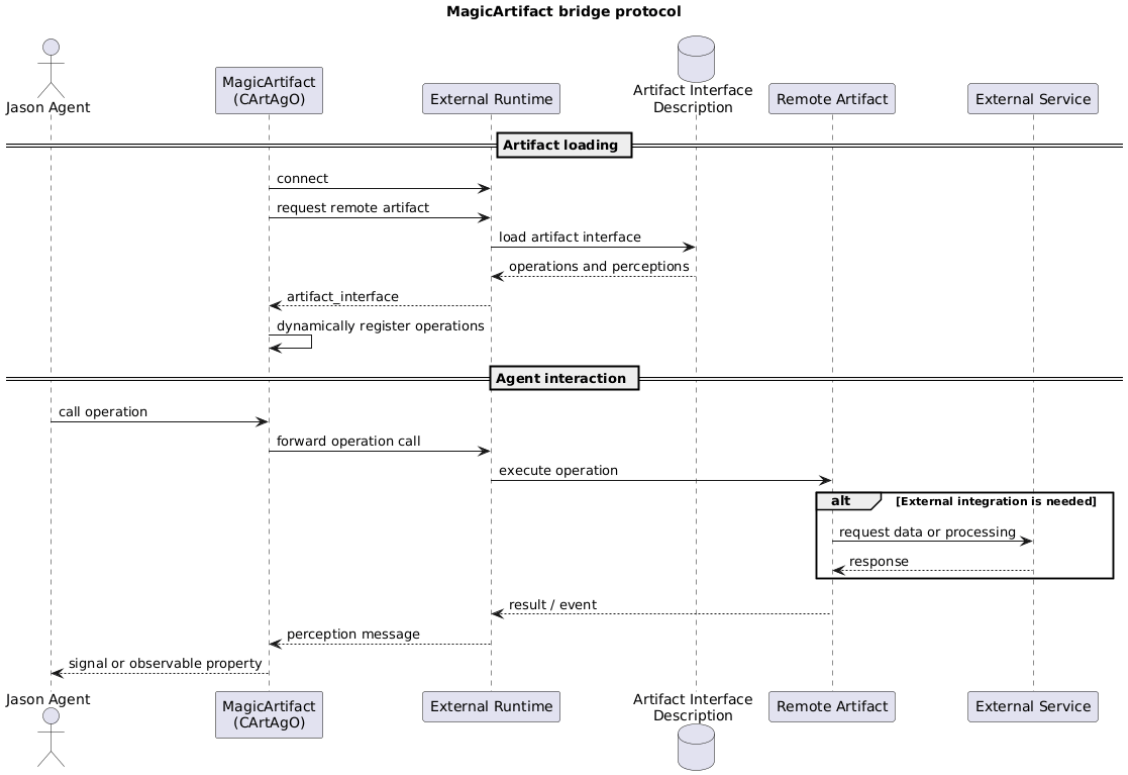

Figure 2. Interaction sequence used by both *MagicArtifact* instances in the proof of concept.

eration calls, signals, observable properties, completion notifications, and analysis results could be exchanged between the Jason agent and the external runtime without application-specific Java artifact classes. These observations demonstrate the functional feasibility of the proposed approach in the evaluated scenario. However, this execution does not constitute a quantitative evaluation of performance, scalability, reliability, or development effort.

5. Qualitative Comparison

To examine how *MagicArtifact* affects the adoption of JaCaMo in an existing application, three implementations of the same social media monitoring scenario were compared qualitatively. The comparison considers the application-specific changes required to introduce a Jason agent and expose collection and sentiment analysis through artifacts, treating the generic bridge and protocol as previously available reusable infrastructure. The implementations were compared according to four criteria: coordination mechanism, artifact interface, location of the application logic, and application-specific integration elements. No quantitative measurements of development time, complexity, or productivity were performed.

In the original TypeScript implementation, a class named `ReflexAgent` directly coordinated collection and analysis, processed the resulting classifications, and maintained the sentiment counters. Despite its name, it was not a Jason agent and did not use BDI reasoning, CArtAgO workspaces, or artifacts in the Agents and Artifacts sense.

Table 2. Qualitative comparison of the three implementations.

Implementation	Coordination and artifact interface	Treatment of the existing TypeScript logic
Original TypeScript	TypeScript controller interacting directly with local TypeScript components.	Collection and sentiment analysis logic used directly, without JaCaMo integration.
Conventional JaCaMo	Jason agent interacting with two application-specific CArtAgO artifacts implemented in Java.	Relevant collection and analysis functionality was implemented again in Java.
JaCaMo with *MagicArtifact*	Jason agent interacting with instances of the same generic Java artifact through Artifact Interface Descriptions.	Existing logic was preserved and adapted in TypeScript. Integration required JCM configuration, Artifact Interface Descriptions, TypeScript adapters, and runtime dispatch, but no application-specific Java artifacts.

In the conventional JaCaMo implementation, coordination was transferred to a Jason agent that joined a workspace, focused on two artifacts, requested reply collection, forwarded the texts for analysis, and processed the resulting observable properties. Exposing these functionalities through CArtAgO required two application-specific Java classes. `TwitterArtifact` implemented social media API communication, parsing, text processing, caching, and signal generation, while `SentimentArtifact` implemented reply storage, prompt construction, language-model communication, response normalization, and observable-property generation. Consequently, relevant functionality previously available in TypeScript was reimplemented in Java.

With *MagicArtifact*, coordination remained in the Jason agent, while the concrete collection and sentiment analysis logic remained in the TypeScript ecosystem. Two instances of the same generic Java artifact represented different remote artifacts, and the existing TypeScript logic was preserved and adapted into services exposed through Artifact Interface Descriptions and protocol-facing artifact classes. This approach still required a Jason agent, JCM configuration, TypeScript adapters, Artifact Interface Descriptions, and runtime dispatch configuration, but avoided application-specific Java artifact classes.

Table 2 summarizes the main differences among the three implementations in terms of coordination, artifact interface, and treatment of the existing TypeScript logic.

The comparison indicates that *MagicArtifact* changes the type of adaptation required when incorporating JaCaMo into an existing non-Java application. It does not eliminate integration work, since Artifact Interface Descriptions, protocol-facing adapters, and runtime configuration are still required. However, it avoids transferring the concrete artifact functionality to application-specific Java classes and allows the original implementation ecosystem to be retained while preserving the interaction model used by Jason agents and CArtAgO artifacts.

The current evaluation is limited to a proof of concept implemented with Node.js and TypeScript. Although the communication protocol is designed to be independent of the external implementation language, this work does not yet validate runtimes developed in other ecosystems. The experiment also considers a single application scenario with two artifacts and does not evaluate systems containing larger numbers of agents, artifacts, workspaces, or concurrent operation requests.

The prototype demonstrates functional integration, but it does not provide quantitative measurements of latency, throughput, scalability, reliability, or development effort. In addition, some implementation decisions remain specific to the current prototype. Re-

mote artifacts and operations are dispatched explicitly by the TypeScript server, the protocol supports a limited set of argument types, and the agent uses a fixed waiting interval before requesting sentiment analysis. Authentication, encrypted communication, reconnection strategies, and recovery from runtime failures were also outside the scope of this evaluation.

6. Conclusion

This paper presented *MagicArtifact*[3], a generic CArtAgO artifact designed to connect JaCaMo agents with artifact implementations executed in external runtimes. The proposal preserves the standard interaction model used by Jason and CArtAgO while separating the artifact interface from its concrete implementation. Through Artifact Interface Descriptions and the MagicArtifact Runtime Protocol, the same Java artifact class can dynamically expose different operations and translate remote responses into signals, observable properties, completion notifications, and errors.

The proof of concept showed that two TypeScript artifacts for social media collection and sentiment analysis could be accessed by a Jason agent without application-specific Java artifact classes. The qualitative comparison also indicated that, unlike the conventional JaCaMo implementation, the proposed approach allowed the existing TypeScript logic to remain in its original ecosystem, although manifests, adapters, and runtime configuration were still required. These results support the functional feasibility of the approach and motivate further evaluation with additional languages, application domains, concurrent executions, and quantitative measurements.

Future work includes implementing external runtimes in additional programming languages, extending the Artifact Interface Description format and protocol to support richer data types and interaction patterns, and replacing fixed synchronization intervals with explicit completion mechanisms. Further experiments should evaluate different application domains, multiple concurrent agents and artifacts, failure conditions, and distributed deployments. Quantitative studies may also compare the conventional CArtAgO implementation process with the use of *MagicArtifact* in terms of integration effort, code reuse, performance, and maintainability.

References

Baiardi, M., Burattini, S., Ciatto, G., and Pianini, D. (2023). JaKtA: BDI Agent-Oriented Programming in Pure Kotlin. In *EUMAS 2023*. DOI: 10.1007/978-3-031-43264-4_4.

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). DOI: 10.1016/j.scico.2011.10.004.

Bordini, R. H. and Hübner, J. F. (2006). BDI Agent Programming in AgentSpeak Using Jason. In *CLIMA 2005*. Springer. DOI: 10.1007/11750734_9.

Fernández Díaz, Á., Earle, C. B., and Fredlund, L.-Å. (2013). eJason: An Implementation of Jason in Erlang. In *ProMAS 2012*. Springer. DOI: 10.1007/978-3-642-38700-5_1.

[3]Available at: `https://github.com/chon-group/MagicArtifact`

Hübner, J. F., Sichman, J. S., and Boissier, O. (2007). Developing organised multi-agent systems using the MOISE+ model: Programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering*, 1(3/4):370–395. DOI: 10.1504/IJAOSE.2007.016266.

Kampik, T. and Nieves, J. C. (2020). JS-son: A Lean, Extensible JavaScript Agent Programming Library. In *EMAS 2019*. Springer. DOI: 10.1007/978-3-030-51417-4_11.

Mellado, A. L. L., Borges, A. P., and Alves, G. V. (2025). MASPY: A Python-Based Framework for Developing BDI Multi-Agent Systems. In *PAAMS 2025*. Springer. DOI: 10.1007/978-3-032-07638-0_18.

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009). Environment Programming in CArtAgO. In *Multi-Agent Programming:Languages, Tools and Applications*. Springer. DOI: 10.1007/978-0-387-89299-3_8.

Ricci, A., Viroli, M., and Omicini, A. (2007). CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS. In *E4MAS 2006*. Springer. DOI: 10.1007/978-3-540-71103-2_4.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. John Wiley & Sons.