Intelligent Application Deplyoment with MAMS

Maximillian Schöll and Rem W Collier[0000−0003−0319−0797]

School of Computer Science, University College Dublin, Ireland
maximillian.scholl@ucdconnect.ie, rem.collier@ucd.ie

Abstract. This paper presents a novel Multi-Agent MicroServices (MAMS) framework, centered on a formal deployment ontology. This ontology models the environment as a dynamic knowledge graph, separating deployment intent from runtime reality. We demonstrate how Belief-Desire-Intention (BDI) agents autonomously reason over this graph to assemble, deploy, and manage containerized systems. Agents explore the environment via a hypermedia-driven API following Hypermedia as the Engine of Application State (HATEOAS) principles, react to system-wide events using WebSub notifications, and coordinate tasks through direct messaging. This approach enables agents to translate high-level goals into low-level Docker API commands, providing a practical blueprint for bridging the gap between declarative agent reasoning and the imperative nature of modern cloud infrastructure.

Keywords: Agent-Oriented Programming · Hypermedia Multi-Agent Systems.

1 Introduction

The growth of cloud native and edge computing has led to increasingly complex, distributed systems [28,30] that require sophisticated deployment and management strategies. Traditional automation methods, such as imperative scripts and declarative configurations, often fall short in dynamic environments [29] necessitating human intervention to adapt to unforeseen circumstances.

Recent work on Multi-Agent MicroServices (MAMS) has established agents as first class, containerized microservices [32] that have the potential to enable the use of agents to autonomously manage the deployment and lifecycle of other microservice ecosystems. This paper presents **MAMS-Deploy**, a framework that seeks to do this through the combination of web standards, semantic knowledge representation, and BDI reasoning.

A distinctive property of our approach that emerges from the adoption of MAMS is its recursive nature: because agents are themselves containerized microservices, the framework could potentially deploy new agents in response to workload changes, geographic distribution needs, or fault recovery scenarios. This means the orchestration layer is not static; it can grow, shrink, or relocate itself without human intervention. Further, the same approach can be applied seamlessly to the deployment of both MAS, traditional distributed systems or any combination of the two.

This recursive capability is particularly valuable in environments where human intervention is costly, dangerous, or impossible [17], [15]. In such cases, lightweight orchestration agents may be deployed alongside application services at the network edge, for example, on satellites, autonomous underwater vehicles, or polar research stations, to manage local resources and maintain service continuity. Examples include space missions, where communication delays and physical inaccessibility prevent real-time human control [8], [4]; deepsea or polar research stations, where harsh conditions limit maintenance opportunities [33]; disaster recovery zones, where safety concerns restrict human presence [25]; and large-scale industrial IoT deployments, where the cost of on-site intervention is prohibitive [18]. In these contexts, the ability of the orchestration layer to autonomously expand, relocate, and repair its distributed components reduces the need for costly or dangerous human intervention and ensures sustained operation even when connectivity to central control is intermittent or delayed.

The main contributions of this paper are: the definition of an ontology that provides a rich, machine-readable vocabulary for describing microservices, their dependencies, and infrastructure targets (Section 4). A high-level architecture that supports the deployment and management of distributed systems based on Semantic Web and Hypermedia APIs (Section 3), a prototype implementation of that architecture (Section 5) and a corresponding evaluation (Section 6).

2 Background

Traditional approaches to managing the deployment of containerised applications are based on the use of well established microservices patterns. Newman [22] provides a canonical guide to such patterns, outlining the key challenges in service discovery, configuration, and monitoring that an autonomous system must manage. Traditionally, these challenges are addressed with declarative orchestration platforms like Kubernetes[1] and static service registries, such as Consul[2] and Eureka[3]). While powerful, these tools rely on human-defined rules and lack the adaptive reasoning needed for true autonomy [21] [13].

Automated management of deployed applications is the ultimate goal of autonomic computing, which advocates for systems capable of self-management through self-configuration, self-healing, and self-optimization [20]. This vision provides the conceptual framework for the "why" of agent-driven deployment. Agents, with their inherent goal-driven reasoning, are natural candidates for implementing these autonomic properties. Recent advances in LLM-based agents are turning this vision into a practical reality. Yu et al. (2025) [34] demonstrate with ServiceOdyssey how a self-learning agent can autonomously master microservice management tasks using curriculum learning and multi-layered feedback, progressing from simple observation to complex actions. Similarly, the Agentic AIOps framework proposed by Zota et al. (2025) [35] formalizes the role

[1] https://kubernetes.io/
[2] https://developer.hashicorp.com/consul
[3] https://github.com/Netflix/eureka

of agents in IT operations, defining a clear path from automated monitoring to autonomous resolution.

The increasing complexity of deployed applications has led in a shift in concerns from simple monitoring of deployed systems (understanding what is happening) to the notion of observability [2] (understanding why it is happening). Observability has led to the emergence of open standards, such as OpenTelemetry [4] that supports the capture and linking of metrics, event/log data, and traces in order to support the explanation of system behaviour via the purpose-built dashboards developed using tools such as Grafana Dashboards [7]. More recently, we have seen the emergence of knowledge graphs as tools for better representing the relationship between OpenTelemetry artifacts [5] not only providing better tools for human observation, but also opening, through techniques such as Semantic Retrieval Augmented Generation (Semantic RAG) the potential for not only LLM-driven human-in-the-loop observation tools but also Agentic tools that can automate the observation of deployed systems, integrating with the AIOps frameworks that are emerging.

3 The MAMS-Deploy Approach

This paper advocates the use of Hypermedia Multi-Agent Systems (hMAS) [16] as a means to delivering frameworks for intelligent management and deployment of applications. Hypermedia MAS are a class of MAS that operate in Hypermedia Environments with the aim of addressing the use of agents as per the original vision of the Semantic Web [10]. Such environments embody the principles of REpresentational State Transfer (REST) [14], and in particular Hypermedia As The Engine Of Application State (HATEOAS), enabling clients - in this case agents - to navigate a web of services by following links embedded within resource representations, eliminating the need for hard-coded knowledge of service endpoints or interfaces [27]. This principle is central to engineering world-wide MAS [9].

MAMS-Deploy gets its name from the decision to adopt the MAMS architectural style in its design [32]. MAMS is an approach to integrating agents into microservices architecture that is well suited to the intelligent management and deployment of applications because Microservices are the predominant approach to implementing applications. According to Market Growth Reports[6] more than 62% or Fortune 500 companies used containerized microservices and the cloud microservices market is projected to be worth more than 1.3B US$ in 2026 alone. MAMS introduces a new class of Agent-Oriented Microsevices (AOMS) that can be easily integrated with Plain Old MicroServices (POMS) simplifying the adoption and deployment of agents. This allows us to treat the MAMS-Deploy

[4] https://opentelemetry.io/
[5] https://grafana.com/events/observabilitycon/agenda/2025/grafana-cloud-knowledge-graph-root-cause-analysis/
[6] https://www.marketgrowthreports.com/market-reports/cloud-microservices-market-106525

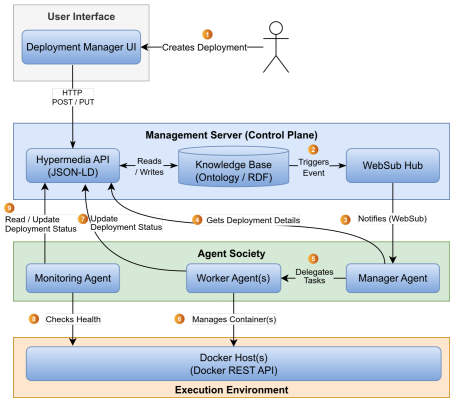

Fig. 1. The MAMS-Deploy architecture illustrating the flow from human intent (1) to reactive agent notification (2-3), planning (4-5), execution (6), and continuous monitoring (8-9).

agents as containerized microservices, that are able to manage the lifecycle of other (possibly agent-oriented) microservices. The proposed solution, which is presented through the architecture diagram in Figure 1 builds on this and four key design principles:

- **Reactive Coordination:** Agents react to events using the W3C WebSub standard [7], eliminating the need for polling and creating a real-time, event-driven system.
- **Semantic Knowledge:** A formal RDF ontology provides a shared, machine-readable understanding of the deployment domain, decoupling agent logic from specific service implementations.
- **Web-Native Integration:** All interactions occur over standard HTTP and a RESTful API that adheres to HATEOAS principles. Agents discover actions and resources at runtime.
- **Distributed Autonomy & Observability:** Agents make independent, goal-oriented decisions, while a human-centric UI provides a transparent window into their state and actions.

The architecture is composed of four distinct layers that enable the flow from human intent to autonomous execution and monitoring. The system is designed around a central, declarative knowledge base, with agents acting as intelligent controllers that perceive and act upon this shared world model.

At the top level, a **User Interface** allows the human operators to initiate process by interacting with the **Deployment Manager UI**. Through this, they define high-level deployment goals that are translated into triples that are sent to the **Management Server** via a HTTP POST request (1). This layer serves

[7] https://www.w3.org/TR/websub/

as the systems authoritative control plane acting as the central hub for state management and event coordination. It combines:

- A **Knowledge Base** modelled as a RDF triple store that contains the authoritative state of the entire system, structured by our deployment ontology.
- A **Hypermedia API**, which serves as the sole gateway to the Knowledge Base, exposing its contents as linked data (JSON-LD).
- A **WebSub Hub** that publishes changes in the Knowledge Base as events, allowing agents to subscribe to relevant topics. This enables real-time notifications of changes, such as new deployments or updates to microservice states (2).

The **Agent Society** layer contains the MAMS-Deploy agents which are responsible for realising the high-level deployment goals of the users. In line with the MAMS philosophy [32], each agent is itself a containerized microservice. This makes the agent society inherently distributed; each agent operates as an independent process and can be deployed anywhere, from a central server to the edge hosts they manage, allowing for a highly scalable and flexible topology. The society consists of various agents, whose actions correspond to the numbered steps in Figure 1:

- The **Manager Agent** subscribes to the WebSub hub and is notified of new deployments (3). It then queries the Hypermedia API to get the full deployment details (4) and delegates specific microservice deployment tasks to available workers (5).
- The **Worker Agent(s)** start managing the container lifecycle upon receiving a task by interacting directly with the Docker REST API (6). After execution, it updates the Knowledge Base via the API (7). This involves updating the status of the Deployment resource and creating a new `ContainerInstance` resource to capture the reality of the running container.
- The **Monitoring Agent** provides the system's self-healing capabilities. It periodically checks the health of running containers directly against the Docker API (8) and compares this with the desired state in the Knowledge Base. If it detects a discrepancy or failure, it updates the relevant `ContainerInstance` status via the API (9), potentially triggering a recovery plan.

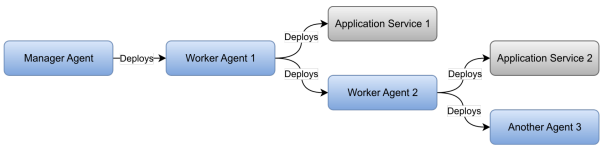

Fig. 2. The recursive nature of the MAMS-Deploy framework, where agents can deploy other agents to manage the orchestration layer itself.

Treating agents as containerized microservices implies that the system can deploy its own orchestration components. For example, a Manager Agent could instruct a Worker Agent to deploy additional Worker Agents on new servers to handle increased load, or to replace failed agents. This recursive capability allows the orchestration layer to adapt its own topology dynamically, enabling self-scaling and self-healing not only for application services but also for the orchestration infrastructure itself, as is shown in Figure 2.

Finally, the **Execution Environment** layer consists of one or more Docker Host(s). All interactions with this layer are performed by the agents via the standard Docker REST API, ensuring broad compatibility without requiring custom client-side software on the hosts.

4 The Deployment Ontology

A core contribution of this work is a formal OWL ontology that provides the semantic foundation for agent reasoning. It is exposed to the agents via the Management Server's hypermedia API as JSON-LD. The ontology's design is predicated on a crucial principle for autonomic systems: the explicit separation between the desired state (the intent) and the actual state (the reality). This allows agents to perceive the gap between what should be and what is, and to formulate plans to close that gap. Figure 3 illustrates the key relationships.

The separation of concerns highlighted above is primarily modelled through three core classes:

- **dep:Deployment**: The top-level resource representing a complete application or system to be deployed. It acts as a container for a collection of `dep:Microservice` resources.
- **dep:Microservice**: This class represents the blueprint of a service. It is a static resource that defines the intent for a deployment, including the Docker `imageName`, `imageTag`, `environmentVariables`, the `portMapping`, and the intended `targetServer`. It describes what should be.
- **dep:ContainerInstance**: This class represents the realization of that intent, an actual, running container. It is a dynamic resource created by a Worker Agent after a successful deployment. It captures the reality of the runtime, including the unique `containerId`, its live `instanceStatus` (e.g., `dep:Running`, `dep:Exited`), and a link back to the `dep:Microservice` definition it fulfills via the `dep:hasInstance` property.

This distinction is fundamental. The Monitoring Agent, for example, operates by comparing the desired state of `dep:Microservice` resources with the actual state of their corresponding `dep:ContainerInstance` resources. Any discrepancy triggers a self-healing plan.

To ground the ontology in a familiar context, we can map its concepts to those found in a standard `docker-compose.yml` file. This is illustrated through the simplified WordPress `docker-compose.yml` file presented in Figure 4 alongside the equivalent semantic description modelled using our ontology. Docker

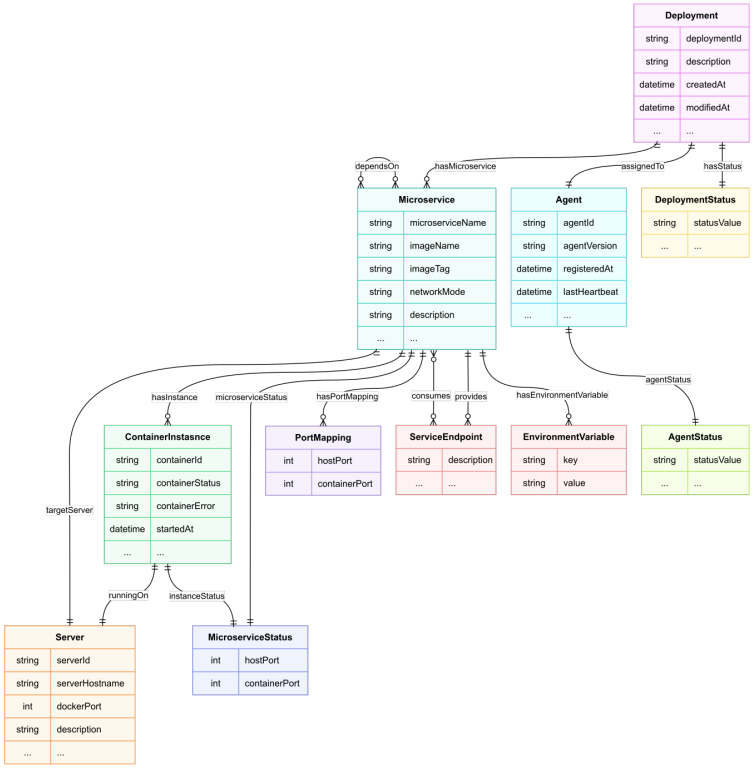

Fig. 3. Key classes and relationships in the deployment ontology.

Compose provides a declarative YAML format for defining multi-container applications. The ontology serves as a semantic, machine-readable equivalent, transforming a static configuration file into a dynamic knowledge graph that agents can reason over.

Table 1 explicitly maps the key concepts between the two representations. The ontology not only mirrors the declarative structure but enriches it with formal semantics that enable advanced agent reasoning. For example, instead of a hardcoded service name like `DB_HOST=db`, the ontology uses an abstract `dep:ServiceEndpoint`. This allows an agent to dynamically discover and bind to any service that fulfils the required capability (`se:mysql-connection`).

The ontology supports two complementary mechanisms for expressing relationships between microservices:

- **Direct Dependencies:** The `dep:dependsOn` property creates an explicit ordering constraint between two `dep:Microservice` definitions. For example, a web application may declare that it `dep:dependsOn` a database service, ensuring that the database is deployed and running before the application is started. This is a simple, static relationship that enforces deployment order.

– **Abstract Service Endpoints:** While direct dependencies capture ordering, they do not describe what a service provides in a reusable way. To address this, the ontology introduces `dep:ServiceEndpoint`, an abstract identifier for a capability or interface that a service offers. A microservice can declare that it `dep:provides` a given endpoint, and other services can declare that they `dep:consume` it. This indirection decouples consumers from specific providers and removes the need for hardcoded service names or addresses

To illustrate this, look at the example on the right of Figure 4. On line 17, the MySQL microservice `ms:wordpress-db` declares that it `dep:provides` the abstract endpoint `se:mysql-connection`. Following this, on line 23, the WordPress application microservice `ms:wordpress-app` declares that it `dep:consumes` this endpoint and uses it in an environment variable template (see line 28).

When deploying `wordpress-blog-app`, the agent identifies that it consumes `se:mysql-connection`; locates the deployed `dep:ContainerInstance` of the MySQL service that provides this endpoint; retrieves the hostname from the instance's `dep:runningOn` server, and substitutes the value into the environment variable before creating the container. This allows agents to dynamically bind services at runtime without hardcoded addresses. By combining direct dependencies for ordering and abstract endpoints for capability matching, the ontology enables agents to reason about both when and how services should be connected.

The MAMS-Deploy environment is also incorporated into the model through the **dep:Server** and **dep:Agent** classes. The **dep:Server** class represents a host machine where containers can be deployed, identified by its `serverHostname`

```
1  services:
2    db:
3      image: mysql:8.0
4      environment:
5      - MYSQL_ROOT_PASSWORD=rootpassword123
6      - MYSQL_DATABASE=wordpress
7      ports:
8      - "3306:3306"
9
10 wordpress:
11  image: wordpress:latest
12    depends_on:
13    - db
14    environment:
15    - DB_HOST=db
16    - DB_NAME=wordpress
17    ports:
18    - "8080:80"
```

```
1  @prefix dep: <http://.../deployment-ontology#> .
2  @prefix ms: <http://.../microservices/> .
3  @prefix srv: <http://.../servers/> .
4  @prefix se: <http://.../service-endpoints/> .
5
6  ms:wordpress-blog-mysql
7    rdf:type dep:Microservice ;
8    dep:imageName "mysql" ;
9    dep:imageTag "8.0" ;
10   :environmentVariable
11     "MYSQL_ROOT_PASSWORD=rootpassword123",
12       "MYSQL_DATABASE=wordpress" ;
13   dep:hasPortMapping [
14     dep:hostPort 3306 ;
15     dep:containerPort 3306
16   ] ;
17   dep:provides se:mysql-connection .
18
19 ms:wordpress-blog-app
20   rdf:type dep:Microservice ;
21   dep:imageName "wordpress" ;
22   dep:imageTag "latest" ;
23   dep:dependsOn ms:wordpress-blog-mysql ;
24   dep:consumes se:mysql-connection ;
25   dep:environmentVariable
26     "DB_HOST={se:mysql-connection.host}",
27     "DB_NAME=wordpress" ;
28     dep:hasPortMapping [
29       dep:hostPort 8080 ;
30       dep:containerPort 80
31     ] .
```

Fig. 4. Example Docker Compose file (left) and its MAMS-Deploy equivalent (right).

Docker Compose (YAML)	**Ontology Equivalent (Turtle)**
`services:` `  db:`	`ms:mysql rdf:type` `  dep:Microservice .`
`image: mysql:8.0`	`dep:imageName "mysql";` `dep:imageTag "8.0" .`
`environment:` `- VAR=val`	`dep:environmentVariable` `  "VAR=val" .`

`ports:`
`- "3306:3306"`
	`dep:hasPortMapping [`
	`  dep:hostPort 3306; ...`
	`] .`

`depends_on: [db]`	`dep:dependsOn ms:mysql .`
Implicit Service Name	`dep:provides se:mysql-connection .`
Hardcoded Hostname	`dep:consumes se:mysql-connection .`

Table 1. Mapping Docker Compose Concepts to the Ontology

and API port, while the**dep:Agent** class represents the actors in the system and links to their current status (e.g., `dep:Idle`, `dep:Busy`).

Finally, to ensure observability, all states are modelled as individuals of status classes (e.g., `dep:Running`, `dep:Failed`, `dep:Busy`). When a failure occurs that an agent cannot resolve, it updates the status of the relevant resource to `dep:Failed` and also populates the `dep:errorMessage` data property with a detailed message from the underlying system (e.g., the Docker API). This makes the reason for failure a queryable part of the knowledge graph, which is then surfaced to the human operator via the UI.

5 Prototype Implementation

To demonstrate how the ontology described in Section 4 can be used to deliver the approach outlined in Section 3, a prototype system has been developed using the reference implementation of the MAMS architectural style [24] that has been implemented using ASTRA [12], a BDI-style programming language based on AgentSpeak(L), and CArtAgO, a framework for building agent environments based on the Agents & Artifacts conceptual model [26]. The approach is inspired by prior applications of MAMS [23,3].

Section 5.1 focuses the design of the Hypermedia API while Sections 5.2 to 5.4 provide details on the design of the agents that consume the API in order to realise the deployment design goals. Details on the User Interface can be found in Appendix C. The prototype source code is available on GitLab [8].

5.1 The Hypermedia API

While the ontology defines the conceptual model of our system, it is the hypermedia API that makes this model a navigable, interactive environment for

[8] https://gitlab.com/mams-ucd/mams-deploy

the agents. This API is the part of the Management Server that exposes the knowledge graph (a collection of all deployments, microservices and their properties) as a set of Hypermedia resources that agents can interact with in order to understand the state of the system. JSON-LD [1] is used as the primary representation, meaning that each JSON object is not just a data structure but a piece of a linked-data graph that can easily be navigated by the agents. The navigable entry points, collections, item resources, and action forms follow the sitemap summarized in Figure 5.

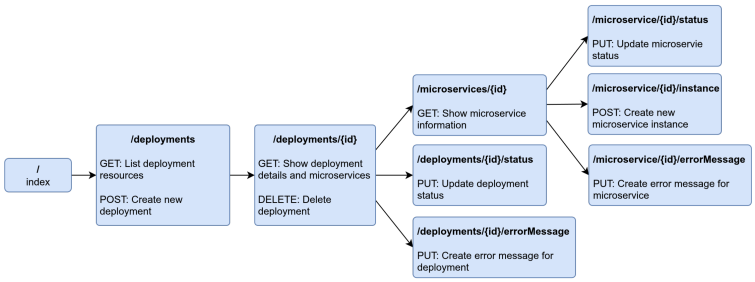

Fig. 5. HATEOAS sitemap for the Management Server API.

The left hand side of Figure 6 presents an example of the root entry point to the API which allows the consumer to discover two things: the URI for the deployments collection and the URI for the ontology itself. Agents can navigate from deployments to their constituent microservices, and from microservices to the target server by following the hypermedia links provided in the JSON-LD responses. The approach is based on approaches used in **WoT Thing Descriptions**[9] and the associated **WoT Hypermedia Controls Ontology**[10] and is not limited to navigation; it is also how agents learn to modify the state of the system. Actions are exposed through hypermedia forms embedded within the resource representations. These forms provide agents with all the necessary metadata to construct a valid state-changing request. For example, after a Worker Agent has started a container, it needs to update the status of the corresponding Microservice resource. By inspecting the resource's representation (the right hand side of Figure 6), it discovers a form for this purpose.

This form tells the agent everything it needs to know to perform the action: the **href** field identifies the target URL for the request; the **htv:methodName** field identifies the HTTP method to use (PUT); the **wot:op** field provides a semantic identifier for the operation, which the agent can use in its reasoning logic; and the **properties** field lists the required fields for the request body, which, in this case, is a `dep:microserviceStatus` property.

[9] https://www.w3.org/TR/wot-thing-description11/
[10] https://www.w3.org/2019/wot/hypermedia

```
1 {
2  "@context": { ... },
3  "@id": "http://192.168.2.10:5000",
4  "@type": "schema:EntryPoint",
5  "links": [{
6   "href": "http://.../deployments",
7   "rel": "collection"
8  }, {
9   "href": "http://.../deployment-ontology",
10   "rel": "describedby"
11  }]
12 }
```

```
1 {
2  "@id": "http://.../microservices/mysql-database",
3  "forms": [{
4   "href":
5    "http://.../microservices/mysqldatabase/status",
6   "contentType": "application/json",
7   "htv:methodName": "PUT",
8   "wot:op": "updatemicroservicestatus",
9   "properties": [{
10    "name": "dep:microserviceStatus",
11    "required": true
12   }]
13  }]
14 }
```

Fig. 6. The API Entry Point (/) response in JSON-LD (left) and excerpt of a hypermedia form for updating a microservice's status (right).

The agent can infer that it needs to send a PUT request to the specified href with a JSON body containing the new status. This hypermedia-driven approach allows agents to adapt to changes in the API without requiring code changes, as they discover available actions and resources at runtime.

5.2 Agent Navigation and Interaction

The framework's autonomy is realized through a collaborative navigation and interaction process between the Manager and Worker agents. They navigate the environment by following the hypermedia affordances and action forms summarized in the HATEOAS sitemap (Fig. 5) using the following interaction sequence:

1. **Task Discovery (Manager Agent)**: The Manager Agent is triggered reactively by a WebSub notification for a newly created deployment. It parses the response to identify any deployments with a status of `dep:Pending`.
2. **Task Delegation (Manager Agent)**: The Manager Agent selects an available (`dep:Idle`) Worker Agent for each pending deployment and delegates the deployment task that Worker through a messages that includes the specific URI of the deployment resource (e.g., `/deployments/wordpress-blog`). This delegation by URI is inherently RESTful and decouples the Manager from the execution details.
3. **Information Gathering (Worker Agent)**: The Worker Agent receives the deployment URI. It performs a GET request on this resource and extracts the list of URIs for all constituent `dep:Microservice` resources that belong to this deployment from the response.
4. **Plan Formulation (Worker Agent)**: The Worker Agent iterates through the list of microservice URIs and performs a GET request for each one to retrieve its full definition. It adds all retrieved properties such as image name, dependencies, environment variables and target server to its local belief base. Once it has gathered all the information, its BDI reasoning engine formulates a concrete execution plan, respecting the `dep:dependsOn`, `dep:provides` and `dep:consumes` constraints to determine the correct deployment order.

5. **Execution and State Update (Worker Agent)**: The Worker executes the plan, updating the shared knowledge graph after each microservice deployment attempt. On success, it POSTs a new `dep:ContainerInstance` resource to the server. On failure, it PUTs an update to the `dep:Microservice` resource, setting its status to `dep:Failed` and adding a `dep:errorMessage`.
6. **Finalization (Worker Agent)**: Once all microservices have reached a terminal state (`dep:Running` or `dep:Failed`), it performs a final PUT request, updating the parent `dep:Deployment` resource's status to `dep:Deployed` or `dep:Failed`, completing the workflow.

The Monitoring Agent runs independently of the Manager and Worker agents. It is responsible for observing the current state of the deployed microservices and alerting the system. It periodically monitors the Knowledge Graph for information on available deployments. For each deployment it gathers information on the expected state of the underlying containers and checks that each container is in the expected state. If a container is not in the expected state, then the Monitoring Agent attempts to restart it. If it fails 3 times, then it updates the status of the container in the Knowledge Graph to reflect that it is failed. The act of changing the state of the container in the Knowledge Graph triggers an update that is passed to the Manager agent using WebSub.

5.3 WebSub-Based Reactive Coordination

A custom WebSub library has been implemented for ASTRA enabling agents to become fully reactive participants in a web-native ecosystem. An agent first initializes the module (line 2), which starts an embedded web server to listen for callbacks from the hub. The agent's callback URL is automatically exposed based on its name. It can then subscribe to any topic offered by the hub (lines 5-8). Once subscribed, the agent uses plans such as the one defined on lines 11-14 to handle incoming event notifications.

```
1 rule +!main(list args) { // agent's initial goal
2   !websub_init("http://localhost", 8080);
3   // Starts a server and exposes the callback URL
4   // http://localhost:8080/worker1
5   !websub_subscribe("http://192.168.2.10:5000/websub",  // Hub URL
6                      "deployment.created");              // Topic URL
7 }
8 rule $websub.event(string topic, string content, string contentType) :
9   strings.endsWith(topic, "deployment.created") {
10  // Actions to be performed on event notification
11 }
```

Fig. 7. ASTRA agent initializing and subscribing to WebSub together with example rule to handle incoming WebSub events.

```
rule $knowledgeStore.read(string dep_uri) :
    strings.contains(dep_uri, "/deployments") {
   // Iterate through microservices just
   foreach(dep.hasMicroservice(dep_uri, string uri)) {
      // Add belief for each microservice
      +microservice(uri);
   }
   // Trigger the goal to start deployment
   !deploy_microservice();
}
// Goal: Deploy a microservice
// Deploy microservices that have no dependencies
rule +!deploy_microservice() :
  microservice(string ms_uri)
  & microservice_image(ms_uri, string image, string tag)
  & ~microservice_depends_on(ms_uri, string _1)
  & ~microservice_deployed(ms_uri, true) {
    !deploy_microservice(ms_uri);
}
```

Fig. 8. Agent rule for processing deployment resources.

5.4 Semantic Knowledge Processing and Planning

Consumption of the JSON-LD representations by ASTRA agents is handled through the `KnowledgeStore` module which acts as a wrapper around an Apache Jena Triple Store[11]. As described in [23], this module integrates the triples into ASTRA reasoner allowing the agent to reason across its private beliefs and any facts contained in the triple store. This module provides actions to allow the agent to consume the JSON-LD representation associated with a given URL (it can handle multiple Semantic Web formats) and notifies the agent when the resource has been consumed. For example, when an agent consumes a deployment resource, the `KnowledgeStore` inserts a set of `dep:hasMicroservice` triples linking the deployment to its constituent services which it can reason about as is illustrated through the first plan in Figure 8. This rule iterates through each `dep:hasMicroservice` fact and adds a new, belief, `+microservice(uri)`, for each service URI it discovers.

The agent then uses a processing rule, shown in Figure 8, to react to this new information. Specifically, the agent transforms the declarative knowledge from the environment into a curated set of facts that subsequent planning rules, like the second plan in Figure 8, can easily query. This two-step process, parsing the environment into semantic triples and then refining them into simple, actionable beliefs is what enables the agent to autonomously build its deployment plan.

[11] https://jena.apache.org/

6 Evaluation

To evaluate MAMS-Deploy, a sample Wordpress deployment[12] is created an used to validate the core claims of the MAMS-Deploy framework: its capacity for autonomous reasoning and goal-driven resilience. Our evaluation is structured into two main qualitative assessments. Section 6.1 verifies the framework's ability to correctly execute deployment plans under normal operating conditions while Section 6.2 evaluates the system's robustness by introducing runtime failures. These scenarios test both reactive and proactive self-healing capabilities, as well as the system's ability to diagnose and transparently report unrecoverable errors. To ensure a realistic and non-trivial benchmark, all tests were conducted using the same WordPress deployment, and each scenario was executed following the information flow depicted in the system architecture (Fig. 1). Detailed execution flows for each scenario are provided in Appendix B.

6.1 Correctness and Scalability

Semantic Reasoning & HATEOAS Navigation: This test validated that an agent can autonomously navigate the hypermedia API to gather information, formulate a plan respecting dependencies, and execute it. The outcome was successful, confirming the division of labor (Manager/Worker) and the agent's ability to derive a multi-step plan directly from the hypermedia environment. (App. B-1)

Reactive Self-Healing: This test evaluated an agent's ability to detect and correct a dependency failure *during* its own execution plan. The agent successfully detected a stopped dependency, executed a corrective restart, and resumed its original plan, demonstrating goal-driven resilience. (App. B-2)

Proactive Self-Healing: This test focused on the dedicated Monitoring Agent's ability to correct failures *post-deployment*. The agent successfully detected a state mismatch, restored the system to its desired state, and correctly reported unrecoverable crash loops, ensuring system observability. (App. B-3)

Failure Diagnosis & Reporting: This scenario assessed the system's ability to handle unrecoverable errors. The system correctly diagnosed a port conflict, prevented dependent deployments, and made the specific error reason queryable in the knowledge graph, demonstrating transparency. (App. B-4)

Scalability via Parallelism: This test demonstrated that deployment throughput increases by distributing independent tasks across multiple Worker Agents. Parallel execution of two deployments significantly reduced total completion time compared to sequential execution, confirming the architecture's scalability. (App. B-5)

[12] The sample RDF deployment document can be found on Gitlab both at the following URL: https://gitlab.com/mams-ucd/mams-deploy/-/blob/main/management/temp/deployment-wordpress-blog.ttl and in Appendix A. This directory also includes two other sample RDF deployment documents

6.2 Resilience and Failure Handling

A key measure of an autonomous system's utility is its behavior in the face of unrecoverable errors. The "Failure Diagnosis and Reporting" scenario (App. B-4) specifically tested this by defining a deployment with a port conflict, an issue the agent cannot resolve on its own. When the Worker Agent attempted to create the container, the Docker API returned a '409 Conflict' error. The agent's web modules captured this response. Recognizing that this was not a transient error it could fix (unlike a stopped container), the agent correctly abandoned its deployment plan for that microservice. Crucially, it then performed two actions to ensure system transparency:

1. It performed a `HTTP PUT` request to the `dep:Microservice` resource, updating its status to `dep:Failed`.
2. It populated the `dep:errorMessage` data property with the exact error message received from the Docker API: "port is already allocated."

This process makes the failure transparent and auditable. The reason for the failure becomes a queryable fact in the shared knowledge graph, which is then surfaced to the human operator via the UI. This prevents silent failures and provides clear, actionable diagnostic information, which is essential for building trust in the system's autonomy.

6.3 Comparison with Traditional Orchestration

While industry-standard orchestrators like Kubernetes are powerful, mature platforms, our framework explores a different paradigm focused on higher-level, goal-driven autonomy. As a research prototype, a direct feature-by-feature comparison is premature. Instead, the value lies in contrasting the foundational principles that motivate our agent-based approach. Traditional orchestrators exhibit several inherent limitations in highly dynamic environments:

- **No Semantic Reasoning**: They operate on static manifests (YAML/JSON) without an explicit, queryable knowledge model of the deployment domain.
- **Limited Adaptability**: Recovery is typically limited to predefined restart policies. They cannot diagnose the cause of novel failures or adapt their strategy dynamically [5].
- **Static Control Topology**: The orchestration layer itself is fixed and cannot autonomously expand, relocate, or replace its own control components [5].
- **Human-Centric Change Management**: Any significant change in deployment intent requires human intervention to modify and reapply manifests [31,6].

Table 2 provides this conceptual comparison. Traditional orchestrators excel at enforcing a desired state defined by a human in a static manifest. Our system,

Aspect	Traditional Orchestration	MAMS-Deploy Framework
Decision Making	Rule-based execution of human-authored YAML/JSON files.	Autonomous, goal-driven reasoning using BDI agent architecture.
Adaptability	Limited to predefined rules (e.g., restart policies). Cannot adapt strategy to novel failures.	Designed for autonomous adaptation. Agents can diagnose failures, re-plan, and change strategy at runtime.
Knowledge Storage	Static YAML/JSON configuration files.	A dynamic, queryable RDF knowledge graph (ontology) that evolves with the system.
Service Discovery	Built-in DNS (Kubernetes) or network-based resolution.	Hypermedia navigation (HATEOAS) and semantic endpoint resolution.
Coordination	Centralized control plane (e.g., Kubernetes API Server).	Distributed multi-agent collaboration via web standards (WebSub, REST).
Coordination	Centralized control plane (e.g., Kubernetes API Server).	Distributed multi-agent collaboration via web standards (WebSub, REST).
Failure Handling	Reactive (restarts failed containers based on health probes).	Diagnostic and proactive. Agents can reason about the cause of failure, attempt corrective actions, and escalate unrecoverable issues.
Human Interaction	Humans write and apply declarative manifests (the "what" and "how").	Humans define high-level goals; agents autonomously determine the "how" and adapt plans dynamically.
Recursive Deployment	Not applicable — orchestrators cannot deploy themselves.	Agents can deploy other agents, enabling self-expansion, self-replication, and adaptive scaling of the orchestration layer itself.

Table 2. Comparison of Deployment Paradigms

by contrast, is designed to *derive* and *autonomously adapt* the plan to achieve a high-level goal, even in the face of unexpected environmental changes.

The most significant differentiator is the framework's **recursive nature**. Because agents are themselves containerized microservices, they can deploy other agents. This enables the orchestration layer itself to grow, shrink, or relocate without human intervention. This capability opens the door to self-propagating systems that could bootstrap entire distributed infrastructures from a minimal seed, a concept beyond the scope of traditional, statically-defined orchestrators.

7 Conclusions

This paper has presented **MAMS-Deploy** a novel agent-based framework for managing and deploying container based systems that uses Hypermedia APIs to expose services and knowledge that can be easily consumed by BDI agents. The primary value of this approach is its inherent adaptability, which contrasts with the rigidity of static configuration files that require human intervention. The proposed ontology enables the creation of a dynamic, machine-readable knowledge graph of the deployment environment. This not only enables the agents' current reasoning but also opens the door for higher levels of abstraction.

As discussed in Section 2, recent trends (under the name AIOps) include the use of Large Language Models (LLMs) that are not only able to transform operator intents into deployment action but also able to help in the diagnosis or failures or the optimisation of the system through the introduction of tools and frameworks to support not only monitoring (what has happened), but also observability (why it happened) [2]. .

State of the art industry approaches, such as Grafana Cloud Knowledge Graph, currently build the knowledge graph bottom up inferring, for example, what Kuberenetes pods and clusters exist through analysis of the metrics, events, logs and traces captured from OpenTelemetry feeds. In contrast, our approach is more integrated in that the knowledge graph constructed explicitly incorporates these higher level features, exposing not only the deployed components but also the configurations of those components. The other main difference is our adoption of Semantic Web technologies over the Neo4j solution used by Grafana. Both approaches are `LLM ready`, with Grafana already providing an AI assistant that allows users to explore the knowledge graph through a human friendly interface. Similar features could be easily integrated into our prototype through the creation of a bespoke User Interface.

The prototype presented in this paper is a proof-of-concept rather than an industry hardened system that is ready for use in production. It focuses on the creation of a knowledge graph that demonstrates the ability to connect deployment descriptions to deployed services but lacks integration with observability tools such as OpenTelemetry. Achieving this integration is a crucial next step and it would provide the basis for further expanding the prototype to support more advanced features such as fault diagnosis. Our plan is to explore the use of our recently developed `astra-langchain4j` library [11] to integrate LLMs using approaches inspired by recent work on fault detection for 5G networks [19].

Other planned improvements include the integration of robust security mechanisms, integration with industry standard orchestrators, such as Kubernetes and the extension of the agent's capabilities to manage the full application lifecycle, including versioned updates (e.g., blue/green or canary deployments). Future work will also focus on enhancing the agents' reasoning and recovery strategies, moving beyond simple restarts to more sophisticated diagnostic plans, such as re-provisioning containers on different hosts or triggering automated rollbacks, eventually incorporating LLMs as described above.

Ultimately, this work represents a significant step toward realizing the long-standing vision of autonomic computing systems that can self-configure, self-heal, and self-optimize, all while operating as first-class, transparent citizens of the modern web.

References

1. JSON-LD - JSON for Linked Data, https://json-ld.org/
2. Agarwal, S.: Observability in microservices: From traditional monitoring to distributed system intelligence. International Journal of Computer Technology and Electronics Communication **5**(6), 16220–16226 (2022)

3. Beaumont, K., Collier, R.: Do you want to play a game? learning to play tic-tac-toe in hypermedia environments. In: European Conference on Multi-Agent Systems. pp. 441–448. Springer (2024)
4. Bualat, M., Barlow, J., Fong, T., Provencher, C., Smith, T.: Astrobee: Developing a free-flying robot for the international space station. In: AIAA SPACE 2015 conference and exposition. p. 4643 (2015)
5. Burns, B., , Grant, B., Oppenheimer, D., Brewer, E., Wilkes.: Borg, Omega, and Kubernetes. Queue **14**(1), 70–93 (Jan 2016). https://doi.org/10.1145/2898442.2898444, publisher: Association for Computing Machinery
6. Burns, B., Beda, J., Hightower, K.: Kubernetes: up and running: dive into the future of infrastructure. O'Reilly M, Beijing Boston Farnham Sebastopol Tokyo, second edition edn. (2019)
7. Chapman, R., Holmes, P.: Observability with Grafana: Monitor, control, and visualize your Kubernetes and cloud platforms using the LGTM stack. Packt Publishing Ltd (2024)
8. Chien, S., Sherwood, R., Tran, D., Castano, R., Cichy, B., Davies, A., Rabideau, G., Tang, N., Burl, M., Mandl, D., et al.: Autonomous science on the eo-1 mission (2003)
9. Ciortea, A., Boissier, O., Ricci, A.: Engineering world-wide multi-agent systems with hypermedia. In: International Workshop on Engineering Multi-Agent Systems. pp. 285–301. Springer (2018)
10. Ciortea, A., Mayer, S., Gandon, F., Boissier, O., Ricci, A., Zimmermann, A.: A decade in hindsight: the missing bridge between multi-agent systems and the world wide web. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (2019)
11. Collier, R., Beaumont, K., Ciortea, A.: astra-langchain4j: Experiences combining llms and agent programming. arXiv preprint arXiv:2601.21879 (2026)
12. Collier, R.W., Russell, S., Lillis, D.: Reflecting on agent programming with agentspeak (l). In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 351–366. Springer (2015)
13. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering pp. 195–216 (2017)
14. Fielding, R.T.: Architectural styles and the design of network-based software architectures. University of California, Irvine (2000)
15. Fong, T., Nourbakhsh, I., Dautenhahn, K.: A survey of socially interactive robots. Robotics and autonomous systems **42**(3-4), 143–166 (2003)
16. Gandon, F.: Merry hmas and happy new web: A wish for standardizing an ai-friendly web architecture for hypermedia multi-agent systems. In: Dagstuhl-Seminar 21072: Autonomous Agents on the Web. p. 3 (2021)
17. Gat, E., Bonnasso, R.P., Murphy, R., et al.: On three-layer architectures. Artificial intelligence and mobile robots **195**, 210 (1998)
18. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of things (iot): A vision, architectural elements, and future directions. Future generation computer systems **29**(7), 1645–1660 (2013)
19. Hatami, P., Majlesara, A., Majlesi, A., Khalaj, B.H.: Automated fault detection in 5g core networks using large language models. arXiv preprint arXiv:2512.19697 (2025)
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)

21. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. MartinFowler. com **25**(14-26), 12 (2014)
22. Newman, S.: Building microservices: designing fine-grained systems. " O'Reilly Media, Inc." (2021)
23. O'Neill, E., Beaumont, K., Bermeo, N.V., Collier, R.W.: Building management using the semantic web and hypermedia agents. In: ATAC@ ISWC. pp. 32–37 (2021)
24. O'Neill, E., Lillis, D., O'Hare, G.M., Collier, R.W.: Delivering multi-agent microservices using cartago. In: International Workshop on Engineering Multi-Agent Systems. pp. 1–20. Springer (2020)
25. Queralta, J.P., Taipalmaa, J., Pullinen, B.C., Sarker, V.K., Gia, T.N., Tenhunen, H., Gabbouj, M., Raitoharju, J., Westerlund, T.: Collaborative multi-robot search and rescue: Planning, coordination, perception, and active vision. Ieee Access **8**, 191617–191643 (2020)
26. Ricci, A., Viroli, M., Omicini, A.: Cartago: A framework for prototyping artifact-based environments in mas. In: International Workshop on Environments for Multi-Agent Systems. pp. 67–86. Springer (2006)
27. Richardson, L., Ruby, S.: RESTful web services. " O'Reilly Media, Inc." (2008)
28. Sekar, A.: The future of large-scale distributed systems: Trends, challenges, and opportunities. International Journal of Scientific Research in Computer Science, Engineering and Information Technology **11**(2), 3374–3390 (Apr 2025). https://doi.org/10.32628/CSEIT25112792, https://ijsrcseit.com/index.php/home/article/view/CSEIT25112792
29. Söylemez, M., Tekinerdogan, B., Kolukısa Tarhan, A.: Challenges and solution directions of microservice architectures: A systematic literature review. Applied sciences **12**(11), 5507 (2022)
30. Urblik, L., Kajati, E., Papcun, P., Zolotová, I.: Containerization in edge intelligence: A review. Electronics **13**(7), 1335 (2024)
31. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC). pp. 583–590 (Sep 2015). https://doi.org/10.1109/ColumbianCC.2015.7333476, https://ieeexplore.ieee.org/document/7333476/
32. W. Collier, R., O'Neill, E., Lillis, D., O'Hare, G.: Mams: Multi-agent microservices. In: Companion proceedings of the 2019 world wide web conference. pp. 655–662 (2019)
33. Yoerger, D.R., Jakuba, M., Bradley, A.M., Bingham, B.: Techniques for deep sea near bottom survey using an autonomous underwater vehicle. The International Journal of Robotics Research **26**(1), 41–54 (2007)
34. Yu, F., Yang, F., Qin, X., Zhang, Z., Zhang, J., Lin, Q., Zhang, H., Dang, Y., Rajmohan, S., Zhang, D., et al.: Enabling autonomic microservice management through self-learning agents. arXiv preprint arXiv:2501.19056 (2025)
35. Zota, R.D., Bărbulescu, C., Constantinescu, R.: A practical approach to defining a framework for developing an agentic aiops system. Electronics **14**(9), 1775 (2025)

A Wordpress RDF Deployment Document

```
@prefix dep: <http://localhost:5000/deployment-ontology#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix deploy: <http://localhost:5000/deployments/> .
@prefix ms: <http://localhost:5000/microservices/> .
@prefix srv: <http://localhost:5000/servers/> .

# WordPress Blog Deployment
deploy:wordpress-blog rdf:type dep:Deployment ;
    dep:deploymentId "wordpress-blog" ;
    dep:hasStatus dep:Pending ;
    dep:hasMicroservice ms:wordpress-blog-mysql,
                        ms:wordpress-blog-app,
                        ms:wordpress-blog-nginx .

# MySQL Database - Foundation layer
ms:wordpress-blog-mysql rdf:type dep:Microservice ;
    dep:microserviceName "wordpress-blog-mysql" ;
    dep:imageName "mysql" ;
    dep:imageTag "9.3.0" ;
    dep:networkMode "host" ;
    dep:environmentVariable
        "MYSQL_ROOT_PASSWORD=rootpassword123",
        "MYSQL_DATABASE=wordpress",
        "MYSQL_USER=wpuser",
        "MYSQL_PASSWORD=wppassword123",
        "MYSQL_HOST=192.168.0.128" ;
    dep:targetServer srv:docker-server-01 ;
    dep:microserviceStatus dep:PendingDeployment ;
    dep:hasPortMapping [
        rdf:type dep:PortMapping ;
        dep:hostPort 3306 ;
        dep:containerPort 3306
    ] .
    dep:provides se:mysql-connection .

# WordPress Application - Depends on MySQL
ms:wordpress-blog-app rdf:type dep:Microservice ;
    dep:microserviceName "wordpress-blog-app" ;
    dep:imageName "wordpress" ;
    dep:imageTag "6.8.2-apache" ;
    dep:networkMode "host" ;
    dep:dependsOn ms:wordpress-blog-mysql ;
    dep:environmentVariable
        "WORDPRESS_DB_HOST={se:mysql-connection.host}",
        "WORDPRESS_DB_NAME=wordpress",
        "WORDPRESS_DB_USER=wpuser",
        "WORDPRESS_DB_PASSWORD=wppassword123",
        "WORDPRESS_TABLE_PREFIX=wp_",
        "WORDPRESS_DEBUG=1" ;
    dep:targetServer srv:docker-server-01 ;
    dep:microserviceStatus dep:PendingDeployment ;
    dep:hasPortMapping [
        rdf:type dep:PortMapping ;
        dep:hostPort 8080 ;
        dep:containerPort 80
    ] .
    dep:consumes se:mysql-connection .

# -- Individuals for Service Endpoint --
se:mysql-connection rdf:type dep:ServiceEndpoint ;
    rdfs:label "MySQL Database Connection" .

# -- Server for Deployment --
srv:docker-server-01
    rdf:type dep:Server ;
    dep:serverId "docker-server-01" ;
    dep:serverHostname "192.168.0.128" ;
    dep:serverPort 2375 .
```

B Evaluation Scenarios

This appendix provides the detailed execution flows for the scenarios summarized in the Evaluation section.

B.1 Scenario 1: Semantic Reasoning and HATEOAS Navigation

Objective: Validate the collaborative workflow where a Manager Agent delegates tasks and a Worker Agent autonomously forms and executes a detailed plan by navigating the hypermedia environment.

Execution Flow:

1. The Manager receives a notification for the new `deploy:wordpress-blog` resource via WebSub.
2. The Manager confirms its status is `dep:Pending`, selects an available Worker Agent, and delegates the task by sending it the deployment URI.
3. The Worker Agent performs a GET request on the deployment URI to retrieve the list of its constituent microservice URIs.
4. The Worker iterates through this list, performing a GET request on each microservice URI to gather its full definition (image, dependencies, etc.) and asserts these facts into its local belief base.
5. With a complete model of the deployment, the Worker's BDI engine forms an execution plan, respecting the `dep:dependsOn` property to deploy mysql before wordpress-app.
6. The Worker executes the plan. Upon successfully deploying the mysql container, it creates a `dep:ContainerInstance` resource in the knowledge graph.
7. To resolve the `{se:mysql-connection.host}` template, the Worker queries its belief base for the new `dep:ContainerInstance` of the mysql service to find the server's hostname and substitutes it.

B.2 Scenario 2: Autonomous Self-Healing (Reactive)

Objective: Evaluate the Worker Agent's ability to react to runtime issues during its execution plan.

Execution Flow:

1. The mysql container is deployed successfully. Before the wordpress-app is deployed, the mysql container is manually stopped.
2. The Worker Agent, tasked with deploying wordpress-app, first validates its dependencies as part of its plan. It queries the status of the mysql instance.
3. It detects that the container's status is `exited`, which contradicts the required `Running` state for the dependency to be met.

4. The agent adopts a new, immediate goal: `!start_container(...)` for the mysql instance. It sends a POST request to the Docker API to restart the container.
5. Once it confirms the mysql container is `Running` again, it resumes its original plan of deploying the wordpress-app.

B.3 Scenario 3: Proactive Self-Healing via Monitoring Agent

Objective: Evaluate the framework's autonomic capability to maintain the desired system state post-deployment.

Execution Flow:

1. The WordPress deployment is successfully running. The mysql container is manually stopped.
2. The Monitoring Agent, on its periodic cycle, queries the Management Server for all resources of type `dep:ContainerInstance`.
3. For each instance that should be `Running`, it inspects its live state via the Docker API.
4. It discovers the mysql container's live state is `exited`, a mismatch with its desired state.
5. This deviation triggers a self-healing plan. The agent sends a POST request to the Docker API's start endpoint.
6. **Success Case**: The container restarts. The system returns to its desired state without human intervention.
7. **Failure Case**: If the container enters a crash loop, the agent's restart attempts fail. After a set number of retries, it updates the `ContainerInstance` status to `dep:Failed` and annotates it with a `dep:errorMessage`.

B.4 Scenario 4: Failure Diagnosis and Reporting

Objective: Evaluate the system's behavior when faced with an unrecoverable deployment error.

Execution Flow:

1. The mysql microservice is defined with a hostPort of 80, which is already in use on the target server.
2. The Worker Agent attempts to create the mysql container. The Docker API returns a 409 Conflict error.
3. The agent's HTTP module captures this error. The agent, unable to resolve a port conflict, recognizes the deployment has failed.
4. It performs an HTTP PUT request to the mysql `dep:Microservice` resource, updating its status to `dep:Failed` and adding the captured error message from Docker to the `dep:errorMessage` property.
5. The Worker Agent marks the overall deployment status as `Failed` and halts further actions.

B.5 Scenario 5: Scalability through Parallel Deployment

Objective: Evaluate the framework's ability to improve throughput by distributing work across multiple agents.

Execution Flow:

1. Two independent deployments are created simultaneously with at least two idle Worker Agents available.
2. The Manager Agent receives two separate WebSub notifications.
3. It assigns the first deployment URI to `worker1` and the second to `worker2`.
4. The two Worker Agents execute their assigned deployments in parallel, each independently gathering information and interacting with their respective target servers.

C User Interface

A critical component for the practical adoption of autonomous systems is the ability for human operators to understand and trust them. Our Deployment Manager UI provides this layer of observability. It is a real-time representation of the same underlying knowledge graph that the agents are operating on. This allows an operator to define goals, monitor the agent society, and observe the entire deployment lifecycle.

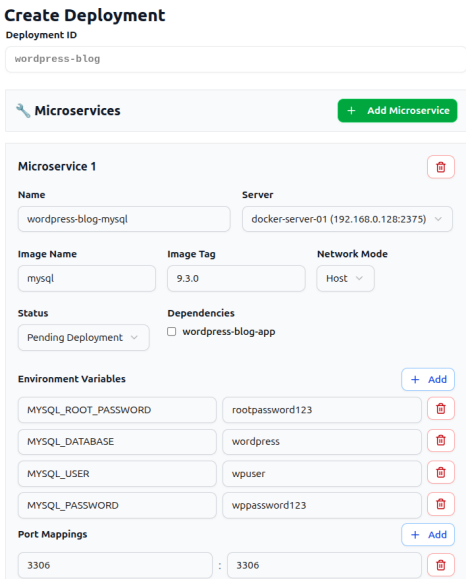

Fig. 9. The UI for creating a new deployment. Operators define the dep:Microservice resources, their properties, and dependencies, setting the goal for the agents.

An operator begins by defining the desired state using the “Create Deployment" form, as shown in Fig. 9. This interface directly populates the dep:Microservice resources in the ontology, providing a user-friendly way to specify the system’s goals.

Once a deployment is initiated, the operator can monitor the agent society’s status in the “Agents" view (Fig. 10). This screen provides insight into which agents are available (idle) and which are actively executing tasks (busy), making the distributed workforce transparent.

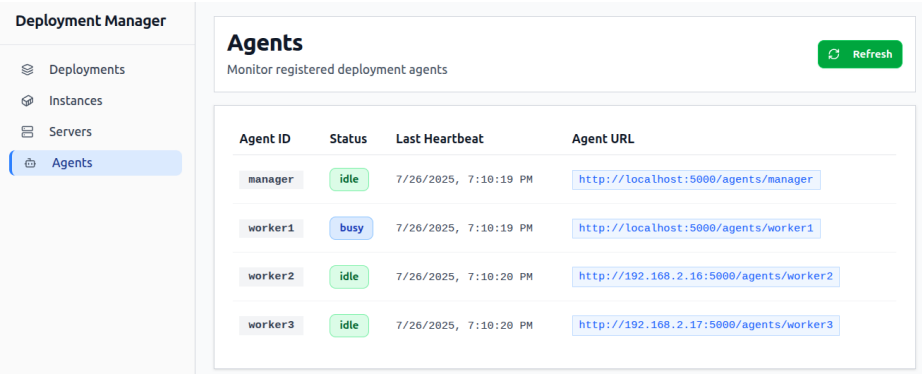

Fig. 10. The Deployment Manager UI showing the status of the registered agent society. The manager and two workers are idle, while worker1 is busy with a task.

The “Deployments" view (Fig. 11) visualizes the agent’s reasoning process. In the WordPress example, it shows that the `wordpress-blog-mysql` service is `Deploying` while the dependent `wordpress-blog-app` is still `PendingDeployment`. This directly reflects the agent’s understanding and adherence to the `dep:dependsOn` relationship defined in the ontology.

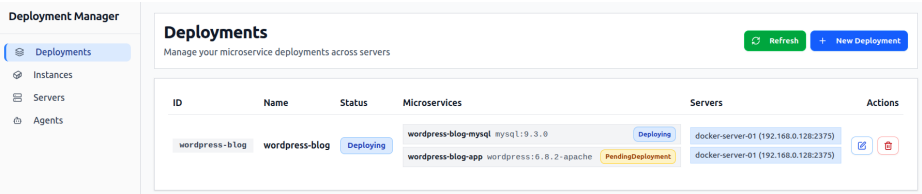

Fig. 11. The UI showing a ’wordpress-blog’ deployment in progress. The agent is actively deploying the mysql service, while the dependent wordpress-app is correctly held in a pending state.

Finally, the “Container Instances" view (Fig. 12) displays the actual runtime state by visualizing the `dep:ContainerInstance` resources. This screen confirms which containers are actually running, on which servers, and provides their runtime identifiers. This completes the observability loop, allowing an operator to see the system progress from intent (Fig. 9) to process (Fig. 11) to reality (Fig. 12).

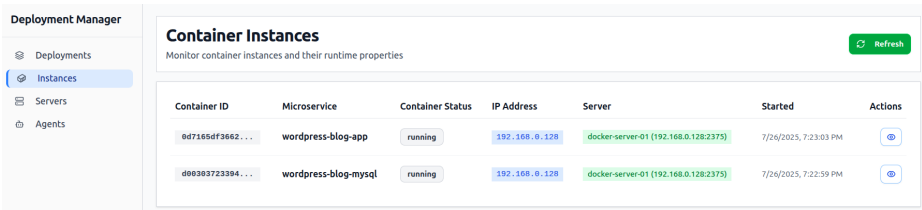

Fig. 12. The UI showing the successfully created dep:ContainerInstance resources after a deployment, reflecting the actual runtime state of the system.