

ALMA MATER STUDIORUM · UNIVERSITÀ DI
BOLOGNA

Master's Degree in Computer Science and Engineering

Intelligent Systems Engineering Project

ARES-MAS

**Autonomous Rovers for Exploration and
Safety**

A Hybrid Symbolic AI System for Multi-Agent Planetary Exploration

Student:

Marco Costantini

0001189584

Academic Year 2025/2026

Abstract

ARES-MAS is a simulated, grid-based orchestrator designed to manage collaborative planetary exploration through a hybrid symbolic AI architecture. The project addresses the complex challenges of autonomous multi-agent coordination, specifically focusing on dynamic hazard avoidance, spatial pathfinding, and safety-critical resource management. To achieve this, the system enforces a strict architectural separation of concerns. The physical layer, implemented in Kotlin, provides a discrete-time 2D spatial simulator that manages environmental state, battery consumption, and mutually exclusive resource locks. The deliberative logical layer encapsulates the system’s intelligence utilizing BDI (Belief-Desire-Intention) agents (Scouts and Harvesters) programmed in Jason and AgentSpeak(L). These autonomous agents communicate via the FIPA Agent Communication Language (ACL). By leveraging the Contract Net Protocol (CNP) for task negotiation, the agents evaluate real-time spatial heuristics (Manhattan distance) to effectively orchestrate task assignment. Crucially, a logic-based safety layer utilizing tuProlog acts as a runtime guardrail, preventing fatal collisions and ensuring fail-safe operations when agents make over-optimistic economic decisions.

GenAI Usage Declaration

During the development of this project and the writing of this report, Generative AI tools were used to assist in tasks such as code formatting, generation of documentation templates, and structural review. All core logic, system architecture, and domain modeling are original work.

Contents

1	Introduction and Project Goals	3
1.1	Mapping ARES-MAS to MAS Characteristics	3
2	System Architecture	5
2.1	Backend (JVM - Kotlin / Jason / tuProlog)	5
2.1.1	State Management and Concurrency	5
2.1.2	Neuro-Symbolic Integration: tuProlog Engines	5
2.2	Frontend	6
2.3	Integration via File-Polling	6
2.4	Technology Stack	7
2.5	Repository Structure	7
3	World Model and Scenario Definition	8
4	Hybrid Intelligence Engine	8
4.1	Collaborative BDI Loop (Jason / AgentSpeak)	8
4.2	Social Coordination: Atomic Claiming and Contract Net Protocol	8
4.2.1	Spatial Bidding and Economic Underbidding	9
4.3	Logic-Based Safety Guardrails (tuProlog)	9
4.4	Automated Planning (STRIPS)	10
4.4.1	Forward State-Space Search Implementation	10
4.5	Sequence Diagram	11
5	Environmental Dynamics and Resource Lifecycle	13
5.1	The Moving Sandstorm	13
5.2	Battery Model	13
5.3	Resource Lifecycle	14
6	Explainability and the Streamlit Dashboard	15
7	Deployment & Execution	17
7.1	Prerequisites	17
7.2	Executing the Backend	17
7.3	Executing the Frontend	17
8	Validation and Known Limitations	18
9	Conclusions & Future Work	19
9.1	Conclusions	19
9.2	Future Work	19

1 Introduction and Project Goals

ARES-MAS is a lightweight, hybrid intelligent system designed to coordinate a fleet of autonomous rovers performing collaborative exploration of an uncharted planetary surface, under safety, environmental, and resource constraints. The system’s central hypothesis is that no single symbolic AI paradigm is sufficient to model the full range of reasoning a physical multi-agent system requires: deliberative goal pursuit, social coordination under uncertainty, hard safety guarantees, and long-horizon logistics planning each call for a different formalism. ARES-MAS therefore combines three classical, well-understood paradigms.

The environment is a discrete 2D grid loaded from a declarative JSON scenario file. Two rover roles cooperate: **Scouts**, which explore the terrain and discover mineral samples, and **Harvesters**, which negotiate the right to extract a discovered sample and deliver it back to a Base Station. A dynamic **hazard** a sandstorm that patrols a fixed cycle of cells introduces time-varying risk that the system must reason about at runtime.

1.1 Mapping ARES-MAS to MAS Characteristics

Beyond its concrete implementation, ARES-MAS is deliberately designed to satisfy the defining properties of Multi-Agent System.

- **Autonomy:** Each agent maintains its own belief base, shaped by its role: Scouts track their current position and the set of minerals they have already handled, using these beliefs to trigger discovery and claim attempts autonomously; Harvesters, while not tracking an explicit position belief, independently evaluate each incoming CFP and decide their own bid. For both roles, battery level is tracked only implicitly through the Environment rather than as an agent-held belief. No central scheduler dictates pathfinding or auction outcomes; the only deliberately centralised decisions are the small set of constants documented as demonstration aids, and even those flow through the same autonomous decision procedures rather than bypassing them.
- **Social ability and coordination:** Agents share no memory. Every act of coordination announcing a discovery, negotiating an extraction contract, awarding a task, is mediated exclusively by **FIPA-ACL-style** message passing (`.broadcast`, `.send` with the `tell` performative). Physical resource contention is handled by an Environment mediated atomic action that plays the role of a shared, contended artefact without agents ever inspecting each other’s internal state.
- **Reactivity and proactivity:** Agents are goal-directed by default: Scouts proactively pursue `!explore` / `!do_patrol`, Harvesters proactively pursue `!do_extract` and `!plan_return`. At the same time, both roles are genuinely reactive to a changing world: a denied move or a denied extraction triggers an immediate, unscripted recovery plan (`!do_patrol`, `!do_extract`), and the specific recovery path taken

depends on the real-time position of the hazard and the real-time battery level at the moment of failure.

- **Situatedness:** Agents are physically placed on a shared 2D grid and cannot reason their way around that fact. Reaching a resource requires physically traversing adjacent cells one step at a time, each step has a real cost, and the safety of a given action depends on the environment's current physical configuration. Cognitive reasoning is therefore always coupled to, and constrained by, spatial and physical limitations enforced by a single authoritative Environment.

2 System Architecture

The architecture strictly decouples the simulation and reasoning backend from the visualization frontend.

2.1 Backend (JVM - Kotlin / Jason / tuProlog)

The backend is a Kotlin/Gradle project that hosts three cooperating engines inside a single custom Jason Environment class (`AresEnvironment`):

- The **Jason** runtime, which executes the **Belief-Desire-Intention** reasoning cycle for every agent.
- A **tuProlog** engine, queried before every safety-critical physical action.
- A custom **STRIPS** planner, exposed to the agents as a Jason Internal Action.

2.1.1 State Management and Concurrency

The `AresEnvironment` class is the single authoritative source of world state. To accurately reflect the physical constraints of the simulation, entities and resources are managed using native Kotlin data structures. Crucially, per-agent battery levels are not global constants but are tracked individually via a `mutableMapOf`, physically grounding each agent’s situatedness.

To prevent race conditions between the BDI reasoning threads and the independent hazard timer, state mutations are strictly protected by explicit concurrency locks (`resourceLock` for physical entities, `engineLock` for Prolog queries):

```
companion object {  
    var hazardList = listOf<Pair<Int, Int>>()  
    var mineralStates = mutableListOf<MineralState>()  
    var batteryLevels = mutableMapOf<String, Int>()  
    private val resourceLock = Any()  
    private val engineLock = Any()  
    // ...  
}
```

When an agent attempts a physical action, the environment safely decrements its specific battery counter within a synchronized block, leaving other agents unaffected:

```
synchronized(resourceLock) {  
    val current = batteryLevels.getOrDefault(agName, 100)  
    batteryLevels[agName] = (current - 2).coerceAtLeast(0)  
}
```

2.1.2 Neuro-Symbolic Integration: tuProlog Engines

A major architectural challenge in hybrid systems is querying a logic engine with transient, agent-specific facts without permanently corrupting the shared knowledge base. ARES-MAS solves this during the `extract` action by instantiating a disposable, throwaway

tuProlog engine for the duration of the query, dynamically injecting the agent’s real-time physical constraints:

```
val isSafe = synchronized(engineLock) {
    val tempEngine = Prolog()
    tempEngine.setTheory(Theory(rulesFile.readText()))

    // Inject dynamic hazard facts
    for ((hx, hy) in hazardList) {
        tempEngine.addTheory(Theory("hazard($hx, $hy)."))
    }
    // Inject agent-specific physical facts
    tempEngine.addTheory(Theory("is_harvester($agName)."))
    tempEngine.addTheory(Theory("battery_level($agName, $batteryLevel)."))
})

tempEngine.solve("safe_to_extract($agName, $x, $y).").isSuccess
}
```

Using a throwaway engine per query avoids any risk of one agent’s transient facts leaking into another agent’s concurrent query, ensuring thread-safe neuro-symbolic reasoning.

2.2 Frontend

A Streamlit dashboard renders the grid, the rover trails, the hazard, and a set of explainability panels. It never talks to Jason or tuProlog directly.

2.3 Integration via File-Polling

The JVM backend appends one structured JSON object per line to `events.jsonl` for every state-changing event (movement, negotiation message, safety violation, resource claim, extraction, hazard displacement, planning delegation, mission completion). This append-only logging is implemented directly in the Environment actions:

```
val ev = """{"tick": $tick, "type": "MOVE", "rover": "$agName", "to": [
    $x, $y], "battery": ${batteryLevels[agName]}}"""
logFile.appendText(ev + "\n")
```

The Streamlit frontend re-reads this file on every script rerun and reconstructs the full simulation state up to a user-selected point in time, exposed through a “Time Machine” slider. This is deliberately a pull-based, replay-oriented design: it trades strict real-time latency for the ability to freely scrub backward and forward through the causal history of a run, which is more valuable for explainability than for live monitoring.

2.4 Technology Stack

Component	Technology
BDI reasoning engine	Jason (<code>io.github.jason-lang:jason-interpreter</code> 3.3.1), AgentSpeak(L)
Logic-based reasoning	tuProlog 3.3.0 (<code>it.unibo.alice.tuprolog</code>)
Backend language / runtime	Kotlin, JVM 21
Build system	Gradle (Kotlin DSL), reproducible via the Gradle Wrapper
JSON parsing/serialization	Gson (<code>com.google.code.gson</code>)
Frontend	Python 3.12, Streamlit
Visualization	Matplotlib
Data interchange	JSON (declarative scenario), JSON Lines

2.5 Repository Structure

- `backend/`
 - `build.gradle.kts` — Gradle build configuration.
 - `ares.mas2j` — Jason MAS configuration: agent roster and Environment binding.
 - `scout.asl`, `harvester.asl` — shared AgentSpeak source, one file per role, reused across multiple agent instances via `.my_name(Me)`.
 - `src/main/kotlin/ares/AresEnvironment.kt` — the Environment: world state, guardrail queries, hazard patrol, battery model.
 - `src/main/kotlin/strips/plan_return.kt` — the STRIPS planner, exposed as a Jason Internal Action.
- `data/`
 - `scenario.json` — declarative initial world state (grid size, minerals, hazards).
 - `rules.pl` — tuProlog safety theory (`safe_to_move/2`, `safe_to_extract/3`).
 - `events.jsonl` — generated at runtime; the full causally-ordered event log consumed by the dashboard.
- `frontend/`
 - `app.py` — the Streamlit dashboard.

3 World Model and Scenario Definition

The scenario is declarative: a JSON file specifies the grid size and the initial entities.

```
{
  "grid": { "width": 10, "height": 10 },
  "entities": [
    { "id": "mineral1", "type": "MINERAL", "position": [3, 4] },
    { "id": "sandstorm1", "type": "HAZARD", "position": [5, 5] }
  ]
}
```

Four agents are declared in the Jason configuration file (`ares.mas2j`): two Scouts (`scout1`, `scout2`) sharing the same `scout.asl` source, and two Harvesters (`harv1`, `harv2`) sharing `harvester.asl`. Sharing a single AgentSpeak source file across multiple instances of the same role is made possible throughout the codebase by the internal action `.my_name(Me)`, which lets a shared plan library branch on, or report, the identity of the specific agent instance executing it.

4 Hybrid Intelligence Engine

4.1 Collaborative BDI Loop (Jason / AgentSpeak)

Each rover is an autonomous BDI agent. Scouts execute a per-agent patrol plan that walks a fixed sequence of grid cells via a shared `!go(X,Y)` sub-goal. After every successful move, the agent updates its own position belief using Jason's belief-revision operator (`--+at(X,Y)`, which atomically removes the previous `at/2` belief and asserts the new one).

A key design decision concerns how a Scout discovers a mineral. A `mineral(X,Y)` percept globally and have every agent react to it the instant the simulation starts which is unrealistic and, with more than one Scout, would cause every Scout to race for the same resource regardless of its actual location discovery is tied to physical presence:

```
@mineral_found[atomic]
+at(X,Y) : mineral(X,Y) & not already_handled(X,Y) <-
  +already_handled(X,Y);
  !try_claim(X, Y).
```

This plan only triggers when an agent's own position belief changes to coincide exactly with an un-claimed mineral's coordinates an inexpensive approximation of a limited sensing range, implemented entirely in AgentSpeak.

4.2 Social Coordination: Atomic Claiming and Contract Net Protocol

Discovering a mineral does not automatically grant the right to auction it. The discovering Scout must first call an Environment action, `claim_mineral(X,Y)`, before doing anything else. On the Kotlin side this action performs a check-and-set on the mineral's claimed flag

inside a synchronized block, guaranteeing that if two Scouts discover the same mineral at nearly the same time a genuine race condition once more than one Scout is present only one of them can ever win the claim; the other receives `false`, and its `!try_claim` goal fails, invoking a dedicated recovery plan that stands down gracefully.

Only the Scout that wins the claim proceeds to run the Contract Net Protocol: it logs and broadcasts a Call-For-Proposal (FIPA-ACL `tell` performative) to every agent in the system, waits a fixed interval to let Harvesters respond, then queries its own belief base with `.findall/.min` to select the lowest-cost proposal among all `propose(mineral(X,Y),Cost)` beliefs that were auto-asserted by incoming `tell` messages, and finally sends an `accept` only to the winner:

```
.findall(offer(Cost, H), propose(mineral(X,Y), Cost)[source(H)], Offers)
.min(Offers, offer(BestCost, Best));
.send(Best, tell, accept(mineral(X,Y)));
```

4.2.1 Spatial Bidding and Economic Underbidding

Every Harvester that receives a CFP computes a bid based on spatial metrics. To strictly couple the cognitive reasoning to the situatedness of the environment, the Harvester natively calculates the Manhattan distance to the target using Jason’s math operators.

To demonstrate the necessity of safety guardrails, an *Underbidding* logic was intentionally introduced for demonstration purposes:

```
+cfp(mineral(X,Y))[source(S)] <-
    // Spatial metric: Manhattan distance from base (0,0)
    Distance = math.abs(X - 0) + math.abs(Y - 0);
    EstimatedCost = Distance * 2;

    .my_name(Me);
    // Underbidding strategy for harv1 to force guardrail intervention
    if (Me == harv1) { Cost = EstimatedCost - 5; }
    else { Cost = EstimatedCost + 10; };

    .send(S, tell, propose(mineral(X,Y), Cost)).
```

This ties the negotiated cost to a genuine spatial quantity while intentionally exposing the BDI agent to a hazardous, over-optimistic economic decision, thus justifying the need for the logic-based guardrail.

4.3 Logic-Based Safety Guardrails (tuProlog)

A lightweight tuProlog engine evaluates two independent safety predicates, both defined in an external theory file (`rules.pl`), before the Environment allows a critical physical action to take effect:

```
safe_to_move(X, Y) :- \+ hazard(X, Y).
safe_to_extract(Rover, X, Y) :-
    is_harvester(Rover),
```

```
\+ hazard(X, Y),
battery_level(Rover, Level),
Level > 20.
```

`safe_to_move/2` is queried on the shared, persistent Prolog engine before every move action; its `hazard/2` facts are kept in sync with the sandstorm’s current cell. When either query fails, `Environment.executeAction` returns `false`. In Jason’s operational semantics this causes the agent’s current intention to be aborted and a goal-deletion event to be raised for the goal that triggered the action; a dedicated recovery plan then takes over for extraction, the failure path still delegates to the STRIPS return planner, so a rover that is denied an extraction for having insufficient battery reserves nonetheless returns safely to base.

4.4 Automated Planning (STRIPS)

When a Harvester needs to return to base whether after a successful extraction or after an extraction was denied by the guardrail it delegates route computation to a custom, from-scratch State-Space Search planner implemented in Kotlin, modelled in the STRIPS style: each candidate action is an explicit triple of preconditions, an add-list, and a delete-list over a state represented as a set of ground literals.

```
data class StripsAction(
    val name: Literal,
    val pre: Set<String>,
    val add: Set<String>,
    val del: Set<String>
)
```

4.4.1 Forward State-Space Search Implementation

The planner performs forward search: from the current state it generates every adjacent, in-bounds, non-hazardous cell as a legal `move/2` action, applies $State' = (State \setminus del) \cup add$, and explores the resulting state graph breadth-first until the goal state `at(baseX, baseY)` is reached. The BFS loop inherently guarantees the shortest unweighted path:

```
// Core BFS routing logic in STRIPS Internal Action
val queue: Queue<Node> = LinkedList()
val visited = mutableSetOf<Set<String>>()
queue.add(Node(initialState, emptyList()))

while (queue.isNotEmpty()) {
    val current = queue.poll()
    if (current.state.contains("at($baseX, $baseY)")) return current.plan

    for (action in generateValidMoves(current.state)) {
        val newState = (current.state - action.del) + action.add
        if (visited.add(newState)) {
```

```

        queue.add(Node(newState, current.plan + action.name.toString
    )))
    }
}
}

```

Because the only state feature currently tracked is position, this reduces mathematically to an unweighted shortest-path search equivalent to breadth-first A* with a null heuristic.

A significant architectural decision was to implement plan delegation as a Jason Internal Action (`strips.plan_return`) rather than as an ordinary Environment action. `Environment.executeAction` can only return a boolean success flag to the agent it has no channel for returning a computed value such as a route. An Internal Action executes synchronously inside the agent's own reasoning cycle and can unify one of its arguments with an arbitrary Jason term:

```

strips.plan_return(Me, X, Y, 0, 0, Plan);
!execute_plan(Plan).

+!execute_plan([]) <-
    .print("Safely returned to base!");
    mission_complete.

+!execute_plan([move(NX, NY) | Rest]) <-
    move(NX, NY);
    .wait(1000);
    !execute_plan(Rest).

```

Crucially, the returned plan is not merely displayed: a recursive `AgentSpeak` plan walks the list and issues one genuine `move` Environment action per leg, so every step of the STRIPS-computed route is independently re-validated by the `tuProlog` movement guardrail and independently logged.

4.5 Sequence Diagram

The following diagram traces one complete resource lifecycle end to end from a Scout's physical discovery of a mineral to a Harvester's guardrail-checked extraction and STRIPS-planned return showing exactly which of the three reasoning paradigms is responsible for each step and how they hand control to one another.

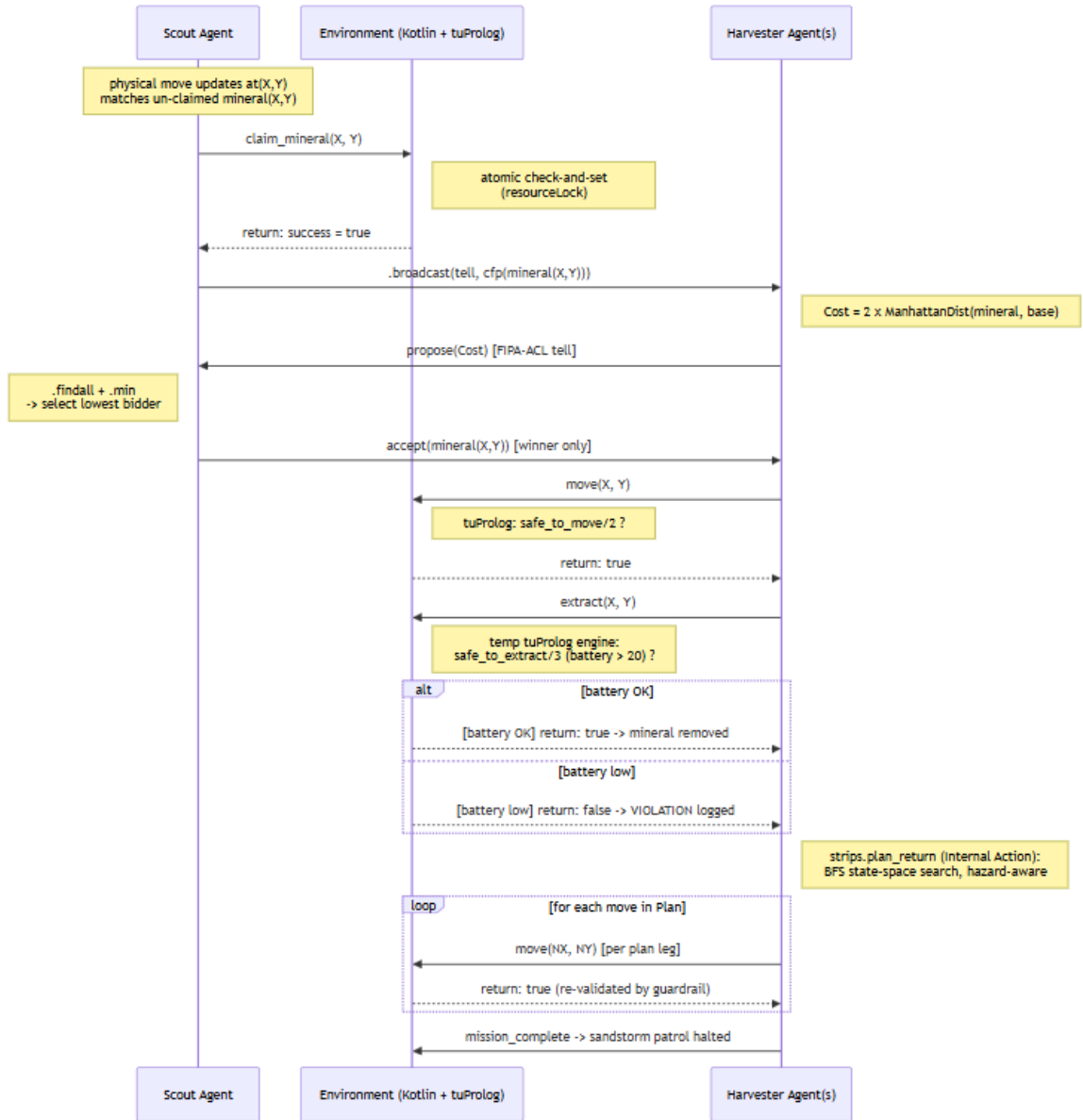


Figure 1: Sequence diagram of one mineral discovery-to-delivery cycle, spanning claim arbitration, Contract Net negotiation, the movement and extraction guardrails, and STRIPS-delegated return planning.

5 Environmental Dynamics and Resource Lifecycle

5.1 The Moving Sandstorm

The hazard patrols a fixed, four-cell cycle on a background timer, independent of any agent’s reasoning cycle. Each relocation is protected by the same `engineLock` used to guard tuProlog queries, ensuring an agent’s safety check can never observe the Prolog theory mid-rebuild.

The following Kotlin snippet shows the background thread injection and the thread-safe state mutation:

```
private fun startHazardPatrol() {
    hazardTimer = Timer(true)
    hazardTimer.scheduleAtFixedRate(object : TimerTask() {
        override fun run() { moveHazard() }
    }, 3000, 4000)
}

private fun moveHazard() {
    synchronized(engineLock) {
        if (patrolMovesDone >= maxPatrolMoves) {
            hazardTimer.cancel()
            return
        }
        // Advance patrol and mutate state securely
        patrolIndex = (patrolIndex + 1) % patrolRoute.size
        hazardList = listOf(patrolRoute[patrolIndex])
        rebuildPrologTheory()
        patrolMovesDone++
        updatePercepts()
    }
}
```

The architecture reloads the entire theory from the rule file and re-populates it with the hazard’s current position on every relocation, trading a small, constant performance cost for absolute state robustness.

A guardrail intervention occurs organically because an agent’s purpose-driven route intersects the hazard’s currently patrolled cell a natural consequence of two independently scheduled dynamic processes. The patrol halts either after a fixed number of relocations or the moment any Harvester signals `mission_complete` after a successful return to base.

5.2 Battery Model

Every agent carries an integer battery counter, initialised per-agent and decremented by a fixed amount after every successful physical move. This value is the one queried by the `safe_to_extract/3` guardrail described in Section 4.3. As detailed in Section 2, this is managed natively via a Kotlin `mutableMapOf<String, Int>` to enforce true physical situatedness.

5.3 Resource Lifecycle

A mineral resource moves through three explicit states, all managed atomically by the Environment. To guarantee consistency during concurrent discovery, resources are modelled using a native Kotlin Data Class:

```
data class MineralState(val id: String, val x: Int, val y: Int, var
    claimed: Boolean = false)
var mineralStates = mutableListOf<MineralState>()
```

The lifecycle proceeds as follows:

- **Available:** Perceived by any Scout as a `mineral(X,Y)` percept.
- **Claimed:** Removed from the shared percept stream the instant one Scout successfully calls `claim_mineral`. This is enforced via an atomic check-and-set:

```
synchronized(resourceLock) {
    val m = mineralStates.find { it.x == x && it.y == y && !it.
        claimed }
    if (m != null) { m.claimed = true; true } else false
}
```

This prevents any other Scout from independently discovering and re-auctioning the same resource.

- **Extracted:** Permanently removed from the Environment's internal resource list once a Harvester's `extract` action succeeds (`mineralStates.removeAll { it.x == x && it.y == y }`), so it can never again be perceived, claimed, or negotiated over.

6 Explainability and the Streamlit Dashboard

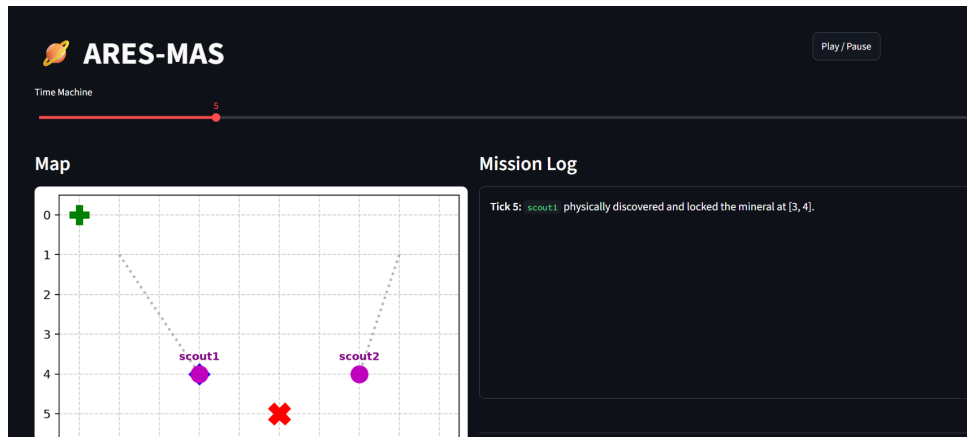


Figure 2: The Streamlit dashboard rendering

Every state-changing event in the system is serialized as a single-line JSON object appended to `events.jsonl`, tagged with a monotonically increasing tick counter shared across the entire simulation. This guarantees that the full causal ordering of events negotiation messages, safety interventions, resource claims, extractions, hazard relocations, and mission completion can be reconstructed exactly.

The following is a representative snippet of the `events.jsonl` file generated by the Kotlin environment:

```
{ "tick": 6, "type": "CLAIM", "rover": "scout1", "at": [3, 4], "success": true }
{ "tick": 8, "type": "NEGOTIATION", "msg_type": "propose", "content": "offer(mineral(3,4),9)", "sender": "harv1", "receiver": "scout1" }
{ "tick": 16, "type": "VIOLATION", "rover": "harv1", "attempted_to": [3, 4], "action_type": "EXTRACT", "reason": "Battery level 19 is below the 20-unit safe threshold" }
```

The Streamlit dashboard exposes this history through several coordinated views:

- **Grid Map and Time Machine:** A grid map showing rover trails, current positions, the mineral (until extracted), the hazard’s current cell, and the Base Station. This is driven entirely by a user-controlled “Time Machine” tick slider with an optional auto-advancing Play mode. The Python frontend dynamically filters the event array up to the selected tick:

```
import streamlit as st
import json

# Reconstructing causal history
events = [json.loads(line) for line in open('events.jsonl')]
max_tick = max(e['tick'] for e in events)

# The Time Machine slider
```

```
current_tick = st.slider("Time Machine", 0, max_tick)
visible_events = [e for e in events if e['tick'] <= current_tick]
render_grid_state(visible_events)
```

- **Mission Log:** A natural-language panel that translates each structured event type into a human-readable sentence (e.g. a CLAIM event becomes "scout1 physically discovered and locked the mineral at [3, 4]"), giving a reader who has not inspected the raw JSON an immediate narrative account of what happened and why.
- **Auction Analysis:** A bar chart comparing the Contract Net bids received for the most recent auction, extracted from the negotiation log.
- **Safety Guardrail Explanations:** A dedicated safety-reasoning panel that, for every logged VIOLATION event, distinguishes between the two guardrail predicates (`safe_to_move` vs. `safe_to_extract`) and renders a structured, rule-by-rule explanation of which condition failed via a fixed textual template per violation type, populated with the logged `reason` field for extraction denials and the target coordinates for movement denials a pseudo proof-tree that mirrors the two guardrail predicates without literally parsing from the log.

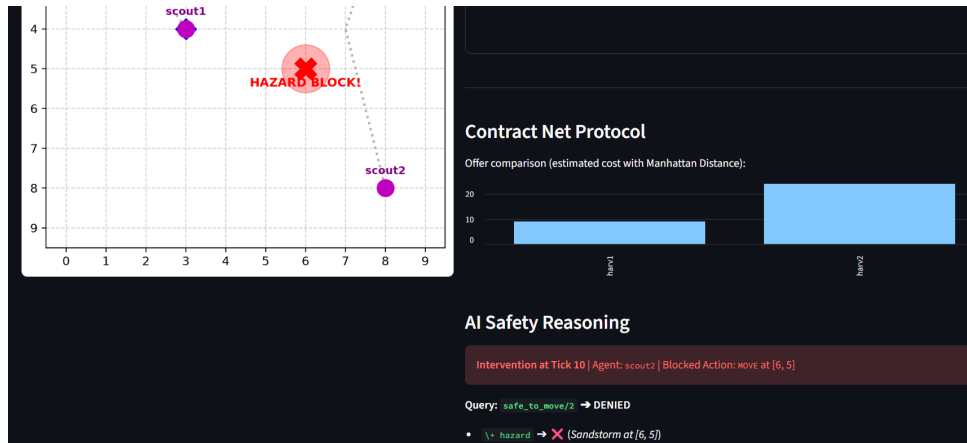


Figure 3: The dashboard’s explainability panels with Contract Net bid comparisons, and tuProlog guardrail interventions.

7 Deployment & Execution

To ensure reproducibility and ease of evaluation, ARES-MAS is designed to be executed via standard build tools.

7.1 Prerequisites

Running the full multi-agent simulation and its dashboard requires the following toolchains installed on the host machine:

- **Java Development Kit (JDK) 21:** Required by the `jason-interpreter` artifact and the Kotlin JVM target.
- **Python 3.12+:** Required for the Streamlit frontend.

7.2 Executing the Backend

The physical environment, the tuProlog guardrails, and the Jason BDI agents are completely encapsulated within the Kotlin/Gradle backend. To start the simulation, navigate to the `backend` directory and use the Gradle Wrapper:

```
cd backend
./gradlew run --args="ares.mas2j"
```

This command will compile the Kotlin source, resolve the Jason and tuProlog dependencies, launch the MAS environment, and begin writing causality logs to the `events.jsonl` file.

7.3 Executing the Frontend

The visualization layer runs independently and can be started before, during, or after the backend execution. Navigate to the `frontend` directory, create and activate an isolated Python virtual environment, install the dependencies, and launch Streamlit:

```
cd frontend
python3 -m venv .venv
source .venv/bin/activate          # On Windows: .venv\Scripts\activate
pip install streamlit matplotlib pandas
streamlit run app.py
```

8 Validation and Known Limitations

Correctness was validated through repeated, instrumented manual runs, cross-checked against the structured `events.jsonl` log to ensure causal consistency and correctness of the physical layer.

Furthermore the final system diverges from the initial proposal in several documented ways:

- **Hazards:** Only one dynamic hazard type (the sandstorm) is implemented.
- **Complexity Guard:** The contingency plan a graceful degradation from the custom STRIPS planner to A* heuristic search under state-space complexity pressure has no corresponding code path.
- **Demonstration Constants:** The Contract Net bid formula and the Harvesters' initial battery levels contain explicit constants chosen to make the negotiation outcome and the extraction guardrail reliably reproducible for a live demonstration. The underlying mechanisms remain general and were validated independently.
- **Clock Sync:** The simulation's "tick" is a monotonically increasing event counter rather than a synchronized real-time clock.
- **Rendering Latency:** The Streamlit dashboard relies on Matplotlib to synchronously redraw the entire grid state on every tick. This introduces noticeable UI latency and blocking behaviour during rapid playback or when quickly scrubbing the Time Machine slider.

9 Conclusions & Future Work

9.1 Conclusions

ARES-MAS demonstrates that three independently well-established symbolic AI paradigms BDI reasoning, logic-based safety verification, and classical automated planning can be composed into a single coherent multi-agent system without collapsing into a monolithic controller, provided the boundaries between them are drawn deliberately:

- **Jason** owns deliberation and social coordination.
- **tuProlog** owns hard safety constraints evaluated immediately before irreversible physical actions.
- **STRIPS** owns long-horizon logistics, invoked only when the BDI layer recognises it needs a computed route rather than a single reactive step.

9.2 Future Work

The following extensions were identified as natural next steps, in rough order of expected effort-to-value ratio:

- **Testing and CI:** Implementation of an automated test suite (unit tests for the STRIPS planner and atomic claim mechanism) together with a continuous integration pipeline.
- **System Expansion:** A richer scenario with multiple concurrent minerals and a second independent hazard type reusing the existing infrastructure.
- **Algorithmic Fallback:** A genuine complexity guard on the STRIPS planner that falls back to a heuristic A* search when the state space exceeds a configurable bound.
- **Frontend Optimization:** Migrating from Matplotlib to a native web-based rendering library (such as Plotly) to eliminate the UI playback latency, while simultaneously gaining native hover and zoom interactivity.