ECOSHIELD-MAS

Explainable Hybrid Intelligence for Collaborative Forest-Firefighting UAVs

Final Technical Report

July 2026

System	Implementation	Evaluation
Multi-agent UAV simulation	Python and Streamlit	Four deterministic scenarios

All experimental results in this report are reproduced from the final submitted code and structured JSONL traces.

Contents

Abstract

EcoShield-MAS is a lightweight hybrid intelligent system for coordinating Scout and Fighter unmanned aerial vehicles in a discrete forest-fire environment. The system combines a collaborative Belief-Desire-Intention (BDI) loop, Prolog-style declarative policy guardrails, and STRIPS-inspired logistics planning. Fire evolves through a wind-influenced cellular automaton, while environmental policy prohibits chemical action in protected habitat and under high-wind conditions. A Streamlit dashboard exposes the operational map, agent cognition, policy proofs, and causal event traces. Four deterministic experiments evaluate baseline response, high-wind safety, protected-zone safety, and low-resource return-to-base planning. The final traces confirm 26 extinguishing actions in the baseline mission, correct blocking of all tested unsafe chemical actions, and successful STRIPS return with full resource restoration at tick 13.

1 Introduction

Wildland firefighting requires rapid sensing and suppression while avoiding new risks to responders, aircraft, and ecologically sensitive terrain. Multi-UAV coordination can improve coverage, but autonomous tactical decisions must remain constrained by safety policy and must be explainable after execution. EcoShield-MAS addresses this requirement as a transparent simulation prototype rather than a black-box controller.

The project demonstrates how complementary symbolic AI paradigms cooperate inside one simulation cycle. BDI represents agent cognition and commitment, declarative policy rules decide whether proposed actions are permissible, and state-space planning handles resource recovery without embedding a long chain of hard-coded tactical branches.

2 System Requirements and Scope

- Represent a forest as a discrete two-dimensional grid with forest, fire, burnt, base, water, and protected cells.
- Propagate fire through a deterministic-seed cellular automaton influenced by wind direction and wind speed.
- Coordinate Scout thermal detection and Fighter suppression through shared beliefs.
- Block chemical suppression when the target is protected or the wind exceeds the declared threshold.
- Generate a return-to-base route when battery or payload crosses a resource threshold.
- Expose decisions, proofs, plans, and causal events in a real-time dashboard and JSONL trace.

The prototype does not control physical aircraft. Motion is grid-based, communication is reliable and instantaneous, and weather is fixed within each scenario. These boundaries make the symbolic decision chain reproducible and auditable.

3 Architecture and Design

3.1 Simulation cycle

A declarative JSON scenario defines grid dimensions, tick limit, seed, wind, spread probability, burn duration, safety thresholds, fixed locations, initial fires, and UAV state. Each tick executes fire propagation,

Scout scanning and broadcast, BDI deliberation, policy evaluation, movement or suppression, and structured event recording.

The layers communicate through a single `Simulation` object rather than through duplicated interface state. The environment updates `grid` and `fire_age`; the Scout phase reads the resulting active-fire positions and writes `known_fires` beliefs; BDI deliberation converts those beliefs and resource levels into a target and intention; the planner converts the intention into a sequence of positions; and the policy engine evaluates the next position or suppression target before the engine commits the change. The event logger records the accepted or rejected transition. Consequently, information flows from environment state to cognition, then to a proposed action, then through a policy gate, and finally back to environment state.

Table 1: Hybrid intelligence layers.

Layer	Responsibility	Primary output
Environment	Grid state, fire age, wind-biased spread	Updated cells and fire events
BDI coordination	Belief update, desire selection, intention commitment	Target or recovery intention
Policy engine	Evaluate ecological and safety predicates	Allow/block result and proof facts
Planner	Search valid move states toward a base or target	Sequential move action list
Observability	Render state and export causality	Dashboard and JSONL trace

3.2 Collaborative BDI loop

Scouts maintain a belief set of fires visible within their thermal scan radius. Their observations are merged and broadcast to Fighters, preventing one Scout from overwriting another Scout's information. A Fighter with adequate resources desires fire suppression and commits to an extinguishing intention. When battery or payload is low, the tactical objective is suspended and replaced by resource restoration and return-to-base intentions.

The BDI cycle is recalculated on every tick. Resource restoration has priority: if battery or payload crosses its threshold, the method sets the recovery desire and returns after one planned movement. Otherwise, reported coordinates that no longer occur in `fire_age` are discarded, and the nearest remaining fire is selected by Manhattan distance. A Fighter at the target requests chemical suppression; a Fighter away from the target requests the first step of a route. This repeated reconsideration allows an agent to change from firefighting to recovery without maintaining a separate deterministic state machine.

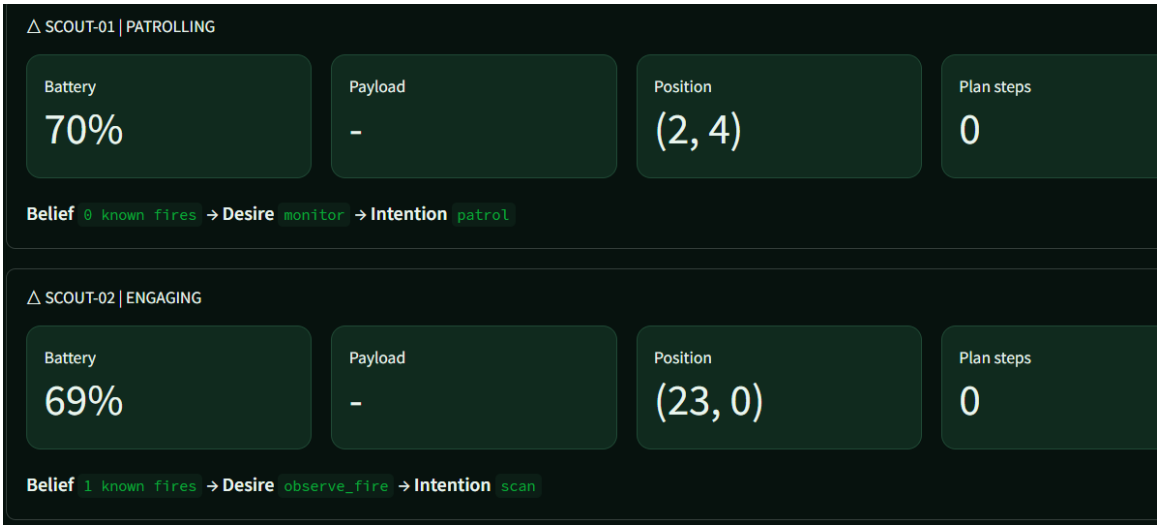

Figure 1: Scout belief, desire, and intention states in the baseline mission.

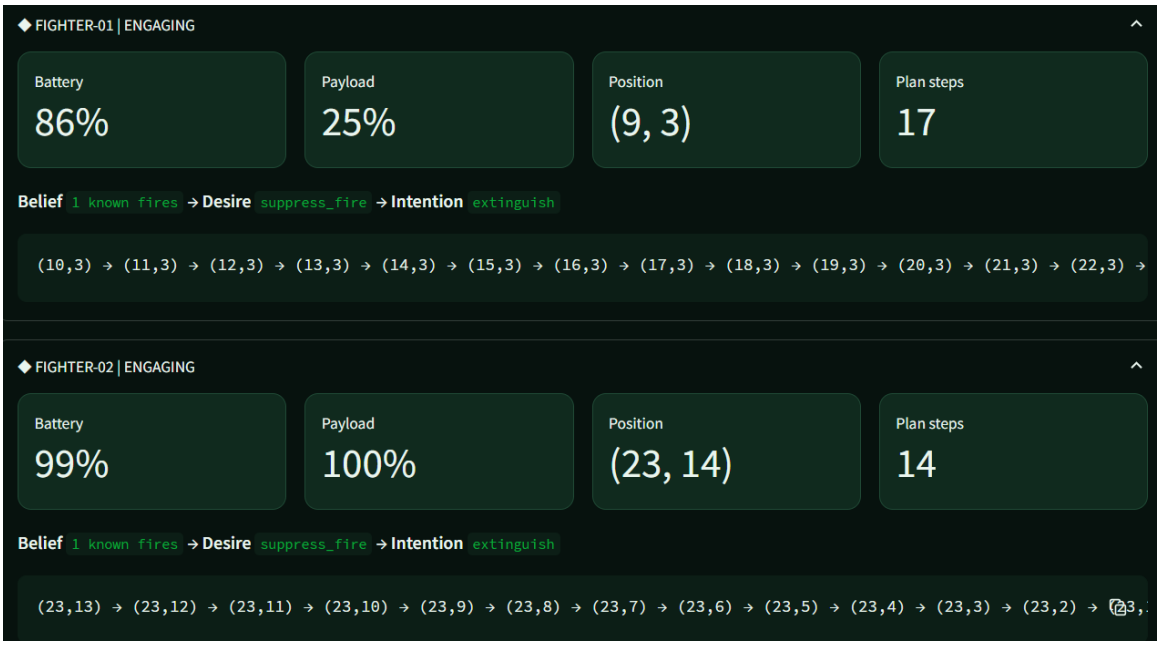

Figure 2: Fighter cognition and ordered plan state in the baseline mission.

3.3 Prolog-style guardrails

The policy layer evaluates explicit predicates and records the rule, result, supporting facts, and negated conditions. Chemical suppressant is permitted only when the UAV is a Fighter, the target cell is not protected, and wind is below the high-wind threshold. Fighter movement into protected habitat is also rejected. The implementation is a lightweight Prolog-style evaluator in Python; it does not embed a separate Prolog runtime.

The policy result is generated from the same facts displayed in the proof panel. For example, `eco_protected=true` is stored as a positive environmental fact, while its `negated` marker records that the rule requires the fact to be false. If either this negated condition or the negated high-wind condition fails, the conjunction fails. The engine receives the resulting `Proof` object and checks `proof.result` before modifying a cell, position, battery, or payload value. Policy evaluation is therefore part of action execution rather than a descriptive check performed after the action.

```
can_use_chemical(UAV, Cell) :-
    is_fighter(UAV),
    \+ eco_protected(Cell),
    \+ high_wind.
```

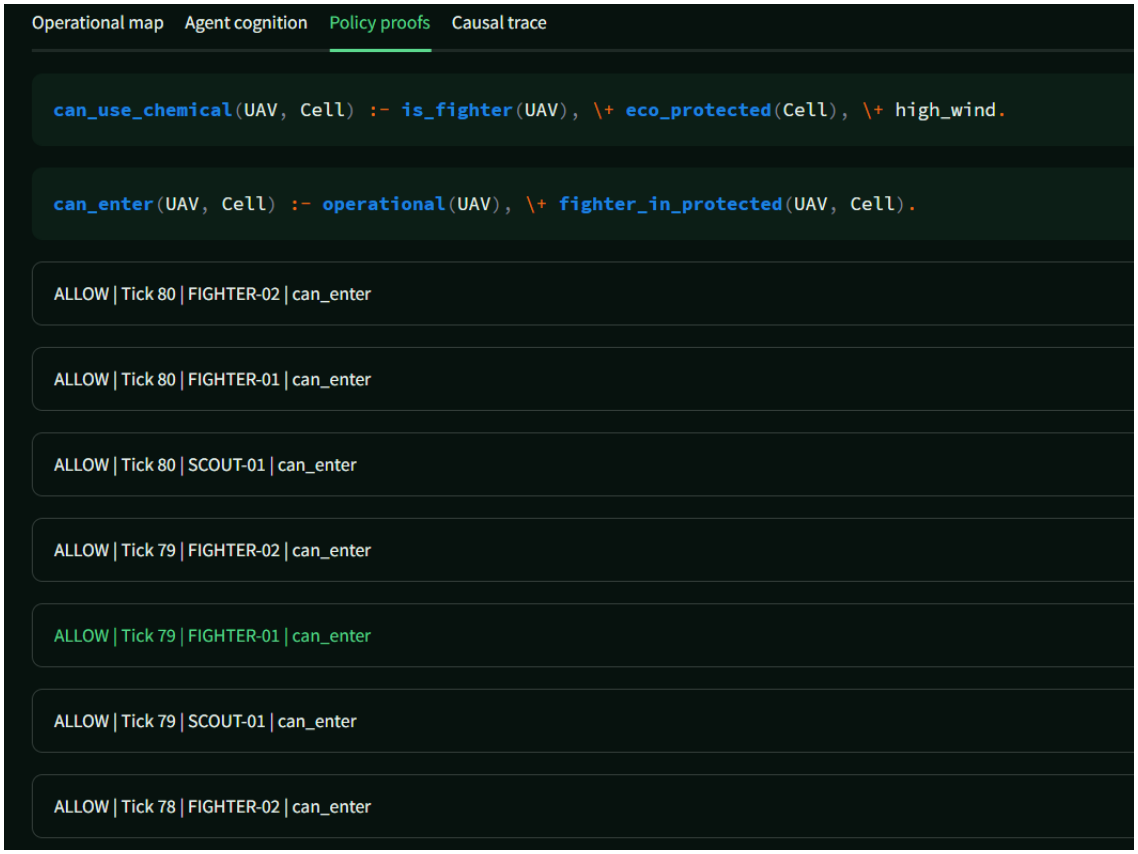

Figure 3: Inspectable policy rules and proof results in the dashboard.

3.4 STRIPS-inspired logistics planning

The planner models the UAV location as a state and each valid grid transition as a move action. Breadth-first forward search finds a shortest route to any base while excluding protected cells for Fighters. Low battery or payload activates the recovery goal. On arrival, a base-service effect restores battery and payload to 100 percent. This is a focused STRIPS-inspired logistics planner rather than a general-purpose domain parser.

The search queue initially contains the UAV's position. Each expansion generates in-bounds neighbours that are not protected and have not been visited. A `parent` entry is stored when a neighbour is first discovered. Once a base is found, following these parent entries back to the start constructs the route, which is reversed into execution order. The engine stores that list in `uav.plan` and removes one position per tick. Reaching a base triggers the service effect: both resources are set to 100 and the completed plan is cleared.

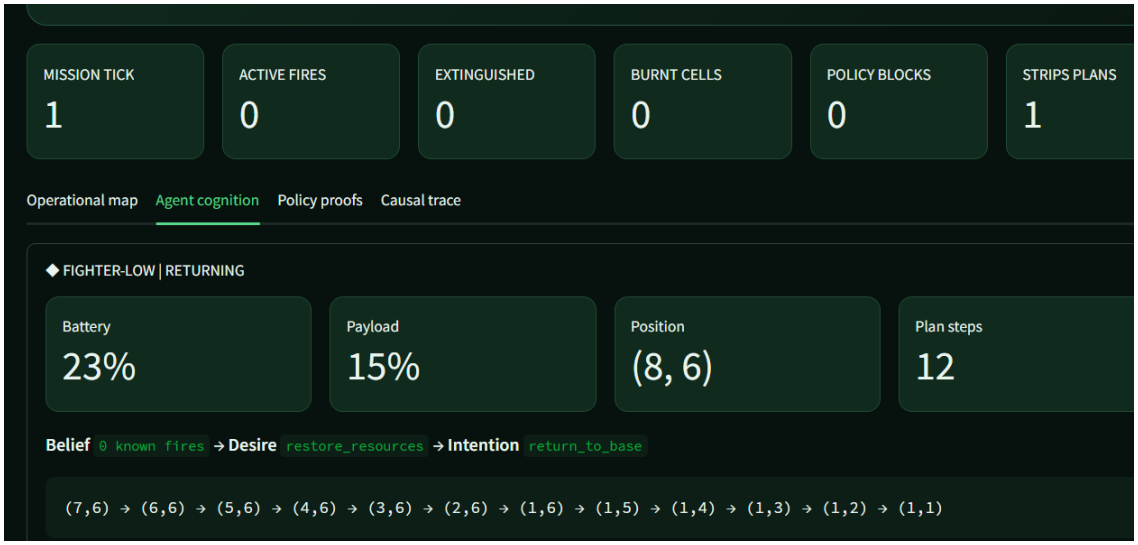

Figure 4: Return plan after the resource threshold is detected at tick 1.

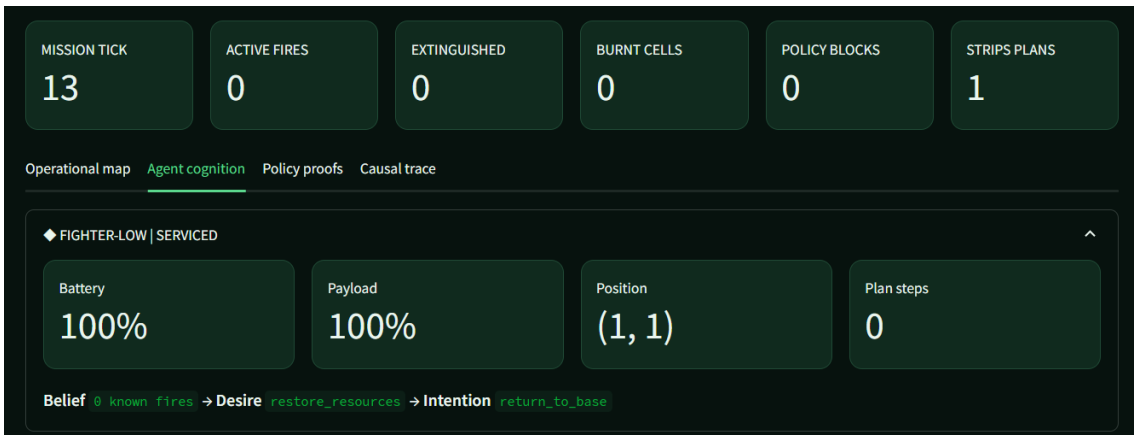

Figure 5: Goal state at tick 13: base reached and both resources restored.

4 Implementation

The implementation separates the simulation state from the Streamlit view. `models.py` converts a scenario into typed objects, `engine.py` owns all state transitions, `policy.py` evaluates action permissions, and `planner.py` searches for routes. The dashboard never implements a tactical rule; it only calls the engine and renders its public state. This separation is important because the same engine can be executed by the test suite without starting a browser.

Table 2: Project structure.

File or directory	Purpose
`app.py`	Streamlit mission-control dashboard
`ecoshield/engine.py`	Tick scheduler, fire propagation, BDI coordination, actions, metrics, and event logging
`ecoshield/policy.py`	Prolog-style rules and proof objects
`ecoshield/planner.py`	Forward breadth-first route search
`scenarios/`	Baseline and three targeted alternative experiments
`tests/test_system.py`	Eight automated behavioural tests

4.1 Scenario construction and validation

`Scenario.from_dict` is the boundary between untrusted JSON input and the simulation. Coordinate arrays such as `[6,4]` are converted into immutable `Position` objects; UAV roles are normalised to lower case; optional parameters receive documented defaults. The method then calls `validate()`, which rejects grids outside 5–80 cells, missing bases, duplicate UAV identifiers, unsupported roles, and coordinates outside the grid. Consequently, the algorithms can assume that their inputs satisfy these basic invariants.

Listing 1. Scenario conversion and early validation.

```
pos = lambda p: Position(int(p[0]), int(p[1]))
uavs = [UAV(id=u["id"], role=u["role"].lower(),
            position=pos(u["position"]),
            battery=float(u.get("battery", 100)),
            payload=float(u.get("payload", 100)))
        for u in raw["uavs"]]
scenario.validate()
```

`Position` is declared as a frozen dataclass. This makes coordinates hashable, so they can be used as keys in `fire_age`, members of the protected-cell set, and nodes in the planner's `parent` dictionary. Distance is Manhattan distance, $|x_1 - x_2| + |y_1 - y_2|$, because movement is restricted to four neighbours.

4.2 Tick scheduling and fire-state update

`Simulation.step()` defines the exact semantics of one discrete tick. The environment advances first, Scouts observe the new fire state second, and then every UAV deliberates and performs at most one movement or suppression action. Finally, a metric snapshot is appended to the event trace.

Listing 2. Complete tick scheduler.

```
self.tick += 1
self._spread_fire()
self._scout_phase()
for uav in self.scenario.uavs:
    self._agent_bdi(uav)
self.log("tick_completed", "system",
         cause="scheduler", details=self.metrics())
```

Fire cells are stored in `fire_age: dict[Position,int]`; therefore the dictionary represents both the active-fire set and the age of every fire. For each active cell, `_spread_fire()` examines four neighbours. A neighbour is eligible only when its cell type is forest or protected habitat. The vector from the fire to the neighbour is compared with the scenario wind vector. The implemented ignition probability is

$$p = \min\left(0.95,\ p_0\left(1 + s\max(0, v_x d_x + v_y d_y)\right)\right),$$

where p_0 is `spread_probability`, s is wind speed, (v_x, v_y) is the neighbour direction, and (d_x, d_y) is wind direction. Thus a downwind neighbour receives a positive multiplier, while an upwind neighbour receives no bonus.

Listing 3. Wind-biased cellular-automaton transition.

```
vx, vy = cell.x - fire.x, cell.y - fire.y
alignment = max(0, vx * wind.dx + vy * wind.dy)
probability = min(
    .95,
    spread_probability * (1 + wind.speed * alignment)
```

```
)
if self.rng.random() < probability:
    additions.append(cell)
```

New fires and burnt-out cells are first collected in temporary lists and are applied only after the iteration. This two-phase update prevents a cell ignited in the current tick from spreading again in that same tick. The random generator is local to the simulation and initialised with `random.Random(seed)`, which makes the state trajectory repeatable for a fixed scenario and action order.

4.3 Collaborative BDI deliberation

The BDI data is represented directly in each `UAV`: a belief dictionary, a selected desire, a committed intention, a target, and an ordered plan. During the Scout phase, each Scout filters active fires by scan radius. The system unions all observations in a set before broadcasting them, so one Scout cannot overwrite a fire detected by another.

Listing 4. Merging Scout observations into Fighter beliefs.

```
visible = [p for p in fires
           if scout.position.distance(p) <= 5]
shared_fires.update(visible)

merged = [asdict(p) for p in
          sorted(shared_fires, key=lambda p: (p.y, p.x))]
for fighter in fighters:
    fighter.belief["known_fires"] = merged
```

`_agent_bdi()` implements an explicit priority ordering. First, a UAV with zero battery is grounded. Second, battery at or below 25 percent, or a Fighter payload at or below 20 percent, selects `restore_resources/return_to_base`; this branch returns immediately, so resource recovery pre-empts firefighting. Otherwise, stale fire beliefs are removed by checking membership in `fire_age`. A Fighter selects the nearest remaining target and commits to `suppress_fire/extinguish`; a Scout commits to `observe_fire/scan`. With no valid target, the UAV selects `monitor/patrol`. This is how beliefs cause goal and intention changes rather than merely being labels shown by the interface.

4.4 Policy evaluation before state mutation

The policy engine is called at the action boundary, immediately before a move or chemical release changes the world. `can_use_chemical()` constructs the three relevant facts and then evaluates their conjunction. It returns a `Proof` containing the rule string, Boolean result, and fact values.

Listing 5. Executable chemical guardrail.

```
facts = [
  {"predicate": "is_fighter", "value": role == "fighter"},
  {"predicate": "eco_protected", "value": cell in protected,
   "negated": True},
  {"predicate": "high_wind", "value": speed >= threshold,
   "negated": True},
]
result = (role == "fighter" and cell not in protected
          and speed < threshold)
return Proof(rule, result, facts)
```

In `_extinguish()`, a successful proof and at least 15 payload units are both required. Only then is the target removed from `fire_age`, changed back to forest, and charged 15 payload units. If the proof is false,

no world state is changed; instead, `chemical_blocked` is logged with the proof. Movement follows the same pattern through `can_enter()`. This placement means an unsafe intention can be proposed but cannot become an executed action.

4.5 STRIPS-inspired route generation

The planner maps logistics to a small state-space problem: a state is `at(Position)`, the initial state is the UAV's current position, a goal is any base position, and an action moves to one valid four-neighbour cell. For Fighters, protected coordinates are passed as the `blocked` set.

Listing 6. Breadth-first state-space expansion and reconstruction.

```
frontier = deque([start])
parent = {start: None}
while frontier:
    current = frontier.popleft()
    for nxt in four_neighbours(current):
        if valid(nxt) and nxt not in blocked and nxt not in parent:
            parent[nxt] = current
            if nxt in goals:
                found = nxt
                break
            frontier.append(nxt)

while node != start:
    path.append(node)
    node = parent[node]
return list(reversed(path))
```

Breadth-first search expands all routes of length k before routes of length $k + 1$, so the first base reached is a shortest path when every move has equal cost. The `parent` map records how each state was reached and reconstructs the ordered action sequence backwards. Its time and memory complexity are $O(V + E)$; in a four-neighbour grid, V is at most width times height and E is proportional to $4V$.

4.6 Logging, metrics, and presentation

The central `log()` method appends a dictionary with UTC timestamp, tick, event, actor, cause, and structured details. Metrics are not separately edited counters: they are derived from the current world or counted from events. For example, `active_fires` is `len(fire_age)`, while `fires_extinguished` counts `fire_extinguished` events. This reduces the risk that a displayed counter disagrees with the trace. Finally, `jsonl()` serialises one event per line. `app.py` reads these objects to draw the map, cognition panels, proofs, and causal trace; it does not reimplement the decisions.

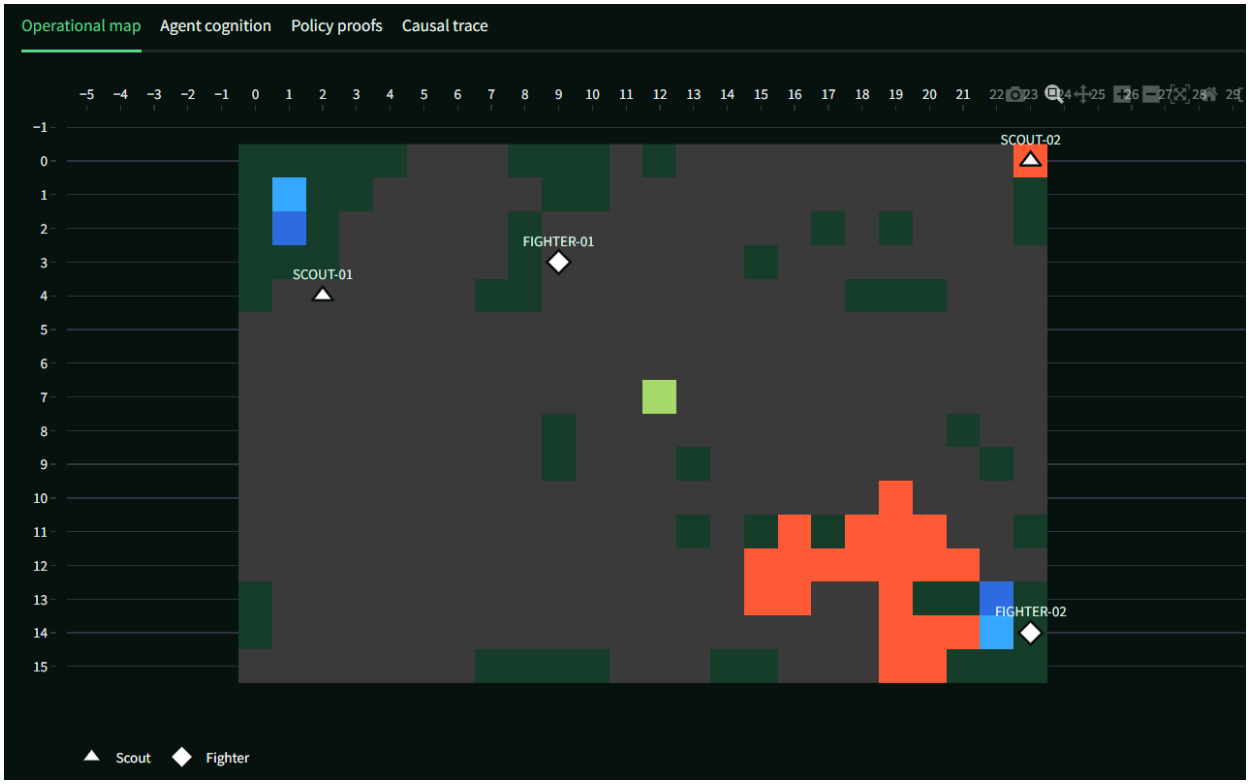

Figure 6: Baseline operational map at tick 80.

5 Experimental Method

The evaluation combines scenario-level integration tests with code-level automated tests. Each experiment starts from a newly constructed `Simulation`; no UAV state, random-generator state, event, or proof is shared between runs. The JSON scenario is therefore the complete controlled input to an experiment.

5.1 Controlled scenario design

The baseline exercises the complete mission over 80 ticks. The other three scenarios change only the conditions necessary to isolate one decision path. The high-wind case sets speed to 0.90 while the declared threshold remains 0.75. The protected-zone case places a detected fire and a Fighter on protected cell (6,4), which bypasses route selection and directly tests the chemical rule. The low-resource case starts a Fighter at (9,6) with battery 24 and payload 15, so the recovery branch must be selected at the first deliberation.

Table 3: Deterministic evaluation scenarios.

Experiment	Seed	Ticks	Purpose
Pine Ridge Response	17	80	End-to-end baseline mission
High Wind Chemical Guardrail	31	12	Verify high-wind chemical prohibition
Protected Zone Chemical Guardrail	41	1	Verify ecological target prohibition
Low Resource STRIPS Return	51	13	Verify route generation and resource restoration

5.2 Execution and evidence-collection protocol

Each scenario was evaluated with the same procedure:

1. Load the named JSON file with `Scenario.from_dict()` and confirm the `simulation_started` event contains the expected scenario, wind speed, threshold, and seed.
2. Create a fresh `Simulation` and call `step()` exactly the number of times declared by the scenario. No manual state edits are performed after construction.
3. Inspect rule proofs when a policy query occurs and inspect the UAV's belief, desire, intention, position, and plan at the stated checkpoint.
4. Export `Simulation.jsonl()` and parse every line as a JSON object. The final `tick_completed` event supplies the terminal metric snapshot; action events provide independent causal evidence.
5. Compare event counts and terminal state against the expected assertions in `tests/test_system.py`.

For the low-resource scenario, tick 1 is recorded separately because it exposes the newly generated 13-action plan. Twelve further calls reach tick 13, where the Fighter must be at base (1,1), have an empty plan, status `serviced`, and battery and payload equal to 100. For the high-wind scenario, the absence of `fire_extinguished` events is checked in addition to the terminal active-fire count, because fires may disappear by natural burn-out rather than suppression.

5.3 Metric computation

Table 4: How reported measures are computed.

Metric	Computation and interpretation
`active_fires`	Number of keys currently in `fire_age`.
`burnt_cells`	Count of grid cells whose enum value is `BURNT`.
`protected_at_risk`	Protected coordinates that are also active-fire keys.
`policy_blocks`	Number of `action_blocked` and `chemical_blocked` events.
`fires_extinguished`	Number of successful `fire_extinguished` events.
`strips_plans`	Number of `strips_plan_created` events.
`resources_restored`	Number of completed `resources_restored` events.

The measures distinguish environmental outcomes from agent actions. In particular, `active_fires=0` does not itself mean that UAVs extinguished the fire; the interpretation requires `fires_extinguished`, burnt-cell counts, and the underlying event types.

5.4 Automated verification

The final command `pytest -q` produced eight passing tests. Deterministic replay creates two independent simulations, advances both four ticks, and asserts equality of their grids and metrics. Policy unit tests exercise protected, high-wind, and permitted chemical queries. The planner test asserts that the returned route reaches its goal and has no intersection with blocked cells. Integration tests run each targeted scenario and inspect both metrics and proof facts. A collaboration test invokes the Scout phase and verifies that both Fighters receive the union of the two expected fire coordinates. The JSONL test parses the export and checks that every event contains `tick`, `event`, `actor`, `cause`, and `details`. Together, these tests cover the main state transformations, policy outcomes, planning behaviour, collaboration, and trace structure.

6 Results

Table 5: Final experimental metrics.

Scenario	Ticks	Active	Exting.	Burnt	Risk	Blocks	Plans	Restored
Baseline	80	21	26	302	0	0	4	4
High wind	12	0	0	3	0	8	0	0
Protected zone	1	1	0	0	1	1	0	0
Low resource	13	0	0	0	0	0	1	1

6.1 Baseline mission

The baseline completed all 80 ticks. Fighters executed 26 successful extinguishing actions, generated four resource-recovery plans, and completed four base-service cycles. No protected habitat remained actively burning at the final tick and no policy violation was executed. The 302 burnt cells show that the scenario is intentionally challenging: the prototype demonstrates coordinated and constrained behaviour, not optimal containment.

The event sequence explains these totals. Scout broadcasts place active fire coordinates in both Fighter belief sets. Each Fighter repeatedly selects the nearest still-active coordinate, follows a planned route, and invokes the chemical rule on arrival. A successful proof removes that coordinate from `fire_age` and emits `fire_extinguished`; consuming payload and battery eventually activates the recovery branch. The four `strips_plan_created` events are therefore paired with four later `resources_restored` events. Meanwhile, the cellular automaton continues to spread before agent actions, so containment and environmental growth occur in the same run. This interaction produces both 26 extinguishing events and 302 burnt cells.

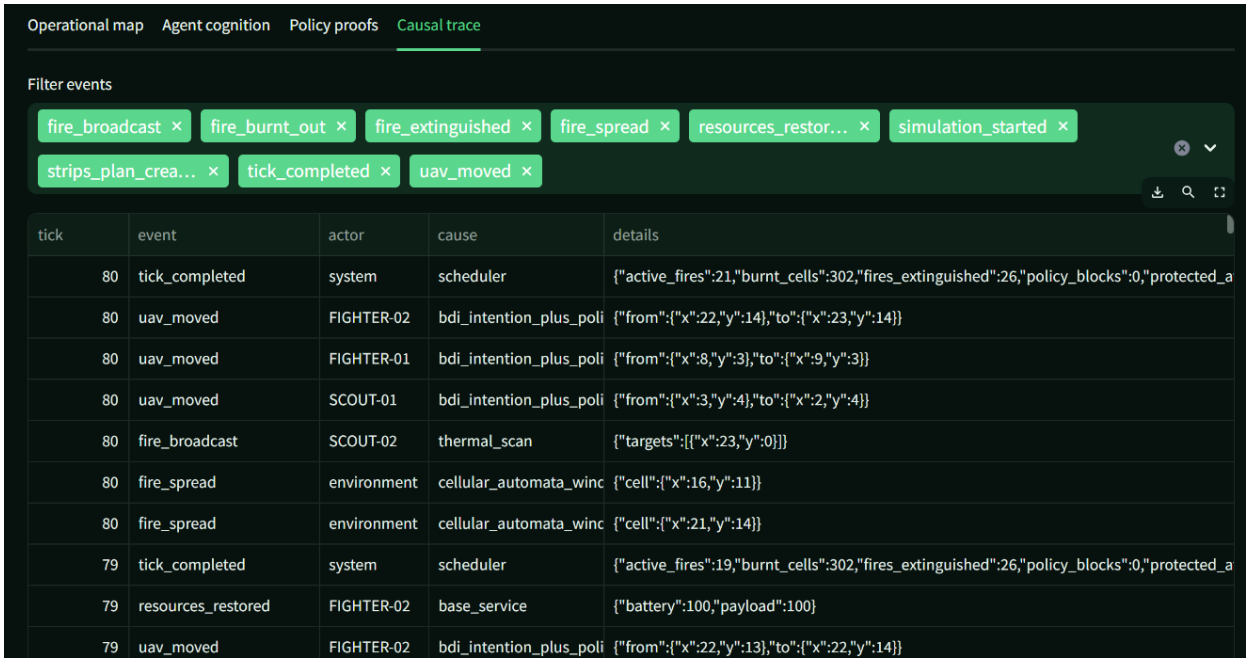

Figure 7: Baseline causal trace with final metrics and resource-restoration evidence.

6.2 High-wind guardrail

Wind speed was 0.90 against a high-wind threshold of 0.75. Eight attempted chemical actions were blocked. The first proof records `is_fighter=true`, `eco_protected=false`, and `high_wind=true`; therefore

`can_use_chemical=false`. No fire was extinguished by chemical action. Three cells eventually burnt out through the environmental model, which explains the zero active fires at tick 12 without contradicting the prohibition.

At each attempted release, the Fighter role fact is true and the protected-cell fact is false, but `wind_speed >= high_wind_threshold` evaluates to true. Because the rule requires `not high_wind`, the proof result is false and `_extinguish()` leaves the fire and payload unchanged. Repeated deliberation causes further attempts and eight blocked events. The active-fire count later becomes zero only when the burn-duration transition changes the remaining fire cells to `BURNT`.

6.3 Protected-zone guardrail

At tick 1 the fire occupied protected cell (6,4). The rule proof records `is_fighter=true`, `eco_protected(6,4)=true`, and `high_wind=false`. The chemical action was blocked, `policy_blocks` increased to one, and `protected_at_risk` remained one. This isolates ecological policy from the weather constraint.

The one-tick configuration places the Fighter at the target so that route planning cannot obscure the policy test. The wind condition passes, but `cell in protected` evaluates to true. The negated ecological condition therefore fails, producing `chemical_blocked`. Since no suppression state change occurs, the same coordinate remains in both the protected set and `fire_age`, which directly yields `protected_at_risk=1`.

6.4 Low-resource STRIPS return

The Fighter began at (9,6) with battery 24 percent and payload 15 percent. At tick 1 it selected `restore_resources`, committed to `return_to_base`, and produced a 13-action route to base (1,1). The route avoided the protected barrier. At tick 13, `resources_restored` was recorded and both battery and payload became 100 percent. The before-and-after interface states and JSONL events agree.

Both initial resource values satisfy the low-resource condition, so this branch is selected before any fire-target branch. Breadth-first search records a route around the protected barrier and the engine consumes its first position during tick 1. Each subsequent tick removes one more route position and applies the movement battery cost. On the tick that position (1,1) is reached, the base service block runs in the same deliberation call, clears the plan, changes the status to `serviced`, restores both values, and records `resources_restored`.

7 Explainability and Traceability

Every event contains timestamp, tick, event type, actor, cause, and structured details. Decision chains can therefore be reconstructed from perception to action: `thermal_scan` produces `fire_broadcast`; the broadcast updates Fighter beliefs; the BDI loop proposes a tactical or recovery intention; the policy engine supplies an allow/block proof; and the executed result updates the environment.

The trace is produced at the same locations where decisions and state changes occur. A broadcast is logged immediately after a Scout constructs its visible target list; a plan is logged immediately after search returns its ordered actions; movement is logged only after `can_enter` succeeds; and a chemical block contains the exact proof returned by the policy engine. A typical blocked event has the following structure:

```
{
  "tick": 1,
  "event": "chemical_blocked",
```

```
  "actor": "FIGHTER-HIGH-WIND",
  "cause": "prolog_guardrail",
  "details": {
    "cell": {"x": 6, "y": 4},
    "proof": {"result": false, "facts": [...]}
  }
}
```

An audit starts with `simulation_started` to confirm the scenario, seed, wind, and threshold. It then filters events by actor and tick. A `fire_broadcast` identifies the perceived targets; the UAV panel exposes the resulting desire, intention, and plan; the next movement or suppression event identifies the proposed action; and the attached proof explains whether that action was admitted. The final `tick_completed` event connects these individual transitions to the reported aggregate metrics. This ordering permits the dashboard state and the JSONL record to be checked against one another.

- **Agent level:** belief count, desire, intention, target, and plan.
- **Rule level:** evaluated rule, result, facts, and negation markers.
- **Planner level:** ordered move actions and resource-threshold cause.
- **System level:** environmental transition and aggregate metrics.

8 Limitations and Future Work

The simulation uses four-neighbour grid motion and does not model altitude, collision avoidance, flight dynamics, communication loss, or continuous time. Wind is fixed inside a scenario rather than changing by tick. The Prolog-style evaluator and STRIPS-inspired planner are deliberately lightweight. Fire behaviour excludes humidity, terrain slope, vegetation class, and suppression uncertainty.

Future work should introduce time-varying weather, richer ecological rules, multi-agent task allocation, communication delay and failure, continuous trajectory planning, collision constraints, and comparative evaluation against non-BDI and non-policy baselines. A separate Prolog runtime and a domain-independent STRIPS/PDDL interface would strengthen the symbolic-AI claims.

9 Conclusion

EcoShield-MAS demonstrates an integrated and explainable approach to collaborative UAV firefighting. A scenario is validated and converted into a shared simulation state; the cellular automaton advances that state; Scout observations update Fighter beliefs; BDI deliberation selects a tactical or recovery intention; route search supplies ordered movement actions; and the policy engine admits or rejects each state-changing action. Event records are written at these transition points, linking inputs, decisions, proofs, and outcomes.

The experiments exercise this chain under normal operation, high wind, protected habitat, and depleted resources. Their results follow from distinct mechanisms: approved proofs permit suppression, failed negated conditions block chemical release without changing the world, and resource thresholds replace the tactical intention with a planned return to base. Within its prototype scope, the system therefore provides a reproducible implementation of collaboration, ecological constraint enforcement, logistics recovery, and decision traceability.

References

[1] R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, no. 3–4, pp. 189–208, 1971.

[2] R. Kowalski, *Logic for Problem Solving*. North-Holland, 1979.

[3] A. S. Rao and M. P. Georgeff, "BDI Agents: From Theory to Practice," in *Proceedings of the First International Conference on Multi-Agent Systems*, 1995.

[4] S. Wolfram, "Statistical Mechanics of Cellular Automata," *Reviews of Modern Physics*, vol. 55, no. 3, pp. 601–644, 1983.

A Reproduction Procedure

Create a Python environment, install `requirements.txt`, and run `streamlit run app.py`. Select each built-in experiment, click *Activate selected experiment*, verify the green *ACTIVE* label, and advance the required number of ticks. Download the JSONL trace after the final tick. The expected checkpoint values are those reported in Section 6.

For the STRIPS experiment, run one tick to capture the generated route, then advance exactly 12 additional ticks. At tick 13 the Fighter status is `SERVICED` at (1,1), with battery and payload both equal to 100 percent.