

A Symbolic Decision Framework for Multi-Agent 3D Simulation

Corrado Santoro¹, Federico Fausto Santoro¹

¹*Department of Mathematics and Computer Science, University of Catania, Catania, Italy*

Abstract

This paper presents the design and implementation of a multi-agent 3D simulation platform in which symbolic decision making is deliberately separated from rendering and physics. The system combines Godot for real-time 3D simulation, a Scala 3 backend for WebSocket transport and state management, and a pluggable Prolog decision layer implemented through tuProlog in the prototype. Each simulated agent periodically sends percepts to the decision server; the server enriches them with state-aware facts, evaluates a per-agent theory through a common predicate, updates the authoritative energy state, and returns a symbolic action to be interpreted locally by the Godot scene. The prototype is validated across four scenarios: simple adversarial agents, a two-player soccer environment, path-following vehicles with traffic lights and intersection management, and top-down tanks with line-of-sight combat. The contribution is primarily architectural and educational: it demonstrates how a lightweight symbolic reasoning layer can be integrated with a game-engine runtime without hard-coding behaviour inside the simulation, while preserving extensibility through hot-swappable theories, a typed functional backend and deployment options ranging from localhost execution to distributed client/server configurations.

Keywords

multi-agent systems, symbolic artificial intelligence, Prolog, Godot, real-time simulation

1. Introduction

Real-time multi-agent simulations are an attractive medium for studying the relationship between perception, decision and action. A game engine offers physics, rendering, collisions, scene composition and immediate visual feedback, while symbolic artificial intelligence offers rule-based policies that can be inspected, modified and compared. In many prototypes, however, these two concerns become entangled: behaviours are encoded directly in the game scripts, rule changes require editing the simulation code, and the difference between world mechanics and decision logic becomes increasingly difficult to maintain.

The project discussed in this paper addresses that problem by separating the simulation substrate from the symbolic decision layer. Godot is responsible for the 3D world, sensing, object state, user interface and physical action execution. A Scala 3 backend exposes a WebSocket endpoint, maintains server-side agent state, and invokes a PrologService implementation to select actions. The current implementation uses tuProlog, while the architecture admits other Prolog engines or custom symbolic backends. Prolog theories implement compact policies for different domains. The same communication backbone supports simple agents, soccer players, vehicles and tanks, each with its own percept vocabulary and action mapping.

The guiding idea is not to build an optimal game AI, nor to claim a novel reasoning algorithm. The goal is to engineer a modular experimental platform in which declarative policies can be changed independently from the real-time simulation. This is especially useful in an educational or research-prototyping context: a student can tune a Prolog file and observe behavioural consequences in a 3D scene without touching Scala transport code or Godot physics code.

The implemented system follows a perception-action loop. Each agent builds a list of symbolic percepts, for example `enemy_close`, `ball_near`, `light_red`, or `enemy_blocked`. The client sends these percepts, together with its identifier and optionally a Prolog theory, to the Scala server. The

WOA 2026: 27th Workshop “From Objects to Agents”, June 15–17, 2026, Salerno, Italy

✉ corrado.santoro@unict.it (C. Santoro); federico.santoro@unict.it (F.F. Santoro)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

server persists theories per agent, enriches percepts with internal facts such as `low_energy` and `last_action_*`, queries Prolog, applies an energy update, and sends back an action. The action remains symbolic at the boundary; Godot decides how to realise it through movement, impulses, shooting, path routing or other local effects.

The rest of the paper is organised as follows. Section 2 positions the work within multi-agent simulation and logic-based behaviour. Section 3 describes design requirements derived from the project. Section 4 presents the core architecture and communication protocol. Section 5 discusses deployment topologies. Section 6 analyses the decision model and Prolog policies. Section 7 discusses the implementation in Scala and Godot. Section 8 describes the four scenarios. Section 9 reports the qualitative validation strategy. Section 10 discusses limitations and future work, and Section 11 concludes.

2. Background and Motivation

Multi-agent systems study collections of autonomous entities that perceive, decide and act in a shared environment [1]. Agent-oriented programming emphasises the explicit representation of agents, their mental or informational state, and the actions available to them [2]. In a real-time 3D setting, these abstractions must coexist with simulation concerns such as collision geometry, frame updates, animation, path following and input handling.

Game engines are well suited to the latter concerns. Godot provides a scene-tree model, physics bodies, areas, ray casts, signals and a scripting runtime suitable for rapid 2D and 3D prototyping [3]. The project uses these facilities to create interactive scenarios and to translate symbolic actions into continuous motion. For instance, the action `move_to_ball` in the soccer scenario becomes a direction vector and a velocity, while `turn_random` in the vehicle scenario becomes a pending path-routing command.

Logic programming is suitable for expressing compact decision policies. Prolog represents behaviour as ordered rules over symbolic facts. In this project, each theory exposes the same predicate:

Listing 1: Common Prolog decision contract.

```
decide_action(+Percepts, -Action)
```

The `Percepts` argument is a list of strings received from the simulation and enriched by the backend. The `Action` argument is unified with a symbolic command. The rules use ordering and cut to obtain deterministic priority-based behaviour. This is intentionally simple: the work favours inspectability and tunability over sophisticated planning.

The prototype uses tuProlog, a lightweight Prolog designed for Java-based and distributed applications [4]. This choice is also culturally meaningful for the project, since tuProlog was developed in the Italian academic ecosystem at the University of Bologna. Architecturally, however, tuProlog is not a hard constraint: the Scala backend depends on the `PrologService` abstraction, so another Prolog engine or a custom symbolic reasoner could be introduced by providing a different service implementation.

Scala 3 was also an engineering choice rather than an unavoidable requirement. It was selected because the backend benefits from strong static types, explicit effect management and functional error composition. Cats Effect, FS2 and http4s structure the WebSocket service as typed, effectful and composable infrastructure [5, 6, 7]. JSON encoding and decoding are handled with Circe [8]. These libraries make the decision server more explicit about side effects, parsing failures and shared mutable state than a more ad-hoc scripting backend would be.

The motivation for introducing a separate decision server is threefold. First, it enforces a boundary between continuous world mechanics and discrete symbolic reasoning. Second, it makes policies hot-swappable at agent granularity: two agents in the same scene may use different Prolog files. Third, it provides a central place for cross-cutting state, such as server-authoritative energy and recent-action memory, without distributing that logic across multiple Godot scripts.

The work is best understood as a systems-integration contribution. It does not introduce a new semantics for agents, a new Prolog interpreter, or a new game-engine feature. Instead, it assembles existing technologies into an experimental architecture with a strict separation of responsibilities. This positioning matters because the value of the system is not in any individual component. Godot already provides scene management and physics; Scala already provides strong abstractions for typed services; Prolog already provides symbolic rule evaluation. The project contribution is the way these components are connected so that changing one policy file can alter the behaviour of a live 3D agent without changing the transport layer or the simulation framework.

This differs from common game scripting patterns in which behaviour is encoded directly in update callbacks. Direct scripting is often effective for small games, but it makes policy comparison more difficult: an aggressive and a cautious agent may differ across many imperative branches. In the presented architecture, such differences can be isolated in two compact theories that share the same percept vocabulary. The effect is similar to using a strategy object, except that the strategy is declarative, serialisable and independent from the language of the simulation.

The project also differs from fully centralised simulation approaches. The server does not simulate positions, collisions or ball dynamics. It receives symbolic summaries and returns symbolic decisions. This avoids duplicating the world model in Scala and keeps the real-time loop local to Godot. The trade-off is that the quality of decisions depends heavily on percept design: if a fact is not exposed by the client or enriched by the server, Prolog cannot reason about it. Percept engineering, therefore, becomes a central design activity.

The transport protocol needs to support bidirectional, low-overhead communication between a long-running Godot scene and a long-running Scala service. WebSocket is a natural fit because it upgrades an HTTP connection into a persistent channel suitable for streams of messages [9]. It was chosen mainly for simplicity and deployability: the same endpoint can be consumed by a Godot desktop client, a browser client, a test script or any external runtime capable of opening a WebSocket connection. The project uses one endpoint for all agent families. This reduces protocol surface area and makes the server a reusable decision service rather than a collection of scenario-specific HTTP routes.

The choice also leaves room for different update policies. A simple polling architecture would force all agents to send at a fixed rate, regardless of whether percepts changed. A pure event architecture would miss a continuous state that changes gradually, such as the distance to a ball or progress along a path. The implemented hybrid uses periodic sending as a baseline and urgent sending for discrete sensor changes. This is especially useful in the vehicle scenario, where entering a semaphore area or right-priority sensor should be communicated immediately.

3. Requirements and Design Goals

The requirements were derived from the implemented project documentation and from the final behaviour of the prototype. They can be grouped into functional, non-functional and implementation-oriented requirements.

The main functional requirement is an end-to-end decision loop: each Godot agent must periodically send its identifier, percepts and optional theory to the server; the server must return an action and an updated energy value; the agent must apply that action in the scene. The same loop must support heterogeneous families of agents, including simple arena agents, soccer players, vehicles and tanks.

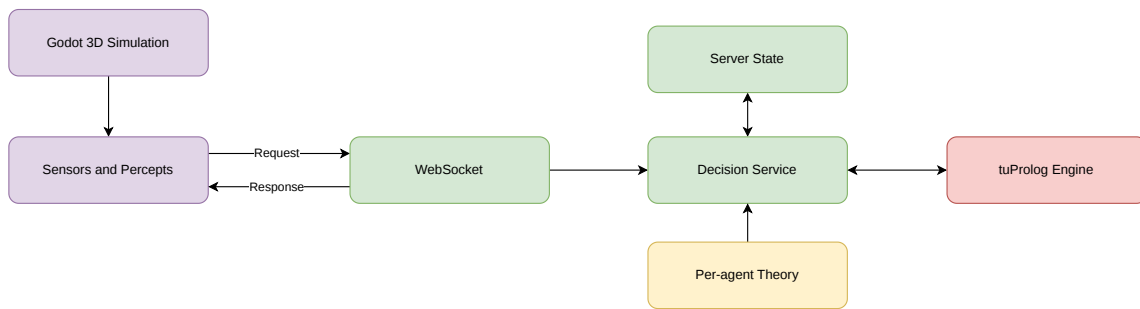
A second functional requirement is theory management per agent. An agent can load a .p1 file from its Godot inspector and send the text to the server. The server stores that theory under the agent identifier and reuses it in subsequent ticks. This avoids resending large theory text on every message while still allowing logic changes without recompiling the Scala backend.

A third requirement is state-aware decision enrichment. The server does not blindly pass client percepts to Prolog; it adds facts derived from authoritative state, especially `low_energy` and `last_action_*`. This makes it possible for Prolog rules to react to recent history while keeping the client-side percept builder focused on the external world.

Table 1

Condensed requirements used to guide the implementation.

Category	Requirement
Functional	Periodic percept-action loop over WebSocket; per-agent Prolog theory; multiple scenarios using one protocol; health endpoint.
Domain	Agent percepts and actions must be symbolic at the network boundary; Godot maps actions to physical effects.
State	Server tracks energy, last action, last percepts and last decision timestamp per agent.
Robustness	Invalid JSON, missing theory and failed Prolog resolution are represented as typed errors.
Extensibility	Adding a scenario should not require a new transport protocol or a new backend service.
Validation	Behaviour must be observable in scene and regressions must be checked scenario by scenario.

**Figure 1:** High-level architecture: Godot produces percepts and applies actions, while Scala orchestrates stateful symbolic decisions through PrologService and the tuProlog adapter used in the prototype.

The non-functional requirements include modularity, maintainability, reactivity and robustness. The backend must be split into modules with explicit responsibilities rather than a monolithic WebSocket handler. The Prolog theories must remain readable and deterministic. The loop must be fast enough for real-time scenes, and invalid JSON or Prolog failures must produce controlled error payloads rather than crashing the server. Table 1 summarises the most relevant requirements.

The implementation requirements constrain the prototype stack: Godot 4.6 and GDScript for simulation; Scala 3 and sbt for the backend; http4s, FS2 and Cats Effect for streaming and effect management; Circe for JSON; and tuProlog 3.3.0 as the current logic-inference adapter. These choices are not arbitrary. They create a JVM-hosted symbolic backend that can communicate with Godot using a standard bidirectional protocol, while keeping the policy language independent from both Scala and GDScript and leaving room for alternative PrologService implementations.

4. System Architecture

The system follows a two-level client/server architecture. The Godot client is the simulation level: it owns scene nodes, colliders, timers, areas, ray casts, rigid bodies and visual state. The Scala server is the decision level: it owns communication, state, error handling and symbolic reasoning through the PrologService interface. Figure 1 shows the component-level view.

4.1. Communication Protocol

The network contract is intentionally small. A request contains the agent identifier, a list of percepts and an optional theory string. A successful response contains the same agent identifier, an action and the updated energy value. Errors are encoded as a single error field.

Listing 2: Scala data transfer objects for the WebSocket protocol.

```
final case class WsRequest(  
  agent: String,  
  percepts: List[String],  
  theory: Option[String]  
)  
  
final case class WsResponse(  
  agent: String,  
  action: String,  
  energy: Int  
)  
  
final case class WsError(error: String)
```

The protocol supports both periodic updates and urgent updates. Periodic updates are useful for continuous behaviour, while urgent updates are triggered by sensor events such as entering an attack area, detecting a semaphore, changing line-of-sight, or entering an intersection. This hybrid pattern keeps the system responsive without requiring every frame to perform a server round trip.

4.2. Server State

The backend stores a `ServerState` composed of two maps: agent identifiers to `AgentState`, and agent identifiers to theory text. Each `AgentState` contains energy, last action, last percept list and last decision timestamp. Energy is updated through a bounded domain rule; for example, `attack` costs more than `move_forward`, `rest` restores energy, and `stop` has no cost.

The server also implements a short anti-jitter decision reuse window of 30 ms. If percepts are unchanged and the previous decision is recent, the backend can reuse the last action and touch the timestamp without applying a new energy delta. This is a pragmatic optimisation for high-frequency WebSocket ticks: it reduces oscillation and avoids recomputing identical Prolog queries when the simulation state has not meaningfully changed.

4.3. Runtime Sequence

The runtime sequence is as follows. First, a Godot agent builds a percept list from local sensors and domain state. Second, it sends a JSON request to `/ws`. Third, the Scala route decodes the frame and delegates to the decision pipeline. Fourth, the pipeline persists any new theory, enriches the percepts, evaluates `decide_action(Percepts, Action)` and updates state. Finally, the server sends a JSON response and the agent applies the action locally. This makes the symbolic action boundary explicit: Prolog decides what should be done, while Godot decides how that action is physically realised.

4.4. Design Rationale

The architecture deliberately avoids placing all intelligence in a single layer. If all decisions were made inside Godot, the project would lose the ability to compare symbolic theories independently from GDScript control code. If all simulation state were mirrored inside Scala, the system would become a distributed simulation with synchronisation problems and duplicated physics. The selected compromise is to send only symbolic percepts and actions. This keeps network payloads small and keeps each component close to its strengths.

Another design choice is to make the server stateful but not world-authoritative. The server tracks only facts that are naturally global to the decision service: energy, last action, last percepts and theory text. It does not track positions, velocities or collisions. The benefit is that the server can enrich decisions and provide continuity without becoming a second game engine. The cost is that long-term world reasoning is limited unless the client exposes additional symbolic facts.

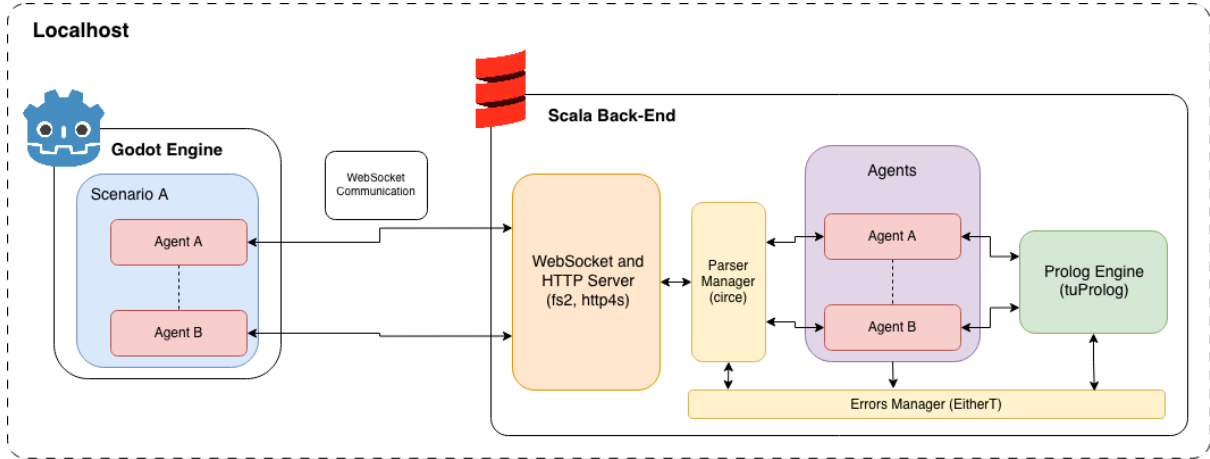


Figure 2: Localhost deployment: Godot and the Scala decision backend run on the same host while communicating through WebSocket.

The per-agent theory map is also important. A simpler implementation could send the theory on every request, but that would waste bandwidth and make logs hard to inspect. Another alternative would be to preconfigure all theories on the server, but that would couple the backend to a fixed scenario catalogue. Persisting a theory after the first non-empty payload supports both efficiency and flexibility. Agents can be configured from the Godot inspector, yet the backend can remain generic.

Finally, the protocol is deliberately unopinionated about action semantics. The server returns strings such as `shoot`, `drive` or `defend_goal`, but it does not know how to execute them. This allows each scenario to interpret an action at the appropriate level of abstraction. A vehicle’s `drive` changes a speed factor on a path-follow controller; a tank’s `shoot` checks line-of-sight and spawns a bullet; a soccer player’s `kick_to_op` applies an impulse to a rigid body. The common protocol therefore supports heterogeneous mechanics without expanding the backend API.

5. Distribution Architectures

The same symbolic decision architecture can be deployed in several ways. The simplest configuration runs the Godot client and the Scala backend on the same machine. This is the mode used most often during development because it minimises network variables and makes debugging easier. In this setting, the WebSocket endpoint is still used, but the client connects to `ws://127.0.0.1:8080/ws`. Figure 2 shows this localhost configuration.

The first distributed configuration separates the simulation host from the decision host. Godot runs on one machine and sends percepts to a Scala backend running on another machine. This is useful when the reasoning layer is deployed on a server, when the simulation is executed on a less powerful client, or when backend logs and metrics need to be centralised. Figure 3 illustrates this client/server topology.

The WebSocket boundary also allows multiple Godot clients to use the same backend. Each client may run a different scenario or a different subset of agents. The backend distinguishes agents by identifier and stores theory and energy state per agent. This makes the architecture suitable for remote demonstrations and distributed experiments in which several simulations share one decision service. Figure 4 shows this multi-client pattern.

A further topology distributes both the clients and the decision backends. Different hosts may run different Scala services, and clients can be assigned to a backend according to scenario, load, locality or experiment configuration. This is not required by the current prototype, but it follows naturally from the fact that the backend is stateless with respect to the physical world and stateful only with respect to agent identifiers and theories. Figure 5 represents this multi-host configuration.

These topologies clarify why WebSocket was a pragmatic first transport: it is simple, widely available

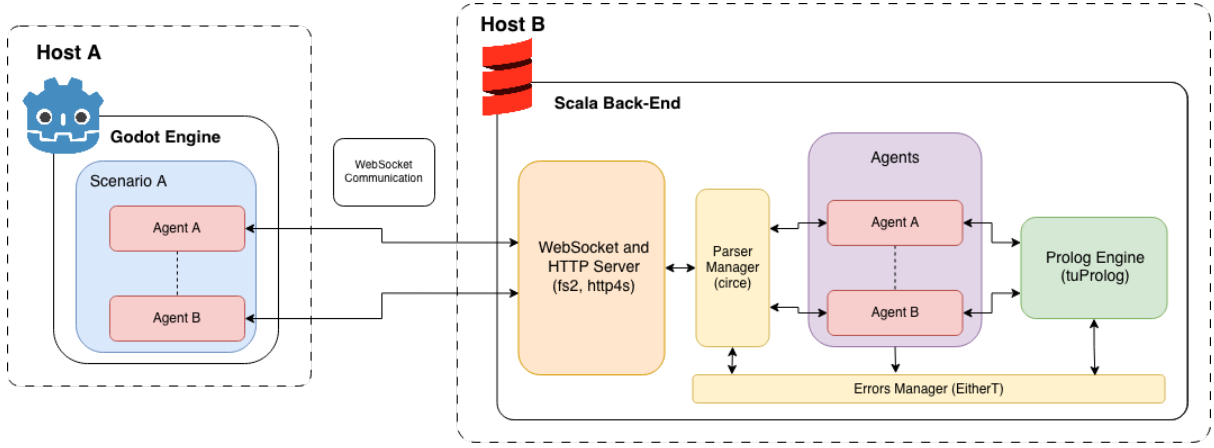


Figure 3: Client/server deployment: a Godot client on Host A delegates symbolic decisions to the Scala backend on Host B.

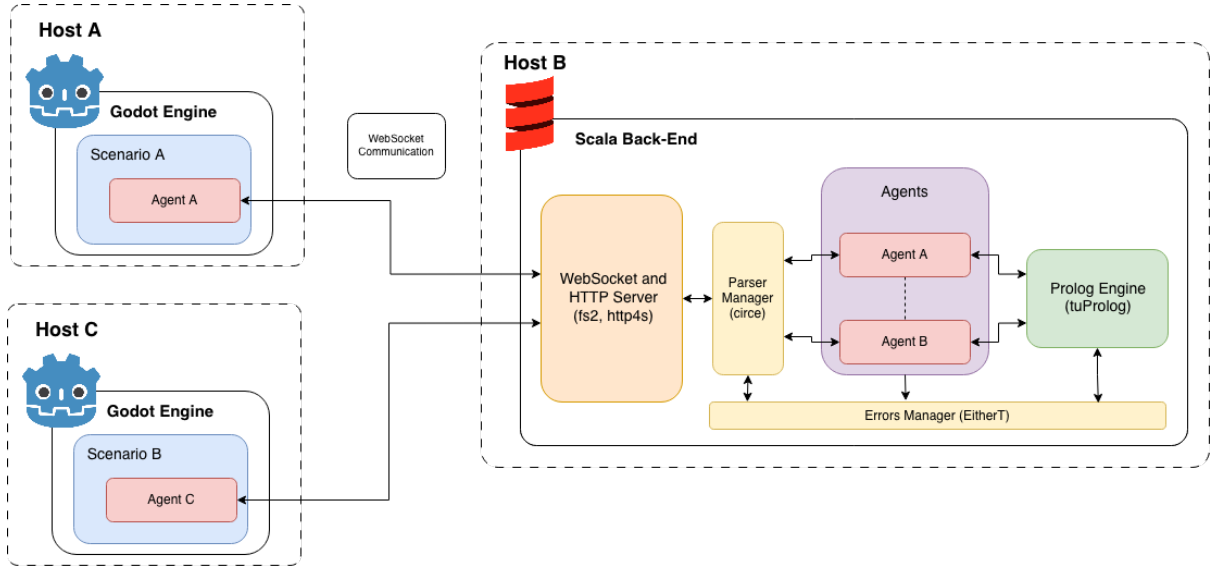


Figure 4: Multi-client deployment: several Godot clients, potentially running different scenarios, share one Scala decision backend.

and independent from Godot-specific networking. However, WebSocket only solves the decision-service communication channel. It does not synchronise a shared Godot world across several live clients. For that reason, a future distributed version should combine the symbolic decision service with Godot's high-level multiplayer API and ENet-based peers [10]. In such an extension, ENet would synchronise the state of one scenario across multiple hosts and clients, while the Scala backend would remain responsible for symbolic decisions or would be integrated as one authoritative service in the distributed simulation.

6. Decision Model

The decision model is based on priority rules. Each Prolog file defines a policy for one scenario or agent type. Rules are ordered from most urgent to least urgent and use cut to stop after the first successful match. This means that the behaviour is deterministic for a fixed percept list and theory.

Listing 3: Excerpt from the vehicle policy. Safety facts are evaluated before routing and cruise behaviour.

```
decide_action(P, stop) :-
    member("must_stop_signal", P), !.
```

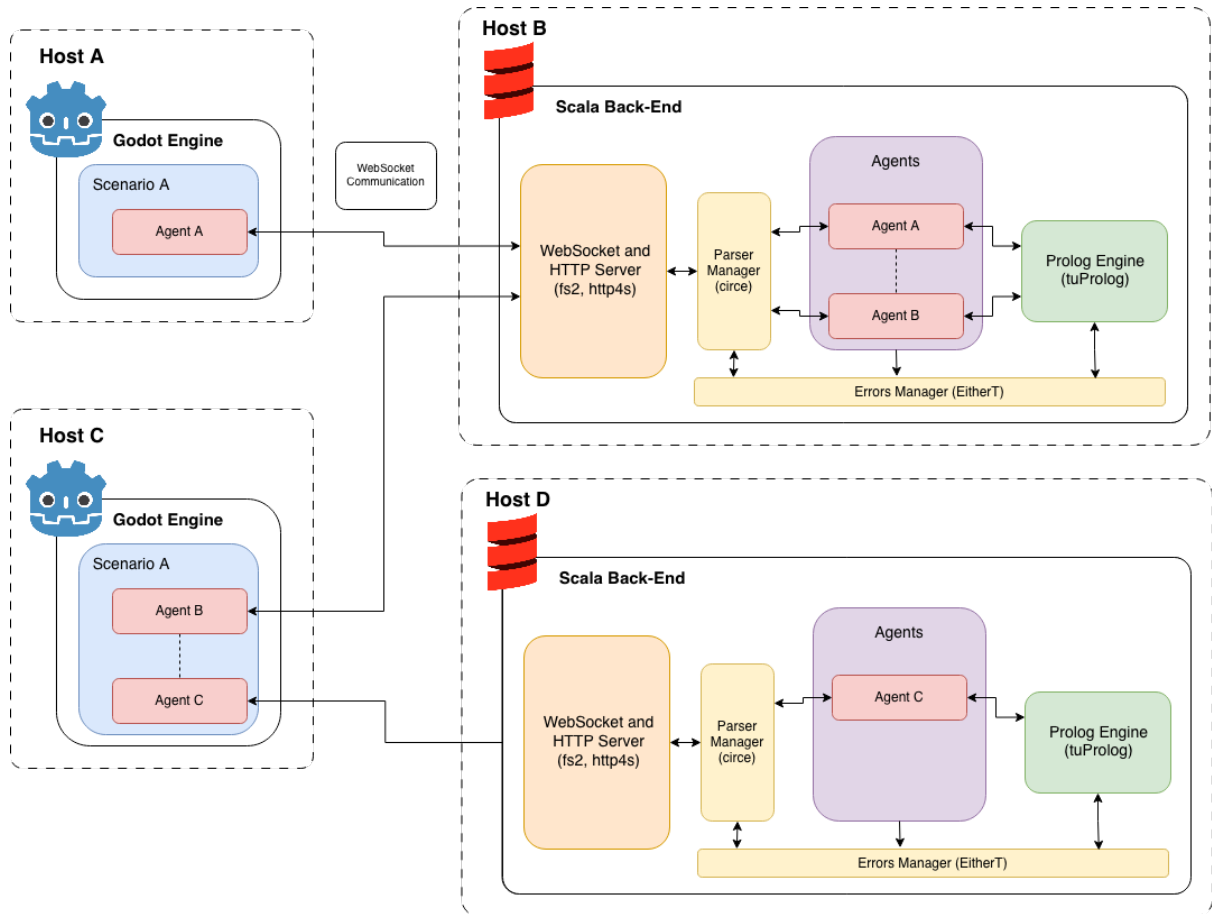


Figure 5: Multi-host deployment: multiple Godot clients and multiple Scala decision backends can be combined by assigning agents or scenarios to different services.

```

decide_action(P, stop) :-
  member("yield_to_right", P), !.

decide_action(P, drive) :-
  member("intersection_unblock", P), !.

decide_action(P, slow_down) :-
  member("vehicle_close", P), !.

decide_action(P, turn_random) :-
  member("at_intersection", P),
  any_turn(P),
  \+ member("last_action_turn_random", P), !.

```

The rule order is part of the policy design. In the vehicle theory, traffic lights, right priority and collision risk appear before intersection routing. In the tank theory, immediate retreat from a close enemy appears before shooting. In the simple-agent theories, goal completion and low energy are checked before attack. This structure is easy to inspect because each rule corresponds to a domain-level priority.

6.1. Percept Enrichment

Percepts originate in Godot but are not exclusively client-controlled. The server removes any client-provided `low_energy` fact and derives that fact from authoritative energy. It also adds `last_action_*`.

Table 2

Prolog theories included in the project.

Theory	Behavioural role
<code>logic_a.pl</code>	Aggressive simple-agent profile: explores and engages enemies actively.
<code>logic_b.pl</code>	Cautious simple-agent profile: shares survival rules with A but falls back to idle.
<code>soccer_left.pl</code>	Balanced soccer policy for the left player: move to ball, kick, sidestep, defend.
<code>soccer_right.pl</code>	Symmetric soccer policy for the right player, avoiding artificial strategic bias.
<code>vehicle_logic.pl</code>	Safety-first traffic policy: lights, right priority, vehicle distance, intersection turns.
<code>tank_hunter.pl</code>	Tactical combat policy: retreat, strafe around blocked targets, shoot on clear line-of-sight.
<code>logic.pl</code>	Generic fallback server policy used when no agent-specific theory is available.

This prevents a client from accidentally or intentionally misrepresenting energy state, and it gives Prolog a small memory of recent decisions. The vehicle theory uses this mechanism to avoid repeatedly issuing `turn_random`: after the first turn decision, the server adds `last_action_turn_random`, allowing the policy to switch to drive.

6.2. Policy Catalogue

The current project includes six scenario-specific theories plus one fallback theory. Table 2 summarises their roles.

The catalogue illustrates why separating policy from simulation is useful. The same WebSocket server can host an aggressive arena agent, a soccer player and a vehicle by merely changing the theory and the client-side percept/action mapping. The server does not need to understand the full semantics of a soccer ball or a traffic intersection; it only needs a stable symbolic interface.

6.3. Action Interpretation

An action is not a low-level command. It is a symbolic intention that must be grounded by the receiving agent. This distinction is crucial for avoiding overfitting the Prolog layer to a specific scene. In the simple-agent scenario, `attack` means orienting toward a target, approaching if necessary, applying damage and knockback when the attack area or contact conditions are satisfied. In the tank scenario, `attack` is not even part of the final policy; the more precise action is `shoot`, which is locally guarded by cooldown and line-of-sight. In the vehicle scenario, `turn_random` does not rotate a car; it stores a pending routing command, and Godot later selects a valid cross path compatible with the current endpoint.

This grounding step is also where safety checks are repeated. Prolog may request `shoot`, but the tank script still refuses to fire if the target is behind an obstacle. Prolog may request `drive`, but the vehicle script still applies lane separation and safe progress limits. This redundancy is intentional. It keeps declarative policies readable while preserving the robustness expected from physics-driven simulation.

6.4. Fallback Behaviour

Every theory includes a final fallback action. This is a small but important engineering practice. A missing fallback would turn an unexpected percept combination into a Prolog failure, which would then produce an error frame rather than behaviour. The fallback is domain-specific: `wander` for aggressive simple agents, `idle` for cautious agents, `defend_goal` for soccer, `drive` for vehicles and `hold` for tanks. These choices encode the safest or most useful default for each domain.

Fallbacks also make policies easier to extend. A new rule can be inserted above the fallback and tested without changing the rest of the contract. If the new rule is too broad, its behavioural impact is visible in the scene. If it is too narrow, the fallback continues to preserve operational behaviour.

7. Implementation

7.1. Scala Backend

The Scala backend is organised into small modules. `Main.scala` creates a shared `Ref[IO, ServerState]`, wires the live Prolog service, and starts the `http4s` Ember server on port 8080. `WebSocketRoutes.scala` exposes `/health` and `/ws`. `DecisionService.scala` implements the request pipeline. `PrologService.scala` adapts `tuProlog`. `State.scala` contains the domain model for energy and decision reuse. `Protocol.scala` contains JSON payload case classes. `AppError.scala` models typed errors, and `MonadTypes.scala` defines the functional composition stack.

The backend uses a Reader-style dependency layer with `Kleisli` and an error layer with `EitherT`. This keeps context access, IO and typed failures explicit. Listing 4 shows the conceptual shape of the decision pipeline.

Listing 4: Simplified structure of the decision pipeline.

```
for
  request  <- decodeRequest(text)
  context  <- askResult
  nowMs    <- liftResult(IO.monotonic.map(_.toMillis))
  snapshot <- liftResult(context.stateRef.get)
  previous = snapshot.agents.getOrElse(request.agent, AgentState.initial)
  percepts = buildCombinedPercepts(previous, request.percepts)
  _        <- liftResult(persistTheory(context, request))
  action   <- decideOrReuse(previous, percepts, snapshot, request)
  response <- liftResult(updateStateAfterDecision(context, request, percepts,
    action, nowMs))
yield response
```

The important property is that transport, state and Prolog execution are not collapsed into one callback. The WebSocket route handles frames; the decision service handles business logic; the Prolog service handles the external engine; the state module handles energy and reuse rules. This modularity is valuable because most future extensions, such as a new error type, a new state field or a new decision backend, have a clear place to be introduced.

The Prolog adapter constructs a Prolog list from percept strings, loads the selected theory, solves `decide_action([...], Action)`, and normalises the returned atom. If no theory is available, it returns `missing_theory`; if the goal fails, it returns `no_solution`; if `tuProlog` raises an exception, it returns a `prolog_failure` message. These errors are converted to `wsError` JSON frames, giving the Godot client a controlled failure mode.

7.2. Scala Library Stack

The Scala dependencies were selected to keep the backend small but disciplined. Cats Effect provides the IO runtime, resource safety and the `Ref` used for shared in-memory state. FS2 provides the stream model used by `http4s` WebSocket routes. `http4s` Ember supplies the HTTP/WebSocket server and the route DSL. Circe provides JSON parsing and semi-automatic derivation for the request, response and error payloads. `tuProlog` is used by the live reasoning adapter to evaluate Prolog theories on the JVM.

This library stack supports the functional style chosen for the backend. Side effects are explicit in `IO`; mutable state is isolated inside `Ref`; transport is represented as streams; domain failures are represented as `AppError` values and propagated with `EitherT`. The result is not merely idiomatic Scala: it gives

the decision pipeline a clear failure model and makes the boundary between pure domain rules and effectful infrastructure easier to test.

7.3. Pluggable Prolog Service

Although the current implementation uses tuProlog, the backend is intentionally written against the PrologService trait rather than directly against the concrete engine. The production implementation, LivePrologService, is only one adapter. A different adapter could wrap another Prolog engine, a remote reasoning service, a cached rule interpreter, or even a non-Prolog symbolic policy evaluator, provided it implements the same method:

Listing 5: Decision engine abstraction used by the Scala backend.

```
trait PrologService:
  def decideAction(
    percepts: List[String],
    theoryOverride: Option[String]
  ): Either[AppError, String]
```

This design keeps the scientific focus on the architecture rather than on a single Prolog implementation. tuProlog is a convenient and meaningful default for a JVM-based Italian academic prototype, but the system boundary is the service interface. Replacing the engine would not require changing the WebSocket protocol, the Godot agents, or the scenario-specific action interpreters.

7.4. Godot Agent Base

Most agent types reuse a common GDScript base, Agent.gd. It centralises WebSocket connection management, periodic percept sending, urgent sending, response parsing, energy storage and action dispatch. Subclasses override two methods:

Listing 6: Godot agent extension points.

```
func build_percepts() -> Array:
  return []

func perform_action(action: String) -> void:
  pass
```

This is a Template Method pattern: the base class controls the communication loop and delegates domain-specific behaviour to subclasses. TestAgent.gd, player.gd and tank.gd use this pattern directly. The vehicle controller follows the same conceptual pattern but embeds the WebSocket logic inside a PathFollow3D script because path progression and routing are central to that scene.

The base agent only understands that a response may contain energy and action. It does not know what shoot, kick_to_opp or turn_random mean. That interpretation is deliberately local. For example, kick_to_opp in soccer applies an impulse to a rigid body, while shoot in the tank scenario instantiates a bullet scene if line-of-sight and cooldown permit it.

7.5. Theory Loading and Hot Swap

In Godot, agents expose a prolog_path property through the inspector. At initialisation they read the file content and call set_theory. The next payload includes the theory text, after which the agent marks it as sent. If the theory is changed or the connection is rebuilt, the theory can be resent. The server persists non-empty theory text by agent identifier.

This mechanism supports heterogeneity in the same runtime. In the simple-agent test, two otherwise similar agents can use logic_a.pl and logic_b.pl. In soccer, the left and right players use symmetric theories. In tank combat, both teams may share tank_hunter.pl while local team percepts guide opposite strafe choices.

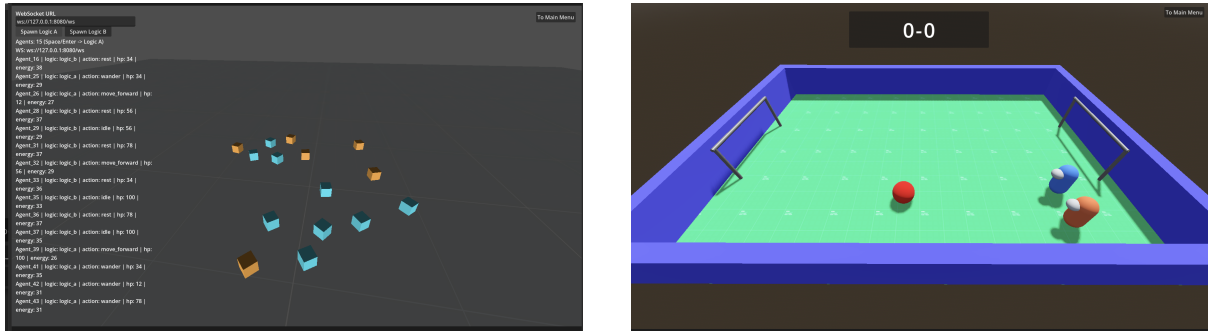


Figure 6: Left: Simple-agent scenario: agents with different Prolog policies interact in a compact arena. Right: Soccer scenario, symbolic policies select ball pursuit, kicking, sidestepping and defence while Godot handles physics and scoring.

7.6. Operational Workflow

The operational workflow is intentionally lightweight. The Godot project is opened from the godot directory and starts from the main menu scene. Each scenario can be launched from that menu, and WebSocket URLs can be configured in the UI or inspector depending on the scene.

This workflow makes the project suitable for iterative experimentation. A typical change cycle is: edit a Prolog rule, restart or reconnect the relevant agent so the theory is resent, observe the new behaviour, and then adjust either the rule priority or the percept builder. For backend changes, the developer can restart the Scala process and perform a smoke test against `/health` before launching a Godot scene. For simulation changes, the developer can run the target scene and then perform a quick pass over the other scenarios to ensure the shared agent base or protocol has not regressed.

The project documentation records this process in Markdown. Requirements, architectural design, detailed design, implementation notes, testing strategy, retrospective and sprint artefacts are versioned alongside the source. This is useful for scientific reporting because it provides traceability from requirements to implementation artefacts.

8. Scenarios

The prototype includes four scenarios. They are not isolated demos; they exercise different stress points of the same architecture.

8.1. Simple Agents

The simple-agent scenario validates the minimal closed loop. Agents can be spawned from the UI and are assigned either an aggressive or a cautious Prolog policy. Each agent senses enemies through a detection area, identifies enemies close enough for attack through an attack area, tracks health locally, receives server energy, and dies when health or energy reaches zero.

The percepts include `enemy`, `enemy_close`, `can_attack`, `under_attack`, `obstacle`, `see_food`, `goal_reached` and `low_hp`. The action set includes movement, turning, attack, flee, rest, wander, idle and celebrate. The aggressive policy tends to continue moving in fallback, while the cautious policy falls back to idle. This produces visible differences without changing the physics script.

8.2. Soccer

The soccer scenario introduces a shared object: the ball. Two Prolog-controlled players try to move the ball into the opponent's goal. The player script builds percepts such as `ball_visible`, `ball_near`, `ball_stuck`, `enemy_near`, `ball_in_own_half`, `ball_in_opp_half` and `ball_in_opp_goal`. The actions are `move_to_ball`, `kick_to_opp`, `sidestep`, `defend_goal` and `celebrate`.

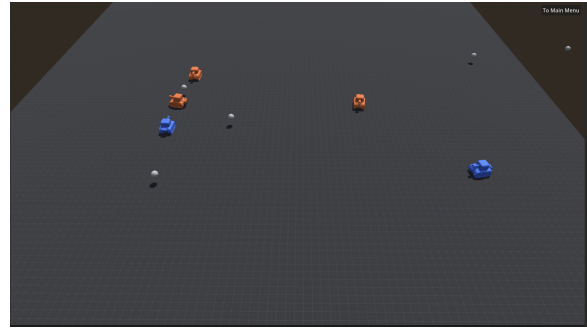


Figure 7: Left: Vehicle scenario: path-following cars combine Prolog decisions with traffic lights, local sensors and anti-deadlock guards. Right: Top-down tank scenario: agents use symbolic tactical decisions grounded by line-of-sight checks, bullets, damage and respawn.

This scenario demonstrates that symbolic policies can coordinate with continuous physics. The Prolog rule `ball_near -> kick_to_opp` does not specify impulse direction, cooldown or collision handling. Those remain local to Godot. The game manager detects goals, updates the score, resets player positions and recentres the ball if it leaves the field. The symbolic layer selects tactical intent; the simulation layer enforces the mechanics.

8.3. Vehicles

The vehicle scenario is the most complex because it combines path following, route choice, traffic lights, right priority, vehicle proximity and anti-deadlock behaviour. Vehicles move on named general paths and cross paths. At an intersection, the script gathers available turn labels by comparing path endpoints and classifying directions as left, right, straight or u-turn. The Prolog action can request a random turn, after which Godot selects a compatible cross path and executes the transition.

The percept set includes traffic-light facts (`light_red`, `light_yellow`, `light_green`, `light_none`), intersection facts (`at_intersection`, `can_turn_*`), proximity facts (`vehicle_ahead`, `vehicle_close`, `vehicle_very_close`, `vehicle_in_area_sensor`), priority facts (`yield_to_right`) and anti-stall facts (`deadlock_risk`, `intersection_unblock`, `intersection_congested`). The policy prioritises red/yellow lights and right priority before intersection routing, and allows an `intersection_unblock` exception when the local guard detects prolonged blocking.

This scenario shows an important design lesson: Prolog policy alone is not enough for physical traffic. Local continuous guards are still necessary for lane separation, braking caps, safe progress limits, semaphore memory and randomised unblock windows. The symbolic layer encodes high-level safety priorities; Godot enforces low-level physical constraints.

8.4. Top-Down Tanks

The tank scenario tests tactical combat. Each tank belongs to a team, searches for the closest enemy in perception range, checks line-of-sight with a ray cast, and shoots only when the target is visible, in range and the cooldown has expired. Percepts include `team_1`, `team_2`, `enemy_visible`, `enemy_in_range`, `enemy_too_close`, `enemy_blocked`, `enemy_left`, `enemy_right`, `can_shoot`, `no_enemy`, `low_hp`, `stuck` and `unstuck_mode`.

The policy first retreats from enemies that are too close. If the target is blocked, the tank strafes left or right depending on team. If the target is in range and shooting is possible, it fires. Otherwise it advances, patrols or holds. Godot implements bullet instantiation, damage, death, respawn, team visuals and anti-stall lateral overrides. This scenario validates that the same symbolic protocol can support richer action semantics without backend changes.

Table 3

Scenario comparison by architectural stress point.

Scenario	Main challenge	Architectural lesson
Simple agents	Differentiating agent profiles.	Per-agent theories are enough to create visible behavioural contrast.
Soccer	Shared dynamic object and scoring.	Symbolic intent must be grounded in local physics and game-manager rules.
Vehicles	Traffic safety and deadlock.	Declarative policy needs local continuous guards for robust physical interaction.
Tanks	Tactical line-of-sight combat.	Symbolic decisions benefit from sensor facts but must be rechecked before irreversible actions.

Table 4

Functional validation matrix used during development.

Scenario	Test focus	Expected observation
Simple agents	Spawn A/B, connect to WebSocket, attack, flee, rest, death.	Agents receive actions, show distinct A/B behaviour and are removed when depleted.
Soccer	Ball pursuit, kick, goal detection, score update, reset.	Players move to the ball, kick toward the opponent goal and recover after goal or out-of-bounds.
Vehicles	Path following, light compliance, right priority, distance control, intersection unblock.	Vehicles stop on red/yellow, slow or stop near others, choose turns and recover from local stalling.
Tanks	Target acquisition, line-of-sight, shooting, damage, respawn, patrol.	Tanks shoot only with clear line-of-sight, strafe blocked targets, die and respawn.

8.5. Cross-Scenario Comparison

The four scenarios were selected because each stresses a different aspect of the architecture. Table 3 compares them by dominant challenge.

The comparison highlights a recurring pattern. The symbolic layer is most effective when the percept vocabulary is compact but semantically meaningful. Too few percepts make policies blind; too many low-level percepts make rules noisy and brittle. The vehicle scenario required the richest vocabulary because traffic depends on light state, path state, local sensors, right priority and deadlock recovery. The soccer scenario required fewer facts because the domain goal is simpler: approach the ball, kick when near, sidestep when contested and defend otherwise. The tank scenario sits between these extremes because line-of-sight and target direction matter, but the tactical action set remains small.

9. Evaluation

The project was evaluated primarily through functional end-to-end tests. This is appropriate for the current maturity of the prototype: the central claim is architectural feasibility and scenario coverage, not measured performance superiority. The test process starts the Scala backend, opens a Godot scene, observes percept-to-action logs and verifies the resulting behaviour in the scene. Regression checks are repeated on stable scenarios after changes to shared code.

Table 4 summarises the manual validation matrix. The observations are qualitative and should be read as functional evidence rather than statistical evaluation.

The most informative findings came from interaction effects. The simple-agent and soccer scenarios mainly validated the communication backbone and action mapping. The vehicle scenario exposed

emergent deadlocks at intersections and required additional local guards, randomised unblock windows and lane separation. The tank scenario exposed the need to combine symbolic line-of-sight facts with local ray-cast verification before shooting. These findings support the architectural premise: symbolic rules are useful as high-level policies, but physical simulations still require local continuous safeguards.

9.1. Engineering Outcomes

The final implementation demonstrates several engineering outcomes.

First, the same server and protocol support multiple domains. No endpoint or payload shape had to be redesigned for soccer, vehicles or tanks. New scenarios were introduced by defining percept builders, action interpreters and Prolog theories.

Second, the backend refactoring improved maintainability. The decision pipeline is no longer mixed with WebSocket route code. State updates, Prolog invocation, JSON decoding and error modelling are separated. This makes future automated tests plausible: pure or near-pure parts such as energy updates, percept enrichment and request decoding can be tested without launching Godot.

Third, the per-agent theory mechanism supports experimentation. A policy change can be made in a Prolog file, assigned to an agent and observed in the runtime. This is a useful affordance for teaching symbolic AI because students can see the consequences of rule ordering and fallback choices in a visual environment.

9.2. Observed Development Risks

Communication and parsing issues occurred early, especially around resource loading and JSON decoding. These were addressed by centralising the protocol, introducing typed application errors and making the client responsible for sending explicit theory text. Movement fluidity was another risk: agents could oscillate, overlap or become stuck if symbolic actions changed too quickly or if local separation was absent. The solution combined decision reuse, urgent sends only on meaningful changes, separation forces and improved collision handling.

The vehicle scenario produced the most difficult emergent behaviour. Deadlocks at intersections could arise even when each individual vehicle followed a reasonable rule. This is a classic multi-agent problem: local rationality is not sufficient for global progress. The implemented solution combines right-priority facts, traffic-light compliance, queue sensing, randomised waiting and an unblock percept that temporarily permits progress. This is not a formal traffic protocol, but it is an effective prototype-level mitigation and a useful case study for future work.

10. Discussion and Future Work

The project suggests that a lightweight symbolic decision server can be a practical bridge between game-engine simulation and declarative agent policies. The boundary works best when each side keeps its natural responsibilities. Godot should not be reduced to a thin renderer: it must still handle physics, collision, sensing and safety constraints. Prolog should not be asked to control continuous motion directly: it should express intent and priorities over symbolic facts. Scala provides the coordination layer where state, transport and engine invocation can be made explicit and typed.

Several future directions are natural. The first is automated testing. The Scala backend should include unit tests for JSON decoding, error conversion, theory persistence, percept enrichment, energy updates and decision reuse. Prolog theories could be tested through a small table-driven harness that asserts expected actions for percept sets. This would catch rule-ordering regressions, especially in the vehicle policy.

The second direction is runtime measurement. The system should record decision latency, Prolog solve time, WebSocket round-trip time, message frequency and per-scenario agent counts. Such metrics would make it possible to evaluate whether the architecture remains responsive as the number of agents grows.

The third direction is richer distributed execution. WebSocket is a good first protocol for the decision service because it is simple and can be consumed by desktop, web or scripted clients. However, synchronising a single Godot scenario across several hosts requires a game-state networking layer rather than only a decision channel. A planned improvement is therefore to introduce communication through ENet and Godot’s high-level multiplayer API [10]. This would allow several clients to participate in the same scenario, with Godot synchronising scene state across hosts while the symbolic backend remains responsible for agent decisions.

The fourth direction is a more formal traffic model. The current vehicle scenario combines symbolic safety rules with local anti-deadlock heuristics. A reservation-based intersection model, cell occupancy protocol or temporal rule layer could reduce reliance on randomised unblock windows. Prolog could represent explicit reservations or priorities rather than only reactive facts.

The fifth direction is richer agent memory. The current server adds only low energy and last action. Future theories could use event histories, cooldown facts, goal stacks or temporal predicates. Care is required: adding memory increases expressive power but also makes policies harder to inspect.

The sixth direction is deterministic replay. Multi-agent bugs are often emergent and difficult to reproduce. Capturing percept streams, actions and random seeds would allow a failed run to be replayed without relying on visual observation alone.

11. Conclusion

This paper described a multi-agent 3D simulation platform that integrates Godot, Scala 3 and Prolog through a small WebSocket protocol. The system separates real-time simulation from symbolic decision making: Godot produces percepts and applies actions, Scala manages state and transport, and a pluggable PrologService evaluates per-agent policies. The prototype uses tuProlog, but the architecture permits alternative Prolog engines or custom symbolic reasoners. Four scenarios demonstrate the breadth of the approach: simple adversarial agents, soccer players, path-following vehicles and top-down tanks.

The main contribution is an extensible architecture for experimenting with symbolic policies in visual, physical environments. The implementation shows that Prolog theories can be hot-swapped per agent, that a typed functional Scala backend can keep decision orchestration maintainable, and that Godot can remain responsible for continuous mechanics. The current evaluation is qualitative and functional, but it identifies concrete next steps: automated tests, runtime metrics, formal traffic coordination, ENet-based multi-host synchronisation and deterministic replay. In this form, the project provides a practical foundation for teaching and prototyping symbolic multi-agent behaviour in real-time 3D simulations.

Acknowledgments

We thank student Antonio Rotundo from the University of Bologna for helping us in the first phase of software development.

Data availability statement

The source code is available in the public repository <https://github.com/UniCT-ARSLab/Vela>.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly (Free Edition) to correct typos and improve readability and language. After using this tool/service, the authors reviewed and edited the content as needed and took full responsibility for the content of the publication.

References

- [1] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2 ed., John Wiley & Sons, 2009.
- [2] Y. Shoham, Agent-oriented Programming, *Artificial Intelligence* 60 (1993) 51–92. doi:10.1016/0004-3702(93)90034-9.
- [3] Godot Engine contributors, Godot Docs – 4.6 branch, 2026. URL: <https://docs.godotengine.org/>, accessed 2026-05-03.
- [4] E. Denti, A. Omicini, A. Ricci, tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures, in: *Practical Aspects of Declarative Languages*, Springer, 2001, pp. 184–198. URL: <https://lia.disi.unibo.it/~ao/pubs/pdf/2001/padl.pdf>.
- [5] Scala Center and contributors, Scala 3 Documentation, 2026. URL: <https://docs.scala-lang.org/>, accessed 2026-05-03.
- [6] Typelevel, Cats Effect, 2026. URL: <https://typelevel.org/cats-effect/>, accessed 2026-05-03.
- [7] http4s contributors, http4s: Typeful, functional, streaming HTTP for Scala, 2026. URL: <https://http4s.org/>, accessed 2026-05-03.
- [8] Circe contributors, Circe: A JSON library for Scala, 2026. URL: <https://circe.github.io/circe/>, accessed 2026-05-03.
- [9] I. Fette, A. Melnikov, The WebSocket Protocol, Technical Report RFC 6455, Internet Engineering Task Force, 2011. URL: <https://www.rfc-editor.org/rfc/rfc6455>.
- [10] Godot Engine contributors, High-level multiplayer, 2026. URL: https://docs.godotengine.org/en/latest/tutorials/networking/high_level_multiplayer.html, accessed 2026-05-03.