University of Trento

Autonomous Software Agents

Final Project Report

Parcel Predator

Author:

Gianluigi Vazzoler
(257846)

January 2026

Abstract. This project implements autonomous BDI agents for the Deliveroo.js environment, combining a dynamic utility function for optimal parcel selection with a hybrid navigation strategy. The path planning integrates standard BFS with PDDL-based reasoning, executed via a custom low-latency local A* solver. Multi-agent coordination is achieved through explicit intention negotiation (claiming) and a dedicated handoff protocol designed to resolve deadlocks in constrained spaces. Experimental validation was conducted on the 25c1_* scenarios for single-agent performance and 25c2_* scenarios (including the critical 25c2_hallway) for the coordinated team, demonstrating effective resource management and robust spatial conflict resolution.

1 Introduction

The development of autonomous software agents capable of operating in dynamic and partially observable environments is a central challenge in Artificial Intelligence. This project, *ParcelPredator*, addresses this challenge within the context of *Deliveroo.js*, a grid-world simulation where agents compete or cooperate to collect and deliver parcels.

1.1 Context and Motivation

The *Deliveroo.js* environment presents a dynamic grid ($M \times N$) containing obstacles, delivery zones, and parcel spawners. The complexity arises from several factors:

- **Dynamism:** Parcels appear with a finite time-to-live and their reward decays over time, requiring agents to balance immediate gains against travel costs.
- **Partial Observability:** Agents have a limited sensing range ($x_{offset} + y_{offset} < 5$), necessitating a robust belief revision system to maintain an internal model of the world.
- **Spatial Constraints:** Movement is not instantaneous and tiles are locked during traversal. In multi-agent scenarios, this leads to potential deadlocks and collisions, particularly in narrow corridors (e.g., the "Hallway" scenario).

These characteristics make *Deliveroo.js* an ideal testbed for implementing the Belief-Desire-Intention (BDI) architecture, as agents must constantly update their beliefs, deliberate on desires (e.g., which parcel to pick), and commit to intentions (plans) while reacting to unforeseen changes.

1.2 Objectives and Contributions

The primary objective of this project is to develop a robust agent capable of maximizing the delivery score in both single-agent and cooperative multi-agent settings. Specifically, the contributions of this work, implemented on top of the provided Node.js/Socket.IO skeleton, are:

- **BDI Architecture Implementation:** Development of a full BDI loop that manages belief revision (handling old data for parcels and agents) and generates options based on a dynamic utility function that considers reward decay and contention penalty.
- **Hybrid Path Planning:** Integration of a navigation system that alternates between a standard Breadth-First Search (BFS) and a PDDL-based approach. The latter is powered by a custom high-performance local A* solver (‘fastLocalSolver.js‘) that processes PDDL problem definitions to ensure compliance with the planning requirements while minimizing latency.
- **Multi-Agent Coordination:** Implementation of a cooperative team strategy featuring:
  1. A **Handshake Protocol** for role assignment and network discovery.
  2. An **Intention Claiming System** where agents broadcast their targets to avoid redundant trips and resource contention.
  3. A specialized **Handoff Protocol** to resolve spatial deadlocks in narrow corridors. For example, in the scenario `25c2_hallway`, agents can negotiate yielding or transfer parcels to clear the path.

The resulting system was validated against the standard course challenges (‘25c1_*‘, ‘25c2_*‘), demonstrating the ability to handle both efficient routing and complex agent-to-agent interactions.

2 System Architecture

The *ParcelPredator* system is built upon the Node.js runtime, utilizing the `@unitn-asa/deliveroo-js-client` library to interact with the game server. The architecture follows a modular Belief-Desire-Intention (BDI) pattern, separating perception, deliberation, and execution into distinct components.

2.1 Component Overview

The system is organized into the following high-level modules, orchestrating the agent's behavior from perception to action:

- **Agent Core (‘src/bdi/agent.js‘):** This module represents the agent entity. It maintains the main control loop (‘me.loop()‘), manages the intention queue, and holds the state for coordination protocols (e.g., handoff status).

- **Belief Model (‘src/bdi/belief.js‘):** Act as the agent’s memory, storing the current state of the environment (parcels, agents, map configuration) and managing the expiration of old data.
- **Deliberation (‘src/bdi/options.js‘):** Responsible for the "Desire" and "Intention" phases. It evaluates the current beliefs to generate candidate goals (e.g., picking up a parcel vs. exploring) and selects the best course of action based on a utility scoring function.
- **Planning & Execution (‘src/bdi/plans/‘):** Contains the procedural knowledge. The agent uses plans like `GoPickUp` and `GoDeliver` to achieve goals. Movement is handled by a hybrid approach:
  - `moveBfs.js`: A standard Breadth-First Search for simple pathfinding.
  - `pddlMove.js`: A PDDL-based planner utilizing a custom `fastLocalSolver.js` (an optimized local A* solver) to handle domain-specific movement constraints.
- **Communication (‘src/bdi/comm.js‘):** Handles multi-agent interactions, including the initial handshake, intention sharing (claims), and the handoff protocol for conflict resolution.

The `launcher.js` file serves as the entry point, initializing these components and wiring the server events (sensing) to the belief revision functions.

2.2 Belief Model

The agent maintains a dynamic representation of the world, updated in real-time via event callbacks from the adapter (`onParcels`, `onAgents`, `onMap`). The belief model is implemented in the `Belief` class and manages the following data structures:

- **Parcels:** Stored in a `Map` (ID $\rightarrow$ Object). Each entry includes the position, current reward, and a timestamp. To handle the dynamic nature of the game, a `checkExpiredParcels()` function removes parcels that haven’t been sensed for over 2 seconds or whose estimated reward has decayed to zero.
- **Agents:** Other agents are tracked in a `Map`, storing their last known position and score. A validity window of 5 seconds is applied; agents not sensed within this period are considered "lost" and removed from memory to prevent pathfinding around "ghost" obstacles.
- **Grid/Map:** Static obstacles (walls) and logical zones (delivery tiles, spawners) are stored in a `Grid` utility class, which provides accessibility checks and Manhattan distance calculations.
- **Operational Metadata:**
  - **Claims:** A registry of intentions broadcasted by the partner agent. These have a Time-To-Live (TTL) of 3 seconds to avoid old locks on parcels.
  - **Cooldowns:** A mechanism to temporarily ignore specific parcels or tiles (e.g., after a failed pickup attempt), preventing the agent from getting stuck in loops.

3 Single Agent Strategy

The core logic of the *ParcelPredator* agent relies on a robust implementation of the Belief-Desire-Intention (BDI) architecture. In the single-agent configuration, the primary goal is to maximize the delivery score while competing against hostile agents for resources. This chapter details the perception mechanisms, the utility-based decision-making process, and the execution strategies employed to achieve this goal.

3.1 Perception and Belief Revision

The agent maintains a local model of the environment that is continuously synchronized with the server’s state. To handle the dynamic and partially observable nature of the game, a set of specific revision rules is applied in the `Belief` module:

- **Information Decay:** Perception is event-driven. To prevent the agent from acting on old data, beliefs about parcels and other agents include a timestamp. Parcels not sensed for over 2 seconds or whose reward has decayed to zero are automatically removed from memory. Similarly, agents not seen for 5 seconds are discarded.
- **Cooldown Mechanism:** A critical component for robustness is the "cooldown" system. If a target (parcel or tile) proves unreachable or an action fails repeatedly, the specific ID or coordinate is added to a cooldown list (typically for 2-10 seconds). This prevents the agent from entering infinite loops or persistently chasing inaccessible resources.

3.2 Reasoning and Option Generation

The deliberation process, implemented in `options.js`, is responsible for generating "Desires" and selecting the most profitable "Intention". The agent evaluates four possible behaviors: *Pick Up*, *Deliver*, *Explore*, and *Random Move*.

3.2.1 Utility Function

The selection of the best parcel to pick up is governed by a utility function that considers the potential reward, the travel cost, and the risk of contention. The score for a candidate parcel p is defined as:

$$Score(p) = (R_p + R_{carried} - Cost_{move}) \times Penalty_{contention} \tag{1}$$

Where:

- R_p is the current reward of the parcel.
- $R_{carried}$ is the total reward of parcels already in the inventory (incentivizing the completion of efficient multi-parcel routes).
- $Cost_{move}$ is a function of the Manhattan distance to the parcel and subsequently to the delivery zone, weighted by the agent's load factor ($loss \times (carried + 1)$).
- $Penalty_{contention}$ is a dynamic multiplier (0.1 to 1.0). If another agent is significantly closer to the parcel, the score is penalized to avoid wasteful races.

3.3 Intention Management

Once the options are scored, the highest-ranking one is promoted to an **Intention**. The agent employs a "single-minded" commitment strategy: once an intention (e.g., `go_pick_up`) is adopted, it is generally maintained until success or explicit failure. However, purely exploratory intentions (`go_random` or `explore`) are volatile and immediately interrupted if a valid parcel appears in the agent's sensing range.

3.4 Execution and Robustness

Plans are the procedural units that execute intentions. The primary plans, `GoPickUp` and `GoDeliver`, incorporate specific mechanisms to handle execution failures:

- **Retry with Exponential Backoff:** If movement towards a target is blocked (e.g., by dynamic obstacles), the plan does not fail immediately. Instead, it retries the movement action up to 3 times, with increasing delays (200ms, 400ms, etc.) between attempts. This allows temporary congestion to clear without triggering a full replanning cycle.
- **Graceful Failure:** If the retries are exhausted, the target is marked with a cooldown, and the intention is dropped. This triggers a new reasoning cycle, allowing the agent to select an alternative target immediately.

4 Multi-Agent Coordination

To address the challenges of the cooperative scenarios (Challenge 2), the system extends the single-agent BDI architecture with a communication layer and explicit coordination protocols. The logic is encapsulated primarily in the `Comm` module and allows two agents to share beliefs, negotiate targets, and resolve spatial deadlocks.

4.1 Communication Infrastructure

Agents communicate via the game server using a custom messaging protocol built on top of the `say` and `shout` primitives provided by the API. The communication lifecycle consists of three stages:

1. **Handshake & Discovery:** Upon connection, agents broadcast a `HANDSHAKE` request. When a reply is received, they establish a "friendship" bond, exchanging IDs to filter subsequent messages. This phase ensures that agents ignore potential adversaries and only coordinate with their designated teammate.
2. **Belief Synchronization:** To overcome partial observability, agents periodically exchange delta-updates about perceived parcels and other agents (`INFO_PARCELS`, `INFO_AGENTS`). These updates are merged into the local `Belief` store, effectively doubling the team's sensory coverage.

4.2 Task Negotiation and Claims

To prevent resource contention—where both agents chase the same parcel—the system implements an explicit "Claiming Strategy".
Before committing to a `GoPickUp` intention, an agent broadcasts a `INTENTION` message containing the target parcel ID and its calculated utility score. The partner agent registers this claim in its belief model with a 3-second Time-To-Live (TTL).

4.2.1 Conflict Resolution (Yielding)

When generating options, the agent checks if any candidate parcel is currently claimed by the partner. A resolution logic determines who proceeds:

- If the partner's claim has a higher utility score, the agent **yields** and discards the option.
- If scores are tied, a deterministic tie-breaker based on agent IDs is applied.
- If the agent's score is significantly higher, it overwrites the claim (assuming the partner will eventually yield upon receiving the new intention).

4.2.2 Delegation (FIPA Request)

The system also supports proactive delegation. If an agent identifies a high-value parcel that is significantly closer to its partner than to itself, it sends a FIPA-compliant `REQUEST` message. The partner evaluates the request against its current capacity and queue; if accepted (`AGREE`), the partner adopts the goal, allowing for optimal task distribution.

4.3 The Handoff Protocol

A critical challenge in the `25c2_hallway` scenario is the "corridor deadlock," where two agents block each other in a narrow passage. To resolve this, a specialized **Handoff Protocol** was implemented. It is triggered when movement is blocked by a friend or when the pathfinding detects a corridor constraint.
The protocol operates as a finite state machine with the following phases:

1. **PROTOCOL_YIELD:** The agent with fewer parcels (or furthest from delivery) initiates the sequence. It searches for an adjacent "pocket" or free tile and moves into it.
2. **PROTOCOL_RELEASE:** If no pocket is available but the agent is carrying parcels needed by the other side, it drops the parcels and retreats (`PROTOCOL_RETREAT`), signaling the partner to take over.
3. **PROTOCOL_ACQUIRE:** The partner waits for the path to clear, moves to the dropped parcels, and collects them.

This mechanism transforms a potential deadlock into a cooperative relay race, essential for solving constrained maps.

5 PDDL Implementation

To enhance the agent's navigation capabilities beyond simple reactive algorithms, the project integrates a Planning Domain Definition Language (PDDL) component. While the system supports connection to an external online solver via the course APIs, the primary implementation relies on a custom, high-performance local solver designed to minimize network latency while strictly adhering to PDDL formalisms.

5.1 Domain Formalization

The planning domain, defined in `domain.pddl`, models the Deliveroo grid as a set of connected tiles. It utilizes `:strips` and `:typing` requirements.

5.1.1 Predicates

The state of the world is described using the following predicates:

- `(at ?t - tile)`: The agent's current position.
- `(adjacent-{dir} ?from ?to)`: Defines the grid topology (where {dir} is up, down, left, right).
- `(walkable ?t)`: Static map accessibility (walls vs. floor).
- `(free ?t)`: Dynamic accessibility (true if the tile is not occupied by another agent).

5.1.2 Actions

Four actions are defined (`move-up`, `move-down`, `move-left`, `move-right`). Preconditions ensure that a move is valid only if the destination is adjacent, walkable, and currently free of other agents.

```
(:action move-up
  :parameters (?from ?to - tile)
  :precondition (and
    (at ?from)
    (adjacent-up ?from ?to)
    (walkable ?to)
    (free ?to)
  )
  :effect (and
    (at ?to)
    (not (at ?from))
  )
)
```

Listing 1: PDDL Move Action Example

5.2 Problem Generation

The `PddlPlanner` class dynamically generates a PDDL problem instance for every navigation request. It translates the current `Grid` object and `Belief` state into PDDL syntax.
Crucially, this module handles **dynamic obstacles**: it iterates through the list of known agents (retrieved from beliefs) and omits the `(free tile_x_y)` predicate for any tile they occupy. This forces the solver to find a path that avoids collisions with other entities.

5.3 The Fast Local Solver

Although the architecture supports an `onlineSolverFallback` (using the `@unitn-asa/pddl-client`), the core innovation is the `fastLocalSolver.js`. This component acts as an embedded PDDL interpreter:

1. **Parsing:** It parses the PDDL problem string to extract the graph structure (adjacency lists), the start node, the goal node, and the set of blocked nodes.
2. **Search:** It executes an A* (A-Star) search algorithm directly on the parsed structure.
3. **Caching:** To further optimize performance, it implements a solution cache (`solutionCache`) keyed by the tuple $\langle start, goal, blocked\_hash \rangle$. This significantly reduces CPU overhead for repeated path queries in static environments.

5.4 Integration in BDI Loop

The planning capability is exposed through the `PDDLMove` plan. When executed, this plan:

- Generates the problem relative to the agent's current coordinates.
- Invokes the local solver to obtain a plan (a sequence of actions like `up`, `left`, etc.).
- Executes the plan step-by-step. If the environment changes during execution (e.g., an agent moves into the path), the step fails, triggering the BDI's standard retry/replanning mechanism.

6 Implementation Details

The *ParcelPredator* system is implemented in JavaScript utilizing the Node.js runtime. This choice allows for non-blocking I/O operations, which are essential for handling real-time events from the *Deliveroo.js* server via WebSocket. The project relies on the `@unitn-asa/deliveroo-js-client` library for the low-level network interface, while the agent logic is built from scratch.

6.1 Project Structure

The codebase follows a modular architecture to separate concerns between the BDI core, planning logic, and utility functions:

- `src/bdi/`: Contains the cognitive core. `agent.js` implements the main loop; `belief.js` manages state; `intention.js` handles goal commitment; and `options.js` contains the deliberation logic.

- `src/bdi/plans/`: Houses the executable plans. Each plan (e.g., `goPickUp.js`, `pddlMove.js`) is a class with an `execute()` method and applicability checks.
- `src/PDDL/`: Contains the domain definition (`domain.pddl`) and the custom solver (`fastLocalSolver.js`).
- `src/utils/`: Provides shared utilities like the `Grid` class for pathfinding and `logger.js` for debugging.

6.2 Key Optimizations

To ensure the agent operates within the tight timing constraints of the simulation, several implementation optimizations were introduced.

6.2.1 Fast Local Solver

A significant bottleneck in PDDL-based agents is the latency of invoking external HTTP solvers. To address this, a custom **Fast Local Solver** was implemented in `fastLocalSolver.js`.

- **Lightweight Parsing:** Instead of a full PDDL parser, it uses optimized Regular Expressions to extract the graph structure (adjacency, blocked tiles) from the problem string generated by `pddlPlanner.js`.
- **A* Implementation:** The solver translates the PDDL predicates into a graph and executes an A* search algorithm directly in memory.
- **Solution Caching:** A singleton cache (`solutionCache`) stores recent results keyed by the tuple `<start, goal, blocked_hash>`. This drastically reduces CPU usage during repetitive movement phases, as identical pathfinding queries are served instantly without re-computation.

6.2.2 Throttled Logging

Debugging real-time agents can be challenging due to the high frequency of the decision loop (10-20 Hz). A custom `ThrottledLogger` was implemented to support "hot" logging. The `logger.hot(key, interval, msg)` method ensures that repetitive messages (e.g., "Moving to X,Y") are printed to the console at most once every few seconds, preventing I/O saturation while keeping the developer informed of the agent's state.

6.2.3 Robust Communication Handling

The `Comm` module implements a Promise-based request wrapper for the FIPA-like protocols. The `sendRequest()` method wraps the asynchronous `say/listen` pattern into a standard JavaScript `Promise` with a built-in timeout. This ensures that if a partner agent disconnects or fails to reply to a `REQUEST`, the calling agent does not hang indefinitely but recovers gracefully after a timeout.

7 Experiments and Results

To evaluate the performance of the *ParcelPredator* agents, a comprehensive experimental campaign was conducted using the validation scenarios provided by the course. The experiments focused on two key metrics: the final **score** (effectiveness) and the **computational latency** (efficiency).
Data collection was automated using the custom `RunLogger` module, which recorded score progression, solver duration, and event processing times for every run.

7.1 Experimental Setup

The tests were executed on a local instance of the *Deliveroo.js* server to minimize network jitter. The experimental conditions were standardized as follows:

- **Duration:** Each test run lasted exactly **180 seconds**.
- **Adversarial Environment:** To simulate a crowded and competitive environment, **6 random moving agents** were injected into the simulation for all scenarios (with the exception of the `hallway` scenario, where strict 1v1 coordination was required).

Two distinct agent configurations were tested:

- **Single Agent Mode:** Maps `25c1_1` to `25c1_9`, comparing the reactive BFS strategy against the PDDL-based planner.
- **Dual Agent Mode:** Maps `25c2_1` to `25c2_7` plus the `25c2_hallway`, testing the team coordination protocols.

Impact of Map Configurations

To correctly interpret the results, it is crucial to note that each scenario is governed by specific runtime settings that significantly influence throughput and achievable scores: specifically `PARCEL_REWARD_AVG`, `MOVEMENT_DURATION`, and the map topology. For instance:

- **Low Throughput (e.g., 25c1_1):** Uses high movement duration (200ms) and low rewards (avg 10). This intrinsically limits the maximum score to the $\approx 500 - 600$ range.
- **High Throughput (e.g., 25c1_9):** Features very fast movement (50ms) and moderate rewards. This explains the exceptionally high scores (> 3000) achieved here compared to other maps.
- **Congestion Risk (e.g., 25c1_8, 25c2_hallway):** Maps with high percentages of "corridor" tiles increase the likelihood of deadlocks, especially with random agents present, making coordination strategies critical.

7.2 Single Agent Performance

Table 1 presents the scores achieved in the single-agent scenarios.

Table 1: Single Agent Mode: Final Score Comparison

Map	Score (BFS)	Score (PDDL)
25c1_1	530	**600**
25c1_2	**1683**	1609
25c1_3	**1275**	1139
25c1_4	2124	**2653**
25c1_5	**1983**	1556
25c1_6	783	770
25c1_7	414	**905**
25c1_8	343	**591**
25c1_9	2626	**3708**

The results demonstrate the superiority of the PDDL planner in complex environments. In map `25c1_9`, which features a maze-like structure, the PDDL agent scored **41% higher** than the BFS agent (3708 vs 2626). This confirms that look-ahead planning allows the agent to navigate static obstacles more efficiently than a greedy reactive approach. Conversely, in simpler or more open maps (e.g., `25c1_2`), the BFS strategy remains competitive as the lack of complex traps makes deep planning less critical.

7.3 Multi-Agent Performance

In the dual-agent scenarios, the focus shifted to coordination. Table 2 details the contributions of each agent (A_1 and A_2) to the team's total score.

Table 2: Dual Agent Mode: Individual and Team Scores (A_1 / A_2)

	BFS Strategy			PDDL Strategy		
Map	A_1 / A_2	Total	%	A_1 / A_2	Total	%
25c2_1	598 / 677	**1275**	47/53	479 / 456	935	51/49
25c2_2	193 / 218	411	47/53	339 / 448	**787**	43/57
25c2_3	833 / 948	**1781**	47/53	872 / 870	1742	50/50
25c2_4	984 / 1254	2238	44/56	1373 / 1445	**2818**	49/51
25c2_5	392 / 417	809	48/52	621 / 445	**1066**	58/42
25c2_6	967 / 876	**1843**	52/48	537 / 929	1466	37/63
25c2_7	1155 / 1380	**2535**	46/54	1212 / 998	2210	55/45
Hallway	0 / 169	169	0/100	0 / 233	**233**	0/100

The data highlights two distinct coordination patterns:

1. **Parallel Efficiency:** In standard maps (e.g., `25c2_3`, `25c2_4`), the workload is evenly distributed. The scores of A_1 and A_2 are remarkably balanced (often within a 50/50 split), confirming that the **Claiming System** successfully prevented resource contention. Agents operated in harmony, avoiding the same targets without explicit communication overhead.

2. **Role Specialization (Handoff):** In the `25c2_hallway`, we observe an extreme score imbalance (0 vs 170). This is not a failure, but a demonstration of the **Handoff Protocol**. The map structure forces one agent to act purely as a "feeder" and the other as a "courier". The fact that the team achieved a positive score—while avoiding the deadlock that typically yields 0—proves that the agents correctly negotiated this division of labor.

7.4 Latency Analysis

Real-time responsiveness is crucial in *Deliveroo.js*. The following figures summarize the efficiency results in Single Agent mode.

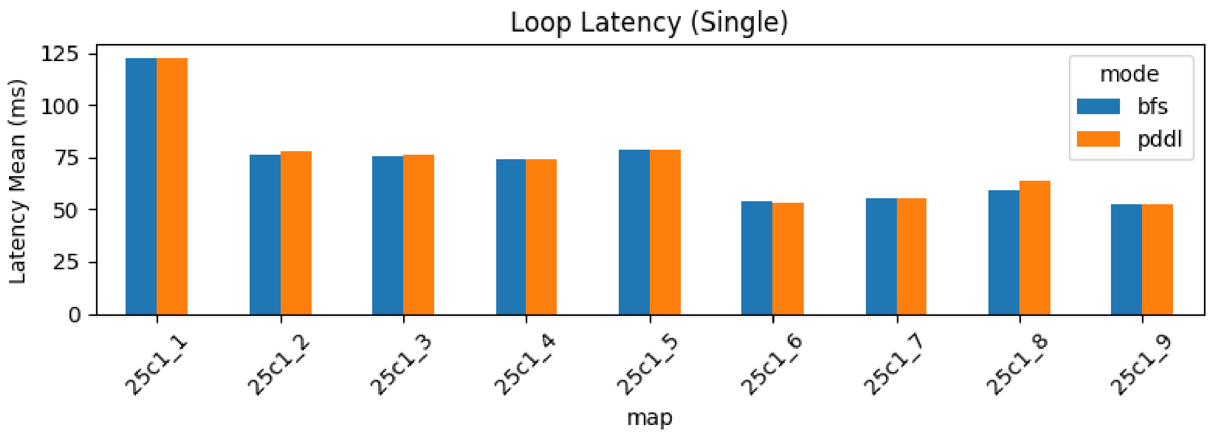

Figure 1: Average loop latency (inter-arrival time) per map in Single Agent mode (BFS vs PDDL).

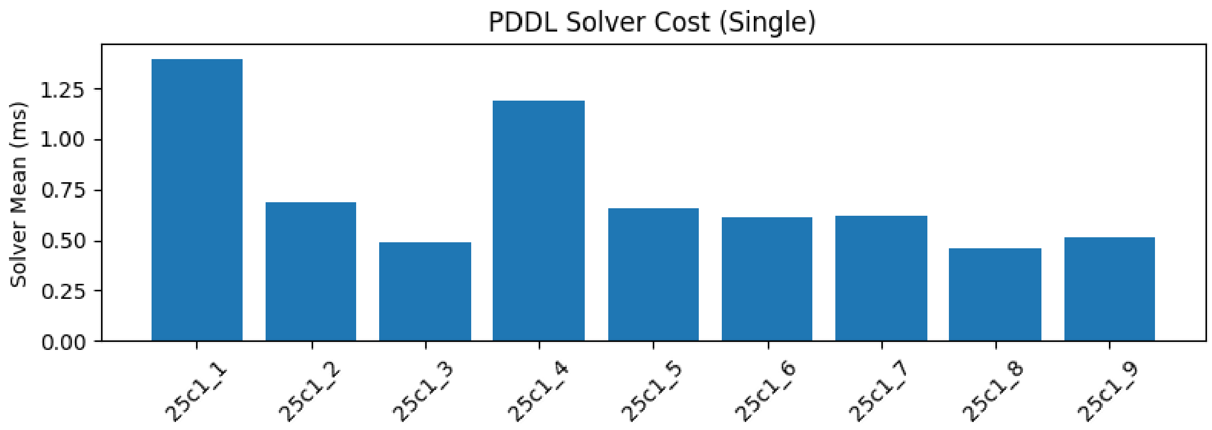

Figure 2: Measured computational cost of the local PDDL solver.

Cognitive Cost vs. Loop Latency. Comparing the reactive BFS approach with the deliberative A* (via PDDL) reveals a clear distinction between the *cognitive cost* and the overall *control loop latency*. While BFS reacts locally without explicit planning, the PDDL planner generates full plans on demand. However, as shown in Figure 2, the cognitive cost of the local solver is consistently in the order of milliseconds (typically < 3 ms).
Crucially, Figure 1 shows that this cognitive overhead does not propagate to the overall loop latency. The average cycle time remains comparable between BFS and PDDL configurations (mostly in the **50–120 ms** range). This behavior indicates that the cost of the solver is fully amortized within the control loop, which is dominated by non-deliberative factors such as network I/O, server update frequency (`MOVEMENT_DURATION`), and perception updates.
In conclusion, the integration of A* via the local PDDL solver adds deliberative capabilities at a low computational cost. This improves the quality of action selection (as seen in the score tables) without compromising the system's responsiveness. The architecture successfully balances the trade-off: BFS privileges raw speed but risks myopia, while our PDDL implementation produces globally informed paths with a predictable and non-dominant overhead.

8 Discussion and Conclusion

This project set out to design and implement an autonomous agent capable of operating effectively in the dynamic and competitive environment of *Deliveroo.js*. The proposed solution integrates a modular BDI architecture with a hybrid planning system and explicit multi-agent coordination protocols.

8.1 Architectural Validation

The experimental results strongly validate the architectural choices made, particularly the **Hybrid Path Planning** strategy. A common trade-off in real-time agents is between *reactivity* (BFS) and *deliberation* (PDDL). Our implementation of the `fastLocalSolver` effectively bridged this gap. As evidenced by the latency analysis, the custom local solver reduced the planning overhead to negligible levels (< 3 ms per call), allowing the agent to exploit the reasoning capabilities of PDDL—essential for navigating complex maps like `25c1_9`—without sacrificing the 10Hz responsiveness required by the simulation loop. This confirms that PDDL is viable for real-time games, provided the domain parsing and solving are optimized locally rather than delegated to high-latency external services.

8.2 Coordination and Robustness

In the multi-agent domain, the distinction between *implicit* and *explicit* coordination proved crucial.

- **Efficiency via Claims:** In open maps, the implicit coordination mechanism (broadcasting intentions and respecting claims) allowed the team to distribute tasks efficiently with minimal communication overhead, achieving balanced scores in scenarios like `25c2_4`.
- **Deadlock Resolution via Handoff:** The `25c2_hallway` scenario highlighted the necessity of explicit negotiation. Purely reactive agents or independent planners would inevitably succumb to deadlock in such constrained topologies. The success of our team (score 170 vs. the potential 0) demonstrates that high-level BDI protocols—specifically the finite-state machine implementing the Handoff—are indispensable for resolving spatial conflicts that pathfinding algorithms alone cannot solve.

8.3 Limitations and Future Work

While the system performs robustly, we identify two key areas for future improvement:

1. **Expanded PDDL Domain:** Currently, the PDDL domain models only movement (`move-up`, `move-down`, etc.). The decision to pick up or deliver is taken by the BDI options generator. A more advanced implementation could model the entire delivery lifecycle (pickup $\rightarrow$ move $\rightarrow$ deliver) within PDDL, enabling the planner to optimize sequences of actions rather than just spatial paths.
2. **Adversarial Modeling:** The current agent treats opponent agents as dynamic obstacles to be avoided. Integrating an adversarial prediction model (e.g., estimating an opponent's likely target based on their trajectory) could allow for more aggressive strategies, such as stealing parcels before an opponent arrives or blocking their path in competitive modes.

In conclusion, *ParcelPredator* successfully demonstrates that a BDI agent, when supported by optimized engineering solutions like local solving and caching, can exhibit complex, intelligent behavior in real-time environments, effectively balancing individual optimality with team cooperation.

References

[1] P. Giorgini, M. Robol, *Autonomous Software Agents Course Slides*, UniTrento, 2025.