Design and Implementation of a BDI Multi-Agent System for the Deliveroo Game

Autonomous Software Agents (2024/25)

Jie Chen – 256177 – jie.chen-2@studenti.unitn.it
Nancy Kalaj – 257021 – nancy.kalaj@studenti.unitn.it

University of Trento
DISI Department
Povo (TN), Italy

Introduction

Intelligent agents require architecture that allow them to operate autonomously in dynamic environments. The Belief-Desire-Intention (BDI) architecture represents a fundamental model for designing rational agents. It enables agents to act based on their beliefs about the environment, their goals (desire), and their plans (intentions). This report describes the design and implementation of a BDI agent-based system for the Deliveroo game. We start by first describing the single-agent implementation and then extend it to the multi-agent scenario.

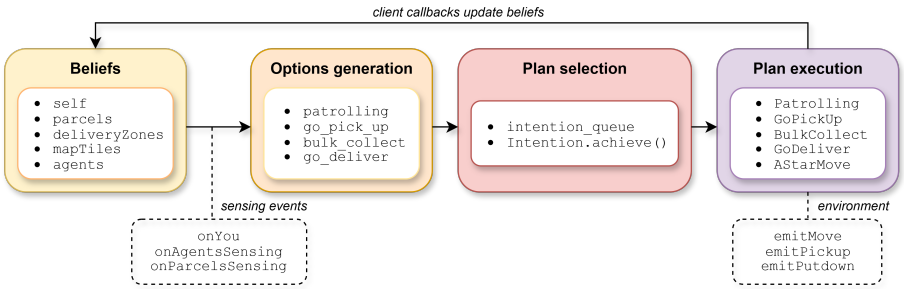

Figure 1: BDI control loop showing the flow from beliefs through options generation, intention revision, plan selection, and plan execution.

1 Belief

Belief	Structure	Source	Purpose
Self (`me`)	Object	`onYou`	Tracks own position, ID, and score.
Parcels	Map	`onParcelsSensing`	Tracks parcel location, reward, and status.
Delivery Zones	Array	`onMap`, `onTile`	Stores delivery zone coordinates.
Map Tiles	Map	`onMap`, `onTile`	Stores tile types and lock status.
Other Agents	Map	`onAgentsSensing`	Tracks other agents' positions and scores.

Table 1: Summary of agent beliefs, structures, sources, and purposes.

The belief represents the agent's knowledge about the environment, which it continuously updates based on sensor inputs and events. In this implementation, the agent maintains several types of beliefs:

- **Self-beliefs (`me` object):** The agent tracks its own position and score. These beliefs are updated continuously through the `client.onYou` event handler, ensuring the agent always knows its current state.
- **Parcel beliefs (`parcels` map):** The agent maintains a map of all known parcels, tracking their location, reward, and whether they are contested by other agents. Additional tracked attributes are:
  - `lastSeen`: timestamp of last observation
  - `spawnTime`: when the parcel first appeared
  - `initialReward`: the reward when spawned
  - `expectedUtility`: the adjusted reward considering contestation by rival agents

These beliefs are updated using the `client.onParcelsSensing` handler. For each sensed parcel, the agent:

1. Checks if the parcel has been stolen by another agent and deletes it from the beliefs if so;
2. Determines if the parcel is contested by checking whether any rival agent is within `CONTEST_RADIUS` and closer or equally close compared to itself, i.e.:

```
const contested = rivals.some(a =>
  distance(a, p) <= distance(me, p)
);
```

3. Adjusts the parcel's `expectedUtility` based on contestation by applying the `CONTEST_PENALTY` if contested.
4. Inserts new parcels or updates existing entries accordingly.
5. Removes parcels if:
   - They are carried by other agents;
   - They were carried by self and are no longer visible;
   - Their reward has decayed to zero;
   - Their lifetime has exceeded the maximum allowed duration.
6. The `suspendedDeliveries` set is also updated accordingly when parcels are no longer present.

- **Beliefs about other agents (`agents` map):** The agent keeps track of the positions and scores of all other agents using the `client.onAgentsSensing` event handler. For each sensing cycle, it:

  1. Inserts or updates each sensed agent;
  2. Removes agents that were not sensed in the current cycle to ensure beliefs remain up-to-date.

  After updating the agents, all old locks in `mapTiles` are cleared and tiles under each sensed agent are locked as follows:

```
const key = `${a.x},${a.y}`;
const tile = mapTiles.get(key);
if (tile) tile.locked = true;
```

  This prevents the planner (A* or patrol) from planning paths through tiles occupied by agents.

- **Map tile beliefs (`mapTiles` map):** The agent maintains a map of all tiles, storing their type (wall, walkable, delivery zone) and whether they are currently locked (occupied). This is initialized using `client.onMap` and updated via `client.onTile` to reflect any dynamic changes.

- **Delivery zones (`deliveryZones` array):** The agent maintains a list of delivery zone coordinates where parcels can be delivered. This is initialized at the start using `client.onMap` and updated dynamically through `client.onTile` if tile types change to or from delivery zones.

2 Desire

Desires represent the goals the agent can potentially achieve and are derived from the agent's beliefs about the environment. In this implementation, the main desires include:

- Delivering parcels to delivery zones if the agent is carrying any.
- Picking up parcels to maximize utility, either individually or through bulk collection to optimize trips.
- Patrolling the map to explore and discover new parcels when no parcels are available or reachable.

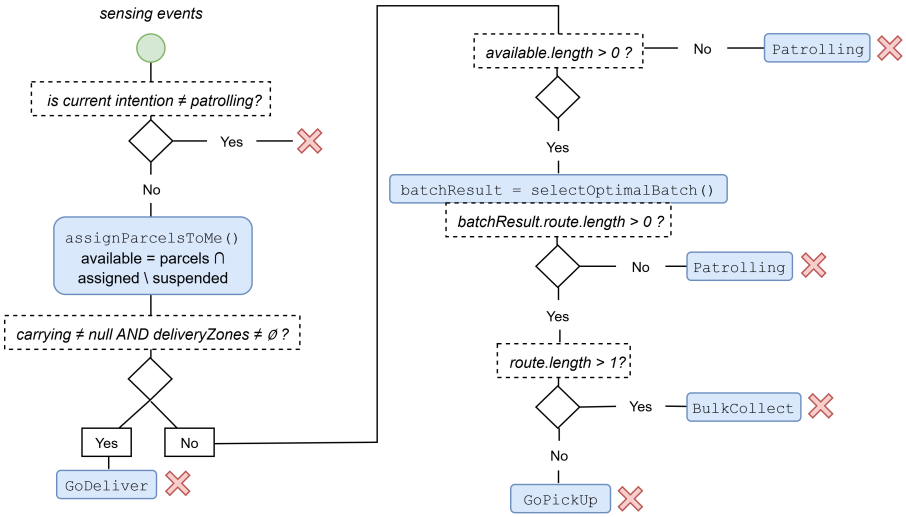

Figure 2: Decision flow inside `optionsGeneration()`: checks busy state, assigns parcels, and chooses between delivery, bulk collection, single pickup, or patrolling.

The `optionsGeneration` function is responsible for evaluating these desires based on the agent's current beliefs and generating possible options to pursue next, pushing new intentions into `myAgent.intention_queue`. It is triggered whenever new sensing data arrives, ensuring the agent continuously re-evaluates its options in response to environmental changes:

```
client.onParcelsSensing(optionsGeneration);
client.onAgentsSensing(optionsGeneration);
client.onYou(optionsGeneration);
```

To avoid interrupting ongoing tasks such as bulk collection or delivery, the function checks if the current intention is not patrolling:

```
const current = myAgent.intention_queue[0];
if (current && current.predicate[0] !== 'patrolling') {
  return;
}
```

Within `optionsGeneration`:

- `options`: stores the possible intentions under consideration

- `carried`: identifies parcels the agent is currently carrying
- `available`: lists all parcels not carried by any agent and not suspended
- `localCount`: counts how many parcels are within a `LOCAL_RADIUS`, used later to evaluate batch collection utility

The decision logic proceeds as follows:

1. If carrying parcels: The desire is to deliver them to the nearest delivery zone.
2. Else if parcels are available:
   - The agent first attempts bulk collection by calling `selectOptimalBatch`, which computes the optimal set of parcels to collect in one route based on utility (reward minus cost) and proximity.
   - If bulk collection is feasible and the route includes multiple parcels, it adds a `bulk_collect` option with their IDs. If the route includes only one parcel, it falls back to a `go_pick_up` option for that parcel.
   - If bulk collection fails, it tries single parcel pickup. It sorts available parcels by distance, then for each parcel (starting from the closest) it runs A* pathfinding to check if the parcel is reachable. If a reachable parcel is found, it adds a `go_pick_up` option for it and stops searching. If no parcel is reachable, it adds patrolling as the fallback option.
3. Else (no parcels available): It defaults to patrolling to explore the map and potentially find new parcels.

Finally, the function avoids pushing duplicate intentions by comparing the best new option to the last intention in the queue. If it is different, it pushes the new option as the agent's next intention.

Net Utility Computation The net utility for a batch of parcels is computed as follows:

$$U_{\text{net}}(B) = \sum_{p \in B} U_{\text{exp}}(p) \; - \; \alpha \; \times \; \frac{T_{\text{travel}}(B)}{\Delta_{\text{decay}}}$$

where B is the chosen batch of parcels, $U_{\text{exp}}(p)$ is the expected utility of parcel p, computed as:

$$U_{\text{exp}}(p) = \begin{cases} \text{reward}(p), & \text{if not contested,} \\ (1 - \text{CONTEST\_PENALTY}) \times \text{reward}(p), & \text{if contested.} \end{cases}$$

Here, α is the movement penalty (the `PENALTY` constant), $T_{\text{travel}}(B)$ is the total estimated travel time (in seconds) to pick up all parcels in B in sequence and deliver them to the nearest delivery zone, and Δ_{decay} is the reward-decay interval in seconds (i.e., `DECAY_INTERVAL_MS` divided by 1000).

In other words, the first term computes the sum of each parcel's expected utility, which also takes into account the discounted utilities of "contested" parcels. The `CONTEST_PENALTY` is a discount factor applied to a parcel's expected utility when it is contested by other agents, reflecting the risk that another agent might pick it up first. If a rival agent is closer to or equally far from a parcel compared to our agent, the parcel is marked as

contested, indicating a high chance of being lost. To account for this competition risk, the agent reduces the parcel's utility by multiplying its reward by $(1 - \texttt{CONTEST\_PENALTY})$. For example, with `CONTEST_PENALTY`=0.5, the expected utility becomes half its original reward if contested. This adjustment ensures that contested parcels appear less attractive in planning decisions, helping the agent prioritize parcels with a higher likelihood of successful pickup and avoid wasting time on risky targets.

The second term simply subtracts the time-cost: travel time estimation is done by invoking `AStarDaemon`, which computes pairwise path lengths (in steps) between (1) the agent's current position and the first parcel, (2) between successive parcels in the candidate batch, and (3) from the last parcel to the nearest delivery zone. The movement penalty α converts travel time into a utility penalty, and dividing the travel time by Δ_{decay} normalizes it against the rate at which parcel rewards decay (a longer route not only delays delivery but also causes parcels' values to drop, so the penalty should scale with both factors). By combining reward gains and time costs in the same formula, the agent can compute whether picking up certain parcels in a certain amount of time is worthwhile in terms of overall utility.

Batch Selection Strategy The `selectOptimalBatch` module selects the optimal batch of parcels for an agent to collect in a single route to maximize its net utility, balancing reward gains against time decay penalties. The algorithm uses both a greedy approximation and exact combinatorial optimization: for small batch sizes, it prefers the exact approach to guarantee optimality; for larger sizes, it switches to greedy to maintain efficiency.

The greedy approach incrementally builds a batch by repeatedly choosing the parcel with the highest net utility. It continues adding parcels until reaching a predefined maximum batch size or no parcel yields positive net utility, and finally adds the travel cost from the last picked parcel to the nearest delivery zone to compute total net utility.

The exact approach evaluates all possible visit orders of a subset of max K parcels to find the truly optimal route with the highest net utility, at the cost of increased computation due to combinatorial explosion.

Underlying both strategies is the use of A* pathfinding to compute accurate distances between parcels and delivery zones, ensuring realistic route planning.

3 Intention and Intention Revision

Intentions are desires that the agent has committed to achieving. In this implementation, the agent maintains an intention queue (`intention_queue`), which is managed by the `IntentionRevisionReplace` class. Each intention is an instance of the `Intention` class and consists of a predicate (which defines the goal—for example, `['go_pick_up', x, y, id]`, `['go_deliver', x, y]`, or `['patrolling']`) and a corresponding plan to achieve that goal.

The intention revision mechanism manages the queue by selecting and executing intentions in order. It operates in a continuous loop, repeatedly checking the current intention, executing the associated plan, and revising intentions by replacing or stopping them as necessary (for example, if parcels become unavailable or if a better goal emerges).

The agent employs an Intention Revision Replace strategy: when a new, better intention is generated (such as discovering a parcel with higher utility), the current intention is stopped and replaced with the new one. To avoid redundant work, the agent compares

the last intention in the queue with the new intention and only pushes the new one if it is different. If there was a previous intention, it is stopped to ensure that only one is pursued at a time. This approach ensures adaptive and reactive behavior, which is a key feature of BDI agents.

4 Planning

Each intention is fulfilled by executing a plan, implemented as a class in the `planLibrary`. The main plan types are: **GoPickUp**, which moves to a parcel's location and picks it up; **GoDeliver**, which moves to a delivery zone and delivers any carried parcels; **BulkCollect**, which collects multiple parcels in an optimized route before delivering them together; **Patrolling**, which explores under-visited sectors when no parcels are available; and **AstarMove**, which navigates to a specific location using A* pathfinding.

Each plan class exposes two main methods: `isApplicableTo`, to check if the plan can handle a given intention, and `execute`, which performs the required sequence of actions to achieve the intention.

Plan Class	Goal	Key Actions
GoPickUp	go_pick_up	Move to parcel → `emitPickup()`
GoDeliver	go_deliver	Move to delivery zone → `emitPutdown()`
BulkCollect	bulk_collect	Batch pickup → Delivery
Patrolling	patrolling	Explore oldest map sectors
AstarMove	go_to	Pathfinding + opportunistic pickup/delivery

Table 2: Summary of plan classes, their goals, and key actions.

The **GoPickUp** plan implements the logic for picking up a parcel. If the agent is already at the parcel's location, it attempts to pick it up via `client.emitPickup()`; if successful, the parcel is removed from both the list of known parcels and suspended deliveries. If unsuccessful, the parcel is removed and the intention is aborted. If the agent is not at the parcel location, a sub-intention is created with `this.subIntention(['go_to', x, y])` to navigate there. If movement fails, the parcel is removed and the plan is aborted. Upon arrival, the plan retries pickup, handling failure as above.

The **GoDeliver** plan is responsible for delivering carried parcels. It first filters parcels carried by the agent and not suspended. If there are none, the plan terminates. Otherwise, it attempts to deliver to delivery zones, sorted by Manhattan distance. For each zone, it tries up to three times to reach it (using the sub-intention mechanism) and upon arrival, invokes `client.emitPutdown()` to drop parcels. If all zones are blocked, the parcels are suspended to avoid immediate retries and the intention is stopped, prompting a replan.

The **Patrolling** plan implements a sector-based exploration strategy to efficiently cover the map. The agent prioritizes sectors that have not been visited recently, selects the least recently visited sector, and attempts to move to a set of spawn tiles within it. Upon reaching such a tile, the agent performs a series of scouting movements to adjacent tiles to sense for parcels. If all attempts fail, the agent tries other sectors, up to a set maximum, ensuring broad and efficient exploration and avoiding redundant patrolling.

The **BulkCollect** plan implements an efficient strategy for collecting and delivering multiple parcels. The plan maps the intended parcel IDs to parcel objects, filtering out

any that are no longer valid. It then loops over the remaining parcels, moving to each and attempting pickup. If pickup fails (e.g., the parcel is stolen), it penalizes the parcel's expected utility and breaks the loop. After attempting to collect, if at least one parcel has been picked up and a delivery zone is available, the agent moves to the nearest delivery zone and sequentially puts down all carried parcels. This plan handles dynamic failures and aims for efficient high-reward collection.

The **AstarMove** plan executes the "go_to" intention using A* pathfinding. The plan invokes `aStarDaemon.aStar()` with the agent's current position, the target, and a Manhattan distance heuristic. If planning fails, the intention is aborted. The agent then executes each step along the planned path, checking for parcels to opportunistically pick up and for delivery zones where parcels can be dropped off. The plan also checks for locked (occupied) tiles and aborts if blocked, triggering replanning. This approach ensures dynamic, collision-free navigation while maximizing opportunities for pickup and delivery.

At the core of navigation, the `AStarDaemon` provides a self-contained pathfinding engine built atop the tile map. It exposes a single `aStar(start, goal, h)` call, returning an ordered list of {`x,y,action`} steps or the string `"failure"` if no route exists. The daemon maintains three core data structures: a min-heap open set keyed by f-score (the sum of g-score and heuristic), a map of g-scores for tracking minimal known costs from the start, and a map of f-scores for estimated total costs to the goal. During search, the daemon extracts the lowest-f-score node, generates cardinal neighbors, and prunes those that are off-map, walls, or currently locked. For valid neighbors, it computes the tentative g-score,

$$g_{\text{score}}(n) = g_{\text{score}}(\text{parent}(n)) + 1$$

and adds any tile-specific penalty. If this undercuts the neighbor's recorded g-score, the daemon updates its maps and reinserts the neighbor into the heap. When the heuristic at a popped node is zero, the daemon reconstructs the optimal path by backtracking via the `cameFrom` map and annotates each move with its cardinal direction (`right`, `left`, `up`, or `down`). By centralizing pathfinding, cost accounting, and avoidance logic, the daemon allows higher-level planning code to request reliable routes without managing graph search intricacies directly.

Planning with PDDL

In order to perform planning with PDDL (Planning Domain Definition Language), we first define a `domain.pddl` that specifies the types (such as the agent itself, other agents, parcels, and map cells), predicates (describing the environment and agent's state), and actions (including movement and parcel handling), each with their parameters, preconditions, and effects. This domain definition enables the planner to generate sequences of move and pickup/putdown actions to achieve goals while respecting obstacles, the map layout, and the positions of other agents.

The `pddl_planner.js` file contains parsing functions to translate the environment states into PDDL-compatible predicates and objects, construct dynamic problem definitions with the desired goals, and invoke an online PDDL solver to compute a feasible action plan. Specifically, it reads the `domain.pddl` file to obtain the action definitions and uses the agent's current beliefs to generate a problem description. Its main planner function (`async planner(parcels, agents, me, goal)`) builds a new PDDL problem

every time a plan is needed, ensuring up-to-date plans that reflect the agent's current perceptions.

By translating high-level desires—such as picking up parcels, delivering them, or navigating to specific coordinates—into formal PDDL goals and retrieving executable plans, this module allows the agent to act intelligently within a dynamic environment.

5 Multi-Agent System

In the multi-agent extension, each agent continues to run an independent local BDI loop as described previously, but a lightweight messaging protocol is layered on top to enable effective communication and coordination between teammates.

The most basic form of inter-agent messaging serves two crucial purposes: first, it allows each agent to maintain an up-to-date belief about its teammate's position by periodically broadcasting position updates (e.g., via `client.emitSay(teamMateId, { action: 'position_update', x: me.x, y: me.y })`). Upon receipt of such a message, the agent updates a local `lastSeenMate` belief, which is then used for cost-estimation and parcel assignment. Second, agents share their current observations about parcels by broadcasting a `parcel_snapshot` message after each sensory update. This enables each agent to integrate parcels observed by the teammate into its own belief map, recompute contestation status, and merge temporal information, thus ensuring both agents maintain a comprehensive and synchronized view of all parcels, even in the presence of occlusions or temporary loss of line-of-sight.

In the multi-agent system, each agent calls the `assignParcelsToMe()` function to decide which parcels it should pursue. First, any stale entries in the global `pickupCoordination` table older than `CLAIM_TTL` (10 000 ms) are removed. Then, for each parcel p in the shared `parcels` map that is not already carried, the agent computes two A*–based cost estimates via `estimateCost()`: one from its own position (`me.x`, `me.y`) and one from the teammate's last known coordinates (`matePos` from `agents.get(teamMateId)` or `lastSeenMate`). These yield `myCost` and `theirCost`, respectively. Next, the agent calculates net utilities `myNet` and `theirNet`; if `myNet` exceeds `theirNet`, the parcel's ID is added to the agent's `assigned` set. In the case of a near-tie (within floating-point epsilon), the agent invokes the pickup coordination protocol: it sends an `emitAsk(teamMateId,{action:'pickup',parcelId:id})` RPC and awaits a Boolean reply. The listener in `client.onMsg(...)` checks the same `pickupCoordination[id]` entry (after a brief randomized delay), grants the claim by writing `{id:requesterId,ts:Date.now()}` into `pickupCoordination[id]` if unclaimed, or denies it otherwise. A denial causes the local `GoPickUp` plan for that parcel to abort immediately. All claims persist for `CLAIM_TTL` before expiring, ensuring that exactly one agent commits to each contested parcel at any time. Finally, only parcels in the resulting `assigned` set enter the agent's local `available` list for option generation, thereby balancing workload and maximizing overall team efficiency.

To guarantee safe traversal of narrow passages, the agent first calls `detectCorridors(mapTiles)` during map initialization, which flags every walkable cell with exactly two opposing neighbors (north–south or east–west) and clusters them into segments via BFS. The `buildCorridorMap` function then populates two lookup tables: `corridorCellMap` (cell → segment ID) and `corridorCellsById` (seg-

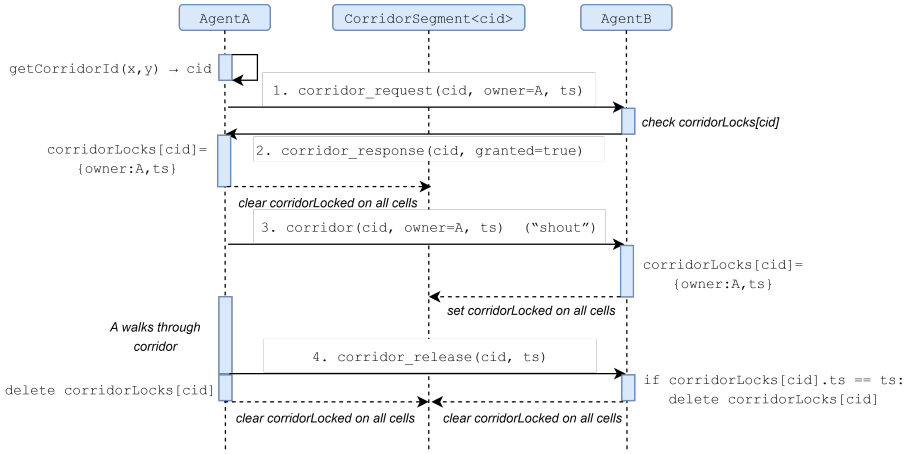

Figure 3: Sequence diagram of the corridor-locking protocol between two agents across segment `cid`. Agent A requests, Agent B responds, then A "shouts" its new ownership, travels, and finally releases.

ment ID → [cell keys]) and augments each tile object with a `corridorLocked` flag initialized to `false` (or `true` if resurrecting an existing lock).

During planning, the `AstarMove` plan invokes `cleanupCorridorLocks()` to expire any locks older than `CORRIDOR_LOCK_TTL`. Before stepping into each next tile of its A*-computed route, it calls `getCorridorId(x,y)`. If that ID differs from the one it's already holding (or is non-null when entering a corridor for the very first time), it must acquire exclusive access by awaiting the promise returned from `askCorridorLock(teamMateId, cid, me.id)`, which under the hood emits a `corridor_request` via `client.emitSay`. When the promise resolves to `true`, the agent immediately clears `corridorLocked` on all cells in `corridorCellsById.get(cid)`, then broadcasts a corridor "shout" (again via `client.emitSay`) so that the teammate sets its own `corridorLocked = true` on those very same cells. Upon exiting the segment (detected by a new `getCorridorId` call), the agent emits a `corridor_release` and resets that segment's tiles to `corridorLocked = false`. If a lock request is denied (or if a corridor-locked cell remains occupied), the `AstarMove` loop will either pause briefly (up to `MAX_LOCK_WAIT` cycles) or call `aStarDaemon.addTempPenalty(x, y, penalty)` to nudge A* toward an alternate route. Together with the periodic textttcleanupCorridorLocks, this fully decentralized mutual-exclusion protocol—tied directly into the A* navigation and intention execution— ensures that two agents never collide in one-tile-wide passages while preserving autonomous, responsive path planning.

Finally, a full "physical handoff" capability has been integrated alongside existing BDI and A* logic. Three new global states—`STATE_IDLE`, `STATE_HANDOFF_PENDING` and `STATE_HANDOFF_ACTIVE`—together with a `state` variable and a `handoffInProgressParcels` set allow each agent to signal and track a handoff protocol via periodic `emitSay(..., {action:'position_update',state})` broadcasts. Incoming messages are queued in `pendingMsgs` and consumed by the new `waitForMsg(testFn,timeout)` helper, which enables plans to await specific partner events with timeouts. When one agent's distance to a delivery zone exceeds its teammate's, `optionsGeneration()` invokes `proposeHandoff(parcelIds)`, which uses `chooseCorridorMeetpoint()`—itself built on `getCorridorId()`, BFS-clustered corridor segments, and `findAdjacentSpots()`—to select a rendezvous cell and ad-

jacent wait spot inside the corridor. A `handoff_request` RPC is sent; upon grant, a special `handoff_execute` intention is pushed. The new `HandoffExecute` plan handles both roles: the giver signals ready, travels to rendezvous (or proxy), drops parcels with `emitPutdown()`, and emits `handoff_done`; the receiver goes to its wait spot, signals ready, waits for `handoff_done` (via `waitForMsg`), then travels to the rendezvous, picks up with `emitPickup()`, and finally pushes a `go_deliver` intention. Timeouts (`startHandoffTimeout()`, `abortHandoff()`) and loop preemption in `IntentionRevision.loop()` ensure that only the handoff plan runs to completion and that the system recovers if messages are lost or one partner stalls. Opportunistic pickup/delivery and corridor-locking logic have been adapted to respect the handoff state, yielding a robust, end-to-end cooperative handoff mechanism.

Note that the entire handoff protocol—and the temporary disabling of corridor locks via `globalThis.disableCorridorLocks`—is only activated when both agents detect they occupy the same corridor segment (i.e. their `getCorridorId(x,y)` values coincide). In all other situations, the usual `corridorLocked` flags remain in force, and the standard mutual-exclusion locking protocol governs passage through every corridor. This design ensures that locks are only lifted for genuine in-corridor handoffs, while preserving collision avoidance and independent navigation elsewhere.

6 Validation and Testing

In order to evaluate the performance of our agent, we tested the code on various predefined maps with different obstacle configurations, observing its behavior in parcel pickup and delivery tasks and confirming that it achieved its goals without getting stuck or violating map constraints. Additionally, we participated in the game challenges to assess how our agent reacted to different environments and to the presence of multiple other agents. Although we did not achieve excellent results in the first challenge, we identified the limitations of our implementation and applied several improvements. Specifically, we enhanced the agent's exploration strategy by focusing on parcel-spawnable tiles rather than wandering around non-spawnable areas, introduced proper handling for parcels with negative rewards, and generalized the agent's behavior to work effectively on diverse maps instead of relying on specific cases. In the second challenge, our agent performed correctly; however, compared to other agents, its decision-making and execution speed remained insufficient to achieve a competitive ranking.

7 Conclusion

In conclusion, the development of our Deliveroo agent provided valuable insights into BDI architectures and planning strategies such as A* and PDDL integration. Through iterative testing and participation in challenges, we identified and addressed several weaknesses, resulting in an agent capable of performing parcel pickup and delivery tasks reliably across diverse maps. Although the agent did not achieve top rankings in competitive settings, the project enhanced our understanding of practical AI implementation, planning under constraints, and decision-making in dynamic environments. Future improvements could focus on optimizing the planning speed, integrating more advanced utility-based intention revision strategies, and enhancing adaptability to opponents' behaviors to further increase performance in competitive scenarios.