AUTONOMOUS SOFTWARE AGENTS

Project Report

257565 - Donald Gera

donald.gera@studenti.unitn.it

UNIVERSITÀ
DI TRENTO

Contents

1 Introduction

The *Deliveroo* game models a parcel delivery task on a grid map where agents seek to maximize score by collecting items and bringing them to designated delivery tiles. The environment is dynamic: new parcels are generated during execution, rewards are time-sensitive due to decay, and the presence of other agents can create interference on shared routes and targets. As a result, effective behavior is not only a matter of reaching a destination, but of continuously prioritizing actions under uncertainty, balancing exploration of the map with exploitation of high-value opportunities, and maintaining reliable navigation despite changing conditions.

1.1 Design perspective

We treat the agent as a repeated decision cycle that integrates sensing, reasoning, and acting. At each step, the agent updates an internal state from observations (parcels, other agents, and map constraints), evaluates a set of candidate objectives, commits to a plan, and revises that commitment when new information makes the current choice suboptimal or infeasible. This control loop is naturally supported by a *Belief–Desire–Intention (BDI)* structure: beliefs represent the current world model, desires encode the objectives that are worth pursuing (e.g., collect, deliver, explore, recover), and intentions capture the plan currently being executed. The explicit separation between these components allows the system to combine reactivity (responding to newly observed parcels or conflicts) with deliberation (following through on selected plans instead of oscillating among alternatives).

1.2 From single agent to cooperation

Development progressed in two stages. We first implemented a single-agent baseline focused on robust movement and decision-making, combining utility-driven target selection with path planning and recovery strategies for blocked or unreachable goals. We then extended the architecture to a decentralized multi-agent setting by adding communication and coordination mechanisms. Agents exchange messages to share intent, reduce overlap, and self-assign spatial responsibilities (e.g., via zone partitioning). In maps that include hallway-like layouts, where agents cannot bypass each other, we introduce dedicated handover mechanisms to prevent deadlocks and reduce mutual blocking. In these constrained areas, agents coordinate the transfer of parcels and responsibilities so that goal achievement remains feasible and progress continues even under strict movement constraints.

1.3 Goals of the project

The goals of the project are:

- design and implement autonomous agents for a dynamic parcel delivery environment with time-varying rewards;
- integrate perception updates, goal management, and committed planning within a BDI reasoning loop;
- implement decision-making based on a score-oriented utility model that balances exploration and exploitation;
- introduce decentralized coordination (intent sharing, spatial assignment, conflict avoidance, and role-based cooperation);
- evaluate the resulting system in terms of score, delivery efficiency, robustness, and behavioral stability.

2 System Design

We organise the agent as a persistent reasoning process that runs for the whole match and continuously couples perception with action. The adopted design follows a *Belief–Desire–Intention (BDI)* style: the agent maintains an explicit internal model of the world, derives a set of candidate objectives from that model, commits to one objective at a time, and executes a corresponding plan while monitoring whether it should be revised. This separation keeps responsibilities well-scoped in the code and makes the system easy to extend from a single agent to a coordinated pair by adding communication and shared decision constraints, without rewriting the entire control logic.

2.1 BDI Control Loop

Figure **??** depicts the control flow used by our agents. Percepts gathered from the environment are first incorporated into the belief state, which maintains the agent's current world model. From these beliefs, the agent derives a set of candidate options, applies filtering rules, and commits to an intention that captures the objective to pursue. The intention is then operationalized into an executable plan, whose actions are carried out in the environment. Crucially, this process is *iterative* rather than sequential: while executing a plan, the agent keeps sensing and updating its beliefs, and it can revise either the current intention or the remaining plan when the context changes. This allows the agent to remain responsive to events such as parcels being collected by others, new parcels appearing, or movement constraints emerging, without degenerating into unstable behaviour driven by constant target switching.

Algorithm 1 reports the control loop we follow. After each percept, the belief revision function updates the internal state. From the resulting beliefs and the current intention set, the agent generates candidate options and filters them to obtain the next intention(s). A plan π is then produced and executed step-by-step. During execution, the loop keeps checking three termination conditions: the plan becomes empty, the intention has been achieved, or the intention is no longer achievable. After every action, the agent incorporates a new percept and may (i) reconsider intentions, and (ii) replan if the remaining plan is no longer sound with respect to the updated beliefs. This yields a lightweight form of online planning: the agent does not continuously recompute everything, but it is able to revise decisions promptly when there is a concrete reason to do so.

Algorithm 1 Agent Control Loop

$B := B_0$;
$I := I_0$;
while true **do**
  get next percept p;
  $B := brf(B, p)$;
  $D := options(B, I)$;
  $I := filter(B, D, I)$;
  $\pi := plan(B, I)$;
  while not ($empty(\pi)$ **or** $succeeded(I, B)$ **or** $impossible(I, B)$) **do**
    $\alpha := hd(\pi)$;
    $execute(\alpha)$;
    $\pi := tail(\pi)$;
    get next percept p;
    $B := brf(B, p)$;
    if $reconsider(I, B)$ **then**
      $D := options(B, I)$;
      $I := filter(B, D, I)$;
    end if
    if not $sound(\pi, I, B)$ **then**
      $\pi := plan(B, I)$;
    end if
  end while
end while

2.2 Core entities and implementation units

- **Grid and tiles.** The world is a discrete 2D map whose cells are represented as tiles, with special subsets for *spawners* and *deliveries*. These tiles also define the neighborhood structure used by the pathfinder.
- **Belief snapshot.** At runtime the agent maintains an internal snapshot of the current state: its own position/score, the last perceived parcels, and the last perceived agents. When coordination is enabled, the same state also stores the partner id and (optionally) an assigned area.
- **Parcels.** Parcels are volatile entities indexed by id, position, reward and carrying status; they are refreshed at each sensing update and may disappear or become unavailable at any time.
- **Plans and actions.** The agent commits to an intention by activating a concrete plan. Plans generate the next low-level action (`move`, `pickup`, `putdown`) and monitor failures to decide when to replan or abort.
- **Messages and coordination state.** In multi-agent mode, structured messages are exchanged to discover a teammate, share information, and negotiate interaction constraints (including handover when narrow layouts prevent bypassing).

Main modules.
Rather than embedding all logic in a monolithic controller, the reasoning loop is realized as a set of modules, each responsible for a specific role:

- **World model:** `environment.js` builds the map and exposes relevant tile sets (spawner/delivery/normal); `tile.js` encapsulates per-cell attributes and utilities.
- **Belief maintenance:** `beliefs.js` stores the internal snapshot and updates parcels/agents from sensing callbacks; it also derives configuration-dependent constants (e.g., decay per movement step).
- **Navigation:** `pathfinding.js` runs A* on the grid neighborhood and translates path steps into movement actions.
- **Plan backbone:** `plan.js` provides shared mechanisms such as path caching, replanning, failure counters, collision checks, and a unified utility evaluation used by pickup/detour decisions.
- **Reactive plans:** `explore.js`, `pickup.js`, and `deliver.js` implement the core task-level plans used in single-agent mode.
- **Coordination layer:** `coordination.js` handles handshake, information sharing, and interaction policies when a partner is present.
- **Handover support:** `handover_detector.js` detects when handover is beneficial and selects a suitable handover tile; `handover.js` implements the role-specific plans for transferring parcels in non-bypassable hallways.
- **Local recovery:** `banlist.js` provides time-based bans to temporarily exclude problematic parcels/tiles and reduce oscillations.

3 Single - Agent

The initial phase of our development focused on building a single autonomous agent able to navigate and operate effectively across a wide variety of map configurations. Since the environment is fully online—parcels appear unpredictably, rewards decay, and other agents can invalidate a previously good choice—the agent must repeatedly reassess its next objective from the current beliefs. In our implementation, this corresponds to the logic in `pickCandidateIntention()` within `agent.js`: at each iteration the agent evaluates which high-level option is currently viable (deliver, pickup, or explore) and commits to it by instantiating the corresponding *plan*. Each option therefore becomes actionable only through the plan's internal logic (e.g., target refresh via `updateTarget()`) and through the pathfinding layer, which grounds the chosen target into a concrete sequence of movement steps. The priority structure is deliberately simple: if the agent is carrying parcels it proposes a delivery intention first; otherwise it selects a pickup intention only when a suitable parcel target exists; if no worthwhile parcel is available, it switches to exploration to increase coverage and improve future parcel discovery.

3.1 Exploration

Exploration is driven by a simple coverage policy: the agent keeps track of where it has already been and deliberately moves toward places it has visited less. Each tile accumulates a visit count, and the next destination is selected by minimizing this count, with heuristic distance used only to break ties and avoid unnecessarily long walks. Since parcels can originate exclusively from spawn tiles, the exploration objective is defined over that set, encouraging the agent to circulate through spawning regions to maximize the chance of spotting new parcels early. To avoid getting trapped in unproductive cycles (for instance, repeatedly committing to a target that turns out to be unreachable), exploration is coupled with a time-based ban mechanism: when a chosen target cannot be reached, its tile key is temporarily banned and the plan promptly switches to an alternative.

3.2 Parcel pickup via utility scoring

The pickup strategy is driven by a utility function that estimates how much score a candidate parcel can realistically contribute once travel time and decay are taken into account. Rather than favoring the largest reward in isolation, the agent compares parcels by combining (i) the reward available at the moment of observation, (ii) the number of movement steps required to reach the parcel and then reach a delivery tile, and (iii) the expected value lost while performing those steps. This time-dependent loss is summarized by a single parameter L, the *loss per movement step*, which models the average reward reduction caused by taking one additional step (i.e., the "cost" of time passing while moving).

Formally, let L be derived from the environment configuration (`PARCEL_DECADING_INTERVAL` and `MOVEMENT_DURATION`), let k be the number of parcels currently carried, and let R_{carried} be the total reward of the carried load. From the current position (x, y), each candidate parcel p is then evaluated by estimating its net gain after the expected travel loss, using path lengths returned by the planner:

$$U_{\text{pickup}}(p) = \big(r(p) + R_{\text{carried}}\big) - \big(d((x,y),p) + d(p, \text{deliver}(p))\big) \cdot L \cdot (k+1),$$

where $d(\cdot,\cdot)$ is the path length returned by A* and $\text{deliver}(p)$ is the closest delivery tile from the parcel position. The plan compares this against the utility of immediately delivering the current load:

$$U_{\text{deliver}} = R_{\text{carried}} - d((x,y), \text{deliver}(x,y)) \cdot L \cdot k.$$

A parcel is considered worth pursuing only if it improves over direct delivery, and the best such parcel becomes the current pickup target. To avoid inefficient behaviour, a parcel is discarded if another observed agent is significantly closer and the agent also relies on a lightweight *ban* mechanism that temporarily excludes problematic targets from consideration. A ban is triggered when the agent repeatedly fails to make progress toward a parcel (e.g., the path is consistently blocked by another agent), or when the parcel disappears / becomes unreachable after the intention has been selected. Once banned, the corresponding parcel id (or tile key, depending on the failure)

is ignored for a limited time horizon, allowing the intention selector to redirect the agent toward alternative opportunities instead of oscillating on the same target.

3.3 Delivery with opportunistic detours

When at least one parcel is carried, the delivery plan is activated. If the agent is already on a delivery tile it immediately performs `putdown`. Otherwise it computes a path to the closest delivery tile and follows it.

A useful refinement is *opportunistic detouring*: while moving toward delivery, the agent may still pick up an additional parcel if doing so increases overall utility compared to delivering immediately. This detour decision reuses the same utility comparison described above, ensuring that detours are taken only when they are expected to be beneficial under decay and travel cost. If the chosen delivery tile is unreachable (e.g., temporary obstruction), it is banned for a short horizon and an alternative delivery target is selected.

3.4 Failure handling and replanning

All plans inherit shared recovery mechanisms from the base `Plan` class. Paths are cached and executed step-by-step; if a path becomes stale (e.g., after replanning or external interference), it is cleared and recomputed. Movement also checks whether the next step is occupied: when blocked, the plan attempts local resolution by replanning; in coordination mode, partner blocking can trigger a dedicated collision-resolution protocol, while in non-partner cases the plan tries to find an alternative route. Finally, each plan tracks consecutive failures and aborts when a configurable threshold is exceeded, allowing the intention selector to switch to a different plan rather than repeatedly repeating a failing action.

4 Multi - Agent

In the two-agent setting, performance depends not only on each agent's individual decision-making, but also on how well the pair avoids interfering with itself. For this reason, we augment the single-agent control loop with a minimal coordination layer that allows agents to exchange a small amount of information and resolve conflicts locally. Rather than introducing a central controller, coordination is kept decentralized: each agent still plans autonomously, but uses shared signals to limit duplicated effort and to handle situations where mutual blocking would otherwise lead to wasted moves and stalled progress (e.g. hallway layout).

4.1 Partner discovery and state synchronization

Coordination starts with a brief partner - discovery phase. Each agent periodically broadcasts a `HANDSHAKE` message containing its identifier and current position; upon receiving the first valid request, the receiver records the sender as its partner and replies directly with a `HANDSHAKE_ACK`. Once acknowledgments are exchanged, the partner link is marked as confirmed and coordination features are activated.

After the handshake, the agents keep their belief models loosely aligned through lightweight information sharing. At fixed time intervals, each agent broadcasts (i) the set of currently available parcels it knows about and (ii) the most recent observations of other agents. These messages extend local perception beyond the limited sensing radius and reduce stale choices during option generation, especially when parcels spawn in distant regions of the map.

Finally, whenever an agent commits to a pickup intention, it explicitly announces the selected target (with its estimated utility). The teammate stores this intention and, if both agents converge to the same parcel, a simple yielding rule resolves the conflict so that only one agent continues the chase while the other immediately reconsiders alternative options. In addition to intention-level coordination, the same communication channel is reused for dedicated collision-handling and (when enabled) handover role negotiation.

4.2 Environment Partition

We reduce duplicated work by assigning each agent an *area of responsibility*. Rather than partitioning the whole map, our split is defined over *spawner tiles*: since parcels can only originate

there, controlling the spawner set is enough to steer both exploration and pickup decisions toward complementary regions.

When the partition is computed After the handshake confirms a teammate, the belief module computes a static partition of the current map's `spawnerTiles`. The partition is computed once (per map) and stored as `myArea`, so that subsequent decisions can be filtered without additional coordination overhead. The assignment is made deterministic: the agent with the smaller id takes the first cluster (Zone A), and the other agent takes the second cluster (Zone B). This guarantees that both agents agree on the split without needing extra messages.

Seed selection (farthest-pair heuristic) Let S be the set of spawner tiles, with $|S| = n$. We first choose two seed tiles that are as far apart as possible under Manhattan distance, to encourage a wide separation between the two zones. To keep the computation lightweight even on large maps, the code uses two modes controlled by a parameter `PARTITION_LIMIT`:

- If $n \leq$ `PARTITION_LIMIT`, we evaluate all pairs (s_i, s_j) and pick the pair with maximum Manhattan distance (exact farthest pair, $O(n^2)$).
- If $n >$ `PARTITION_LIMIT`, we use a linear heuristic: pick an arbitrary tile a, select s_A as the farthest tile from a, then select s_B as the farthest tile from s_A (approximate farthest pair, $O(n)$).

Balanced assignment with confidence ordering Once seeds (s_A, s_B) are selected, we assign every remaining spawner to one of the two clusters while keeping sizes balanced. The target sizes are:

$$|A| = \left\lceil \frac{n}{2} \right\rceil, \qquad |B| = n - |A|.$$

For each spawner $t \in S \setminus \{s_A, s_B\}$, we compute:

$$d_A(t) = d(t, s_A), \qquad d_B(t) = d(t, s_B),$$

and define a *confidence* score $\Delta(t) = |d_A(t) - d_B(t)|$. Intuitively, tiles with large Δ are *clearly* closer to one seed and should be assigned first. We therefore sort remaining tiles by: (i) decreasing $\Delta(t)$, then (ii) increasing $\min(d_A(t), d_B(t))$, and finally (iii) a deterministic coordinate tie-break (lexicographic on (x, y)) to guarantee reproducibility.

Assignment then proceeds greedily in that sorted order: each tile is pushed to its preferred side (the seed it is closer to), unless that side already reached its target size, in which case it is placed in the other cluster. This produces two similarly sized zones while remaining computationally cheap and stable across runs.

How the partition is used by plans The partition directly constrains single-agent routines when coordination is active:

- **Exploration:** the exploration plan selects its next target among spawner tiles in `myArea`, so that each agent patrols a different portion of the spawning space.
- **Pickup:** parcel candidates are filtered by zone whenever possible, preferring parcels whose coordinates fall inside `myArea`. This reduces direct competition between teammates and lowers the probability that both agents chase the same opportunities.

Overall, the partition acts as a simple but effective decentralization mechanism: agents keep the same BDI-style control loop, but their option set is biased toward disjoint regions, which improves coverage and reduces redundant motion without requiring a centralized controller.

4.3 Handover mode in constrained layouts

Some maps contain hallway-like structures where two agents cannot reliably bypass each other. In these cases, to keep progress stable, we switch to an explicit *handover mode* that splits the end-to-end task into two shorter routes and turns the interaction point into a controlled transfer.

Detecting when handover is worth enabling. After the handshake has confirmed a teammate, the coordination layer evaluates the static map using the `HandoverDetector` class. The detector returns a handover configuration only when it finds a convincing reason to prefer transfer over standard operation. In particular, we activate handover on **bottlenecked layouts**, where the routes between the spawn region and the delivery region necessarily funnel through the same narrow passage and agents cannot bypass each other. In this case, the detector searches for a *bottleneck* tile that lies on (almost) every feasible spawn→delivery route. Selecting this tile as the handover point turns an unavoidable meeting location into a controlled interaction: instead of repeatedly blocking the corridor, agents synchronize at a fixed transfer position and continue on complementary sides of the map.

Role assignment and synchronization. Once handover mode is initialized, agents assign complementary roles using a simple and deterministic rule: the agent with smaller Manhattan distance to the selected spawn becomes the *COLLECTOR*, and the other becomes the *COURIER*. The chosen role is sent directly to the teammate through a `HANDOVER_ROLE` message together with the handover coordinates, allowing both agents to instantiate the matching plan immediately.

Collector plan (spawn → handover). The collector executes a tight loop: it waits on the designated spawn until a parcel is available, picks it up, and then travels toward the handover tile. The transfer is performed with an explicit synchronization condition: if the collector reaches the handover position, it drops the parcel only when the courier is already adjacent, so the parcel is not left exposed for long. If the handover tile is temporarily occupied, the collector avoids planning directly onto an occupied goal and instead moves to a *staging* neighbor on the spawn-side, where it can wait without blocking the bottleneck passage.

Courier plan (handover → delivery). The courier remains parked on a *waiting* neighbor on the delivery-side of the handover point. When a parcel becomes available on the handover tile, the courier approaches and picks it up, then completes delivery to the delivery area. As with the collector, if the handover tile is occupied at the moment, the courier does not force a path to a blocked goal: it waits on the waiting position until the pickup becomes feasible.

5 Pathfinding

Efficient navigation is central to the agent, since movement time directly affects score due to parcel decay. Our navigation layer models the map as a 4-neighborhood graph and relies on a customized A* search that is lightweight enough to be called repeatedly inside an online control loop. The computed path is not executed all at once: plans consume it incrementally, validate each next step against the current (dynamic) state, and trigger replanning whenever the cached route becomes invalid.

5.1 Grid graph representation

As mentioned previously the environment is stored as a 2D array of `Tile` objects. Each tile encodes whether it is reachable and, when applicable, whether it has a special role (spawner or delivery). During initialization, the environment precomputes adjacency by linking each reachable cell to its reachable neighbors in the four cardinal directions. This yields a static neighbor graph that can be queried in constant time during search, while dynamic constraints (e.g., agents occupying cells) are handled at planning time and/or execution time.

5.2 A* search

Heuristic and node state We use Manhattan distance as heuristic:

$$h(n) = |x_n - x_g| + |y_n - y_g|.$$

Each search node stores its coordinates (x, y), the cost-to-come g, the heuristic h, the evaluation value $f = g + h$, and a parent pointer used to reconstruct the final path.

Algorithm 2 A* search used by the agent

```
initialize frontier with start node
bestInFrontier[start] ← start
explored ← ∅
while frontier not empty do
    pick n ∈ frontier with minimal (f, h)
    remove n from frontier
    if bestInFrontier[pos(n)] ≠ n then continue
    end if
    delete bestInFrontier[pos(n)]
    if n is goal then return ReconstructPath(n)
    end if
    add pos(n) to explored
    for all neighbors m of n do
        if Blocked(m) then continue
        end if
        if pos(m) in explored then continue
        end if
        g' ← g(n) + 1
        if m unseen or g' < g(m) then
            update m: g(m) ← g', h(m) ← Manhattan(m, goal), parent(m) ← n
            bestInFrontier[pos(m)] ← m
        end if
    end for
end while
return empty path
```

Returned path When the goal is reached, the path is reconstructed by following parent pointers back to the start and reversing the sequence. The returned list excludes the starting cell, so its length corresponds to the number of movement steps. If the frontier is exhausted without reaching the target, the algorithm returns an empty path.

5.3 From paths to actions: incremental execution and replanning

Paths are consumed incrementally by the active plan. Each plan caches the current target and its route, recomputing A* only when the target changes or when the cached path becomes invalid. At each decision step, the plan extracts the next tile on the path and converts the displacement into a primitive simulator action (left/right/up/down).
Because the world is dynamic, execution includes additional consistency checks:

- **Blocked-step handling:** if the next cell is occupied, the plan either requests coordination-based resolution (when the blocker is the teammate) or triggers local recovery (e.g., replanning and bans) when the blocker is an opponent. In handover mode, this check is intentionally relaxed because staging/waiting around the handover tile is handled explicitly by the handover plans.

This design keeps the planner fast and simple, while the surrounding execution logic ensures robustness under online changes and multi-agent interference.

5.4 PDDL planning

Design rationale A first possible approach would be to use the planner as a pathfinding tool – encoding tiles and directional predicates, then solving for `(at ?target)`. In practice, this adds little value: the online solver introduces network latency (per call), while the A* implementation already computes optimal paths synchronously. Using PDDL to replicate what A* does is redundant. For this reason, we decided to implement PDDL for what it is genuinely designed: reasoning about a *sequence of heterogeneous actions* – movement, pickup, and putdown – to achieve a high-level goal. Rather than hardcoding the task decomposition (first navigate to the parcel, then pick

up, then navigate to delivery, then put down), we let the solver determine *when* to pick up and *when* to put down as part of a single planning episode. The goal is not "be at tile t" but "parcel p ends up on delivery tile d", and the solver derives the full action sequence autonomously.

Domain formulation The PDDL domain models the environment as a set of tile objects with directional predicates (`up`, `down`, `left`, `right`) encoding adjacency. The agent's position is tracked via the fluent `(at ?tile)`, and parcel locations via `(parcel_at ?parcel ?tile)`. The domain defines six actions:

- **move-up/down/left/right**: Move between adjacent tiles, updating `(at ?tile)` accordingly.
- **pick-up**: Pick up a parcel at the current tile, asserting `(carrying ?parcel)` and removing `(parcel_at ?parcel ?tile)`.
- **put-down**: Deliver a carried parcel to the current tile, reversing the pickup effects.

Problem generation and bounded search For each parcel opportunity, the agent dynamically generates a PDDL problem instance. To keep planning tractable, we construct a *bounded subgraph* around the agent's position, the parcel location, and the target delivery tile:

$$\text{minX} = \max(0, \min(x_{\text{agent}}, x_{\text{parcel}}, x_{\text{delivery}}) - \text{padding})$$
$$\text{maxX} = \min(W - 1, \max(x_{\text{agent}}, x_{\text{parcel}}, x_{\text{delivery}}) + \text{padding})$$

with analogous bounds for the y dimension. Only tiles within this bounded region are included as objects, and directional predicates are generated only for neighbors that also lie within the bounds.

Trade-offs For a fast-paced game like Deliveroo, where parcels spawn frequently, decay over time, and multiple agents compete for them, PDDL has clear drawbacks. The online solver call is orders of magnitude slower than a synchronous A* lookup, so by the time a plan is returned the target parcel may already be gone or its value significantly diminished. Furthermore, once a PDDL plan is produced it is executed as a fixed action sequence: unlike the reactive A*-based plans, which re-evaluate the situation every tick and can replan or switch targets mid-execution, the PDDL plan commits to the full move–pickup–move–putdown sequence without revision. If the environment changes during execution (e.g. a competing agent picks up the parcel), the only recourse is to discard the entire plan and re-deliberate from scratch. For these reasons, A* remains the backbone of the agent's reactive behavior (Pickup, Deliver, Explore, and Handover plans all use it).

6 Evaluation

6.1 Experimental setup

Our experimental evaluation was carried out on the collection of maps from both challenge levels, covering a variety of layouts and difficulty conditions. For every map, we executed 5-minute sessions under the same simulator configuration to keep the comparison fair and reproducible. Within this fixed setup, we measured performance under the two planning variants available in our agent: the default A*-based navigation and the optional PDDL planning mode.

6.2 Results

Table 1: Single-agent scores (A* vs PDDL).

Level	A*	PDDL
25c1-1	1548	342
25c1-2	1195	525
25c1-3	1610	678
25c1-4	834	446
25c1-5	1181	388
25c1-6	2239	401
25c1-7	1144	194
25c1-8	502	367
25c1-9	2471	271

Table 2: Two-agent scores: per-agent (A* vs PDDL).

Level	A1 A*	A1 PDDL	A2 A*	A2 PDDL
25c2-1	1335	468	1195	375
25c2-2	962	450	896	329
25c2-3	820	280	834	211
25c2-4	1174	495	1111	408
25c2-5	947	502	1147	338
25c2-6	600	166	674	169
25c2-7	1341	253	1060	232
25c2-8	659	244	482	274
25c2-9	546	125	476	71
25c2 hall	2713	0	0	0

6.3 Discussion

Across the tested maps, the agent remains competitive both in single-agent and two-agent settings. In two-agent mode, explicit intention exchange and role separation help both instances contribute to the team score, while handover mode provides a practical way to avoid deadlocks in non-bypassable hallways. The comparison between A* and PDDL highlights a typical trade-off in dynamic environments: longer-horizon planning can be advantageous when plans are reusable and the environment is stable enough, but reactive local planning remains essential for fast recovery when the game state changes quickly.