UNIVERSITÀ
DI TRENTO

# Autonomous Software Agents

Master's Degree in Artificial Intelligence Systems

**Authors: Matteo Bando, Jonathan Fin**
**IDs and e-mails: 259068, matteo.bando@studenti.unitn.it; 256178, jonathan.fin@studenti.unitn.it**
**Academic year: 2024-2025**

---

## Contents



## 1. Introduction

Deliveroo.js is a two-dimensional, grid-based parcel-delivery game designed to offer an interactive environment where autonomous agents compete to achieve the highest score while dynamically refining their planning strategies.

**Map and Environment:** at the start of each match, agents are placed on a map represented by a grid: each cell is either walkable or blocked. Walkable cells may also serve as delivery zones or parcel spawning sites. On a parcel-spawning cell, packages appear at random intervals, each carrying a point value that, depending on the game's configuration, may decay over time.

**Parcel Handling:** to pick up a parcel, an agent must move onto its cell and execute a pick-up action; to deliver it, the agent transports it to a delivery zone and performs a drop-off. Agents can carry an unlimited number of parcels, but they only earn the points corresponding to each parcel's value at the moment of delivery, and any parcel whose timer reaches zero disappears immediately.

**Game Modes:** the game can be played in free-for-all mode, where every agent competes individually, or in team mode, where teams of agents

collaborate. The objective in all modes is to accumulate the highest total score on the map.

**Match Configuration:** at the beginning of every round, all clients receive a configuration file that defines the map matrix and sets all relevant parameters, such as the observation distance or the server clock, for that specific match.

# 2. Single Agent

## 2.1. Initial configuration

When a single agent is launched, it undergoes the following initialization steps to prepare for interaction with the Deliveroo simulation server:

1. **Client setup:** we instantiate `DeliverooClient`, a thin wrapper around `DeliverooApi` that centralizes connection parameters (host URL, master/slave tokens) and decouples the code from the external library. Upon construction, it registers callback hooks for all incoming events:
   - `onYou` (agent identity and initial position)
   - `onTile` / `onMap` (map tiles and full map load)
   - `onConfig` (server configuration parameters)
   - `onParcelsSensing` (visible parcels)
   - `onAgentsSensing` (visible other agents)
2. **Model instantiation:** the agent's internal knowledge is organized into five core stores:
   - `Me` holds the agent's identifier, current coordinates, and status.
   - `ParcelsStore` tracks known parcels (pick-up/drop-off points) on the map.
   - `MapStore` maintains the grid layout, tile types, computed distance matrix, and sparsity metrics.
   - `AgentStore` records sensed positions and timestamps of other agents.
   - `ServerConfig` caches global simulation parameters (clock interval, map dimensions, sensing radius, carrying capacity).
3. **Agent injection:** we create an `Agent` instance and wire it to the client and all five stores. Its constructor signature is:

   ```
   new Agent(client,
             me,
             parcels,
             mapStore,
             agentStore,
             serverConfig)
   ```

   This binds the agent's reasoning cycle (belief update, desire generation, intention filtering, action execution) to the shared models.
4. **Event-driven population:** as data arrives from the server, the corresponding stores are updated:
   - `onYou` sets the agent's unique ID and coordinates.
   - `onConfig` populates `ServerConfig` with the parameters.
   - `onMap` and `onTile` build the terrain graph in `MapStore` and trigger distance/sparsity pre-computations.
   - `onParcelsSensing` and `onAgentsSensing` refresh dynamic information about parcels and other agents.
5. **Readiness check:** before entering its main loop, the agent ensures that:
   - Its own `ID` has been received.
   - The map has been fully loaded (`mapStore.mapSize > 0`).
   - It is not currently executing a movement command (`!agent.isMoving`).

   Only when all conditions are satisfied does the agent proceed to its continuous "perceive–deliberate–act" cycle.

This systematic initialization ensures that, at time zero, the agent has a consistent view of both the static environment (map topology, server rules) and the dynamic elements (parcel requests, other agents), enabling robust decision-making from the very first simulation tick.

### Map

The game map is obtained mainly from the `onMap` event and sometimes from the `onTile` one, if there happens to be a change in the environment. After all the map is loaded into the memory, the Floyd-Warshall algorithm is performed: the agent now has a distance matrix available, which contains in each cell $i, j$ the distance (in tiles) from $i$ to $j$. With this method,

the agent has a very good heuristic for future calculations, often regarding the score.

### Connection config

The connection parameters for the Deliveroo server (host URL and authentication tokens) are loaded from a `.env` file by the `Config.js` module at startup:

- `Config.js` imports `dotenv`.
- If no `.env` file is found, an error is printed and the application continues without valid credentials.
- When one is loaded, `process.env` provides three properties:

```
const CONFIG = {
 host:       process.env.HOST,
 token:      process.env.TOKEN,
 tokenSlave: process.env.TOKEN_SLAVE
};
```

- The `CONFIG` object is then exported and consumed by the client, which opens the WebSocket to `host` using `token` (master) or `tokenSlave` (slave).

Example of a `.env` file:

```
HOST='YOUR_URL_HERE'
TOKEN='YOUR_TOKEN_HERE'
TOKEN_SLAVE='YOUR_TOKEN_HERE'
```

## Launch Command

Once you have configured the `.env` file , start the script by running:

```
npm run start-single
```

## 2.2. BDI

### Parcels sensing

The agent has a limited vision, so we can sense parcels (and later agents) only up to a certain distance, based on the game configuration.
Sensed parcels are added to the memory and if they exist, they are updated. They are identified by a unique `id`, given by the server, and other standard attributes are its coordinates, if it is carried by someone and its reward. When the object is generated, another attribute is added, which is the distance from the nearest base, necessary for a precise theorical score calculation.

### Parcels revision

Parcels also have an attribute for storing the timestamp at which the agent last saw them. This is used in the *belief revision* to calculate an existence probability, based on the time passed and the number of agents present in the map: the higher the number of agents, the lower the probability of the parcel still being on the map, not picked up by someone. The score of the parcels are also calculated if the agent does not sense them anymore, subtracting a value based on the decaying factor from the server configuration.

### Agent sensing

Opponent agents are handled in the same way as parcels, being added to memory or updated when sensed. They also have an attribute that represents the direction they are moving in.

### Desires and Intentions

*Desire generation* is one of the core parts of the agent.
It is represented as an array of desires (or possible intentions). Three main desires exists, which consist in the following:

- **Pickup:** for every parcel in the memory a score is calculated, taking into consideration the distance to reach that parcel and the distance to reach the nearest base from that parcel. The score calculation is shown in equation 1, with $a$ being the agent, $p$ the parcel, $b$ the nearest base with respect to $p$, and *penalty* the decaying factor of the parcels score (e.g. 1 point each second). A visual representation is also available in figure 1, where the decay in score for each parcel is considered to be 1 point for each move of the agent.
- **Deposit:** if the agent is carrying at least one parcel, a desire to go deposit the carried parcels is added to the list with its score, calculated similar to the pickup one.
- **Explore:** the agent always has a desire to explore but its score is very low. This ensures that the explore action is executed only if there's not other possibility.

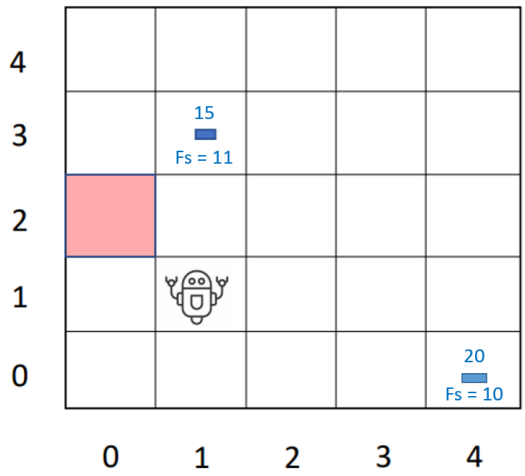


Figure 1: Pickup strategy

The number on top of the parcels is the current score, while the *Fs* value is the final score. The red tile is the base. The agent will pickup the nearest parcel, even though the other one has a higher score currently

$$\begin{aligned} score = (a.carried\_score + p.reward) \\ -((distance(a,p) + distance(p,b)) \\ *(a.carried\_amount + 1) * penalty) \end{aligned} \quad (1)$$

The desires list is then sorted by score, and the one with the highest score becomes the current intention.
All the intentions are immediately transformed into an action, preceded by a planning phase, except for the `pickup` intention. If the agent detects that another agent is going to get to the parcels faster them itself, it discard that intention and picks the second best one, perfoming again the checks if it is another `pickup` intention. The pseudo-code for the *agent-check* is the following:

```
if a senses another agent:
    if agent is closer than a:
        if agent is 1 tile away from
            parcel OR going towards the
            parcel:
             discard parcel and pick next
                 intention
```

Listing 1: Opponent agents checks

If the opponent agent is next to the parcel it is assumed to pick up that parcel. The *"going towards the parcel"* part is computed by analyzing the direction in which the agent is going, with respect to the parcel location.

## 2.3. Actions

The agent moves continuously in the map. Each path is calculated before acting, in a *planning-like* phase, with the $A^*$ algorithm.

### Pickup

The agent performs the `pickup` action by going to the location of the parcel and then picking it up.

### Deposit

The deposit intention is fulfilled by going to the nearest available home and deposit all of the carried parcels, and removing them from the memory.

### Explore

If the selected intention is to `explore`, then the agent selected a reachable random spawn tile and goes to that location. If the map is *sparse*, it stays on the same cell from a small period of time, waiting for a parcel to spawn. The sparseness of the map is calculated at the beginning, based on a proportion between spawn tiles and total tiles in the map, and also spawn tiles and the maximum number of parcels present in the map at the same time.

### Opponents handling

All the movements are preceded by check, that verifies if the path to the agent's destination is clear, that is if no other agents are blocking that path. If the agent sees another agent in the next cell along its path, it waits a configurable amount of time (which can also be zero seconds). If the path is still obstructed afterwards, it calculates a new path to its destination, excluding the tile where the opponent agent is standing.

## 2.4. Results in the Challenge 1

We placed $7^{th}$ out of 20 teams. During the live matches, we noticed that every time we issued a movement command: our agent incurred a penalty due to sending a new command before the server had applied the previous one [1]. To solve this, we introduced a boolean flag

[1] During offline testing we did not encounter these synchronization issues.

`isMoving` in our `Agent` class, which is set to `true` immediately before sending any movement command and reset to `false` once the move completes:

```
// Before moving:
this.isMoving = true;

await moveAndWait(this.client, this.me,
    dir);

// After move confirmation:
this.isMoving = false;
```

Listing 2: Setting `isMoving` before and after moves

Thanks to this change, in *Challenge 2* both agents no longer incurred movement penalties.

# 3. Multi Agent

## 3.1. Launch

The multi-agent system is handled by launching the two agents in the `mainMulti.js` file simultaneously. The agent have different `DeliverooClient` and `Me` objects but the same objects for all the memory (`ParcelStore`, `MapStore`, ...).

### Launch Command

Once you have configured the `.env` file, start the script by running:

```
npm run start-multi
```

## 3.2. Coordination

### Shared memory

By passing the same memory location to all the agents, the whole memory is shared between them and they can write and read in the same location, without the need to exchange messages.

### Map division

The map is divided equally between the agents, so they cover only their part and increase their efficiency. The division affects only the spawn tiles, so when performing an *Explore* action, the agent only goes to one of the tiles that are assigned to it.

To split the map, the k-means algorithm is used: it is an unsupervised clustering algorithm that tries to make $k$ clusters (in this case 2) of tiles to assign one of them to each agent. If the algorithm cannot find a good split, it tries again a few times. If no solution is found, each agent can navigate the entire map for its explorations.

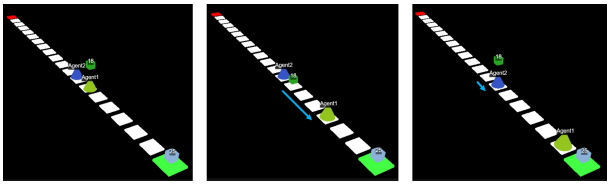

Figure 2: Parcel exchange

The *Agent1* detects its teammate, drops the parcels and goes away. The *Agent2* goes to pickup the dropped parcels and will then deliver them to the base

### Pickup parcels

Each agent, in the `generateDesire` step, calculates the score for itself and for its own teammate. A desire is added to the list of possible intentions if its own score is higher than its teammate's, then the `filterIntentions` step is performed as usual. In this way, no parcel is contended by both agents simultaneously, reducing redundant actions and avoiding collisions due to conflicting goals.

### Parcels exchange

Sometimes it is necessary that agents collaborate to drop off parcels in a base that is reachable by only one of the agents: for instance, if there is a corridor of width equal to 1 tile and the agents can only move in one direction, up or down.
If an agent is carrying some parcels and detects its teammate in a neighbor tile of its current location, the agent which is further from the base drops the parcels and goes away from that location, freeing the tile. It communicates his drop action to the other agent within the shared memory; the other agent can now consider picking up those parcels, always calculating the potential score of its action.
A simple representation of this sequence of actions can be visualized in the figure 2.
Sometimes an agent gets stuck because it cannot move in any direction, because it is blocked by his teammate. If this happens, it tells the other agent to move (through shared memory);

the other agent pushes an intention with highest priority and moves away a few tiles, so the first agent has space to move and exchange the parcels correctly.

### 3.3. Results in the Challenge 2

Althought we placed 8$^{th}$ among 14 teams, we further refined our code introducing the following new features:

- **Map Partitioning via k-Means:** On the first game loop, we now optionally split the map between agents using k-means (configurable with `USE_MAP_DIVISION`, `MAX_TRIES_KMEANS`, etc.). This ensures that each agent is responsible for its own sector, reducing overlap and improving parcel coverage.
- **Dynamic Base Reallocation and Path Fallback:** We added a resilient `getBasePath(target)` routine that:
  - Attempts to compute a path to the given base tile.
  - After a configurable number of failed tries (`BASE_TRIES`), temporarily removes the blocked base tile from the map, waits `BASE_REMOVAL_TIME`, then restores it.
  - Recomputes the nearest available base via `mapStore.nearestBase(this.me)` and updates `currentNearestBase`.
  - Retries pathfinding up to `BASE_SWITCH_MAX_TRIES` before giving up.

## 4. PDDL

After building the `single` and `multi-Agents` another agent was created to assert the capabilities of the `PDDL` domain.
The new agent is almost entirely made up of `PDDL` code, and run on an online solver. It implicitly uses the `BDI` architecture, still sensing the environment, updating its belief, generating the desires to pick up and deposit parcels and generating a plan.

### 4.1. Problem

The automatically generated `deliveroo_problem.pddl` file encapsulates one concrete snapshot of the game world that the planner must reason about. It follows the canonical PDDL layout:

1. **(:objects)** declares every logical object that currently exists:
   - the single agent (e.g., *agent_42*);
   - one symbol per visible parcel (e.g., *p_17*);
   - one symbol per known base (e.g., *base_1_9*);
   - every walkable map tile (e.g., *t_5_4*).
2. **(:init)** lists the ground facts that are *true right now*:
   - *(at agent_42 t_4_4)* — the agent's current position;
   - *(parcel-at p_17 t_5_4)* — parcel #17 is waiting on tile $(5, 4)$;
   - *(base-at base_1_9 t_1_9)* — a base occupies tile $(1, 9)$;
   - *(adjacent t_0_2 t_1_2))* facts describing local connectivity.
3. `(:goal)` a conjunctive goal that requires every currently known parcel to be delivered.

All problem files are stored in the `PDDL` folder and are produced on the fly by `pddlTemplatesjs`.

### 4.2. Domain

Unlike the problem file, the domain file is **static**: it describes the immutable physics and rules of the **Deliveroo world**.

**Types** We have defined the following types: `agent`, `parcel`, `base`, and `tile`.

**Predicates**

- **(at ?a ?t)** agent **a** is located on tile **t**.
- **(parcel-at ?p ?t)** parcel **p** is lying on tile **t**.
- **(carrying ?a ?p)** agent **a** is currently holding parcel **p**.
- **(base-at ?b ?t)** base **b** occupies tile **t**.
- **(delivered ?p)** parcel **p** has been delivered and it is no longer in play.
- **(adjacent ?t1 ?t2)** tiles **t1** and **t2** share an edge (undirected).

**Actions** Below is the definition of the three actions that we have defined

- **move( ?a, ?from, ?to )**
  - **Preconditions:**
    - (at ?a ?from)
    - (adjacent ?from ?to)
  - **Effects:**

  - delete (at ?a ?from)
  - add (at ?a ?to)
- **Intuition:** The agent leaves the source tile and appears on the destination tile.

- **pickup( ?a, ?p, ?t )**
  - **Preconditions:**
    - (at ?a ?t)
    - (parcel-at ?p ?t)
  - **Effects:**
    - delete (parcel-at ?p ?t)
    - add (carrying ?a ?p)
  - **Intuition:** The parcel is removed from the ground and held by the agent.
- **deposit( ?a, ?p, ?b, ?t )**
  - **Preconditions:**
    - (at ?a ?t)
    - (base-at ?b ?t)
    - (carrying ?a ?p)
  - **Effects:**
    - delete (carrying ?a ?p)
    - add (delivered ?p)
  - **Intuition:** The parcel is dropped and marked as delivered.

### 4.3. Limitations

Relying on an external PDDL solver brings two main challenges: network and solver latency, and world drift caused by the evolving game state. In the following, we first describe these issues in detail and then introduce our DEAR (Detect → Execution failure → Abort → Replan) approach as a remedy

#### Core Challenges

The system relies on an external, online PDDL solver accessed via asynchronous API calls. When we request a plan:

- **Network and solver latency:** we must wait for the remote solver to compute and return a plan.
- **World drift:** during this waiting period, the game world continues to evolve (e.g., other agents may pick up parcels, tiles may become blocked), so the returned plan may refer to parcels or states that no longer exist.

*Example:* at time $t_0$ the agent asks for a plan to pick up parcel *p_1* at *tile (5,4)*. Due to latency, the solver's plan arrives at $t_2$, but in the meanwhile, parcel *p_1* has already been collected by another agent, invalidating the plan.

#### Solution: DEAR

In our implementation, if the agent encounters an execution error,such as being unable to move because another agent now occupies the target tile (a situation not reflected in the previously generated plan), it immediately stops executing the current plan. Using the most recent world model stored in `MapStore`, `ParcelsStore`, and `AgentStore`, we then invoke `getPlan()` again to generate a new, up-to date plan.
This approach, which we call *DEAR*: **Detect → Execution failure → Abort → Replan** guarantees that the agent never proceeds on outdated assumptions, providing robustness at the cost of additional solver calls.

## 5. Documentation

#### Documentation Generation

**Using JSDoc** For automatic generation of our API documentation, we use **JSDoc**, a documentation generator for JavaScript that parses specially formatted comments within the source code. JSDoc recognizes standard tags (e.g., `@param`, `@returns`, `@description`, etc.) and produces a set of navigable HTML pages, making it easy to explore functions, classes, and their parameters. By using JSDoc, each function and class in our codebase is documented in a clear, up-to-date manner, directly from the source comments, minimizing the risk of outdated documentation.

**Online Access** The generated documentation is also accessible online at: https://bandomatteo.github.io/Deliveroo/

**Local Generation** To generate the documentation locally, simply run:

```
npm run generate-docs
```

## 6. Version Control and Project Management

- **GitHub Repository:** All code is hosted at: https://github.com/bandomatteo/Deliveroo, with a `main` branch for stable releases and `dev` branch for ongoing development.
- **Pull Requests** Significant modifications are submitted via pull requests.

- **GitHub Issues:** We track bugs and feature requests through issues labeled `bug`, `enhancement`, and `documentation`, organizing milestones for each release.
- **GitHub Projects:** We employ a Kanban board with `To do`, `In progress`, and `Done` columns to plan and monitor tasks in an Agile workflow.

## 7. Future Works

Building on our DEAR framework, we can further enhance the system's robustness, performance, and scalability:

**Hybrid approach** Use the PDDL solver only for a small part of the code, using more plain javascript and enabling to have more control over the agent. In this way the agent can be more complicated without changing much of the domain representation.

**Embed a local PDDL solver** Replace the remote API calls with an embedded planner running in-process. This removes network-induced delays and world-drift inconsistencies, guarantees data coherence, and enables tighter integration between the planner and the world model.

We think that adopting this change will dramatically reduce latency, eradicate API-related synchronization errors, and lay the foundation for a more resilient, high-throughput multi-agent planning platform.