**Universitat Rovira i Virgili**

Department of Computer Engineering and Mathematics (DEIM)

# Minecraft Assignment

Multi-Agent System in Minecraft (mcpi)

*Comprehensive Technical Specification*

**Course:** TAP

**Year:** 2025-2026

# Contents

**1 Introduction** **2**

**2 General Objective** **2**

**3 System Requirements** **3**

**4 Agent Behavior** **4**

4.1 ExplorerBot . . . . . 4

4.2 MinerBot . . . . . 4

4.3 BuilderBot . . . . . 5

4.4 State Management . . . . . 5

**5 Coordination and Synchronization** **6**

**6 Chat Commands** **7**

**7 Technical and Quality Requirements** **9**

**8 Completion Criteria and Demo Protocol** **9**

**9 Optional Extensions** **9**

**10 Submission** **10**

**11 Minecraft Server Configuration** **12**

# 1 Introduction

This document specifies the design and implementation of a **multi-agent system** in Python operating within the Minecraft environment using the `mcpi` API. The objective is to develop an intelligent and cooperative ecosystem where agents autonomously perform exploration, mining, and construction tasks while maintaining real-time coordination, synchronization, and communication.

The assignment emphasizes **software architecture and advanced programming paradigms**—functional, reflective, and object-oriented—prioritizing design quality over graphical complexity. The system must demonstrate the integration of asynchronous programming, modularity, structured logging, and extensibility.

Key aspects to demonstrate include:

- **Functional Programming:** Use of decorators and higher-order functions (`map`, `filter`, `reduce`) to transform and summarize data, such as terrain maps or resource statistics.
- **Reflective Programming:** Automatic discovery and registration of agents and strategies at runtime without manual imports, leveraging Python's reflection capabilities.
- **Object-Oriented Design Patterns:** Proper application of design patterns (e.g., Strategy, Observer, Factory) to improve modularity, flexibility, and maintainability.
- **Engineering Quality:** Compliance with PEP8, comprehensive documentation, and integration with GitHub-based CI/CD pipelines (GitHub Actions, Codecov).

# 2 General Objective

The goal is to create an autonomous ecosystem of agents that cooperate within Minecraft to build a functional base. The system must include the following agents:

- **ExplorerBot:** Analyzes the terrain and identifies optimal building zones.
- **MinerBot:** Extracts or simulates materials required for construction using different mining strategies.
- **BuilderBot:** Constructs the structure using materials provided by the MinerBot, following the plans derived from ExplorerBot's terrain analysis.

The system is considered complete when all three agents cooperate seamlessly and the BuilderBot successfully constructs a functional shelter using resources gathered by the MinerBot based on ExplorerBot's data. Multiple instances of each agent may coexist if synchronization and consistency are guaranteed.

# 3 System Requirements

All agents must inherit from a common `BaseAgent` class and implement the perception–decision–action cycle via the methods `perceive()`, `decide()`, and `act()`.

Agents must be dynamically registered using reflection—manual imports are not permitted. The system should scan directories (e.g., `agents/`, `strategies/`) and automatically import valid modules.

Concurrent execution must be supported through either `asyncio` or multithreading. Inter-agent communication must use structured JSON messages logged consistently on both sender and receiver sides.

## Message Contract and Validation

All inter-agent communication must follow a standardized JSON schema and be validated before processing.

Listing 1: Standardized JSON message format

```
{
  "type": "inventory.v1",
  "source": "MinerBot",
  "target": "BuilderBot",
  "timestamp": "2025-10-21T15:30:00Z",
  "payload": {
    "iron": 12,
    "wood": 24,
    "stone": 8
  },
  "status": "SUCCESS",
  "context": {
    "task_id": "MNR-042",
    "state": "RUNNING"
  }
}
```

Each field must be explicitly validated:

- `type` – message category (e.g., `map.v1`, `inventory.v1`).
- `source`, `target` – agent identifiers.
- `timestamp` – ISO 8601 UTC format.
- `payload` – structured data relevant to the message.
- `status` – message processing outcome.
- `context` – optional metadata (task, state, or correlation ID).

All messages must be validated through a schema or custom validation function before processing.

# 4 Agent Behavior

This section details the individual responsibilities, behaviors, and interactions of the three core agents that compose the system.

## 4.1 ExplorerBot

ExplorerBot surveys the surrounding terrain to identify suitable and stable regions for construction. Using the `getHeight(x, z)` function, it scans defined areas and detects zones with minimal elevation variance that can serve as foundations for BuilderBot's structures.

The user initiates exploration through in-game chat commands, specifying coordinates and optional range parameters. If new coordinates are received while exploration is active, ExplorerBot must confirm whether to interrupt the current process or queue the new request.

During execution, ExplorerBot periodically publishes `map.v1` messages containing structured terrain data, including elevation maps, identified flat regions, and potential obstacles. These messages are consumed by BuilderBot to plan construction. The bot must respond appropriately to control commands (`pause`, `resume`, `stop`) to ensure exploration can be safely suspended, resumed, or terminated while preserving its context.

## 4.2 MinerBot

MinerBot manages the extraction and collection of materials required by BuilderBot. It supports multiple mining strategies—at least two must be implemented:

- **Vertical Search:** Drills downward through layers to extract resources at increasing depths.
- **Grid Search:** Explores a cubic region following a structured grid pattern for uniform coverage.
- **Vein Search:** Detects clusters of identical materials (veins) and recursively mines adjacent blocks to maximize yield.

Upon receiving a Bill of Materials (BOM) from BuilderBot via `materials.requirements.v1`, MinerBot validates its inventory and begins mining operations until the requirements are fulfilled. Progress updates are periodically published through `inventory.v1`. Control commands (`pause`, `resume`, `stop`) must be handled safely, maintaining context such as position, collected materials, and current strategy.

## 4.3 BuilderBot

BuilderBot constructs structures using the materials supplied by MinerBot and the terrain data produced by ExplorerBot. It analyzes terrain maps, generates a Bill of Materials (BOM), and publishes it as a `materials.requirements.v1` message. If resources are insufficient, it suspends activity until materials become available.

When construction begins, BuilderBot places blocks layer by layer while logging coordinates, timestamps, and material types. It supports the same control commands for pausing, resuming, or stopping construction safely. If a building plan changes, BuilderBot recalculates the BOM and republishes updated requirements, ensuring synchronization across dependent agents.

## 4.4 State Management

All agents (`ExplorerBot`, `MinerBot`, `BuilderBot`) adhere to a unified control and state management model ensuring consistent and safe coordination.

### State Model

To ensure consistency and fault-tolerance across all agents, the system adopts a unified finite state machine (FSM) that governs each agent's lifecycle and transitions.

- `IDLE`: Waiting for a command.
- `RUNNING`: Actively executing its task.
- `PAUSED`: Temporarily halted, with full context preserved.
- `WAITING`: Blocked, awaiting data or resources.
- `STOPPED`: Terminated safely with state and data persisted.
- `ERROR`: An unrecoverable issue occurred; the agent halts safely and notifies dependents.

### Control Messages

Agents must implement handlers for the following control messages:

- `pause` – Temporarily halts execution and transitions to `PAUSED`.
- `resume` – Restores execution from the last checkpoint and transitions to `RUNNING`.
- `stop` – Safely terminates operations, ensuring all data and logs are stored before transitioning to `STOPPED`.
- `update` – Dynamically reconfigures parameters (e.g., coordinates, strategy, or build plan). Agents must confirm updates via acknowledgment messages.

**Consistency and Synchronization**

- Transitions between states must be atomic and logged with timestamp, agent ID, and metadata.
- On entering `STOPPED` or `ERROR`, all locks (e.g., mining regions) must be released.
- Agents must notify dependents of state changes to avoid deadlocks or inconsistency.
- Checkpoints must be serialized (e.g., JSON or pickle) for recovery after interruptions.
- Logs must include event type, timestamp, previous and next state, and transition reason.

This unified state management ensures coherent coordination, fault tolerance, and reliable recovery across distributed agents.

# 5 Coordination and Synchronization

Agents communicate and coordinate their actions through structured, asynchronous message exchange. All communication follows the standardized JSON schema defined in Section 3, ensuring consistency, validation, and traceability across all interactions.

## Communication Flow

The logical communication hierarchy is as follows:

1. **ExplorerBot → BuilderBot:** publishes `map.v1` messages with terrain data.
2. **BuilderBot → MinerBot:** sends `materials.requirements.v1` messages with material demands.
3. **MinerBot → BuilderBot:** replies with `inventory.v1` progress or completion reports.
4. **BuilderBot → All:** optionally broadcasts `build.v1` updates to synchronize global state.

## Communication Channel and Delivery

The specific communication mechanism between agents is intentionally flexible. Developers may choose any suitable approach—such as asynchronous queues, event buses, or message-passing abstractions—provided that it guarantees the following properties:

- Asynchronous and non-blocking message exchange.

- Reliable transmission with retry and timeout mechanisms.
- Validation of message structure and integrity before processing.
- Persistent logging of all sent and received messages, including context metadata.

The chosen mechanism must ensure that all messages conform to the standardized JSON schema and that the communication flow between agents remains deterministic and traceable.

### Synchronization and Locking

To avoid conflicts during concurrent execution:

- MinerBots must lock unique spatial regions (identified by `(x, z)` sectors) while mining.
- Locks must be automatically released when entering `STOPPED` or `ERROR` states.
- State changes must be acknowledged between dependent agents to ensure consistency.
- Communication channels must include timeout and retry mechanisms to handle transient failures.

### Integration with Chat Commands

User-issued chat commands (e.g., `/miner start`, `/builder pause`) are internally converted into control messages (`command.*.v1`) following the same schema as inter-agent communication. This unified interface ensures consistent behavior between user interactions and autonomous coordination among agents.

## 6 Chat Commands

This section summarizes the available in-game chat commands to interact with and control each agent.

### Common Commands

```
/agent help
/agent status
/agent stop
/agent pause
/agent resume
```

## ExplorerBot Commands

```
/explorer start x=<int> z=<int> [range=<int>]
/explorer stop
/explorer set range <int>
/explorer status
```

## MinerBot Commands

```
/miner start [x=<int> z=<int> y=<int>]
/miner set strategy <vertical|grid|vein>
/miner fulfill
/miner pause
/miner resume
/miner status
```

## BuilderBot Commands

```
/builder plan list
/builder plan set <template> [params]
/builder bom
/builder build
/builder pause
/builder resume
```

## Workflow Commands

```
/workflow run [x=<int> z=<int>] [range=<int>] [template=<name>]
              [miner.strategy=<vertical|grid|vein>]
              [miner.x=<int> miner.y=<int> miner.z=<int>]
```

The `/workflow run` command executes the full coordinated workflow:

1. Starts **ExplorerBot** to analyze the terrain and publish `map.v1`.
2. Activates **BuilderBot** to compute the Bill of Materials (BOM) and publish `materials.requireme`
3. Invokes **MinerBot** to collect or simulate materials and send `inventory.v1` updates.
4. Once materials are available, **BuilderBot** continues with construction and emits `build.v1` progress.

All parameters are optional and can be omitted. If not specified, each agent uses its default configuration (e.g., predefined exploration area, default building template, and standard mining strategy). Every stage of the workflow and all messages must be logged for traceability.

# 7 Technical and Quality Requirements

- PEP8 compliance and structured logging (`INFO`, `DEBUG`, `ERROR`).
- Clear docstrings for all classes and methods.
- Automated testing with CI/CD integration (GitHub Actions, Codecov), ensuring both unit and integration tests validate synchronization mechanisms.
- All tests must be executable via a single command (e.g., `pytest`).

# 8 Completion Criteria and Demo Protocol

The project is considered complete when the following are demonstrated end-to-end:

- **Multi-material workflow:** BuilderBot constructs at least two distinct structure templates using different material mixes provided by MinerBot.
- **Full coordination loop:** ExplorerBot → BuilderBot → MinerBot → BuilderBot, with validated JSON messages logged at send/receive points.
- **Lifecycle control:** Live demonstration of `pause`, `resume`, and `stop` across all agents, with consistent recovery from checkpoints.
- **Synchronization correctness:** No region overlap during mining (locks), no deadlocks on WAITING, and deterministic replay from logs.
- **Traceability:** Availability of message traces and structured logs that allow reconstructing the full execution.

During the demo, students must execute a scripted scenario and provide the logs/artifacts produced.

# 9 Optional Extensions

Students are encouraged to extend the base system by adding advanced or creative functionalities that demonstrate deeper understanding and technical mastery. These extensions are not mandatory but can significantly improve the overall assessment if well implemented and documented.

- **Remote Control (Pyro):** Use the Pyro library to enable a remote client to interact with the server.

- **Natural Language Interface:** Integrate a simple natural language processing (NLP) layer to translate user-friendly commands such as "Build a small wooden hut" into structured task messages understood by the agents.
- **Adaptive Strategies:** Allow agents to dynamically change their internal strategies (e.g., mining method or building pattern) based on environmental conditions, performance metrics, or available resources.

All optional extensions must comply with the same standards of validation, logging, and modularity defined in the core specification.

# 10 Submission

- Submit a ZIP archive via Moodle containing:
  - The **complete project source code**, organized and documented.
  - A **fully reproducible setup guide**, including environment configuration, dependencies, and execution instructions.
  - A **comprehensive technical report** written in academic style, which must include:
    - A detailed **system architecture explanation**, including diagrams.
    - Complete documentation of all **APIs, JSON message contracts, and communication flows** between agents.
    - A thorough **justification of all design and architectural decisions**, explicitly explaining:
      - **Reflective Programming:** Describe how Python's reflection mechanisms were used for dynamic agent or strategy discovery, module registration, or runtime introspection. Justify why this approach improves flexibility, extensibility, and decoupling.
      - **Functional Programming:** Explain where and how higher-order functions, decorators, or functional transformations were applied (e.g., for data aggregation, logging, or filtering) and justify their benefits for modularity and reproducibility.
      - **Object-Oriented Design Patterns:** Identify all applied design patterns (e.g., `Strategy`, `Observer`, `Factory`, `Singleton`), explain their purpose, and justify their role in improving cohesion, scalability, and maintainability.
    - A clear discussion of the chosen **concurrency model** (`asyncio` or multithreading), including reasoning, synchronization mechanisms, and state recovery procedures (`pause`/`resume`/`stop`).

    - A detailed description of the **testing strategy**, distinguishing between unit, integration, and synchronization tests, and an explanation of the CI/CD configuration used for validation (GitHub Actions or Codecov).
    - An explicit reflection on **modularity, extensibility, and testability** of the final system.

- Share a **private GitHub repository** with your laboratory instructor, including the CI/CD configuration and automated test results.

# 11 Minecraft Server Configuration

The server that will be used for the practice is available directly from the Adventures in Minecraft GitHub. `https://github.com/AdventuresInMinecraft/AdventuresInMinecraft-Linux` Please note that there are different versions depending on the operating system we want to use (Windows, Mac, and Linux).

Regardless of the OS being used, the project will always have the following structure:


As can be seen, there is a file called `StartServer`, which will have the extension `.bat`, `.sh`, or `.command` for Windows, Linux, and Mac, respectively. If this file is executed, the server will start running with the default memory requirements (1024 MB).

If this memory is not enough to perform the necessary tasks, it can be modified directly in the `start.{bat|sh|command}` file, located in the `/Server` directory. By default, the following command is present:

```
1 java -Xmx1024M -jar -DIReallyKnowWhatIAmDoingISwear craftbukkit.jar
```

The previous command can be modified to increase the memory available for the server. In the example below, the memory is increased to 1 GB.

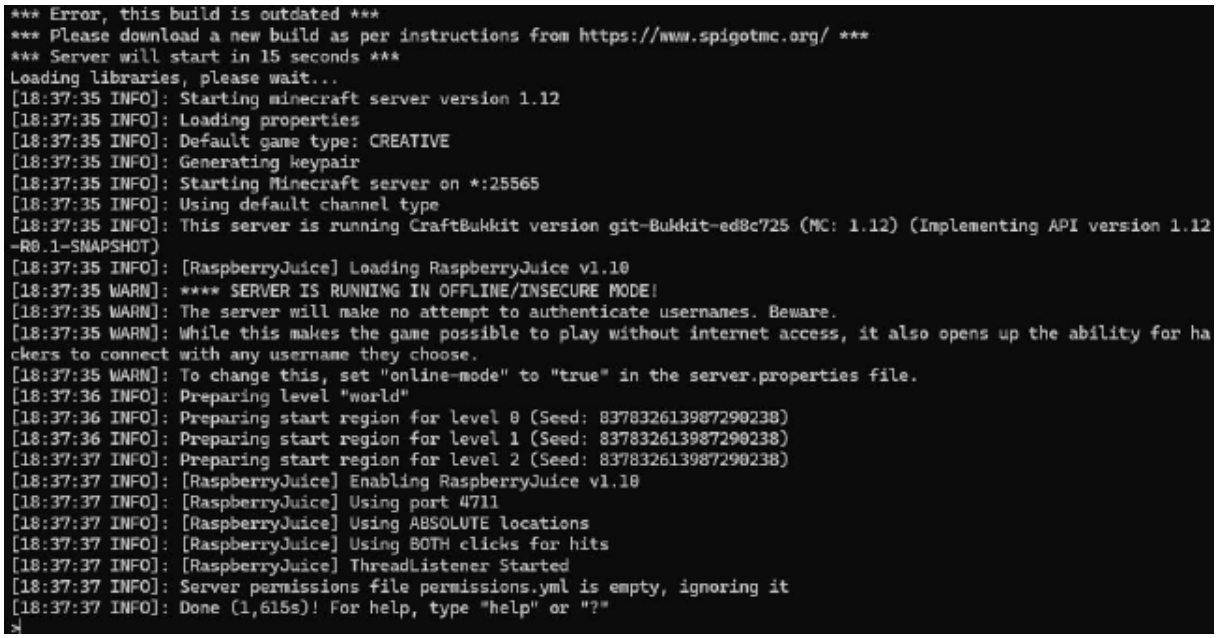
```
1 java -Xms1G -Xmx1G -jar craftbukkit.jar
```

If it becomes necessary to modify some of the server's properties, this can be done directly from `/Server/server.properties`. Once the command to start the server is executed, we need to wait a few seconds, and once it is fully loaded, we should have something similar to what is shown in the following image.

The server can also be started directly from a terminal. While positioned in the `/Server` directory, we can execute the second command to start the server.

## Connect to the Server

Since the server has been opened on our machine, we can access it directly by entering `localhost` in the "server address" field when creating it in the game's "MultiPlayer" tab.




## Modify the World from Python

To interact with the Minecraft world, the AdventuresInMinecraft library contains various API libraries with methods that allow us to perform different actions. The only one that needs to be included in the project is `mcpi`, which can be found in `/AdventuresInMinecraft/MyAdventu`
The other folders are additional API libraries that allow interaction with the game via Arduino, RaspberryPi, MicroBit, etc.

Below is an example of a small Python program that performs several actions in the game.

```python
# Import necessary modules
from mcpi.minecraft import Minecraft
import mcpi.block as block

# Connect to the Minecraft game
mc = Minecraft.create()

# Interact with the Minecraft world
mc.postToChat("Hello Minecraft World")
pos = mc.player.getTilePos()
mc.setBlock(pos.x+3, pos.y, pos.z, block.STONE.id)
```

# References


- **Adventures in Minecraft:**
  https://github.com/AdventuresInMinecraft/AdventuresInMinecraft-Linux
- **Functional Python Programming:**
  https://realpython.com/python-functional-programming/
  https://docs.python.org/3/howto/functional.html
- **Reflective Programming in Python:**
  https://dev.to/bshadmehr/series/23935
  https://gamedevacademy.org/python-reflection-tutorial-complete-guide/
- **Codecov:**
  https://docs.codecov.com/docs/flags
  https://docs.codecov.com/docs/status-badges
  https://about.codecov.io/
  https://docs.codecov.com/docs/quick-start
- **TLauncher (Free Minecraft):**
  https://tlauncher.org/en/


# Appendix A – Example Execution Flow

1. ExplorerBot starts exploration → publishes `map.v1`.
2. BuilderBot receives map → computes BOM → publishes `materials.requirements.v1`.
3. MinerBot mines resources → sends `inventory.v1`.

4. BuilderBot builds structure → sends `build.v1` progress.
5. User updates plan → BuilderBot halts and republishes new requirements.
6. MinerBot adjusts targets → BuilderBot waits.
7. Construction resumes → final `build.v1` confirms completion.
8. All agents return to `IDLE` and log final timestamps.