Game Agent for Minecraft

Prototype Project Report

Wahkiakum School District

07-FA25-SP26-WSD-AI

Lemar Joe Encio

Ethan Olsen

September 10, 2025

Introduction

The idea for this project took shape through the client's experience teaching young students about coding concepts and using tools such as Forge to modify the Minecraft video game. Building on this foundation, the client has a strong desire to help kids not only understand programming but also explore the fundamentals of artificial intelligence (AI); how it works, and how it can be applied in creative, interactive environments like Minecraft.

This project will create a modular AI agent that can operate within Minecraft and serve as a learning platform for others. The agent and its supporting infrastructure will allow students, researchers, and enthusiasts to modify, train, and experiment with reinforcement learning in an engaging context.

Background and Related Work

The project has been attempted by many people interested in Minecraft and machine-learning over the last couple of years. However, no one has made any of their projects a learning platform for other people to experiment with. Our project will be just that. We will design an AI agent that has modular code so that outside users can modify and learn from it as they see fit. We will also create a website in the second semester that explains how this was done and how to work with the AI agent to train it any way the user sees fit.

Project Overview

*** Bolded items will be defined in the glossary**

The system will be built around five main components that connect perception, decision-making, and action into a complete AI loop: the game controller, the observation pipeline, the action interface, the AI bridge, and the learning and inference system.

The game controller will be the part of the system that actually interacts with the Minecraft client. It will read the game state every tick and translate that into "player-like" inputs such as moving, looking, or jumping. To make this practical, the controller will include a human override feature that lets control be instantly switched between the player and the AI through a hot key. It will also buffer the last action, so the agent doesn't freeze between decisions. If the AI connection fails, the client will return control to the player, preventing the game from stalling. This layer will be implemented in one of two ways:

- **Fabric** or **Forge** in Java (**Gradle** build, Fabric API events, **Guava** EventBus)
- Headless client using **Mineflayer** (Node.js, TypeScript)

The observation pipeline will be what gives the AI its "view" of the world. It will collect structured data such as block/fluid info, entity positions and states, and the agent's own condition; its health, hunger, armor, position, and movement stats. It will also record environmental details like time, biome, and weather. Inventory information will be summarized, and **ray casting** will add a form of vision by simulating what the agent can see ahead using Minecraft's client-side ray cast APIs. To keep the data transfer efficient, observations will be **serialized** and **compressed.** During early prototyping we'll use JSON for readability with JSON Schema for validation, then potentially move to **Protocol Buffers** or **FlatBuffers** for performance; **MessagePack** is an alternative if we stay schema-light. For compression we'll evaluate between **zstd** and **LZ4**.

The action interface will turn the AI's outputs into real in-game actions. At the simplest level, this will include moving, attacking, placing blocks, or selecting hotbar slots; later these will be translated into more advance behaviors like "navigate to X", "mine Y", or "craft Z". Action rate limits and safety constraints will be enforced with **token-bucket throttling** using Bucket4j on the JVM or a custom "leaky bucket" in Python/TS to keep ai actions from triggering anticheat flags. When we add navigation and building, we'll leverage **Baritone** for pathfinding and mining; for custom logic we'll keep an **A*** and **JPS** implementation available. Actions will be validated against schemas and return acknowledgments so the model can observe success, failure, or partial outcomes.

The AI bridge connects the Minecraft client to the AI model backend and ensures neither side blocks the other. It manages bi-directional transport, tick alignment, **sequence numbers**, **timeouts**, and **backpressure** so the game never stalls waiting for an action. We'll start with **Websockets** or **FastAPI** in Python and keep a **gRPC** path ready in case we need higher throughput or typed streaming. Asynchronous frameworks like **Kotlin coroutines**, or **Python asynchio** will keep networking off the game thread, and ring buffers/queues like **Disruptor** or **ConcurrentLinkedQueue** will protect the render thread if networking hiccups. Versioned message schemas and heartbeat/reconnect logic will keep the bridge resilient as the codebase evolves.

The learning and inference system is the actual brain of the agent. Implemented in Python with **PyTorch**, it will begin with a **Deep Q-Network** and may expand to **PPO** using **Stable-Baselines3** or **RLlib**. All the knobs and settings will live in one central config (Hydra) so runs are easy to repeat, and we'll use NumPy/PyTorch utilities for basic data prep. Training will stat offline in a mock environment that mirrors our schemas, then move online to collect info from the live client. We'll teach the agent with a curriculum: first basic navigation, then survival, then combat and crafting. To measure real progress, we'll use repeatable test worlds with fixed seeds so results are comparable. We'll track experiments in a dashboard (TensorBoard or Weights & Biases) and ship structured logs to a lightweight monitoring setup (e.g., **ELK** or **Loki/Grafana**). Finally, we'll keep things reliable with unit tests (**pytest** for Python, **JUnit** for JVM code) and a Docker Compose file that spins up the client-model stack locally with a single command.

Client and Stakeholder Identification and Preferences

Primary Client and Sponsor
Our primary client, mentor and sponsor is Ron Wright, a semi-retired teacher from the Wahkiakum School District. He also serves as a coach and mentor for the school's robotics team and has supported students in pursuing creative projects for many years.

Stakeholders

- **Students (current and future)**: Middle and high school students who will use the project as a learning tool.
- **Teachers and Mentors**: Educators who may try to use the project in their class or activities, benefiting from instructional materials and accessible tools.

Glossary (Alphabetical)

- A* – A graph search algorithm widely used in games and robotics for pathfinding. It computes the shortest path between two points while considering both distance and cost.
- **Backpressure** – A flow-control mechanism where the receiving system signals the sender to slow down if data is arriving faster than it can be processed.
- **Baritone** – An open-source pathfinding system for Minecraft that enables automated navigation, mining, and building tasks.
- **Compressed** – The process of reducing the size of data using algorithms (e.g., zstd, LZ4) to improve storage efficiency and network transfer speed.
- **ConcurrentLinkedQueue** – A thread-safe, non-blocking queue implementation in Java that allows multiple producers and consumers to efficiently share data.

- **Deep Q-Network (DQN)** – A reinforcement learning algorithm that combines Q-learning with deep neural networks. It was the first method to achieve human-level control in Atari games by learning directly from raw pixels.
- **Disruptor** – A high-performance inter-thread messaging library for Java that uses a ring-buffer design to achieve very low-latency communication between components.
- **ELK Stack** – A log management and analytics suite consisting of **Elasticsearch** (search engine), **Logstash** (data ingestion pipeline), and **Kibana** (visualization). Used for collecting and analyzing large volumes of system logs.
- **Fabric** – A lightweight modding framework for Minecraft written in Java. It provides APIs and hooks into the game client for adding custom features.
- **FastAPI** – A modern Python web framework for building APIs, designed for high performance using asynchronous I/O and automatic data validation.
- **FlatBuffers** – A serialization library developed by Google that enables zero-copy access to data, making deserialization extremely fast.
- **Forge** – A popular Minecraft modding platform written in Java with a large ecosystem of mods and tooling, designed for heavier modifications.
- **Gradle** – A build automation system used for compiling, testing, and packaging Java projects. It manages dependencies and build scripts.
- **gRPC** – A high-performance remote procedure call (RPC) framework developed by Google that uses HTTP/2 and Protocol Buffers for efficient communication between services.
- **Guava** – A core Java library developed by Google that provides utilities for collections, caching, concurrency, and event handling.
- **JPS (Jump Point Search)** – An optimization of A* pathfinding for uniform-cost grids that reduces the number of nodes explored, improving performance.
- **Kotlin coroutines** – A concurrency framework in Kotlin that simplifies writing asynchronous, non-blocking code using sequential syntax.
- **Loki/Grafana** – A monitoring and observability stack where **Loki** is a log aggregation system optimized for scalability, and **Grafana** is a visualization and dashboard tool. Often used together for system monitoring.
- **LZ4** – A high-speed compression algorithm optimized for real-time scenarios, trading lower compression ratio for very fast throughput.
- **MessagePack** – A binary serialization format similar to JSON but more compact and faster to parse, suitable for performance-sensitive applications.
- **Mineflayer** – A Node.js library that implements a headless Minecraft client, allowing bots to interact with the game world programmatically.

- **PPO (Proximal Policy Optimization)** – A reinforcement learning algorithm developed by OpenAI that improves training stability by limiting policy updates. It is widely used for continuous control and robotics tasks.
- **Protocol Buffers** – A schema-based serialization format developed by Google. It encodes structured data compactly and is faster and smaller than JSON.
- **PyTorch** – An open-source deep learning framework developed by Meta (Facebook). It provides dynamic computation graphs and GPU acceleration, making it popular for reinforcement learning research and production.
- **Python asyncio** – A standard Python library for writing concurrent programs using asynchronous I/O, coroutines, and event loops.
- **Ray casting** – A computational technique for simulating vision by projecting rays into a virtual environment to determine the first object intersected.
- **RLlib** – A scalable reinforcement learning library built on top of Ray. It provides distributed training capabilities and implementations of many popular RL algorithms, including PPO and DQN.
- **Sequence numbers** – Identifiers added to messages to preserve order and ensure reliable communication between systems.
- **Serialized** – The process of converting structured data into a format suitable for storage or transmission, often used before compression.
- **Stable-Baselines3 (SB3)** – A Python library that provides reliable, well-tested implementations of popular reinforcement learning algorithms (e.g., PPO, DQN, A2C) built on top of PyTorch.
- **Timeouts** – Limits placed on how long a system waits for a response before assuming failure, preventing indefinite blocking.
- **Token-bucket throttling** – A rate-limiting algorithm where actions consume tokens from a "bucket" that refills at a fixed rate, controlling request frequency.
- **WebSockets** – A communication protocol that enables persistent, full-duplex connections between client and server, allowing real-time data exchange.
- **zstd (Zstandard)** – A modern compression algorithm developed by Meta (Facebook) that balances fast speeds with high compression ratios.

References

[1] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv:1312.5602*, Dec. 2013. [Online]. Available: https://arxiv.org/abs/1312.5602 . [Accessed: Sep. 12, 2025].

[2] J. Schulman *et al.*, “Proximal Policy Optimization Algorithms,” *arXiv preprint arXiv:1707.06347*, Jul. 2017.
(Introduces PPO — modern, stable RL algorithm widely used in practice.)

[3] PyTorch, “PyTorch: An open source machine learning framework,” *PyTorch Official Website*. [Online]. Available: https://pytorch.org/ . [Accessed: Sep. 12, 2025].

[4] OpenAI, “Learning to Play Minecraft with Video PreTraining (VPT),” *OpenAI Research*, Jun. 2022. [Online]. Available: https://openai.com/index/vpt/. [Accessed: Sep. 12, 2025].
(Trains agents to play Minecraft directly using human video data.)

[5] G. Wang *et al.*, “Voyager: An Open-Ended Embodied Agent with Large Language Models,” *arXiv preprint arXiv:2305.16291*, May 2023. [Online]. Available: https://arxiv.org/abs/2305.16291. [Accessed: Sep. 12, 2025].
(Explores long-term, open-ended skill learning in Minecraft.)