Full Tech Stack

Key decisions (answer these now)

- Use a **headless bot** instead of a client mod for MVP? → **Yes at first**
- Bot flavor: **Mineflayer (Node)** vs **MCProtocolLib (Java)**? → **Mineflayer**
- Bridge: **WebSocket/MsgPack first** (fast) → gRPC later? → **Yes then Maybe**
- For SP worlds, OK to run a **local dedicated server**? → **Yes**
- Add a **Paper plugin** when training on your own server? → Likely **Yes** (clean rewards/resets) → **Yes**

Necessary

1) Client-side game controller (required)

- **Control loop:** Something that can read the game state each tick and send “player-like” inputs (move, look, jump, interact).
- **Human override:** Hotkey or UI switch to hand control back/forth instantly.
- **Action buffering:** Hold last action between decisions; cap decision rate (e.g., 5–15 Hz) to keep latency predictable.
- **Error handling:** If the AI link drops, fall back to idle or human control; never hang the render/game thread.

Software:

- Fabric or Forge mod (Java/Kotlin);
- Headless bot client via Mineflayer (Node.js) or MCProtocolLib (Java);
- Lightweight state machines (XState, SMI);
- Java concurrency (Kotlin coroutines, java.util.concurrent);

- Event buses (Fabric API events, Guava EventBus).

2) Observation pipeline

- **Spatial snapshot:** Local 3D neighborhood of blocks/fluids and metadata (breakability, light, etc.).
- **Entities:** Nearby entities with type, position, velocity, and relevant attributes.
- **Player state:** Health, hunger, armor, effects, position/velocity, orientation, biome/time/weather.
- **Inventory summary:** Slots, item types, durability, counts; optional crafting affordances.
- **Sensors/raycasting:** Forward rays / fan of rays for line-of-sight and target awareness.
- **Compression/serialization:** Efficiently package observations for transmission to the AI.

Software:

- Fabric/Forge APIs for world/entity access;
- Raycasting helpers in Minecraft client API;
- Serialization with MessagePack, Protocol Buffers, FlatBuffers;
- Compression via zstd or LZ4;
- Binary utils (Kryo);
- Data schemas with JSON Schema or .proto.

3) Action interface

- **Low-level actions:** Movement (forward/strafe/jump/sneak), look deltas, attack/use/place, hotbar select.
- **High-level macros (optional):** “go to X,” “mine Y,” “craft Z,” built from low-level primitives.
- **Constraints & safety:** Rate limits, cooldowns, human-plausible bounds to avoid anticheat flags.
  Acknowledge/results: Return success/failure and outcome details to the AI loop.

Software:

- Input simulation via client hooks (Fabric/Forge);
- Task layer using Baritone (navigation/mining/building) or custom A*/JPS implementations;

- Rate limiting with Token Bucket libs;
- Validation with Hibernate Validator (Java) or Pydantic (Python) for action schemas.

4) AI bridge (client ↔ model)

- **Transport:** A request/stream channel between the game and the AI (bi-directional).
- **Schema:** Clear message definitions for Observation, Action, Event, and Episode.
- **Scheduling:** Tick alignment, timestamps, sequence numbers, replay protection.
- **Backpressure:** Queues and timeouts so the game never waits indefinitely for the model.
  Versioning: Message/schema version to prevent drift between game and AI.

Software:

- WebSockets (Java: Jetty/Tyrus, Python: websockets/Starlette/FastAPI);
- gRPC (HTTP/2) with Protobuf;
- ZeroMQ or NATS for pub/sub;
- Async frameworks (Kotlin coroutines, Python asyncio); Ring buffers/queues (Disruptor, Java ConcurrentLinkedQueue).

5) Learning & inference

- **Policy/inference loop:** Consumes observations, outputs actions at a fixed rate under latency budget.
- **Training loop:** Offline or online updates from collected rollouts; supports PPO/DQN or similar.
- **Curriculum:** Simple-to-hard progression (navigation → survival → combat → crafting).
- **Evaluation harness:** Deterministic seeds and standardized scenarios to compare policies.

Software:

- PyTorch;
- Stable-Baselines3 or RLlib;
- JAX/Flax (alternative);
- PyTorch Lightning (optional);
- Hydra for configs;
- NumPy for preprocessing;
- ONNX Runtime/LibTorch (if embedding models in JVM).

Optional

6) Server-side support (optional, but useful for MP)

- **Reward shaping & telemetry:** Server-side hooks to compute rewards fairly and consistently.
- **Scenario orchestration:** Start/stop episodes, reset positions/inventories, spawn targets, enforce rules.
- **Privileged queries:** Access to state that's hard/expensive for a client to infer (kept minimal to avoid "cheating").
- **Anti-cheat cooperation:** Ensure agent actions remain within human-like limits.

Software:

- Paper/Spigot plugin (Java/Kotlin);
- Citizens/Sentinel for NPC testing;
- LuckPerms for permissions;
- Exposed/JPA/Hibernate for telemetry storage;
- REST via Spring Boot/Ktor or embedded HTTP server.

7) World scaffolding (optional)

- **Task arenas & curricula:** Prebuilt maps/chambers for training/eval tasks.
- **Triggers & markers:** Checkpoints, success/failure signals, timers, counters.
- **Reset logic:** Clean restoration of world state between episodes.

Software:

- Datapacks (functions, loot tables, predicates);
- Structure blocks;
- WorldEdit/FAWE for arena building;

- Command blocks for prototyping;
- Paper plugin APIs for resets.

8) Pathfinding & skills layer

- **Navigator:** Converts target positions into feasible step-by-step movement plans.
- **Skill primitives:** Mine block, place block, build stair, bridge gap, kite mob, parkour step.
- **Task planner (optional):** A lightweight state machine or behavior tree to sequence primitives.

Software:

- Baritone (Java) for navigation/building;
- Behavior Trees (btree4j, behavior3js) or custom FSM;
- Graph libs (JGraphT);
- Heuristic search (A*/D* Lite) implementations.

-

9) Data & logging

- **Rollout recorder:** Store (obs, action, reward, done, info) with timestamps.
- **Compression & format:** Efficient on-disk representation for long sessions.
- **Replay buffers:** Sampling APIs for training; support prioritization if needed.
- **Analytics:** Success rates, time-to-goal, damage taken, resources gathered, etc.

Software:

- Parquet/Arrow for columnar storage;
- zarr or NPZ for buffers;
- Weights & Biases or MLflow for experiment tracking;
- Pandas/Polars for analysis;
- Prometheus + Grafana for metrics dashboards;

- Logging via Loguru/structlog (Python) and SLF4J+Logback (JVM).

10) Reliability & performance

- **Latency budget:** End-to-end decision within a few milliseconds; action rate caps to stabilize.
- **Health checks & reconnection:** Detect dead links, restart streams, exponential backoff.
- **Profiling:** Measure time spent in capture, encode, send, infer, decode, apply.
- **Frame safety:** Never block the main thread; isolate network/model work.

Software:

- Watchdogs/health endpoints (FastAPI/Actuator);
- Retry/backoff (Tenacity for Python, resilience4j for JVM);
- Profilers (py-spy, scalene, line_profiler, Java Flight Recorder);
- Async executors (Kotlin coroutines, Java ForkJoinPool).

11) Multiplayer specifics

- **Identity & auth:** Player account or bot identity management; secure secrets storage.
- **Desync tolerance:** Handle packet loss, jitter, and server tick variation.
- **Fairness rules:** Keep to human-plausible input rates and camera speeds.
- **Spectator & control UI:** Admin controls to start/stop agents, observe, and gather metrics.

Software:

- Mojang/Microsoft auth flows (Yggdrasil-compatible libs), Secret storage (dotenv, AWS Secrets Manager);
- Network libs with jitter buffers;
- Admin UIs via server plugin commands or web dashboards (React + FastAPI/Spring).

12) Safety & guardrails

- **Action filters:** Disallow destructive actions outside designated arenas.

- **Rate/area limits:** Cap block changes per minute; whitelist biomes/regions if needed.
- **Emergency stop:** Hard kill-switch from in-game command and from outside (console/HTTP).

Software:

- Server plugin interceptors;
- Region protection (WorldGuard);
- Rate limiters (Guava RateLimiter, Bucket4j);
- HTTP control endpoints (FastAPI/Spring Boot).

13) Developer experience

- **Local dev loop:** Run the client controller and AI locally with hot reload where possible.
- **Config system:** Centralized configs for observation size, action set, ports, and rates.
- **Testing:** Unit tests for schemas and primitives; scenario tests for end-to-end tasks.
- **CI basics:** Lint/format, build artifacts, minimal automated tests on push.

Software:

- Gradle (Kotlin DSL) for JVM;
- pytest/mypy/ruff/black for Python;
- IntelliJ IDEA, PyCharm/VS Code;
- Docker & Docker Compose; GitHub Actions;
- Pre-commit hooks.

14) Documentation & UX

- **Message contracts:** Human-readable spec for obs/actions/events.
- **Runbooks:** “How to run single-player,” “How to connect to a server,” “How to record an episode.”
- **Debug HUD:** On-screen status (AI on/off, action rate, ping, reward).
- **Logs & traces:** Structured logs with correlation IDs to tie game ticks to model decisions.

Software:

- Markdown + MkDocs or Docusaurus;

- OpenAPI/AsyncAPI for service docs;
- In-game HUD via Fabric/Forge HUD overlays;
- Visualization notebooks (Jupyter);
- Tracing with OpenTelemetry (otlp exporters) + Jaeger/Tempo.