April 14, 2026 · 12 min read · By Yiğit Konur

I turned Pokémon Red into a shared multiplayer arcade — here's how

software-engineering

▯ Play it live → (https://yigitkonur.github.io/wasm-pokemon-red/) — one instance, everyone plays together.

the idea started simply: what if visitors to my site could play Pokémon Red together, passing the controller between them like a Twitch Plays setup — but embedded on a blog, backed by a proper save state, and with no server cost worth mentioning?

this is the full build log.

---

what it does

one copy of Pokémon Red runs in the browser. anyone who opens the page can join. a single player holds the controller at a time; after five seconds idle the turn passes to the next person in queue. if nobody is around, an AI bot picks up and navigates the overworld or battles. there is a live chat next to the game. the save state persists across sessions: close the tab, come back tomorrow — the game is exactly where you left it.

-

step 1 — build the ROM from source

everything starts with pret/pokered (https://github.com/pret/pokered), the fully reverse-engineered disassembly of Pokémon Red and Blue. it's a remarkable project: decades of community work turned a 1 MB Game Boy cartridge into readable, annotatable assembly.

```bash
git clone https://github.com/pret/pokered.git
cd pokered
make
# produces pokered.gbc
sha1sum pokered.gbc   # verify against roms.sha1
```

you need rgbds (https://rgbds.gbdev.io) (≥0.9). the assembler, linker, and fix tool are all included. `make` produces `pokered.gbc` — a verified cartridge image. the SHA1 is pinned

in `roms.sha1`; build reproducibility matters when you're going to ship this ROM publicly.

---

step 2 — compile the emulator to WebAssembly

binjgb (https://github.com/nicknassar/binjgb) is a cycle-accurate Game Boy emulator written in C. it is small, well-structured, and exposes the right hooks for Emscripten compilation.

```bash
git clone https://github.com/nicknassar/binjgb.git
source ~/emsdk/emsdk_env.sh

emcc binjgb/src/emulator.c binjgb/src/joypad.c binjgb/src/apu.c \
     web/binjgb/wrapper.c \
     -o dist/binjgb.js \
     -s EXPORTED_FUNCTIONS=@web/binjgb/exported.json \
     -s MODULARIZE=1 \
     -s ALLOW_MEMORY_GROWTH=1 \
     -O3
```

`web/binjgb/exported.json` lists the eight C functions the browser shell needs:

```json
["_emulator_new_simple", "_emulator_run_until_f64",
 "_get_frame_buffer_ptr", "_get_audio_buffer_ptr",
 "_emulator_get_ticks_f64", "_free",
 "_malloc", "_joypad_set"]
```

`wrapper.c` is an overlay that adds a frame-perfect 60fps canvas loop and exposes the audio ring buffer to JavaScript without modifying binjgb's source. the compiled output is two files: `binjgb.js` (runtime and glue) and `binjgb.wasm` (the actual bytecode).

---

step 3 — wire the browser shell

-

`web/player.html` is a standalone HTML page. no framework, no bundler. it loads `binjgb.js`, initialises the emulator with the ROM bytes, then ticks forward in a `requestAnimationFrame` loop.

the loop is straightforward:

```javascript
function tick(now) {
  const cyclesPerFrame = 4194304 / 60;
  Module._emulator_run_until_f64(emulatorPtr,
    Module._emulator_get_ticks_f64(emulatorPtr) + cyclesPerFrame);
  renderFrame();
  scheduleAudio();
  requestAnimationFrame(tick);
}
```

keyboard input goes through a thin dispatcher in `player.js` that maps DOM KeyboardEvent codes to the eight Game Boy buttons and calls `Module._joypad_set`. the canvas is upscaled 3x with nearest-neighbour interpolation so the pixel art stays crisp.

---

step 4 — the multiplayer turn system

the most interesting engineering is the turn system. it lives in two files: `web/multiplayer.js` (browser client) and `server/server.js` (Node.js WebSocket server deployed on Railway).

the protocol is minimal:

-

`join`		
`input`		
`state_sync`		
`turn_granted`		
`turn_revoked`		
`chat`		
`ai_granted`		
`ai_revoked`		

the server tracks one `activeMode`: `idle`, `human`, or `ai`. only the mode holder's inputs are forwarded to the emulator. everyone else spectates.

turn timer:

```javascript
function resetTurnTimer(ws) {
  clearTimeout(turnTimer);
  turnTimer = setTimeout(() => {
    if (activeMode === 'human' && activePlayer === ws) {
      revokeHumanTurn();
      grantNextOrIdle();
    }
  }, TURN_MS); // 5000
}
```

every button press from the active player resets the five-second timer. idle means someone gets granted next. a human leaving means the next person in queue or idle. idle long enough means the AI gets a turn.

-

step 5 — persisting save state with Redis

without persistence the game would restart every time the server restarts or the last player leaves. the solution: serialize the full emulator RAM and store it in Redis on every turn handoff.

```javascript
// on turn handoff (server.js)
const stateBlob = emulator.serializeState();      // ~8 KB base64
await redis.set('pokemon:state', stateBlob);

// on new player joining
const saved = await redis.get('pokemon:state');
if (saved) ws.send(JSON.stringify({ type: 'state_sync', state: saved }));
```

upstash redis is serverless: no persistent connection, no provisioning. the REST API fits perfectly with Railway's ephemeral container model. the free tier covers ~10 000 requests/day — more than enough at hobby scale.

step 6 — the AI bot

`autoplay.js` is a simplified port of bouletmarc/PokeBot (https://github.com/bouletmarc/PokeBot). the bot doesn't do screen scraping; it reads Game Boy RAM addresses directly via the emulator's exported memory pointer.

-

```javascript
// check if we're in a battle
const BATTLE_ACTIVE = 0xD057;
const inBattle = readRam(BATTLE_ACTIVE) !== 0;

if (inBattle) {
  battleRoutine();   // select FIGHT, pick highest-PP move
} else {
  overworldWalk();   // random walk with wall-bounce detection
}
```

the bot is gated by `multiplayerAllowed`: it only activates when the server has granted it a turn via `ai_granted`. it never fires when a human holds the controller.

---

step 7 — embedding or self-hosting

embed the shared arcade (one line):

```html
<iframe src="https://yigitkonur.github.io/wasm-pokemon-red/"
        width="100%" height="640"
        style="border:none;border-radius:12px"
        allow="autoplay">
</iframe>
```

self-host the whole stack:

1. clone yigitkonur/wasm-pokemon-red (https://github.com/yigitkonur/wasm-pokemon-red) and build the ROM and WASM (`make`)
2. deploy `server/server.js` to Railway — one-click from `railway.toml`
3. create an Upstash Redis instance, set `UPSTASH_REDIS_REST_URL` and `UPSTASH_REDIS_REST_TOKEN` in Railway environment
4. update `WS_URL` in `web/multiplayer.js` to point to your Railway deployment
5. push `web/` to any static host (GitHub Pages, Netlify, Vercel, S3)

total cloud cost at hobby scale: **$0**.

---

what's next

the logical next step is more ROMs. the binjgb emulator handles any Game Boy cartridge; swapping the ROM file and redeploying is a five-minute operation. the turn system, Redis persistence, and AI bot are all ROM-agnostic.

the interesting design question is whether to run each ROM as a separate page with its own server, or to extend the server to a multi-room model where each room is one ROM. the latter is more elegant and would let visitors channel-surf between games — like a small arcade.

Tetris is the obvious first addition. Link's Awakening next. eventually a lobby page at the root where you pick a game, join the queue, and play for your five seconds.

---

links

- live arcade: yigitkonur.github.io/wasm-pokemon-red (https://yigitkonur.github.io/wasm-pokemon-red/)
- source: github.com/yigitkonur/wasm-pokemon-red (https://github.com/yigitkonur/wasm-pokemon-red)
- pret/pokered: github.com/pret/pokered (https://github.com/pret/pokered)
- binjgb: github.com/nicknassar/binjgb (https://github.com/nicknassar/binjgb)
- PokeBot: github.com/bouletmarc/PokeBot (https://github.com/bouletmarc/PokeBot)

-

SOURCES

pret/pokered — Pokémon Red/Blue disassembly (https://github.com/pret/pokered)

nicknassar/binjgb — Game Boy emulator (C) (https://github.com/nicknassar/binjgb)

Emscripten — C to WebAssembly toolchain (https://emscripten.org)

rgbds — Game Boy assembler and linker (https://rgbds.gbdev.io)

bouletmarc/PokeBot — RAM-reading Pokémon bot (https://github.com/bouletmarc/PokeBot)

Upstash Redis — serverless Redis (https://upstash.com)

Railway — deploy the WebSocket server (https://railway.app)

Full source and self-host guide (https://github.com/yigitkonur/wasm-pokemon-red)

-