HackMerlin LLM Agent — 6 Levels passed

Repository: https://github.com/unforgettablexD/hackmerlin

Demo Video: https://github.com/unforgettablexD/hackmerlin/blob/main/level6.webm

Attempts : **hackmerlin/session-ollama-20250910-115335/attempts.jsonl at main · unforgettablexD/hackmerlin**

Level Summary : **hackmerlin/session-ollama-20250910-115335/level_summaries.json at main · unforgettablexD/hackmerlin**

Transcript : hackmerlin/session-ollama-20250910-115335/transcript.jsonl at main · unforgettablexD/hackmerlin

---

1) Executive Summary

This document describes a browser-automation LLM agent that solves levels on *hackmerlin.io*. The agent runs a Chromium session via Playwright, converses through the site's chat UI, opportunistically submits candidate passwords, and advances levels only when the page heading changes from **"Level N"** → **"Level N+1."** Modal hints alone do **not** count as success. Strategic prompting leverages a persistent memory of prior tries to avoid repeated mistakes and to escalate from probing questions to decisive submissions.

---

2) System Architecture

2.1 High-Level Components

- **Browser Controller** — Drives navigation, chat input, password submission, modal handling, and robust level detection.
- **Experience Store (Memory)** — Append-only attempts log; compact level summaries (tries, successes, avoid list, recent patterns); helpers to fetch recent attempts for a level.
- **LLM Client (Ollama : GPTOSS)** — Thin HTTP client that requests strictly formatted JSON, extracts one JSON object, and optionally preserves the model's hidden reasoning ("think") to disk for offline analysis.
- **Strategist** — Encodes system rules and level tactics; composes *conversation history* + *state* into the prompt; returns the next **ASK** or **SUBMIT** action in strict JSON.
- **Runloop** — Orchestrates: choose action → execute in the browser → verify success by heading increment → persist artifacts (attempts, transcript, screenshots, think logs).
- **Entrypoint (CLI)** — Simple command-line wrapper to launch a session (headless or visible).

2.2 Success & Safety Invariants

- **Success** = heading increment from *Level N* to *Level N+1* only. Modal hints or UI popups are insufficient.

- **No repetition** — Wrong submissions become part of an **avoid list**; the Strategist never repeats them.
- **Short, format-locked prompts** — The agent prefers short questions and strictly formatted submissions to reduce ambiguity.

---

3) Key Implementation Choices (Why These Decisions)

1. **Hard success signal via DOM heading**
   Queries multiple heading candidates (role-, tag-, and text-based) with a last-resort body regex (`\bLevel\s+(\d+)\b`) to robustly parse the level number. This prevents false positives from modals and UI animation delays.
2. **Modal handling without conflating success**
   Hints are captured and closed via a Continue/Enter/Close(X) sequence, then the runloop re-checks the heading. This preserves contextual clues while protecting the success criterion.
3. **Memory for tactical restraint**
   Every ASK/SUBMIT is logged (`attempts.jsonl`), and level aggregates are stored (`level_summaries.json`), including an **avoid** list. Strategist prompts embed these facts so the LLM avoids re-asking and re-submitting the same thing.
4. **Strict JSON I/O contract with the LLM**
   The client strips any hidden reasoning (e.g., `<think>` blocks) from the visible answer but archives full traces to disk. If the model drifts from JSON, a second pass enforces **"one JSON object, no prose."**
5. **"VC tactic" probing**
   For ambiguous levels, the system prompt encodes a recommended path: ask for **length**, **vowels/consonants**, **first/last character**, then indexed characters as needed; synthesize candidates and submit when confident. This scales from easy to harder levels without hand-coding each level.

---

4) Code Walkthrough (What Each Module Does)

- **`merlin_agent/browser.py`**

    - Launches Chromium, navigates, types in chat, submits passwords, detects level increments (`get_level`, `wait_for_level_increment`), captures screenshots/DOM, and safely handles Mantine modals.

- **`merlin_agent/memory.py`**
    - `ExperienceStore` appends attempts to `attempts.jsonl`, maintains `level_summaries.json` (tried/successes/avoid), and returns recent attempts for strategist consumption.
- **`merlin_agent/ollama_client.py`**
    - Posts to `/api/chat`, tolerates NDJSON or single JSON bodies, extracts one JSON object, preserves `<think>` for disk, and normalizes into **ASK/SUBMIT** action shapes with optional guard-retries for format drift.
- **`merlin_agent/strategist.py`**
    - Defines SYSTEM rules: *never repeat wrong guesses*, *don't ask the same thing twice*, *submit when confident*, *keep prompts short and format-locked*. Embeds **conversation history** + **STATE JSON** (`level`, `attempts_so_far`, `avoid`) and returns the next action in strict JSON.
- **`merlin_agent/runloop_llm.py`**
    - Main loop: close any hint modal; request action; perform ASK (and opportunistically extract a password from the reply) or SUBMIT; verify by heading; persist **attempts**, **transcript**, **screenshots**, and **think logs**; continue to next level upon success.
- **`merlin_agent/utils.py`**
    - Project paths (`ROOT`, `RUNS`) and helpers (`ts_ms`, `write_jsonl`, `load_json`).
- **`main_llm.py`**
    - CLI entrypoint: `--headless`, `--debug`; prints transcript path after the run.

---

5) Runbook (How to Reproduce)

5.1 Prerequisites

- Python 3.10+
- Playwright Chromium:
- pip install -r requirements.txt
- Ollama running locally (or reachable) with a pulled model. Environment variables:
    - `OLLAMA_ENDPOINT` (default: `http://127.0.0.1:11434/api/chat`)

- `OLLAMA_MODEL` (default in code: `gpt-oss:latest`)
- `OLLAMA_TEMPERATURE` (default `0.8`)
- `OLLAMA_NUM_CTX` (default `2048`)

5.2 Launch

 # from repo root
python -m merlin_agent.main_llm
omit --headless to watch the browser

Artifacts are written under:

runs/session-ollama-YYYYMMDD-HHMMSS/
  transcript.jsonl
  attempts.jsonl
  level_summaries.json
  think/
  levelXX_attemptYY.png

---

6) Data & Artifacts (What Is Captured)

`attempts.jsonl` — chronological log of ASK/SUBMIT events, replies, correctness flags, and modal hints. *(Attach/link your file here.)*
Example entry:

{"type":"submit","level":3,"password":"AROMA","submit_ok":false,"modal_hint":""}

- **`transcript.jsonl`** — per-attempt metadata: timestamp `ts`, `level`, `attempt`, `action`, short `why`, and a `think_preview`.
- **`level_summaries.json`** — aggregate per-level stats (`tried`, `successes`, `do_not_try`, `last_k_patterns`).
- **Screenshots & DOM dumps** — visual/debug evidence per attempt; initial DOM snapshot per level.
- **Think logs** — full chain-of-thought traces saved under `runs/.../think/` (kept offline for auditability; not shown in prompts).

---

7) Strategy & Prompting (How It Decides Without Revealing CoT Publicly)

Goal & Contract. The Strategist composes a strict decision prompt that always returns exactly one JSON object representing the next action. Only two shapes are valid:

{"action":"ask","question":"<short>","avoid":["..."],"why":"<short>"}

{"action":"submit","answer":"<word>","avoid":["..."],"why":"<short>"}

System Rules Embedded. The SYSTEM prompt encodes: (1) success is *only* a heading increment from Level N → N+1; (2) modal hints are not success; (3) never repeat wrong submissions; (4) never ask the same thing twice; (5) keep prompts short and format-locked; (6) prefer SUBMIT when confident; and (7) level hints + a general **VC tactic** (length → vowels/consonants → first/last → index probes → synthesize → submit).

Conversation-Aware Prompting. Before each decision, the Strategist builds a **conversation block** from memory:

- For each attempt on the current level, it writes compact lines like `USER ASK: ...`, `ASSISTANT REPLY: ...`, `USER SUBMIT: <pw> → OK/WRONG`, plus any `HINT:` text and `EVENT: advanced to next level` markers.
- It also constructs a **STATE** object: `{ level, attempts_so_far, avoid }`, where `avoid` is deduplicated from **all prior wrong submissions** on that level (capped to 50). This ensures the model does not re-submit previous mistakes and reduces repeated questions.

Two Prompting Patterns (current vs. alternative).

- **Current (active):** The Strategist **merges** SYSTEM + Conversation History + STATE **into the system message** and uses a *minimal* user message: “Return the next ACTION in STRICT JSON ONLY.” This strongly biases the model to emit a single JSON object and keeps features/data in one place for reproducibility.
- **Alternative (commented in code):** A prior variant packaged the entire conversation and state into the **user** message with a thinner system prompt. The active approach proved more robust for JSON-only compliance.

Action Normalization & Fallbacks. After the model responds, the Strategist normalizes the object and applies guardrails:

- If `action` is missing/invalid, it falls back to a safe **ASK** with a default, format-locked question (“What is the password? Reply with the single word only.”).
- The Strategist threads through the computed `avoid` list to the action so downstream logic can respect it.
- A short `debug_think_preview` string and a full `_full_think` payload are attached **only for logging**. Hidden reasoning (“chain-of-thought”) is **saved offline** for audit/debug but **never surfaced** to the webpage, and it is **not** re-consumed as model context.

Decision Cycle Summary.

1. Build Conversation History and STATE from memory for the current level.
2. Merge with SYSTEM rules (GOAL/FACTS/RULES/TACTIC/OUTPUT).
3. Ask the model to return one strict JSON object (ASK or SUBMIT).

4. Normalize and enforce fallbacks; attach `avoid` and think metadata for logs.
5. Hand the structured action to the runloop, which executes in the browser and verifies success strictly by heading increment (modal hints ignored).

Why This Works. The design couples **explicit rules** with **level-aware memory** and a **JSON-only contract**, minimizing prompt drift and repeated errors. The minimal user message and system-merged context reduce formatting failures, while the avoid list and VC tactic provide a disciplined path from probing to decisive submission.

---

1. **If cost and compute requirements were not an issue, how would you improve the agent?**

   If I did not have a cost or compute requirements, I would use the gpt5 or better models for reasoning and api calls.

2. **If the agent had to be run in a cost-constrained environment, how would you redesign it?**

   If the agent had to be run in a cost-constrained environment I would add more 1. Hardcoding guesses and 2. Better prompts with way better guesses and hints or i could guide it with questions as to how to play this game.

3. **What were unique challenges that you faced building this?**

   The unique challenges I faced were:
   - how to tell model when it has passed a level
   - what all information does the model needs to play this game
   - The prompting required to guide agent to pass this test
   - The models decision reasoning model vs normal models: