Experimental Setup and Preliminary Evaluation of an Autonomous Research Agent

Satyam Regmi
CogAI Lab, Caldwell University
mail@satym.me

Abstract
This research investigates whether a large language model, equipped with computer tools, storage, and full permission can autonomously perform research tasks, evaluate its findings, and improve its results through repeated execution and review. The broader objective is to examine whether additional autonomous working time leads to verified progress or new findings. The initial phase focused on establishing a functioning agent environment and evaluating a small computational experiment. Unlimited execution and original discovery were not tested.

System Architecture

The system separated model inference from task execution. The agent executed commands and edited files inside a Linux Docker container.

Component	Configuration
Controller operating system	macOS 27.0.1, ARM64
Host Python environment	Python 3.12.15 virtual environment managed with uv
Package manager	uv 0.13.0
Docker engine	Version 29.8.2
Agent framework	OpenHands SDK 1.54.0
Supporting packages	OpenHandsTools, Workspace, and Agent Server 1.54.0
Model	cyankiwi/Qwen3.6-35B-A3B-AWQ-4bit
Model serving	vLLM, reporting version 0.23.0
API interface	OpenAI-compatible API under /ai/v1
Served model identifier	llm1
Advertised model context limit	100,000 tokens
Execution image	ghcr.io/openhands/agent-server:latest-python
Container architecture	Linux ARM64

The agent received terminal and file-editing tools. Its access to the Mac's filesystem was provided through a designated shared research directory. Graphical desktop control and dedicated web-search tools were not configured during this pilot.

Installation and Connectivity Validation

A project directory, ~/research-agent, was created. A Python virtual environment was initialized, and the OpenHands packages were installed together to maintain compatible versions. Successful imports confirmed that the SDK and Docker workspace components were available.

Docker setup required starting Docker Desktop, selecting its desktop-linux context, and adding Docker Desktop's executable directory to the terminal's search path. These steps resolved an unavailable Docker engine and a missing credential-helper error. A successful hello-world container verified basic container execution. The DOCKER_HOST variable was subsequently configured to allow the Python client to locate the same engine.

The model endpoint was first checked through its model-listing API. A separate chat-completion request supplied a simple function definition and requested a tool call. The model returned a structured check_workspace function call, demonstrating basic compatibility with the tool-calling protocol.

Initial Agent Validation

The first agent task was to create and execute a Python program that calculated the sum of integers from 1 through 100. The model was configured with native tool calling, temperature zero, a maximum output of 4,096 tokens, and an initial request timeout of 180 seconds.

An unsupported visualize argument caused the first attempt to fail during conversation creation. Removing that argument resolved the compatibility issue.

In the successful run, the agent created a program using sum(range(1, 101)) and executed it inside Docker. After the agent finished, the controller separately executed the generated program and checked that its output equaled 5050. The program passed this check and was copied to the Mac before container cleanup. This established an end-to-end path from model inference to tool execution and output verification.

Persistent Research Workspace

A second runner mounted ~/research-agent/workspace into the container at /workspace. This allowed code, measurements, reports, and progress notes to survive container shutdown. The controller verified the mount by writing a marker inside the container and reading it from the Mac.

Research instructions were stored in TASK.md. The controller recorded agent events in timestamped JSONL files and maintained a separate run-status file. Installed Python package versions were recorded in requirements-working.txt.

The research runner used an 80,000-token maximum input setting, a 4,096-token output limit, temperature zero, two configured retries, stuck detection, and a maximum of 40 agent iterations per run. The iteration limit was not a wall-clock time budget.

Each invocation started a new conversation. Continuity depended on reading saved files, particularly PROGRESS.md; exact conversation restoration and an automatic continuous research loop were not implemented.

Pilot Research Task

The agent was assigned a sorting experiment examining how input size and ordering affected insertion sort, merge sort, and Python's built-in sorted().

The task specified three input sizes—100, 1,000, and 3,000—and three input arrangements: random, ascending, and descending. Five fixed seeds were required. The agent selected 10 warm-up runs and 50 measured repetitions per configuration, producing 6,750 measurement records.

Requested deliverables included an experiment plan, executable code, raw CSV measurements, a report, and incremental progress notes. The instructions required matched inputs across algorithms, input preparation outside the timed section, correctness checks, and conclusions based on executed experiments.

Independent Review of the Pilot

An external review of the uploaded code and data found 6,750 unique measurement identifiers and 27 algorithm–size–input groups containing 250 measurements each. All 27 reported means and sample standard deviations matched recalculations from the CSV after rounding. Both custom sorting implementations also passed 205 additional correctness cases each.

However, the review identified weaknesses in reporting and experimental practice. The report included incorrect speedup ranges, an interpretation inconsistent with its own input-order measurements, and an unsupported prediction about alternative C implementations. The insertion-sort function was described as in-place despite copying its input.

The benchmark prepared an input copy before timing, but additional copying and allocation occurred within the sorting functions. Correctness was checked once per configuration rather than for every timed result. Measurements were accumulated in memory and written only after the full experiment, creating a risk of losing intermediate results. Algorithms were also measured in a fixed order. The progress document did not establish that checkpoints had been written incrementally.

These findings demonstrated that numerical consistency and successful execution did not guarantee accurate interpretation or reliable recovery behavior.

Autonomous Audit Attempt

A separate audit session was prepared to test whether the same model could identify problems without receiving the external reviewer's findings. The original workspace was backed up, and SHA-256 hashes were recorded for six original files. The audit instructions required recalculation of numerical claims and examination of methodology, reproducibility, and recovery, while preserving the original experiment.

The first audit attempt failed when a model request exceeded the configured 180-second timeout. The timeout was increased to 600 seconds, and instructions were added to process the CSV programmatically, return compact summaries, and save audit findings incrementally.

A subsequent attempt failed with a server-disconnection error. Small generation requests then succeeded both directly from the Mac and from the Docker image. These tests confirmed basic connectivity at the time of diagnosis, but did not establish why the larger agent request failed. The audit remained incomplete.

Current Scope and Limitations

The pilot established working local-model integration, containerized tool execution, persistent research files, event logging, and a computational experiment whose reported summary statistics could be checked against its data.

It did not establish autonomous discovery, reliable self-correction, uninterrupted long-duration operation, or improvement from increased execution time. Human assistance was involved in installation, troubleshooting, and external evaluation. The use of a mutable Docker image tag also limits exact reproducibility until its image digest and container environment are recorded.

Future evaluation includes measurement of verified task success, factual and methodological errors, recovery after interruption, resource consumption, and progress across controlled time budgets.