Project 04 — Complete Profile of SWE-agent (Agent Runtime as a System)

Treat a coding agent (default: **SWE-agent**) as a complex system: it is a pipeline of tool calls, subprocesses, IO, and (often) remote model calls. Build a **reproducible benchmark harness** and a **profiling report generator** that explains where time goes, how failures happen, and how resource limits change behavior.

Team size **1–2**, duration **10 weeks**.

> ***Baseline expectation (this course): MS level.***

Scope / constraints

- Must run on **laptop + Ubuntu VM**.
- If you use a remote LLM API, you must:
  - state the model + parameters
  - cap spend (your report should include an explicit cost note)
- If API access is unavailable, you can scope to:
  - a small local model via an adapter (if feasible on CPU), and/or
  - profiling the agent's non-LLM runtime (tool execution, testing, repo operations) on a smaller task set.

Do not claim "agent improved" without controlled evidence.

What you must build

1) A task benchmark pack (required)

- 10–20 tasks that run in your VM.
- Each task must have:
  - input repo + issue statement
  - a success criterion (tests pass / diff matches / oracle output)
  - a timeout budget

The pack can be small; consistency matters more than size.

2) A "complete profile" report generator (required)

For each run, generate a machine-readable log (JSONL is fine) and a summary that includes:

- time breakdown:
  - LLM waiting time (if applicable)
  - tool-call execution time (git, test, build, search)
  - local CPU time
- resource usage:
  - peak RSS, major events (OOM, throttling)
  - context switches (approx via `pidstat -w`)
- outcome taxonomy:
  - success
  - failure modes (looping, oscillation, wrong-file edits, flaky tests, etc.)

3) Resource-bound experiments (required)

Use **cgroup v2** to show how SWE-agent degrades under:

- CPU quota throttling
- memory limits

You must provide an explanation tied to observed signals (not just “it got slower”).

Evidence standard (MS baseline)

For **each** interference experiment and mitigation, you must provide:

1. **Two independent pieces of evidence (cross-layer observations)**
    - At least one must come from the **application/user space** (end-to-end latency percentiles, error rate, throughput, hop-level latency, etc.), and another from the **OS/resource control/K8s control plane** (cgroup/PSI/pidstat/iostat/kubectl events, etc.).
    - “Independent” means the two pieces should not both come from the same tool or the same type of metric (for example, two top screenshots do not count).
    - Examples:
        - App latency histogram/logs + cgroup v2 cpu.stat / memory.current
        - PSI (/proc/pressure/*) + iostat -x / pidstat
        - K8s events (kubectl describe pod) + cgroup stats
2. **One exclusionary control (controlled variable / counterexample)**
    - For the most likely alternative explanation, provide evidence showing that it is not the primary cause.
    - Examples
        - Stable iostat → not an I/O bottleneck
        - No throttling in cpu.stat → not a CPU quota issue
        - Low CPU PSI / low CPU usage in pidstat → not CPU contention
3. **Before/after percentiles + corresponding mechanism-level metric changes**
    - Report p50/p95/p99 (or at least p50/p99).
    - Also show one mechanism-level metric moved consistently with your explanation (e.g., `throttled_usec`, PSI, `await`, `oom_kill`).

Perf PMU counters may not work in VMs; don’t depend on hardware cache events.

Research questions (pick at least 2)

Examples:

- When does the agent spend most time locally vs waiting on the model?
- Which tool calls dominate time, and why (cold caches, dependency installs, test suite shape)?
- Which failure mode is most common under tight resource budgets?
- Does adding a simple policy (timeouts, tool allowlist, retry limits) improve reliability?