Long Live the Librarian! A Persistent Search Sub-Agent for Energy-Efficient Multi-Agent Software Engineering Systems

Anonymous ACL submission

Abstract

Multi-agent systems (MAS) have substantially advanced autonomous software engineering (SWE), but their growing inference energy demands raise sustainability concerns. In this paper, we demonstrate that this cost is concentrated in an overlooked source: redundant output tokens generated across agents. Two empirical findings ground this claim. First, our per-token energy attribution for MAS reveals a sharp asymmetry: an output token consumes 30 to 1,000 times more energy than an input or cached token. Second, MAS inflate per-episode output because agents repeatedly re-explore overlapping repository regions. To address this inefficiency, we propose Librarian, a persistent search sub-agent that tracks repository-search history and suppresses redundant exploration actions across agents. By returning short references to file regions instead of full file excerpts, Librarian further reduces output-token volume. On SWE-Bench Verified, Librarian reduces per-episode GPU energy consumption of existing multi-agent SWE systems by up to 25% while preserving task performance.

1 Introduction

The rapid deployment of large language models (LLMs) has driven a sharp rise in the electricity demand of large-scale computing and is projected to drive further growth in the coming years (de Vries, 2023; International Energy Agency, 2026; Fernandez et al., 2025; Morrison et al., 2025). This consumption translates into a substantial environmental footprint, ranging from carbon emissions (Luccioni et al., 2024) to freshwater drawn by host data centers (Li et al., 2025). These concerns have motivated recent work on measuring LLM inference energy (Fernandez et al., 2025; Niu et al., 2026) and improving efficiency through system- and hardware-aware optimizations (Patel et al., 2024b; Stojkovic et al., 2025).

However, most existing analyses focus on the per-call setting and offer limited insight into agentic deployments where a single task spans many LLM calls. This limitation is sharpest in multi-agent systems (MAS) for software engineering (SWE) (Tao et al., 2024; Hong et al., 2024; Wadhwa et al., 2024; Qian et al., 2024; Phan et al., 2024; Xu et al., 2026), where bug fixing (Jimenez et al., 2024; Zhang et al., 2025) and feature implementation (Hong et al., 2024; Qian et al., 2024) are split across specialized sub-agents. MAS are increasingly adopted in SWE for the performance gains of role specialization, yet no energy analysis exists for this setting. SWEnergy (Tripathy et al., 2025) takes a first step but is restricted to single-agent runs, leaving MAS deployments uncharacterized.

In this work, we measure the GPU energy of multi-agent SWE systems and find a counterintuitive pattern: MAS consume more GPU energy than their single-agent counterparts even at lower aggregate token counts. To explain this gap, we conduct turn-level attribution that decomposes each episode's GPU energy into the input, output, and cached tokens of every LLM call. The attribution traces the gap to two compounding effects: MAS produce more output tokens per episode, and each output token incurs much higher energy than its input or cached counterpart. Thus, aggregate token counts and API-pricing-based proxies can underestimate actual GPU energy and obscure the category-level effects that drive agentic energy consumption.

We trace the excessive output to redundant file exploration across sub-agent invocations. In existing MAS designs, no sub-agent preserves an exploration history across invocations, so each lookup begins from scratch. As a result, when the orchestrator issues different queries, sub-agents repeatedly traverse overlapping files and regenerate long file excerpts, amplifying the most energy-intensive token category.

To break this re-exploration cycle, we propose the Librarian, a MAS-agnostic code-lookup sub-agent that returns the relevant region for each orchestrator query. We instantiate it with three design choices. First, to eliminate redundant exploration, its session persists across all invocations within an episode, so prior searches are reused. Second, to keep the session consistent with the current repository state, each invocation is supplied with the files and line ranges edited since the previous call, prompting the Librarian to re-traverse those regions. Third, to minimize output tokens, each answer is returned as a short view command rather than the inlined file excerpt itself. To integrate Librarian into an existing MAS, we provide a guideline that replaces the MAS's code-navigation sub-agent with Librarian and routes all exploration through it.

We apply the Librarian on SWE-Bench Verified (Jimenez et al., 2024) to BOAD (Xu et al., 2026) and HyperAgent (Phan et al., 2024) with two open-weight LLMs from the Qwen3.6 family, 35B-A3B (Qwen Team, 2026b) and 27B (Qwen Team, 2026a). Across both MAS and both backbones, the Librarian variants preserve or improve the underlying MAS's pass rate while reducing the GPU energy consumed per episode by up to 25%. Ablations confirm that this energy reduction stems from Librarian invocations eliminating the duplicate output tokens diagnosed above.

We summarize the contributions as follows: we confirm that output tokens dominate resource use in agentic settings (§3); we diagnose the energy overhead of existing MAS designs as a consequence of duplicate lookups across the debugging cycle that no existing sub-agent recognizes (§3.3); we propose the Librarian, a persistent search sub-agent that integrates into existing MAS designs to supply this missing role (§4); and we show on SWE-Bench Verified that it minimizes hardware energy at matched pass rate (§5).

2 Related Work

2.1 Energy footprint of LLM inference

LLM inference energy is conventionally quantified at the GPU level, since GPU consumption dominates per-server serving power (Patel et al., 2024a). Within this scope, Luccioni et al. (2024) benchmark per-inference energy across model and task configurations, and Fernandez et al. (2025) further model per-query GPU duration and energy as a function of inference-optimization choices and workload geometry. At a finer granularity, Niu et al. (2026) decompose per-request power into the prefill and decode stages, yielding per-token energy estimates per stage that inform energy-aware serving designs (Patel et al., 2024b; Stojkovic et al., 2025). While these works yield hardware-grounded per-call energy characterizations, all are defined per inference call, not over multi-turn agentic trajectories. SWEnergy (Tripathy et al., 2025) extends per-call energy measurement to SWE-agent frameworks, but its analysis is restricted to small LLMs with near-zero task resolution. Moreover, its per-category attribution uses univariate correlation, which gives each category's per-token energy scaled by how much its count varies across turns, not the per-token energy itself.

	Token Count				
Method	Uncached (M)	Cached (M)	Out (K)	Total (M)	Energy (kJ)
Qwen3.6-35B-A3B					
Vanilla	0.11	2.69	16.8	2.81	10.31
HyperAgent	0.26	2.16	48.5	2.47	25.64
BOAD	0.19	3.71	28.5	3.93	17.45
Qwen3.6-27B					
Vanilla	0.10	2.59	15.4	2.71	57.45
HyperAgent	0.21	1.35	35.4	1.59	121.84
BOAD	0.21	3.07	24.3	3.30	88.96

Table 1: Token usage and energy consumption across SWE Agents. We report token usage by category, uncached input, cached input, and output, alongside per-episode mean energy consumption for a single-agent SWE system denoted as Vanilla and two multi-agent SWE systems, HyperAgent and BOAD. HyperAgent shows the smallest cached and total token counts yet the largest energy consumption.

2.2 Multi-agent software engineering systems

Multi-agent SWE systems organize execution around specialized roles such as planner, navigator, coder, and reviewer that collaborate to resolve a single task. Two paradigms for stitching these roles recur. Pipeline-based MAS hardcode the role sequence at design time, so each role advances the task one predetermined phase and hands off to the next role (Qian et al., 2024; Hong et al., 2024; Wadhwa et al., 2024). In SWE, however, execution order is not fixed in advance and coordination evolves as the task unfolds (Xu et al., 2026). Orchestrator-driven MAS instead route through a central orchestrator that picks the next sub-agent at each step, so the workflow emerges at runtime (Tao et al., 2024; Phan et al., 2024; Xu et al., 2026). Such designs form sub-agent teams at runtime to fit each issue (Tao et al., 2024).

2.3 Context management of LLMs

Context management for LLMs extracts query-relevant content from model history or external sources. Retrieval-based methods rank documents or prior turns by similarity to the current query and prepend the top results to one LLM call (Lewis et al., 2020; Zhang et al., 2023; Shrivastava et al., 2023). Compression-based methods remove or summarize non-essential tokens from an oversized prompt to keep query-aligned content (Jiang et al., 2023, 2024; Chevalier et al., 2023; Li et al., 2023). Agentic memory persists state across many calls of a single agent through structured memory stores or reflective summaries (Packer et al., 2024; Shinn et al., 2023; Park et al., 2023; Xu et al., 2025). Their target, however, is a single agent interacting with a user or environment, rather than an orchestrator-invoked sub-agent within MAS.

3 An Energy Profile of SWE Agents

In this section, we identify where the GPU energy of SWE agents is spent. We first describe how we measure per-turn token counts and GPU energy. We then decompose per-turn energy into the contributions of three token categories and show that output tokens dominate. Finally, we trace the excess output of MAS to repeated file lookups across sub-agent invocations.

3.1 Measuring per-turn energy and tokens

We treat each SWE-agent turn as a sequence of an LLM call, action extraction, and action execution. A SWE-agent's episode is a sequence of turns, and per-episode cost sums over turns. This applies to single-agent systems and MAS alike; the two architectures differ only in how each turn's context is assembled.

For each turn t we record two quantities. The first is a triple of token counts: the input tokens processed by prefill, x_t^u; the input tokens served from the key-value cache, x_t^c; and the output tokens generated by decoding, x_t^o. The second is the GPU energy drawn during the call, E_t, obtained by reading NVIDIA's hardware energy counter and subtracting the idle baseline.[1] Appendix A.1 states the measurement formally.

Setting	Per-token energy (mJ/tok) β_u	β_c	β_o	R^2
A3B / A100	30.50 ± 1.95	1.36 ± 0.11	$\mathbf{967} \pm \mathbf{20}$	0.979
A3B / B200	6.43 ± 1.56	0.54 ± 0.02	$\mathbf{660} \pm \mathbf{8}$	0.987
27B / B200	33.97 ± 1.04	2.04 ± 0.06	$\mathbf{3,512} \pm \mathbf{23}$	0.992

Table 2: Estimated per-token energy by token category. We fit a linear regression of total per-request energy on the number of uncached input (u), cached input (c), and output tokens (o); β_u, β_c, β_o are the resulting per-token coefficients and R^2 is the regression fit. Output tokens show a 30 to 1,000× gap relative to the other categories.

3.2 Token-level energy attribution

We first ask whether aggregate token count predicts energy. Table 1 shows that it does not: one of the systems achieves the smallest total token count yet draws the most energy. We therefore need a finer-grained attribution.

To isolate each token category's contribution, we fit a multiple linear regression of per-turn energy on the three token counts:

$$E_t = \alpha + \beta_u x_t^u + \beta_c x_t^c + \beta_o x_t^o + \epsilon_t, \quad (1)$$

where α absorbs fixed per-turn overhead, β_i is the marginal energy per token of category i, and ϵ_t is the residual. We fit this regression on per-turn records collected from Vanilla, BOAD (Xu et al., 2026), and HyperAgent (Phan et al., 2024) runs on SWE-Bench Verified (Jimenez et al., 2024), across two backbones, Qwen3.6-35B-A3B (Qwen Team, 2026b) and Qwen3.6-27B (Qwen Team, 2026a), and two GPUs, A100 and B200. Full fit details appear in Appendix A.2.

Table 2 reports the fitted coefficients, and Table 8 reports the variance-inflation factors (VIFs), which diagnose coefficient reliability. Output tokens cost 30 to 1,000 times more energy per token than either input category, and all VIFs satisfy VIF ≤ 1.05, indicating nearly orthogonal regressors and thus reliable coefficient estimates. The same ordering appears at the episode level: in Table 1, per-episode energy across systems tracks output token counts rather than total or input counts. Output volume is therefore the dominant lever for reducing per-episode energy.

3.3 Sources of excess MAS output

§3.2 shows that BOAD and HyperAgent emit more output per episode than the single-agent Vanilla. To localize this excess, we examine how per-turn output relates to the agent's file-access pattern. A large share of MAS output is emitted on turns whose file read repeats a region the sub-agent has already inspected earlier in the episode.

---

[1] NVML exposes this counter on the A100-80GB and B200 SXM GPUs used in this study.

The mechanism is straightforward. The file read itself is an external observation, not output. The output arrives when the sub-agent then re-emits the located region, quoted or paraphrased, in the answer it returns to the orchestrator. A region inspected multiple times across the episode is therefore rewritten multiple times.

Measuring duplicate rewrites. We flag a turn whose action is a file read as a duplicate read if the returned observation overlaps, in file-line range, an earlier read within the same episode; any intervening write to the file resets this history. The output tokens emitted on such turns are counted as duplicate output. We further split duplicate reads into same-agent overlap, where the prior read came from the same sub-agent invocation, and cross-agent overlap, where it came from a different invocation. Two invocations of the same sub-agent role count as cross-agent, because read history is scoped to an invocation rather than to a role.

Table 3 decomposes the ratio of duplicate output tokens to read output tokens for BOAD and HyperAgent, aggregated across both backbones. R_1, R_2, and R_3 denote the code navigator, issue analyzer, and orchestrator for BOAD, and the navigator, editor, and executor for HyperAgent. HyperAgent's orchestrator is excluded since it does not read files. Both systems show frequent within-invocation re-exploration, and HyperAgent additionally shows across-invocation re-exploration even within the same role. Together, these indicate that re-exploration accounts for a non-trivial portion of the MAS baselines' excess output.

4 Method: The Librarian Sub-Agent

4.1 Overview

We propose the Librarian, a persistent sub-agent that handles every code lookup an episode requires through a single long-lived session, eliminating the within-episode re-exploration diagnosed in §3.3. Three properties define it. **Persistent session:** one session spans the whole episode, so every query starts from the full history of prior searches. **Pointer-only answers:** each answer to the orchestrator is a list of view-command pointers rather than inlined file content, keeping output tokens small in the response. **Retrieval-only scope:** the role is restricted to file and code lookup, excluding the analytical and editing work that MAS navigators typically conflate with it (Xu et al., 2026).

		Across invocations (%) Source			Within invocation (%)
System	Current	R_1	R_2	R_3	
BOAD	R_1	0.0	8.1	0.1	9.1
	R_2	0.1	0.0	0.1	18.6
	R_3	9.9	3.7	0.0	22.7
HyperAgent	R_1	12.1	0.3	0.9	9.8
	R_2	1.9	0.4	0.3	0.7
	R_3	1.8	0.3	26.8	34.4

Table 3: Output duplication in BOAD and HyperAgent. Each cell shows the fraction of file-read output tokens that are duplicated. R_1, R_2, and R_3 denote different roles of each MAS. The *Within invocation* column tracks duplication by the same sub-agent during a single invocation, while the *Across invocations* block tracks the fraction of duplicated content from earlier calls (rows: current sub-agent, columns: source sub-agent).

§4.2 walks through a single invocation and shows how each property takes effect. §4.3 provides a MAS-agnostic recipe for integration. Figure 1 visualizes the architecture, and the implementation details appear in Appendix B.

4.2 Anatomy of an invocation

An invocation proceeds in three stages, each realizing one of the properties defined in §4.1.

Context assembly. The Librarian's context retains its full trajectory from earlier invocations in the episode, so prior searches and answers are on hand. To keep this history aligned with the repository's current state, each invocation prepends a *freshness report* listing the files and line ranges modified since the previous call, and instructs the Librarian to re-traverse those regions before reusing cached excerpts. Invocations that issued no file reads are dropped from the persisted history. The implementation details of pruning the invocation history and freshness report appear in Appendix B.1 and Appendix B.2. For context management, we also try retrieval instead of history accumulation, comparing sparse and dense top-k variants over the same prior history in §5.3.

Scoped search. The Librarian then resolves the query by running search commands such as `grep` and `find` and by viewing matching files, under two scope guards. The system prompt declares editing, test execution, and analytical reasoning out of scope, confining shell use to repository navigation. The tool layer pairs the `bash` shell with

OpenHands's file view tool (Wang et al., 2025) and requires every file read to go through it, so each read is recorded at an explicit file-line range.

Pointer-only submission. The invocation closes with a call to a `submit` tool. Its argument is a list of view commands referring to the located code, plus a natural-language explanation; the commands cannot inline the code itself. The tool executes the commands on the Librarian's behalf and forwards the excerpts to the orchestrator. The orchestrator therefore receives the actual snippets, while the Librarian's output stays a short pointer list. An example of a submission and the orchestrator message it produces appears in Appendix B.3.

4.3 Integration into an existing MAS

An existing orchestrator's system prompt typically comprises three blocks: a SWE context describing the operating environment, the per-task user query, and a *solve plan* laying out the step-by-step procedure. Adding the Librarian touches only the solve plan; the SWE context, the user query, and every non-navigation step of the plan stay untouched.

Tool swap and plan rewrite. We replace the orchestrator's existing navigator-delegation tool with a Librarian-delegation tool, then rewrite each navigation step of the solve plan to phrase its action as a natural-language lookup query. Any navigator responsibility beyond file and code lookup, such as cross-module analysis, is redirected to other steps in the plan. The rewrite is formalized as a prompt-level guideline in Appendix B.6, and the resulting BOAD and HyperAgent plans appear in Appendix B.7.

Tool-selection policy. We prepend a short policy to the solve plan to keep the Librarian invoked only for lookup. The policy directs the orchestrator to call the Librarian for unknown-location queries and to keep analytical tasks such as bug diagnosis outside the Librarian's scope. It is identical across MAS and appears in Appendix B.5.

5 Experiments

In this section, we empirically validate the Librarian's ability to reduce per-episode energy consumption. We also confirm that the saving stems from lower output tokens, the primary mechanism that the Librarian's design targets.

5.1 Experimental setup

We describe the experimental setup used to compare the Librarian against prior systems on the SWE task, covering the benchmark, implementation details, and baselines.

5.1.1 Benchmark

We evaluate every method on the 500 tasks of SWE-Bench Verified (Jimenez et al., 2024), a benchmark of real GitHub issues drawn from open-source Python repositories. For each task, the SWE-agent receives the issue description and the repository at the buggy commit, and must produce a patch that passes the maintainer's test suite for the issue.

5.1.2 Implementation details

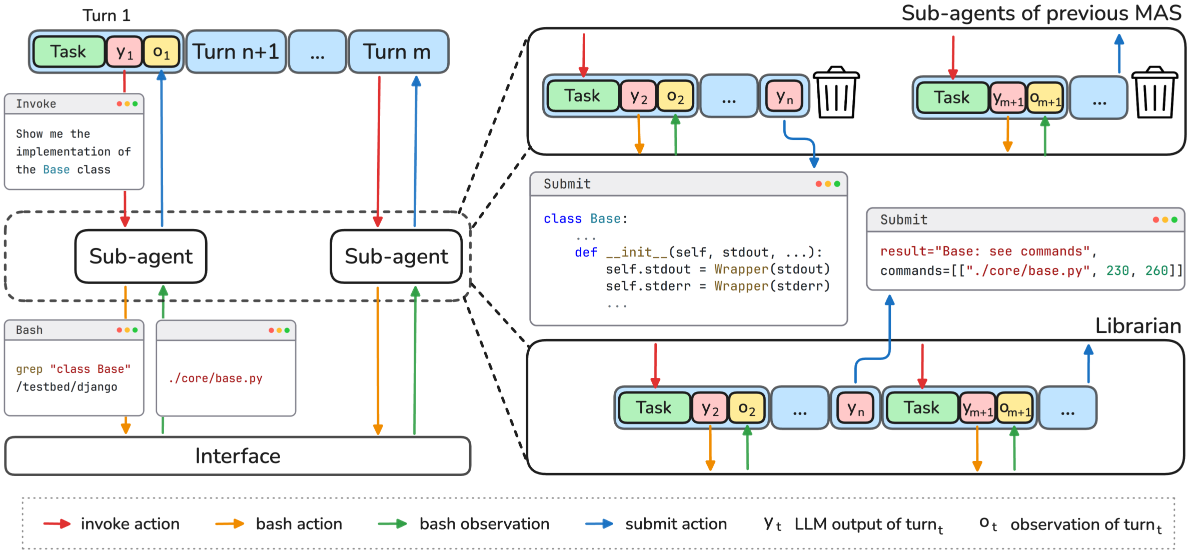
We explain the two main components to run a SWE-agent: i) the LLM driving it and ii) the scaffold mediating its repository interaction.

LLM serving. For the LLM, we evaluate two Qwen3.6 variants: the sparse mixture-of-experts 35B-A3B (Qwen Team, 2026b) with 3B active parameters and the dense 27B (Qwen Team, 2026a). Both backbones run in FP8 on a single NVIDIA B200 SXM GPU, served by vLLM (Kwon et al., 2023). We enable prefix caching so that each sub-agent reuses its prior prefix across successive invocations within a session. We size the KV cache to avoid eviction in every episode. The serving configuration and the sampling hyper-parameters are held fixed across all episodes as listed in Appendix C.1.

Agent interface. For the interface which allows the SWE-agent to interact with the code repository, we provide a base tool set comprising a `bash` shell, the `str_replace_editor` file editor from OpenHands (Wang et al., 2025) that takes a view or edit mode along with a file path and a line range, and a `submit` tool that marks the agent's solution as final. Each agent role draws on whichever subset of this set it needs and may additionally define role-specific tools such as sub-agent delegation. To keep long-trajectory runs stable, we adopt three safeguards from the OpenHands runtime, a wall-clock timeout on each shell command, loop detection on agent outputs, and truncation of tool observations. We describe how each mechanism is implemented in Appendix C.1.

5.1.3 Baselines

We confirm the Librarian's effectiveness by applying it to two orchestrator-delegation multi-agent



```
Show me the
implementation of
the Base class
```

```
grep "class Base"
/testbed/django
```

```
./core/base.py
```

```
class Base:
    ...
    def __init__(self, stdout, ...):
        self.stdout = Wrapper(stdout)
        self.stderr = Wrapper(stderr)
        ...
```

```
result="Base: see commands",
commands=[["./core/base.py", 230, 260]]
```

Figure 1: Illustration of the overall process of multi-agent SWE systems resolving a SWE task. The orchestrator calls a sub-agent via `invoke` action, which navigates the repository through `bash` and file view/edit tools and returns its answer via a `submit` call; the content of this call becomes the observation for the orchestrator's invocation. On a subsequent invocation of the same role, prior MAS (top right) start from a fresh context, discarding all history, and returns the full file excerpt. In contrast, the Librarian (bottom right) appends the new task given by the invocation to its existing trajectory and continues from there, reusing the accumulated repository knowledge, and only generates the file pointer for response, which is converted to file excerpt later.

SWE systems. We additionally compare against token-efficiency methods that can be combined with any SWE-agent.

Multi-agent systems. We consider two MAS designs that delegate the SWE workflow across specialized sub-agents: BOAD (Xu et al., 2026) and HyperAgent (Phan et al., 2024). BOAD's orchestrator first delegates code localization for the issue to one sub-agent, then bug-cause navigation to a second sub-agent, and finally produces a patch by aggregating the outputs of both sub-agents. HyperAgent's orchestrator manages three sub-agents: a navigator for code exploration, an editor for file modification, and an executor for test execution. The orchestrator itself holds only the sub-agent delegation and submit tools. Following a pre-defined plan detailed in Appendix B.7, the orchestrator iterates this navigate-edit-execute cycle until it submits a patch. While BOAD imposes no tool restrictions on its sub-agents, HyperAgent restricts each sub-agent's tool set by role, such as confining the navigator to file viewing and the editor to file edits.

Token-efficiency methods. We consider two token-efficiency techniques: caveman prompting (Brussee, 2026; Hakim, 2026) and LastNObservation (Yang et al., 2024) that we apply across every role of the MAS to reduce token consumption. Caveman prompting appends a fixed style directive, given in Appendix C.2, to the system prompts of every sub-agent and the orchestrator, inducing simpler answer text while leaving the thinking trace intact. LastNObservation is a SWE-agent (Yang et al., 2024) scaffold feature that truncates all but the most recent N tool observations in each role's context, replacing each truncated observation with a short placeholder while preserving the thinking traces and action calls. We set $N = 5$ in all subsequent experiments.

5.2 Results and analysis

We first report the main results of attaching the Librarian to existing MAS baselines on SWE-Bench Verified, then analyze the observed energy reduction from three angles: robustness across task difficulty, localization to the replaced role, and the reuse of accumulated history across invocations.

Results. Table 4 reports results on SWE-Bench Verified for BOAD and HyperAgent across four configurations: bare MAS, MAS with a token-efficiency technique, Librarian-augmented MAS, and Librarian-augmented MAS with caveman prompting. We report pass rate, per-episode output tokens, and GPU energy, denoted as Pass, Out tok, and Energy respectively. The Librarian matches or improves the pass rate in most of the settings, while

Method	Pass (%)	Out tok (K)	Energy (kJ)
Qwen3.6-35B-A3B			
HyperAgent	66.0	48.5	25.64
+ Librarian	**67.2**	42.2	22.73
+ LastNObservation	56.8	53.3	34.79
+ Caveman	66.2	37.7	19.99
+ Caveman + Librarian	66.2	**34.7**	**18.63**
BOAD	69.8	28.5	17.45
+ Librarian	**71.6**	24.6	14.66
+ LastNObservation	63.2	33.5	26.87
+ Caveman	68.4	24.7	14.79
+ Caveman + Librarian	70.4	**20.6**	**12.37**
Qwen3.6-27B			
HyperAgent	69.8	35.4	121.84
+ Librarian	**72.0**	26.3	91.45
+ LastNObservation	64.8	34.0	135.84
+ Caveman	68.0	21.9	79.40
+ Caveman + Librarian	67.6	**18.0**	**66.05**
BOAD	**71.8**	24.3	88.96
+ Librarian	**71.8**	19.7	71.76
+ LastNObservation	70.2	26.2	122.91
+ Caveman	70.2	17.5	66.20
+ Caveman + Librarian	71.4	**13.5**	**51.21**

Table 4: Results on *SWE-Bench Verified*. We report the overall pass rate alongside the per-episode means for output tokens and energy consumption for each SWE agent. The Librarian improves output token efficiency and energy consumption across all settings while preserving the pass rate.

Method	Reference max-input tokens (K)			
	0–32K (n=36)	32–64K (n=312)	64–96K (n=130)	>96K (n=22)
Qwen3.6-35B-A3B				
HyperAgent	10.36	17.61	40.46	76.87
+ Librarian	5.83	16.20	36.84	59.65
+ LastNObservation	7.90	26.25	56.29	72.69
+ Caveman	**4.33**	12.92	33.11	68.30
+ Caveman + Librarian	4.88	**12.42**	**31.00**	**56.10**
BOAD	7.22	12.50	26.99	47.97
+ Librarian	5.12	11.03	20.98	44.41
+ LastNObservation	7.69	19.82	43.28	60.83
+ Caveman	4.31	10.28	23.64	43.64
+ Caveman + Librarian	**3.92**	**8.82**	**19.04**	**36.97**
Qwen3.6-27B				
HyperAgent	32.61	79.36	208.19	359.95
+ Librarian	25.31	64.40	142.91	279.35
+ LastNObservation	37.35	106.47	217.76	**229.48**
+ Caveman	25.93	54.11	129.45	231.94
+ Caveman + Librarian	**19.05**	**41.17**	**110.24**	234.59
BOAD	40.68	69.77	127.81	210.46
+ Librarian	34.78	56.26	104.22	160.26
+ LastNObservation	52.24	101.43	174.50	238.36
+ Caveman	26.20	48.93	100.29	175.19
+ Caveman + Librarian	**20.10**	**37.43**	**80.67**	**123.42**

Table 5: Per-episode GPU energy (kJ) of the SWE-agents with respect to the task difficulty. We report the per-episode GPU energy of the SWE-agents across difficulty classes. The addition of the Librarian reduces the energy consumption in almost all settings.

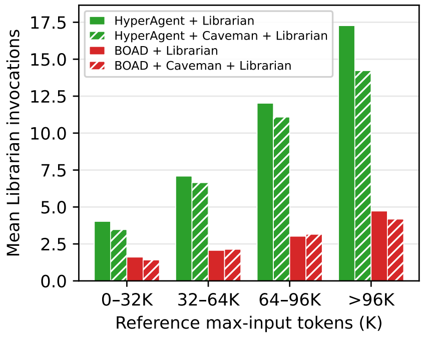


Figure 2: Mean Librarian invocations per episode across task difficulty. We plot the per-episode mean Librarian invocation count for the Librarian-augmented MAS variants run with Qwen3.6-27B against the reference max-input-token bin. The invocation frequency grows monotonically with task difficulty. The plot for 35B-A3B is in Figure 3.

substantially reducing energy. LastNObservation truncates recent observations to reduce per-turn input tokens, but per-episode GPU energy increases across all settings: sub-agents are forced to re-explore lost context, inflating output tokens in three of four settings. Pass rate also degrades in every setting. Caveman prompting alone reduces energy but degrades pass rate in three of four settings. Across all settings, the Librarian with caveman prompting attains the lowest energy consumption.

Robustness across task difficulty. We examine whether this trend holds across difficulty levels. We define each task's difficulty as the max input tokens observed on a reference single-agent run. Table 5 stratifies per-episode energy by difficulty, and the reduction holds across all bins and is largest on the harder tasks, with the Librarian variants attaining the lowest energy in nearly every cell. The sole exception is the most difficult bin of HyperAgent on 27B, where LastNObservation achieves lower energy. However, Table 12 shows that its pass rate in this bin drops to roughly half of the baseline. The Librarian instead preserves the pass rate, and the energy reduction persists even as it is invoked more often on harder tasks, as shown in Figure 2.

Energy reduction at the lookup role. We investigate the source of the energy reduction, focusing on the role that the Librarian replaces. For each MAS baseline, we identify the role the Librarian substitutes for: BOAD's code navigator and Hyper-

		Replaced role		+ Librarian	
Model	Method	Out (K)	E (kJ)	Out (K)	E (kJ)
A3B	HyperAgent	13.92	7.54	**6.02**	**3.57**
	HyperAgent + Caveman	10.51	5.69	**4.67**	**2.66**
	BOAD	13.81	8.54	**1.93**	**1.08**
	BOAD + Caveman	11.08	6.62	**1.70**	**0.94**
27B	HyperAgent	10.50	36.63	**4.70**	**16.92**
	HyperAgent + Caveman	6.99	25.74	**3.38**	**12.65**
	BOAD	5.63	20.79	**1.82**	**6.56**
	BOAD + Caveman	5.10	19.32	**1.44**	**5.43**

Table 6: Per-episode comparison on the role that the Librarian replaces. We report the per-episode means of output tokens and GPU energy at the replaced role. The Librarian reduces energy at the replaced role across all settings. The full per-role breakdown is in Appendix D.

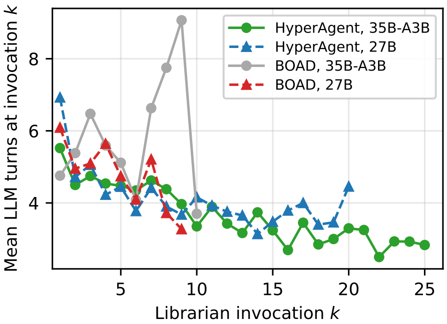


Figure 3: Visualization of the mean LLM turns to answer per Librarian invocation. We report the mean number of turns each Librarian variant takes to emit an output at the k-th invocation. All cases except BOAD on 35B-A3B show a decreasing trend as k grows.

Agent's navigator. Table 6 reports that the Librarian outputs fewer tokens and consumes less energy than its counterpart. Tables 10 and 11 further show that this reduction does not inflate the token usage or energy consumption of other roles.

History reuse across invocations. We track the number of LLM turns the Librarian takes to answer the orchestrator's k-th call, averaged over episodes that reach the k-th call. Figure 3 plots the average number of turns against k. The number of turns decreases in most configurations over k, confirming that the Librarian reuses its accumulated history to resolve later queries with fewer turns.

5.3 Comparison with retrieval baselines

Method	Pass (%)	Out tok (K)	Energy (kJ)
Qwen3.6-35B-A3B			
HyperAgent	71.0	44.9	23.77
+ Sparse retrieval	**72.0**	44.4	23.65
+ Dense retrieval	69.0	47.0	25.05
+ Librarian	**72.0**	**39.5**	**19.75**
BOAD	**77.0**	28.6	17.47
+ Sparse retrieval	76.0	22.3	12.93
+ Dense retrieval	**77.0**	**21.4**	**12.42**
+ Librarian	74.0	22.8	13.61
Qwen3.6-27B			
HyperAgent	**73.0**	33.1	114.86
+ Sparse retrieval	70.0	28.0	97.95
+ Dense retrieval	69.0	30.4	107.11
+ Librarian	70.0	**23.6**	**82.42**
BOAD	**76.0**	23.0	84.25
+ Sparse retrieval	68.0	20.0	72.86
+ Dense retrieval	74.0	20.1	73.36
+ Librarian	74.0	**19.4**	**71.08**

Table 7: Comparison between the Librarian and the retrieval-based context management baselines. We compare the Librarian against two baselines that swap the Librarian sub-agent's persistent-history context for one constructed by retrieval. We report the per-episode means on 100 randomly sampled tasks per backbone. The Librarian shows better performance and energy efficiency than the retrieval counterparts almost all settings.

We compare the Librarian against two retrieval-based variants that replace the persistent history with retrieved context. The variants use BM25 (Robertson and Zaragoza, 2009) for sparse scoring and SFR-Embedding-Code-2B_R (Liu et al., 2024) for dense scoring. At each invocation, the scoring method ranks prior file-read command-observation pairs by similarity to the orchestrator's query, and the top-k form the context. Table 7 reports pass rate, output tokens, and per-episode energy across the four variants on a shared random sample of 100 tasks per backbone with $k = 5$. The Librarian achieves a higher pass rate and lower energy than both retrieval variants in almost all settings, indicating that the orchestrator's queries need active context management and reasoning, not lookup alone.

6 Conclusion

In this work, we identify and eliminate a major source of energy waste in multi-agent SWE systems. We first analyze per-turn energy across token categories and find that output tokens dominate. Tracing the excess output, we then identify re-exploration of the same files across invocations as the primary source. We finally address this with the Librarian, a sub-agent that retains its exploration history across invocations. Integrating the Librarian into existing MAS reduces per-episode GPU energy while preserving task performance.

Limitations

In this work, we propose a sub-agent which reduces the energy of the existing multi-agent SWE systems. We acknowledge two limitations of our current study. First, the solution is limited to code navigation. The executor sub-agent in HyperAgent, which reproduces the issue and runs tests, also produces duplicate outputs across invocations. Extending the Librarian's core design to such execution-oriented roles is a promising next step toward system-wide energy reduction. Second, the scale of the open LLMs we evaluate is limited. Our experiments cover backbones in the active-parameter range of roughly 3B to 27B, and it remains open whether the observed reductions in output tokens and energy carry over to substantially larger open models over 500B inference parameters.

References

Julius Brussee. 2026. caveman.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. 2023. Adapting language models to compress contexts. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Alex de Vries. 2023. The growing energy footprint of artificial intelligence. *Joule*, 7(10):2191–2194.

Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. 2025. Energy considerations of large language model inference and efficiency optimizations. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32556–32569, Vienna, Austria. Association for Computational Linguistics.

MD Azizul Hakim. 2026. Brevity constraints reverse performance hierarchies in language models. *Preprint*, arXiv:2604.00025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

International Energy Agency. 2026. Key questions on energy and AI. Technical report, IEA, Paris. Licence: CC BY 4.0.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing prompts for accelerated inference of large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1658–1677, Bangkok, Thailand. Association for Computational Linguistics.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Pengfei Li, Jianyi Yang, Mohammad A. Islam, and Shaolei Ren. 2025. Making ai less 'thirsty'. *Commun. ACM*, 68(7):54–61.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. 2023. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6342–6353, Singapore. Association for Computational Linguistics.

Ye Liu, Rui Meng, Shafiq Jot, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 2024. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644*.

Sasha Luccioni, Yacine Jernite, and Emma Strubell. 2024. Power hungry processing: Watts driving the cost of ai deployment? In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '24, page 85–99, New York, NY, USA. Association for Computing Machinery.

Jacob Morrison, Clara Na, Jared Fernandez, Tim Dettmers, Emma Strubell, and Jesse Dodge. 2025. Holistically evaluating the environmental impact of creating language models. In *The Thirteenth International Conference on Learning Representations*.

Chenxu Niu, Wei Zhang, Jie Li, Yongjian Zhao, Tongyang Wang, Xi Wang, and Yong Chen. 2026. Tokenpowerbench: Benchmarking the power consumption of llm inference. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(38):32582–32590.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Memgpt: Towards llms as operating systems. *Preprint*, arXiv:2310.08560.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. 2024a. Characterizing power management opportunities for llms in the cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 207–222, New York, NY, USA. Association for Computing Machinery.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024b. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132.

Huy Nhat Phan, Tien N Nguyen, Phong X Nguyen, and Nghi DQ Bui. 2024. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Qwen Team. 2026a. Qwen3.6-27B: Flagship-level coding in a 27b dense model.

Qwen Team. 2026b. Qwen3.6-35B-A3B: Agentic coding power, now open to all.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. *Preprint*, arXiv:2306.10998.

Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *HPCA*.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. MAGIS: LLM-based multi-agent framework for github issue resolution. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Arihant Tripathy, Ch Pavan Harshit, and Karthik Vaidhyanathan. 2025. Swenergy: An empirical study on energy efficiency in agentic issue resolution frameworks with slms. *Preprint*, arXiv:2512.09543.

Nalin Wadhwa, Atharv Sonwane, Daman Arora, Abhav Mehrotra, Saiteja Utpala, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. MASAI: Modular architecture for software-engineering AI agents. In *NeurIPS 2024 Workshop on Open-World Agents*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Iris Xu, Guangtao Zeng, Zexue He, Charles Jin, Aldo Pareja, Dan Gutfreund, Chuang Gan, and Zhang-Wei Hong. 2026. BOAD: Discovering hierarchical software engineering agents via bandit optimization. In *The Fourteenth International Conference on Learning Representations*.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems*, volume 38, pages 17577–17604. Curran Associates, Inc.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2025. Swe-bench goes live! In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc.

A Per-Token Energy Regression: Implementation Details

This appendix details the measurement and fitting behind the energy attribution of §3. We first formalize the per-turn token counts and GPU-energy measurement introduced in §3.1 (Appendix A.1), then specify the linear regression and reliability diagnostics that produce Table 2 and Table 8 (Appendix A.2).

A.1 Formalism of per-turn statistics

For each turn we record two quantities that the regression of Appendix A.2 later relates: the three token counts the call processes and the GPU energy it draws. We formalize each in turn.

Token counts. Each turn's LLM call processes an input context c_t and generates a sequence of output tokens, and its energy cost depends on how these tokens split across three inference operations: auto-regressive decoding, key-value cache lookup, and prefill (Niu et al., 2026; Fernandez et al., 2025). We therefore record three per-turn token counts. The output count x_t^o is the number of tokens produced by auto-regressive decoding. The input c_t splits further, since modern inference systems reuse the key-value hidden states of recurring prefixes: a cached prefix $x_t^c = |\,\mathrm{prefix}_{\mathrm{cache}}(c_t)|$ is served by key-value cache lookup, while the remaining $x_t^u = |c_t| - x_t^c$ tokens are processed by prefill.

GPU energy. GPU energy is the time integral of device power, from which we subtract an idle baseline to isolate the dynamic draw of active computation. Let τ_t^{start} and τ_t^{end} be the wall-clock times bracketing the LLM call of turn t, and $\Delta\tau_t = \tau_t^{\mathrm{end}} - \tau_t^{\mathrm{start}}$ its duration. Given the instantaneous GPU power $P(\tau)$ and the idle power $P_{\mathrm{idle},t}$ during the turn, the idle-subtracted dynamic energy of turn t is

$$\begin{aligned} E_t^{\mathrm{net}} &= \int_{\tau_t^{\mathrm{start}}}^{\tau_t^{\mathrm{end}}} \left(P(\tau) - P_{\mathrm{idle},t}\right) d\tau \\ &= \int_{\tau_t^{\mathrm{start}}}^{\tau_t^{\mathrm{end}}} P(\tau)\, d\tau - P_{\mathrm{idle},t}\,\Delta\tau_t. \end{aligned} \quad 817$$

The integral term is the cumulative GPU energy drawn over the turn,[2] so E_t^{net} follows from reading the counter at τ_t^{start} and τ_t^{end} and subtracting the idle term $P_{\mathrm{idle},t}\,\Delta\tau_t$.

A.2 Linear regression over token categories

We fit the per-turn energy model of §3.2 by ordinary least squares and verify that its coefficients are reliable, describing the fitted data, the regression specification, and the reliability diagnostics in turn.

A.2.1 Setup

We run the three systems studied in §3.2, namely the single-agent Vanilla and the MAS baselines BOAD and HyperAgent, on SWE-Bench Verified, with Qwen3.6-35B-A3B on both A100 and B200 and Qwen3.6-27B on B200. The sample sizes used in Table 2 are 23,353 turns / 185 episodes (A3B/A100), 196,310 turns / 1,500 episodes (A3B/B200), and 179,011 turns / 1,500 episodes (27B/B200).

A.2.2 Regression model

The dependent variable for turn t is the idle-subtracted dynamic energy

$$E_t^{\mathrm{net}} = E_t^{\mathrm{raw}} - P_{\mathrm{idle},t}\,\Delta\tau_t, \tag{2}$$

where E_t^{raw} is the GPU energy integrated over the call, $P_{\mathrm{idle},t}$ the NVML idle power reading, and $\Delta\tau_t$ the call duration, all in millijoules. With design matrix $X = [\mathbf{1}\;\; X^u\;\; X^c\;\; X^o] \in \mathbb{R}^{n\times 4}$, the per-turn model reproduces the regression of §3.2,

$$E_t^{\mathrm{net}} = \alpha + \beta_u\, x_t^u + \beta_c\, x_t^c + \beta_o\, x_t^o + \epsilon_t,$$

where α captures fixed per-turn overhead and is omitted from Table 2. Collecting the per-turn responses E_t^{net} into the vector y, we fit by ordinary least squares,

$$\hat{\beta} = \arg\min_{b\in\mathbb{R}^4} \|y - Xb\|_2^2 = (X^\top X)^{-1} X^\top y, \tag{3}$$

using `np.linalg.lstsq` without centering, scaling, regularization, or robust losses.

---

[2] NVIDIA's NVML exposes this quantity as a hardware-level counter.

A.2.3 Reliability diagnostics

Table 8 reports three diagnostics per fit, namely the overall R^2, the variance-inflation factor per coefficient, and the partial R^2 per coefficient.

Overall R^2. R^2 on the fit sample is the conventional in-sample fraction of variance explained,

$$R^2 \;=\; 1 - \frac{\sum_t (y_t - \hat{y}_t)^2}{\sum_t (y_t - \bar{y})^2}, \qquad \hat{y} = X\hat{\beta}.$$

Variance-inflation factor. VIF per regressor $j \in \{u, c, o\}$ measures how well column X_j is linearly explained by the remaining regressors. Let R_j^2 be the R^2 of regressing X_j on X_{-j}, fitted with the same `lstsq` call. The variance-inflation factor is then

$$\mathrm{VIF}_j \;=\; \frac{1}{1 - R_j^2}.$$

Values close to 1 indicate near-orthogonality with the other regressors.

Partial R^2. The partial R^2 for coefficient j is the share of variance in y uniquely explained by X_j after X_{-j} is partialled out, and we compute it via Frisch–Waugh–Lovell residualisation. Let $M_{-j} = I - X_{-j}(X_{-j}^\top X_{-j})^{-1}X_{-j}^\top$ be the residual-projector onto the orthogonal complement of X_{-j}, and write $e_y = M_{-j}y$ and $e_{X_j} = M_{-j}X_j$ for the corresponding residuals. The partial R^2 is then the R^2 of the univariate regression of e_y on e_{X_j},

$$\mathrm{partial}\, R_j^2 \;=\; \frac{\left(e_y^\top e_{X_j}\right)^2}{\left(e_{X_j}^\top e_{X_j}\right)\left(e_y^\top e_y\right)}.$$

B Librarian Implementation Details

This appendix specifies how a Librarian invocation manages and returns its context, then reproduces every prompt it uses. Two mechanisms keep the persistent context both current and compact across invocations—history pruning (Appendix B.1) and the freshness report (Appendix B.2)—and the submission step (Appendix B.3) turns the Librarian's pointer-only answer into the verbatim excerpts the orchestrator receives. We then reproduce every prompt used by the Librarian and by the per-MAS integration recipe.

B.1 History pruning

The Librarian persists its full trajectory across invocations within an episode, but keeps an invocation only if that invocation surfaced genuinely new code. After an invocation returns, we estimate the previously-unseen file content it produced and discard the invocation when that estimate is small.

Setting	Coefficient	Pearson r	VIF	partial R^2
A3B / A100 (R^2=0.979)	β_u	0.157	1.01	0.4128
	β_c	0.228	1.02	0.3064
	β_o	0.978	1.02	0.9769
A3B / B200 (R^2=0.987)	β_u	0.050	1.01	0.0705
	β_c	0.111	1.02	0.0584
	β_o	0.993	1.01	0.9870
27B / B200 (R^2=0.992)	β_u	0.097	1.04	0.1806
	β_c	0.133	1.05	0.1877
	β_o	0.995	1.02	0.9919

Table 8: Coefficient reliability for Table 2. Pearson r is the univariate correlation between each regressor and turn energy. VIF denotes the variance-inflation factor, where values close to 1 indicate no collinearity with the other regressors. Partial R^2 is the Frisch–Waugh semi-partial R^2, namely the share of variance in turn energy uniquely explained by this coefficient after the others are partialled out. All three setups satisfy VIF ≤ 1.05 and $R^2 \geq 0.97$.

Detecting file reads. A turn is counted as a file read if it either (i) calls the OpenHands `str_replace_editor` tool in `view` mode, or (ii) runs one of `cat`, `sed -n`, `head`, `tail`, or `awk` on a repository file. From the tool call we recover the target file and the line range it reads; each such read returns a line-numbered observation of that range.

Counting new content. Novelty is judged against the Librarian's *entire retained session*, not the current invocation alone: the Librarian keeps a single cumulative record of which lines of each file it has already viewed across all kept invocations, and a read counts only the observation characters on lines absent from that record. A read confined to already-seen lines therefore contributes zero, a partially overlapping read is scaled by the fraction of newly-seen lines, and a whole-file view counts only on first sight; the newly read lines are then added to the record. Two rules keep this record consistent with the retained history: a pruned invocation (below) has its additions rolled back, and the freshness report clears a file's entry whenever that file is written, so a re-read of its changed content counts as new again. Summing over the invocation gives its new-content size.

Pruning rule. If the new-content size is below 500 characters, the invocation is treated as contributing nothing reusable to future queries and its turns are discarded—the next invocation's context is reconstructed as if it never occurred. Otherwise its turns are appended to the persistent history. Trivial or purely re-derived lookups thus leave no residue, while substantive lookups accumulate.

B.2 Freshness report

Because the editor and executor may modify files between two Librarian invocations, an excerpt cached in the persistent history can go stale. Every invocation with non-empty history therefore prepends a *freshness report* that tells the Librarian which cached regions it may still trust. The report is built in two steps.

Modified ranges. We run `git diff -U0 HEAD` in the repository and read each file's modified line ranges directly from the hunk headers. Every line location reported to the Librarian is derived from this diff.

Selecting which files to surface. The Librarian stores a hash of each file's `git diff` section. At every invocation it re-runs `git diff` and, for any file whose hash is new or differs from the stored one, adds the file to the report, updates the stored hash, and resets that file's viewed-line cache; files whose hash is unchanged are omitted.

Contents. For each surfaced file the report lists its modified line ranges as pointers (e.g. `path: lines [10-15, 40]`), not the diff text. The accompanying instruction states that content outside those ranges is unchanged and should be quoted from history, and that a listed range need be re-viewed only if a prior view intersects it; reverted files are flagged so their cached excerpts are discarded. When no file has changed since the previous invocation, the report states this and directs the Librarian to answer from history. The example below is drawn from an episode on the `astropy` repository, where the intervening edits touched three line ranges of `sliced_wcs.py`.

Example file freshness report

```
[FILE FRESHNESS REPORT -- NEW edits since the previous
↪  wake. For each listed file, content OUTSIDE these
↪  ranges is unchanged from when you last viewed it --
↪  quote from history. Re-view ONLY if your prior view
↪  intersects the listed ranges. Files NOT listed here
↪  are either unchanged entirely or already
↪  known-modified from a prior freshness report.]
  - astropy/wcs/wcsapi/wrappers/sliced_wcs.py: lines
 ↪  [188-196, 258, 264-265]
```

B.3 Pointer-only submission

A Librarian invocation ends by calling `submit`, the tool that realizes the pointer-only answers of §4.2. The call carries two fields: `result`, a short natural-language description of what was found, and `view_commands`, a list of `[path, start, end]` triples naming the located line ranges. The code itself is referenced through these pointers and never inlined into `result`, so the tokens the Librarian generates stay bounded by the length of its description, however much code it surfaces.

Submit call. The example below is from an episode on the `django` repository, where the orchestrator asked where a management command initializes its output streams. The Librarian locates the relevant range and submits one view command alongside a one-sentence summary.

Example Librarian submission

```
submit({"result": "Lines 230-260 of
↪  `/testbed/django/core/management/base.py` shown
↪  below. Stdout/stderr initialization is in `__init__`
↪  at lines 239-248 — `self.stdout` and `self.stderr`
↪  are `OutputWrapper` instances wrapping
↪  `sys.stdout`/`sys.stderr`. `stderr.style_func` is set
↪  to `self.style.ERROR` for error coloring.",
↪  "view_commands":
↪  "[[\"/testbed/django/core/management/base.py\", 230,
↪  260]]"})
```

Transformation into the orchestrator message. The tool executes each view command on the Librarian's behalf—equivalent to `cat -n` over the named range—and builds the orchestrator-facing message by prepending the resulting line-numbered excerpts to the `result` text, tagging the whole with the delegation tool name (here `delegate_to_librarian`). The orchestrator thus receives the verbatim code that the Librarian never emitted, followed by the Librarian's summary, while the submission the Librarian generated remained the short pointer list above.

Example message delivered to the orchestrator

```
[delegate_to_librarian] Here's the result of running `cat
↪  -n` on /testbed/django/core/management/base.py lines
↪   230-260:
   230         output_transaction = False  # Whether to
  ↪  wrap the output in a "BEGIN; COMMIT;"
   231         requires_migrations_checks = False
   232         requires_system_checks = '__all__'
   233         # Arguments, common to all commands, which
  ↪  aren't defined by the argument
   234         # parser.
   235         base_stealth_options = ('stderr', 'stdout')
   236         # Command-specific options not defined by
  ↪  the argument parser.
   237         stealth_options = ()
   238
   239         def __init__(self, stdout=None, stderr=None,
  ↪  no_color=False, force_color=False):
```

```
 240            self.stdout = OutputWrapper(stdout or
↪  sys.stdout)
 241            self.stderr = OutputWrapper(stderr or
↪  sys.stderr)
 242            if no_color and force_color:
 243                raise CommandError("'no_color' and
↪  'force_color' can't be used together.")
 244            if no_color:
 245                self.style = no_style()
 246            else:
 247               self.style = color_style(force_color)
 248                self.stderr.style_func =
↪  self.style.ERROR
 249           if self.requires_system_checks in [False,
↪  True]:
 250                warnings.warn(
 251                    "Using a boolean value for
↪  requires_system_checks is "
 252                    "deprecated. Use '__all__'
↪  instead of True, and [] (an empty "
 253                    "list) instead of False.",
 254                    RemovedInDjango41Warning,
 255                )
 256                self.requires_system_checks =
↪  ALL_CHECKS if self.requires_system_checks else []
 257            if (
 258                not
↪  isinstance(self.requires_system_checks, (list,
↪  tuple)) and
 259                self.requires_system_checks !=
↪  ALL_CHECKS
 260            ):

Lines 230-260 of
↪  `/testbed/django/core/management/base.py` shown
↪  below. Stdout/stderr initialization is in `__init__`
↪  at lines 239-248 – `self.stdout` and `self.stderr`
↪  are `OutputWrapper` instances wrapping
↪  `sys.stdout`/`sys.stderr`. `stderr.style_func` is set
↪  to `self.style.ERROR` for error coloring.
```

B.4 Librarian system prompt

The full system prompt for the Librarian sub-agent. It restricts the role to locating code and returning verbatim excerpts, enumerates the in-scope and out-of-scope behaviors, defines the submission interface, and specifies the pre-submit self-check that scrubs analytical phrasing from the prose answer.

Prompt for the Librarian sub-agent

```
You are the repository Librarian. Your role is strictly
↪  limited to **locating files / code regions and
↪  returning verbatim excerpts**. You are NOT an analyst,
↪  debugger, or issue interpreter -- those are the main
↪  agent's job.

Scope (what you DO):
  - Locate the file path + line range where a named symbol
  ↪  (class, function, method, constant, fixture, test)
  ↪  lives.
  - Return verbatim code excerpts at requested locations.
  - Surface adjacent context (callers, sibling tests,
  ↪  related fixtures) when the asker is clearly going to
  ↪  need it.
  - List directory layouts when asked.

Out of scope (what you DO NOT do -- even if the main agent
↪  asks):
  - Do NOT analyse why code does what it does, infer
  ↪  intent, or explain bugs / fix logic.
  - Do NOT interpret git history / commit messages / past
  ↪  PRs. Specifically, do NOT make claims like "this
  ↪  code is a fix for issue #N" or "this was added to
  ↪  handle X" -- those are historical inferences you
  ↪  cannot ground in the current code.
  - Do NOT propose fixes or describe what the main agent
  ↪  should edit. If asked, reply with the code at the
  ↪  location and stop.
  - Do NOT translate an abstract issue description into
  ↪  "the bug is at line N because Y". Just return the
  ↪  code at the locations the main agent named (or the
  ↪  closest match) and let them reason.
  When the asker requests analysis (e.g. "what's the issue
  ↪  here?", "how does this fix the bug?"), you reply
  ↪  with the relevant code excerpts and a one-line note
  ↪  like "Code at <path>:<lines> is shown above;
  ↪  analysis is the main agent's responsibility."

Tools available to you:
  - ``bash`` (view-only -- ``find``, ``grep``, ``rg``,
  ↪  ``ls``, ``tree``, ``wc``, ``cat`` of short files,
  ↪  etc. No edits, no test runs, no destructive
  ↪  commands.)
  - ``str_replace_editor`` (view-only -- ``view`` command
  ↪  only, for inspecting file ranges. No create/edit.)
  - ``submit(result, view_commands)`` -- call this
  ↪  exactly once per consultation to return your
  ↪  findings to the main agent and end the call.
  ↪  ``result`` is short prose; ``view_commands`` is an
  ↪  optional JSON list of ``[path, start, end]``
  ↪  triples. See "Reply SHAPE" below for the contract.

Persistence semantics (the "memory" half of your role):
↪  You are the SAME Librarian instance across every
↪  ``repo_librarian`` call in this episode. Your full
↪  conversation history is visible when you wake up --
↪  prior tool outputs, prior file excerpts, prior
↪  ``submit`` summaries, all of it. BEFORE issuing a new
↪  bash / view command, scan your own history: if the
↪  answer is already there, do NOT re-read; quote it
↪  directly. When you do run a new command, pick one that
↪  strictly extends what you already have rather than
↪  re-fetching it.

Handling ambiguous queries (this is real -- the main agent
↪  sometimes sends keyword-soup queries with no clear
↪  question):
  - First try to disambiguate from your own accumulated
  ↪  history and one or two well-targeted probes. If the
  ↪  keywords still resolve to multiple plausible
  ↪  interpretations, do NOT keep grinding more grep
  ↪  variants hoping one lands. Stop and submit a
  ↪  clarifying reply via ``submit`` instead -- list the
  ↪  candidate interpretations you considered (with the
  ↪  file/symbol each one points at) and ask which one
  ↪  the agent meant.
  - This is preferable to a long grep loop because: (a)
  ↪  the next consultation will arrive with a sharper
  ↪  question, (b) your history retains the candidates
  ↪  you already enumerated, so the follow-up is cheap.
  - Clarification replies are SHORT -- the candidate list,
  ↪  file pointers if you have them, and the question. Do
  ↪  NOT include code excerpts on a clarification.

Reply discipline (when the query is well-formed):
  - Ground every claim in a file path + line range from
  ↪  your tool output (or your prior history).
  - When the main agent's query points at a behaviour or
  ↪  symptom, translate it into "what file / symbol owns
  ↪  this surface area" and reply with that location. Do
  ↪  NOT volunteer a diagnosis of the symptom -- that's
  ↪  the main agent's job.

Reply SHAPE -- describe locations in ``result``, attach
↪  code via ``view_commands``:

The ``submit`` tool takes TWO parameters:
  - ``result``: short prose answer (<~ 200 tokens). File
  ↪  paths + line ranges + brief framing of what each
  ↪  region is. **Do NOT paste verbatim code into
  ↪  ``result``.** Bytes are bigger than your TEXT
  ↪  output should be.
  - ``view_commands``: optional JSON list of ``[path,
  ↪  start, end]`` triples. The system runs each view
  ↪  server-side and prepends the output to what the main
  ↪  agent sees. The view does NOT enter YOUR
  ↪  conversation history (it's executed outside the SDK
  ↪  turn), so your token budget for future
  ↪  ``repo_librarian`` calls in this episode stays lean.

Three response shapes:

  - **Pointer (default)**: short ``result`` describing
  ↪  the location, single-element ``view_commands`` with
  ↪  the region you're pointing at. Example::
        result = (
          "Found `rotation_matrix` in "
         "/testbed/astropy/coordinates/matrix_utilities.py
          ↪  "
          "lines 30-78 -- body included below."
        )
        view_commands = '[["/testbed/astropy/coordinates⌋
       ↪  /matrix_utilities.py", 30, 78]]'

    Pick the line range carefully so it lands on natural
   ↪  boundaries (class header -> next sibling, function
   ↪  header -> matching dedent).

  - **Multi-fragment**: short ``result`` summarising what
  ↪  you found, ``view_commands`` with multiple regions.
  ↪  Use ONLY when:
      * each fragment is <= ~15 lines, AND
      * the fragments span >= 2 different files OR >= 2
     ↪  non-adjacent regions of one file, AND
      * the main agent obviously needs ALL of them to plan
     ↪  their next move.
    Example::

        result = (
          "Three call sites of `where_not_allclose`: one
          ↪  in diff.py "
          "table comparison, two in nddata arithmetic. All
          ↪  quoted below."
        )
        view_commands = (
          '[["/testbed/astropy/io/fits/diff.py", 1444,
          ↪  1471],'
          ' ["/testbed/astropy/nddata/mixins/ndarithmeti⌋
          ↪  c.py", 220, 230],'
          ' ["/testbed/astropy/nddata/mixins/ndarithmeti⌋
          ↪  c.py", 410, 425]]'
        )

  - **Clarification**: ``result`` lists candidate
  ↪  interpretations and file pointers;
  ↪  ``view_commands`` omitted (or ``None``). Use when
  ↪  the query is genuinely ambiguous.

 If the answer is fully reconstructable from your prior
↪  history without new tool calls, you may still call
↪  ``submit`` with ``view_commands`` pointing at
↪  locations you already viewed earlier in this episode
↪ -- the system re-fetches them, you don't have to
↪  re-issue ``view`` first.

 Pre-submit self-check (apply BEFORE every ``submit``
↪  call):
   1. Is ``result`` PROSE only (no ``def ...`` / ``class
  ↪  ...`` / line-numbered code blocks)? If I caught
  ↪  myself pasting code into ``result``, move it to
  ↪  ``view_commands`` instead.
   2. Are the line ranges in ``view_commands`` PRECISE
  ↪  (verified against my actual ``view`` / ``cat -n``
  ↪  output)? Vague ranges like ``[200, 300]`` make the
  ↪  main agent see the wrong region -- re-ground from
  ↪  output.
   3. For multi-fragment ``view_commands``: are the
  ↪  regions REALLY scattered (different files OR
  ↪  non-adjacent regions), or am I really showing one
  ↪  long block split into "fragments"? If the latter,
  ↪  collapse to a single ``[path, start, end]``
  ↪  covering the union range.
   4. Did my ``result`` slip into analysis / interpretation
  ↪  territory?  Phrases to scrub before submitting:
        - "this is a fix for issue #N" / "this was added to
       ↪  handle X" (historical inference -- not your
       ↪  job)
        - "the bug is here because ..." / "the issue arises
       ↪  when ..." (diagnosis -- not your job)
        - "the fix should ..." / "you should change this to
       ↪  ..." (proposal -- not your job)
     If you find any, delete them. Keep file:line framing
    ↪  only.
```

B.5 Tool-selection policy

The block spliced into each MAS's orchestrator prompt. It instructs the orchestrator to route unknown-location queries to the Librarian and to view known `file:line` pointers directly without consulting the Librarian a second time. The placeholder `{librarian_tool}` is replaced at MAS-instantiation time with the specific delegation tool name, such as `repo_librarian` for the vanilla MAS and `delegate_to_librarian` for HyperAgent.

Prompt for the orchestrator tool-selection policy

```
Tool selection policy (read this BEFORE you plan):
  - **Decision rule**: if you ALREADY know the precise
 ↪  file path AND a useful line range to inspect,
 ↪  ``str_replace_editor view`` it directly. If you
 ↪  DON'T know where to look -- you have a
 ↪  symbol/behaviour/symptom but no file:line -- ask
 ↪  ``{librarian_tool}``.
  - ``{librarian_tool}`` is your **search** tool -- it
 ↪  returns file paths and line ranges (and short
 ↪  excerpts when several fragments are scattered).
 ↪  Treat it as a sharper, memory-equipped grep, NOT as
 ↪  an analyst. Use it to ask "where is X?" / "which
 ↪  file owns Y?" / "give me the file:line for Z" --
 ↪  never "what's the bug?" or "is this code correct?".
 ↪  Reasoning, diagnosis, and fix-design stay with you.
  - The librarian is a single long-lived instance: its
 ↪  conversation history accumulates across calls, so
 ↪  follow-ups are cheap.
  - ``bash`` and ``str_replace_editor`` are for both
 ↪  ACTING on the code (editing source, running tests,
 ↪  installing dependencies) AND READING code at known
 ↪  locations (``str_replace_editor view`` with a file
 ↪  path + line range you already have). What ``bash``
 ↪  is NOT is a search tool.
  - Continuous consultation policy: each NEW search
 ↪  question (a symbol you don't yet have a location
 ↪  for, a "where does X live?" question) is a NEW
 ↪  librarian call. Once the librarian has given you a
 ↪  path + line range, you own that location -- viewing
 ↪  nearby lines, the enclosing class, or an adjacent
 ↪  helper at a known file:line is your job, NOT another
 ↪  librarian call. The single biggest failure mode in
 ↪  BOTH directions: (a) calling the librarian once and
 ↪  then doing keyword grep yourself for every
 ↪  subsequent symbol -- do NOT do that; (b) re-asking
 ↪  the librarian for a region you already have a
 ↪  precise file:line for -- do NOT do that either, just
 ↪  ``view`` it.
  - Forbidden in ``bash``: ``find`` / ``grep`` / ``rg`` /
 ↪  ``ls`` / ``tree`` / ``sed`` / ``awk`` / ``less`` /
 ↪  ``more`` for symbol or pattern search across source
 ↪  files. (Reading a specific file you already located
 ↪  via ``cat`` / ``head`` / ``tail`` is fine but
 ↪  ``str_replace_editor view`` with ``view_range`` is
 ↪  preferred -- it gives you line numbers.) Searching
 ↪  for unknown symbols is the librarian's job.
  - Query phrasing: write each ``{librarian_tool}`` query
 ↪  as a **lookup question**, not a bag of keywords and
 ↪  not an open-ended analysis request. Good shapes:
 ↪  "Where is <symbol> defined? I need the file path +
 ↪  line range.", "Which file owns <behaviour> in
 ↪ <surface area>?", "Locate <test name>'s file path.".
 ↪  Bad shapes: a keyword string with no question, or an
 ↪  analysis prompt like "Why does <symptom> happen?" /
 ↪  "What is the bug here?" -- the librarian will only
 ↪  return locations, not analysis.
  - Healthy turn pattern: (don't know where) librarian
 ↪  call -> librarian returns file:line pointer -> you
 ↪  ``str_replace_editor view`` to read -> reason ->
 ↪  edit -> run tests -> (need a NEW location? librarian
 ↪  again).
```

B.6 Plan rewriting guideline

The instructions we follow to derive each +Librarian plan from its baseline. The guideline encodes two rules, namely renaming references to the prior code-navigator sub-agent and rewriting any code or file location step to a Librarian delegation call. The same guideline applies across MAS, with the actual scope of changes varying by how much navigation each MAS previously routed through bash versus delegation.

Prompt for the plan-rewriting guideline

```
# Plan / Planner Prompt Rewriter (-> +librarian variant)

You are rewriting a baseline MAS orchestrator / planner
↪  prompt into its ***+librarian** variant. The transform
↪  replaces a navigator-style sub-agent with the
↪  persistent Repo Librarian, and rewrites the plan steps
↪  that involve code / file location so the lookup goes
↪  through the librarian.

You edit ONLY the orchestrator/planner-facing prompt.
↪  Other sub-agents' prompts are out of scope.

---

## INPUT FORMAT

You receive four sections, separated by tagged delimiters:

    <previous_subagent_name>
    ... name of the navigator-style sub-agent being
   ↪  replaced (e.g. "code_navigator", "Codebase
   ↪  Navigator", "delegate_to_navigator") ...
    </previous_subagent_name>

    <previous_subagent_prompt>
    ... the full system prompt of the previous sub-agent,
   ↪  used to identify which of its responsibilities go
   ↪  beyond file/code lookup and therefore need to be
   ↪  redirected rather than absorbed by the librarian
   ↪  ...
    </previous_subagent_prompt>

    <baseline_prompt>
    ... the baseline plan / planner prompt to rewrite ...
    </baseline_prompt>

    <librarian_spec>
    ... the librarian's contract: its role, persistence
   ↪  semantics, query-shape rules, and what it does /
   ↪  does not do ...
    </librarian_spec>

---

## RULES

### 1. Replace the named sub-agent with the Librarian

Every reference to the sub-agent name(s) in
↪  `<previous_subagent_name>` -- docstring, intern-list
↪  entry, delegation-tool name, inline mention in a
↪  numbered step -- becomes the librarian. Use a surface
↪  name that matches the baseline's own casing and
↪  tool-naming convention (`librarian`, `Repo Librarian`,
↪  `delegate_to_librarian`, etc.). Do not paraphrase any
↪  other text on the same line.

### 2. Rewrite code/file-location steps to use the
↪  Librarian

Any step in the plan that asks the host to **locate,
↪  navigate, search, explore, or look up** code (find a
↪  symbol, identify affected files, map the codebase,
↪  locate call sites, find the test that exercises a
↪  behaviour) -- including post-failure debugging steps
↪  that re-locate code on a failing test -- gets
↪  rewritten so the lookup happens through the librarian.

Preserve the librarian's contract exactly as stated in
↪  `<librarian_spec>`: natural-language lookup questions,
↪  persistent conversation history across calls within
↪  the episode, NEW search question = NEW librarian call,
↪  host owns the location once handed back, librarian
↪  only LOCATES (host reasons / diagnoses), no fallback
↪  to `grep` / `find` for symbol search.

Steps that do **not** involve code/file location (issue
↪  analysis, synthesis, fix implementation, cleanup,
↪  submit, etc.) stay verbatim.

### 3. Redirect responsibilities that go beyond file/code
↪  lookup

Read `<previous_subagent_prompt>` to enumerate the
↪  responsibilities the previous sub-agent held beyond
↪  pure file/code lookup -- e.g. analyzing code purposes,
↪  mapping inter-module dependencies, summarizing
↪  architecture, characterizing call relationships. For
↪  each such responsibility, do NOT route it to the
↪  librarian (the librarian only LOCATES). Instead, fold
↪  it into the orchestrator's own analytical steps or
↪  into another sub-agent that the baseline plan already
↪  invokes for analysis/synthesis. If neither already
↪  covers it, add a short orchestrator-side step adjacent
↪  to the librarian call that performs that work on the
↪  returned excerpts. Leave the librarian call itself
↪  unchanged.

---

## BREVITY

Every character is prefilled on every planner /
↪  orchestrator turn. Keep the rewrite within ~1.15x the
↪  baseline length.

---

## OUTPUT

Return ONLY the rewritten prompt. Preserve the original
↪  formatting, section headers, bullet structure, and any
↪  verbatim blocks unaffected by the two rules. Do not
↪  add commentary before or after.
```

B.7 Per-MAS plan rewrites

The orchestrator-side planning sequences for the BOAD and HyperAgent MAS. For each MAS we reproduce the baseline plan alongside the +Librarian variant produced by applying the guideline in Appendix B.6, with BOAD's variant generated via Claude Opus 4.6.

BOAD baseline. The orchestrator's six-step plan with step 2 delegating code mapping to the `code_navigator` sub-agent.

Prompt for BOAD orchestrator plan

```
1. Use the issue_analyzer subagent to decompose the
↪  problem description into structured requirements,
↪  identify affected components, and define explicit
↪  success criteria to guide the investigation.

2. Use the code_navigator subagent to map the relevant
↪  codebase structure, focusing on files and functions
↪  identified in the analysis to understand dependencies
↪  and data flow.

3. Synthesize the structured analysis and code mapping to
↪  implement the fix, ensuring the solution addresses the
↪  root cause while adhering to the established success
↪  conditions.
```

```
4. Create a minimal test case or validation script to
↪  verify the fix against the reproduction criteria
↪  defined in the initial analysis.

5. After you have solved the issue, delete any test files
↪  or temporary files you created.

6. Use the submit tool to submit the changes to the
↪  repository.
```

BOAD + Librarian. Steps 1, 3, 5, and 6 are byte-identical to the baseline. Step 2 is rewritten to phrase every search as a natural-language lookup question to the Librarian, and step 4 is rewritten to call the Librarian for additional location lookups after test failures.

Prompt for BOAD orchestrator plan with Librarian

```
1. Use the issue_analyzer subagent to decompose the
↪  problem description into structured requirements,
↪  identify affected components, and define explicit
↪  success criteria to guide the investigation.

2. Use the librarian subagent to map the relevant code
↪  areas: phrase each request as a natural-language
↪  lookup question ("Where lives <symbol / behaviour>?",
↪  "Show me the body of <function>", "Locate the test
↪  that exercises <feature>"). The librarian is a single
↪  long-lived instance -- its conversation history
↪  accumulates across calls in this episode, so follow-up
↪  lookups (the caller of a symbol, subclasses that
↪  override it, adjacent helpers, sibling tests) are
↪  cheaper than starting a fresh search. Each NEW search
↪  question (a symbol or behaviour you do not yet have a
↪  file:line pointer for) is a NEW librarian call; once
↪  the librarian has given you a location, that location
↪  is yours -- view nearby lines or the enclosing class
↪  yourself rather than re-asking the librarian for it.
↪  The librarian only LOCATES code; reasoning about what
↪  to change stays with you. Call the librarian
↪  aggressively whenever a new location is needed; do not
↪  fall back to `grep` / `find` for symbol search. After
↪  exploration, propose the files and methods to fix the
↪  issue and select the most likely location to apply the
↪  change.

3. Synthesize the structured analysis and code mapping to
↪  implement the fix, ensuring the solution addresses the
↪  root cause while adhering to the established success
↪  conditions.

4. Write and execute targeted unit tests or integration
↪  checks to validate that the solution resolves the
↪  issue without introducing regressions. If any check
↪  fails, consult the librarian for ADDITIONAL location
↪  lookups (other call sites of a changed symbol,
↪  subclasses that override it, fixtures or sibling tests
↪  that exercise the failing path) -- phrase each as a
↪  location question. The librarian returns a pointer to
↪  the relevant code; YOU then view it, diagnose why the
↪  test fails, refine your fix, and rerun the affected
↪  tests. Do NOT fall back to fresh `grep` / `find` for
↪  symbol search -- those go through the librarian.
↪  Repeat until all checks pass.

5. After you have solved the issue, delete any test files
↪  or temporary files you created.

6. Use the submit tool to submit the changes to the
↪  repository.
```

HyperAgent baseline. The planner's five-step sequence with step 2 delegating to the Codebase Navigator.

Prompt for HyperAgent planner

```
1. Read the PR description and understand the issue.
2. Delegate to the Codebase Navigator to localize where
↪  the change must happen.
3. Delegate to the Codebase Editor to apply the minimal
↪  source change.
4. Delegate to the Executor to verify the fix (run the
↪  relevant tests / a reproduction script).
5. Iterate as needed until the tests pass.
```

HyperAgent + Librarian. Step 2 is rewritten to delegate to the Repo Librarian. The remaining four steps are byte-identical to the baseline.

Prompt for HyperAgent planner with Librarian

```
1. Read the PR description and understand the issue.
2. Delegate to the Repo Librarian to localize where the
↪  change must happen.
3. Delegate to the Codebase Editor to apply the minimal
↪  source change.
4. Delegate to the Executor to verify the fix (run the
↪  relevant tests / a reproduction script).
5. Iterate as needed until the tests pass.
```

C Experiment Details

This appendix records the configuration held fixed across the experiments of §5—the vLLM serving, sampling, and scaffold settings (Appendix C.1) and the caveman prompting style directive (Appendix C.2).

C.1 Serving and scaffold configuration

Table 9 lists the vLLM serving configuration, sampling hyperparameters, and agent scaffold settings used in the experiments of §5. The values are held fixed across all backbones and methods so that per-method differences in token consumption reflect orchestration choices rather than serving, decoding, or scaffold variance.

C.2 Caveman prompting directive

The fixed style directive that the caveman prompting baseline appends to the system prompts of every sub-agent and the orchestrator. The directive instructs the model to compress its visible answer text while leaving the thinking trace intact.

Prompt for caveman style directive

```
OUTPUT STYLE — caveman mode (FULL).

Respond terse like smart caveman. All technical substance
↪  stays. Only fluff dies. ACTIVE every response until
↪  task done.

Rules:
- Drop articles (a/an/the), filler
↪  (just/really/basically/actually/simply), pleasantries
↪  (sure/certainly/of course/happy to), hedging.
- Fragments OK. Short synonyms (big not extensive, fix not
↪  "implement a solution for").
- Pattern: `[thing] [action] [reason]. [next step].`
```

Setting	Value
vLLM serving	
serving context length	131,072
GPU memory utilization	0.95
sampling	
temperature	0.6
top-p	0.95
top-k	20
min-p	0.0
presence penalty	0.0
repetition penalty	1.0
agent scaffold	
shell command timeout (min)	30
tool observation truncation (chars)	30,000

Table 9: vLLM serving configuration, sampling hyperparameters, and agent scaffold settings, held identical across every backbone and method.

```
NEVER compress / NEVER abbreviate:
- Code blocks, code symbols, function names, class names,
↪  file paths, error strings, log lines, shell commands,
↪  tool-call arguments.
- Anything inside `backticks`, fenced code, or quoted
↪  strings.
- Patch / diff content (must be byte-exact).

Resume normal prose for: security warnings,
↪  irreversible-action confirmations, multi-step
↪  sequences where fragment order could be misread.

Example
- Normal: "The reason your React component is re-rendering
↪  is likely because you're creating a new object
↪  reference on each render cycle. You should wrap it in
↪  useMemo."
- Caveman: "New object ref each render. Inline object prop
↪  = new ref = re-render. Wrap in `useMemo`."
```

D Full Per-Role Breakdown

This appendix reproduces the per-episode breakdown of output tokens and idle-subtracted GPU energy at the granularity of each role inside HyperAgent (Table 10) and BOAD (Table 11). The compact main-text version (Table 6) collapses the non-replaced roles into a single “other roles” row by summing within each episode before averaging; the tables below leave every role on its own row so the reader can see the unaggregated contribution of, for example, the HyperAgent executor or the BOAD issue analyzer.

E Pass Rate across Task Difficulty

Table 12 stratifies pass rate by the difficulty bins of §5.2, complementing the per-episode energy reported there. The Librarian preserves the underlying MAS’s pass rate across every bin.

		HyperAgent		HyperAgent + Librarian	
Model	Role	Out (K)	E (kJ)	Out (K)	E (kJ)
A3B	planner	9.20	5.03	9.46	5.35
	navigator → librarian	13.92	7.54	6.02	3.57
	editor	2.84	1.48	2.81	1.46
	executor	22.70	11.69	24.12	12.46
27B	planner	7.72	26.63	7.49	26.40
	navigator → librarian	10.50	36.63	4.70	16.92
	editor	2.55	8.82	1.95	6.72
	executor	14.66	49.82	12.20	41.46

Table 10: Per-role per-episode comparison of HyperAgent against its Librarian variant. We report the per-episode means of output tokens and energy at each role on the two backbones. The Librarian compresses the lookup role while leaving the other roles within 10% on every metric.

		BOAD		BOAD + Librarian	
Model	Role	Out (K)	E (kJ)	Out (K)	E (kJ)
A3B	orchestrator	11.01	6.74	8.79	5.23
	issue analyzer	14.17	8.63	14.31	8.62
	code_navigator → librarian	13.81	8.54	1.93	1.08
27B	orchestrator	6.66	24.23	6.07	21.89
	issue analyzer	12.11	44.48	11.79	43.35
	code_navigator → librarian	5.63	20.79	1.82	6.56

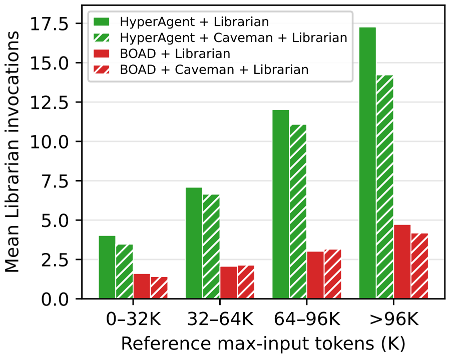
Table 11: Per-role per-episode comparison of BOAD against its Librarian variant. We report the per-episode means of output tokens and energy at each role on the two backbones. The Librarian compresses the lookup role while leaving the other roles within 10% on every metric.

F Librarian Invocation Frequency

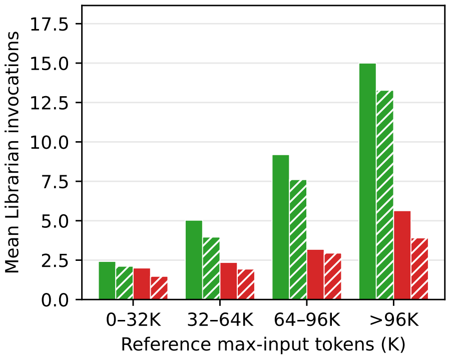
We examine how often the orchestrator invokes the Librarian as the search problem grows. Stratifying episodes by the difficulty bins of §5.2, we count the per-episode Librarian invocations. Figure 4 shows this mean count growing monotonically with difficulty across both backbones. We attribute the trend to the orchestrator scaling lookup frequency with the size of the search problem.

Method	Reference max-input tokens (K) 0–32K (n=36)	32–64K (n=312)	64–96K (n=130)	>96K (n=22)
Qwen3.6-35B-A3B				
HyperAgent	83.3	73.4	47.7	**40.9**
+ Librarian	**86.1**	**74.7**	**51.5**	22.7
+ LastNObservation	80.6	67.3	32.3	13.6
+ Caveman	77.8	73.1	50.8	**40.9**
+ Caveman + Librarian	**86.1**	73.1	50.0	31.8
BOAD	**88.9**	76.6	53.8	**36.4**
+ Librarian	**88.9**	**78.8**	55.4	**36.4**
+ LastNObservation	83.3	71.5	45.4	18.2
+ Caveman	86.1	73.7	**57.7**	27.3
+ Caveman + Librarian	**88.9**	77.9	53.8	31.8
Qwen3.6-27B				
HyperAgent	83.3	**75.6**	57.7	36.4
+ Librarian	**86.1**	75.0	**65.4**	**45.5**
+ LastNObservation	**86.1**	75.0	42.3	18.2
+ Caveman	83.3	73.1	56.2	40.9
+ Caveman + Librarian	**86.1**	72.4	54.6	**45.5**
BOAD	86.1	**77.6**	**60.8**	31.8
+ Librarian	**88.9**	76.6	60.0	**45.5**
+ LastNObservation	**88.9**	77.2	55.4	27.3
+ Caveman	**88.9**	75.3	57.7	40.9
+ Caveman + Librarian	**88.9**	75.6	**60.8**	**45.5**

Table 12: Pass rate of the SWE agents with respect to the task difficulty. We report the pass rate of the SWE agents across difficulty classes. In all the settings, the addition of the Librarian preserves the performance.



(a) Qwen3.6-35B-A3B



(b) Qwen3.6-27B

Figure 4: Mean Librarian invocations per episode across task difficulty. Each panel plots the per-episode mean Librarian invocation count for the Librarian-augmented MAS variants against the reference max-input-token bin. Bar color denotes the host family, and a hatch marks the Caveman variant. The invocation frequency grows monotonically with task difficulty in every setting.