

Evaluating Context Segmentation in Locally Deployable SLMs for Cybersecurity CTF Tasks^{*}

Sebastiano Nordio¹[0009–0004–0586–4673] and Michele Lotto²[0009–0001–9133–4297]

¹ Independent

`norseb21@outlook.com`

² University of Genoa, Via All’Opera Pia, 16145 Genoa, Italy

`michele.lotto@edu.unige.it`

Abstract. The proliferation of highly capable open-weight Small Language Models (SLMs) democratizes access to advanced cybersecurity capabilities, posing an escalating risk as these models can bypass proprietary API guardrails when deployed locally. However, SLMs deployed as autonomous agents often struggle with long-horizon, exploratory tasks like cybersecurity Capture The Flag (CTF) challenges due to context bloat and cognitive degradation from accumulated tool-call outputs. To understand and mitigate this cybersecurity threat, we introduce *context segmentation*, a two-level agentic framework that divides complex exploitation tasks into manageable, contextually isolated sub-problems. Evaluating on the `picoCTF` dataset using memory-constrained `gemma-4` models, we demonstrate that for the E4B model, our strategy acts as an intelligent search, achieving competitive rewards with superior token efficiency compared to brute-force retries, and successfully solving 18.52% of tasks that standard agentic execution fails to complete. Code is available at <https://github.com/9xeb/context-segmentation>.

Keywords: Small Language Models · Cybersecurity · Autonomous Agents · Context Segmentation · Capture The Flag (CTF).

1 Introduction

The integration of Large Language Models (LLMs) into the cybersecurity landscape has introduced a profound paradigm shift, yielding both defensive advantages and severe security challenges. On one hand, LLMs can significantly optimize defensive operations, such as automated vulnerability detection, code analysis, and incident response [19]. On the other hand, their dual-use nature raises critical concerns, as these same capabilities can theoretically be weaponized to generate malicious payloads, automate social engineering, and identify vulnerabilities [13]. Despite these theoretical risks, a practical bottleneck has historically existed: foundational studies demonstrated that only massive, state-of-the-art proprietary models possessed the reasoning and long contextual planning required to successfully execute complex, multi-step cybersecurity tasks, such

^{*} Accepted at the RAISE 2026 Workshop (ESORICS 2026). Non-archival.

as autonomous web hacking [8]. Consequently, the immense computational overhead and API-gated access to these frontier models acted as a natural safeguard. This high barrier made it practically impossible for malicious actors with limited hardware resources to conduct fully automated, large-scale cyberattacks using language models. However, the landscape is rapidly evolving; the continuous proliferation of highly capable open-weight models such as the DeepSeek-R1 [12], Gemma4 [11] and Qwen3 [24] families has democratized access to advanced problem-solving and multi-stage attack analysis [12,4]. Recent demonstrations have proven that models running on consumer hardware can now autonomously discover vulnerabilities, execute exploits, and even self-replicate across network hops without human oversight [4]. Because these models can be fine-tuned, uncensored [5], and deployed locally on consumer-grade hardware, they bypass the safety guardrails of proprietary APIs [7]. This presents a tangible and escalating risk, as the advancing capabilities of these open models could be actively exploited by attackers to automate complex cyber threats without oversight or resource constraints. This automation is made possible by integrating these models into agentic frameworks [23], which transforms LLMs from storytellers to entities capable of autonomously navigating the real world. As a side effect, during the agentic loop, despite LLMs only generating strings, real code is executed under their control by the harness. This means that LLMs paired with tools can now do anything a computer program does, including using email, controlling robotic motors, and cracking IT systems via CLI.

To understand and mitigate the cybersecurity threat of autonomous, LLM-powered agents, it is essential to evaluate the true offensive potential of locally deployable open-weight Small LMs (SLMs). In this paper, we propose a novel methodology designed to improve the performance of these smaller models in complex cybersecurity scenarios. Our approach introduces *context segmentation*, an automated prompt-level ‘divide and conquer’ strategy that breaks down intricate exploitation tasks into manageable, context-specific sub-problems and processes them sequentially. Specifically, the primary contributions of this work are as follows:

- **Token-Efficient Agentic Execution:** We demonstrate that while brute-force iteration of baseline agentic loops can yield higher absolute rewards, *context segmentation* provides a significantly more token-efficient alternative.
- **Resolution of Complex Edge-Cases:** We show that our segmentation strategy intelligently constrains and guides model generation, enabling the successful resolution of complex, edge-case tasks that even the reiterated baseline fails to address entirely.
- **Re-evaluation of SLM Threat Models:** We provide empirical evidence that, when equipped with an optimized prompt-level harness, open-weight SLMs can successfully execute more sophisticated exploitation chains. This lowers the safeguard barrier for malicious actors with limited hardware resources to conduct fully automated cyberattacks using SLM.

2 Background

Small Language Models (SLM) are models designed to maintain the accuracy and/or adaptability of large language models, while being subject to hardware constraints, such as VRAM and computing power [20]. The definitions of ‘small’ and ‘large’ are a function of both context and time. For instance, GPT-2 (1.5B) was considered ‘large’ in 2019 but in 2026 is considered smaller than many ‘small’ language models. In cybersecurity context, while SLMs have demonstrated to nearly reach LLMs performance on tasks such as tactical and technical extraction [25], they still lack behind LLMs in tasks such as Phishing Website Detection [10]. The advent of Agents, which leverage the emerging ability of language models to generate structured text to be parsed by other programs, has been pivotal in cybersecurity [28,1], as well as in other fields [9]. SLM-powered agents can now generate JSON or YAML payloads containing function call signatures that external programs process to execute code on behalf of the model. The code execution results are then appended to the context and sent back to the language model, which can subsequently call additional tools or provide a final response. While a conversation is technically occurring, a command interpreter is actually reading and answering on the other side instead of a human, enabling the continuous agentic loop. Since exploitations are typically performed via shell interfaces (e.g., bash), language models are prompted to generate function call signatures containing command chains or scripts to perform exploitations autonomously.

Threat Model: Hardware Constraints. In evaluating these systems, our threat model specifically assumes an adversary operating under strict hardware limitations. The deployment environment is restricted to standard hardware with limited Video RAM (VRAM) and compute capacities. Consequently, the agent must rely entirely on localized Small Language Models. This constraint dictates that the models cannot leverage massive context windows or trillion-parameter reasoning to solve complex vulnerabilities in a single inference pass, necessitating efficient and tool-assisted interactions through an agentic loop.

Capture The Flag (CTF) Tasks. To safely and objectively assess the cybersecurity capabilities of these hardware-constrained agents, their function calls must be routed to repeatable sandbox environments. A standard and highly effective metric for this evaluation relies on Capture The Flag (CTF) challenges [28,21,17,26]. CTFs are specialized cybersecurity exercises designed to test and develop offensive and defensive security skills. These challenges span various technical domains, including web vulnerability exploitation, reverse engineering, cryptography, and binary exploitation, where the objective is to locate and extract a hidden string of text, known as the ‘flag’, which serves as cryptographic proof that the system was successfully compromised. While originally designed for human cybersecurity learners interacting directly with a terminal, the deterministic environments, isolated nature, and clear success criteria of CTFs make them the ideal benchmarks for assessing an autonomous agent’s exploitation capabilities.

While agentic loops enable SLMs to interface with complex terminal environments, they introduce a severe bottleneck in hardware-constrained settings: *context bloat* [14]. During multi-step CTF challenges, execution logs, verbose tool outputs, and trial-and-error command outputs rapidly consume token budgets, leading to dead-end execution clutter. Furthermore, SLMs suffer from attention degradation and KV-cache overhead long before reaching their theoretical context limits, leading to instruction drift, hallucinated command syntax, and repetitive failure loops [29,22]. *Context segmentation* directly addresses these limitations by automatically applying a ‘divide and conquer’ strategy at the prompt level, breaking down intricate exploitation tasks into manageable, context-specific sub-problems and processing them sequentially.

3 Methodology

To translate this conceptual divide-and-conquer strategy into a practical architecture without violating our strict hardware limitations, we must achieve context segmentation using only a single locally hosted SLM. The core intuition is to separate high-level strategic reasoning from low-level tactical execution, ensuring the verbose outputs of terminal commands do not pollute the agent’s long-term planning capabilities.

To achieve this separation using only one underlying model, rather than forcing a single agent to parse its entire history of failed attempts in a single context, our approach divides the cognitive load. A main agent, named *Explorer*, keeps track of summarized, high-level reports on all problem-solving attempts, and uses those reports to generate new attempt instructions. A secondary agent, named *Worker*, is prompted by the main agent and actually performs the single problem-solving attempts as instructed. The *Explorer* receives the problem and is equipped with an agent-as-a-tool named `run_strategy`, which accepts a proposed course of action in natural language. Upon invocation, this tool starts the *Worker* as an independent secondary agent loop that executes the proposed strategy from a blank context, while the *Explorer* waits for the tool call to return a result. Unlike the *Explorer*, the *Worker* directly interacts with the execution environment (i.e. a bash CLI tool). Upon termination, the *Worker* generates a summary detailing its actions and its reason for stopping, and the summary is returned to the *Explorer* as a tool output. After each report, the *Explorer* agent evaluates whether to call a new strategy or halt execution. Both the *Explorer* and the *Worker* have access to a `give_up` tool, which allows them to signal and justify when they have exhausted all viable options within their respective contexts. Appendix A provides the Python pseudocode for our proposed implementation of *context segmentation*.

This approach is supposed to mitigate *context bloat*, by dynamically pruning irrelevant information [27]. The primary objective of *context segmentation* is to prevent Small Language Model (SLM) agents from becoming disoriented by lengthy sequences of tool interactions, thereby enabling coherent, high-level problem-solving.

4 Experimental Setup

Models. Our evaluation focuses on agents powered by Instruction-Tuned (it) models, optimized for local deployment. To ensure computational feasibility across varied consumer-grade hardware, all selected models were quantized to 4-bit precision to operate within a strict memory envelope of less than 6 GB of RAM³. Specifically, we evaluate the `gemma-4-E2B-it`⁴ and `gemma-4-E4B-it`⁵ variants [11], which successfully meet these resource constraints while maintaining competitive reasoning capabilities. See Appendix C for further discussion on the models utilized.

Task and Environment. Because autonomous agents and their associated harnesses represent a relatively nascent area of research, we encountered a scarcity of cybersecurity-oriented benchmarks tailored to evaluate agents rather than standalone models. For instance, Cybench [28] only supports wiring model APIs and precludes custom agent harnesses; CyberGym [21] focuses on software vulnerability analysis rather than system exploitation; and CAIBench [17] is hidden behind a paywall. Overall, the current evaluation ecosystem remains fragmented and model-centric.

Consequently, we opted to use a modified version of Intercode [26] containing functional Capture The Flag (CTF) challenges, which serve as a proxy for real-world interactive problem-solving and tool manipulation. Specifically, we leverage the interactive `CTFEnv` environment class. The evaluation dataset comprises a diverse set of cybersecurity challenges sourced directly from the `picoCTF` platform [6]. These challenges require the agent to dynamically interact with a bash shell, examine binaries, and retrieve hidden flags. Further details on Intercode are presented in Appendix D.

4.1 Strategies

We evaluate and compare two operational configurations to assess the impact of *context segmentation*.

Plain (Baseline): This strategy represents a standard agent execution wherein the agent interacts with the environment sequentially during a single execution trial. We implement this as a single *Worker* agent loop, initialized with the problem description as its prompt. A task is marked as successful if the correct flag is submitted, and as failed if the agent ‘gives up’.

Explorer: This is our proposed approach. Similar to the baseline, the task is considered successful upon submission of the correct flag and failed if the *Explorer* agent ‘gives up’. Experimental observations revealed that this strategy underperforms compared to a standard agent loop on very simple problems, as

³ Details regarding the utilized hardware and software infrastructure are provided in Appendix B.

⁴ <https://huggingface.co/unsloth/gemma-4-E2B-it-GGUF>

⁵ <https://huggingface.co/unsloth/gemma-4-E4B-it-GGUF>

the *Explorer* agent tends to overcomplicate straightforward solutions. To mitigate this, we refined our approach into the *plain+explorer* strategy. Furthermore, we excluded this from our evaluation in favor of the *plain+explorer* strategy.

Plain+Explorer: This strategy combines the baseline with our proposed approach. The system initially employs the *plain* strategy; if the single *Worker* agent fails (i.e., gives up), the system falls back to the *Explorer* approach. The success criteria remain consistent: the task succeeds if the correct flag is submitted and fails if the *Explorer* agent concedes.

Further details on strategies are discussed in Appendix E.

5 Results

In Table 1, we compare the mean reward and mean token usage of our *plain+explorer* strategy against the *plain* baseline. As shown, our approach achieves a notable improvement in reward, albeit at the cost of significantly higher token consumption. This trade-off occurs because the *plain+explorer* strategy governs the execution of multiple *plain* instances, effectively iterating the baseline approach several times.

Table 1. Comparison of mean reward and token usage between the *plain* and *plain+explorer* strategies.

Strategy	gemma-4-E2B-it		gemma-4-E4B-it	
	Reward	Tokens	Reward	Tokens
plain	0.14	2362.99	0.23	2694.54
plain+explorer	0.19	8031.02	0.39	8652.52

Given these initial insights, it is crucial to determine whether the observed gain in reward is merely the result of iterating the baseline approach multiple times. To investigate this, Table 2 compares the mean reward and token usage of our *plain+explorer* strategy against a *plain best-of-k* baseline, where the baseline reward is evaluated using a logical OR over the per-task success of k independent iterations (reward@ k), and the baseline token usage is calculated as the sum of the tokens consumed across all independent iterations. As shown, for the **gemma-4-E2B-it** model, our approach does not outperform a simple brute-force iteration of the *plain* strategy. Conversely, for the **gemma-4-E4B-it** model, our *plain+explorer* strategy demonstrates a strong advantage in token efficiency. While it does not exceed the absolute maximum reward achieved by the resource-heavy *plain best-of-5* baseline (0.41), it attains a highly competitive reward of 0.39 using only 8652.52 tokens. For comparison, the *plain+explorer* approach outperforms the *plain best-of-4* configuration in reward (0.39 vs. 0.38) while consuming over 2,200 fewer tokens. This suggests that with a sufficiently

capable base model, the *plain+explorer* strategy intelligently guides the generation toward successful outcomes, achieving a highly favorable trade-off between task success and computational cost compared to brute-force retries.

Table 2. Comparison of reward and token usage. Reward is measured as reward@ k for *plain best-of- k* configurations, and as the mean reward for *plain+explorer* experiments.

Experiment	gemma-4-E2B-it		gemma-4-E4B-it	
	Reward	Tokens	Reward	Tokens
plain best-of-1	0.14	2420.47	0.25	2525.10
plain best-of-2	0.20	4799.23	0.31	5262.07
plain best-of-3	0.22	7136.43	0.35	7940.94
plain best-of-4	0.23	9400.00	0.38	10874.63
plain best-of-5	0.25	11814.94	0.41	13472.68
plain+explorer	0.19	8031.02	0.39	8652.52

Finally, in Table 3, we break down the task successes to better understand the overlap and differences between the two strategies. For the more capable **gemma-4-E4B-it** model, the *plain+explorer* strategy demonstrates a substantial advantage, successfully solving 18.52% of the tasks that the baseline *plain* strategy fails to answer. In contrast, the baseline uniquely solves only 2.88% of the tasks, highlighting a strong benefit when employing the explorer configuration. Conversely, for the **gemma-4-E2B-it** model, the vast majority of tasks (77.78%) remain unsolved by either approach. While the *plain+explorer* strategy still recovers more unique tasks than the baseline (7.82% vs. 3.29%), the margin is much narrower. This breakdown proves that the exploration mechanism is highly effective at recovering edge cases and guiding generation, but it requires an underlying model with a sufficient baseline capacity to actually solve the problems once they are explored.

Table 3. Breakdown of task success. The table shows the percentage of tasks solved exclusively by one strategy, by both, or by neither.

Outcome	Tasks Solved (%)	
	gemma-4-E2B-it	gemma-4-E4B-it
Solved only by <i>plain</i>	3.29	2.88
Solved only by <i>plain+explorer</i>	7.82	18.52
Solved by both	11.11	20.16
Solved by neither	77.78	58.44

6 Limitations and Future Work

While our approach successfully enhanced the capabilities of agents powered by locally deployable SLMs in CTF challenges, current results remain preliminary and highlight critical avenues for future research.

Moving forward, our efforts will revolve around **prompt optimization** to move beyond our current hand-crafted implementations, refining the *Explorer* and *Worker* system prompts for maximum strategic coherence and execution reliability. This effort will include introducing new roles to capitalize on aggressive context segmentation, ensuring the architecture supports general problem-solving without relying on specific domain assumptions. Alongside these structural improvements, we will implement an **agentic long-term memory system** to persistently record, retrieve, and reuse successful exploitation strategies. We must also rigorously reassess the implications of the ‘give-up’ tool to determine if it inadvertently inhibits agents and compels premature laziness during repeated failed attempts at exceptionally hard problems.

Finally, because our 6GB RAM constraint limited us to only two models that reliably completed the current benchmark (as detailed in Appendix C), we intend to test a wider variety of models on low-end hardware. This will culminate in comprehensively validating our methodology through **expanded benchmarking** across more diverse, high-fidelity security domains that extend far beyond CTF challenges and better reflect real-world vulnerability discovery usable with custom harness frameworks.

7 Conclusion

In this work, we investigated the capabilities of locally deployable agents powered by open-weight Small Language Models (SLMs) in executing autonomous cybersecurity tasks. Recognizing that standard agentic loops suffer from rapid context bloat and cognitive degradation, we introduced *context segmentation*, a dual-agent framework that dynamically separates high-level strategic reasoning from low-level execution clutter. It successfully resolves complex, long-horizon tasks that single-pass baselines entirely fail to address, achieving a superior trade-off between absolute reward and token efficiency compared to brute-force repetition. This capability effectively lowers the hardware barrier for automated cyberattacks, highlighting a critical and escalating threat in the offensive security landscape. While our empirical results confirm this threat, they also suggest that SLMs possess a latent agentic potential that remains heavily constrained by traditional, manual prompt engineering. The proposed *context segmentation* approach acts as an initial form of automated prompt expansion, demonstrating significant room for optimization.

References

1. Abramovich, T., Udeshi, M., Shao, M., Lieret, K., Xi, H., Milner, K., Jancheska, S., Yang, J., Jimenez, C.E., Khorrami, F., Krishnamurthy, P., Dolan-Gavitt, B.,

- Shafique, M., Narasimhan, K., Karri, R., Press, O.: Enigma: Interactive tools substantially assist lm agents in finding security vulnerabilities (2025), <https://arxiv.org/abs/2409.16165>
2. AI, L.: Lfm2 technical report. arXiv preprint arXiv:2511.23404 (2025)
3. AI9Stars: G9v3-3b. <https://huggingface.co/ai9stars/G9v3-3B> (nd), accessed: 2026-09-04
4. Air, A., Reworr, Kotov, N., Volkov, D., Steidley, J., Ladish, J.: Language models can autonomously hack and self-replicate (2026), <https://arxiv.org/abs/2605.06760>
5. Arditi, A., Obeso, O., Syed, A., Paleka, D., Panickssery, N., Gurnee, W., Nanda, N.: Refusal in language models is mediated by a single direction (2024), <https://arxiv.org/abs/2406.11717>
6. Chapman, P., Burket, J., Brumley, D.: PicoCTF: A Game-Based computer security competition for high school students. In: 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14). USENIX Association, San Diego, CA (Aug 2014), <https://www.usenix.org/conference/3gse14/summit-program/presentation/chapman>
7. Cheng, Z., Zhang, M., Sun, J., Dai, W.: On weaponization-resistant large language models with prospect theoretic alignment. In: Proceedings of the 31st International Conference on Computational Linguistics. pp. 10309–10324 (2025)
8. Fang, R., Bindu, R., Gupta, A., Zhan, Q., Kang, D.: Llm agents can autonomously hack websites (2024), <https://arxiv.org/abs/2402.06664>
9. Ferrag, M.A., Tihanyi, N., Debbah, M.: From llm reasoning to autonomous ai agents: A comprehensive review (2026), <https://arxiv.org/abs/2504.19678>
10. Goldenits, G., König, P., Raubitzek, S., Ekelhart, A.: Small language models for phishing website detection: Cost, performance, and privacy trade-offs. *Journal of Cybersecurity and Privacy* **6**(2) (2026). <https://doi.org/10.3390/jcp6020048>, <https://www.mdpi.com/2624-800X/6/2/48>
11. Google DeepMind: Gemma 4: Open models based on gemini research and technology. <https://ai.google.dev/gemma/docs> (2026), accessed: 2026-06-16
12. Guo, D., Yang, D., Zhang, H., Song, J., Wang, P., Zhu, Q., et al.: Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948 (2025)
13. Gupta, M., Akiri, C., Aryal, K., Parker, E., Praharaj, L.: From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy (2023), <https://arxiv.org/abs/2307.00691>
14. Hong, K., Troynikov, A., Huber, J.: Context rot: How increasing input tokens impacts LLM performance. <https://research.trychroma.com/context-rot> (2025), chroma Research Technical Report. Accessed: 2026-09-04
15. inclusionAI: Ling-3.0-tiny. <https://huggingface.co/inclusionAI/Ling-3.0-tiny> (nd), accessed: 2026-09-04
16. Maan, D., Sanchez, D., Vardhan, A., Montague, D., Trylesinski, M., Hall, A., Colvin, S.: Pydantic ai. <https://github.com/pydantic/pydantic-ai>, documentation: <https://pydantic.dev/docs/ai/>
17. Sanz-Gómez, M., Mayoral-Vilches, V., Balassone, F., Navarrete-Lozano, L.J., Chavez, C.R.J.V., del Mundo de Torres, M.: Cybersecurity ai benchmark (caibench): A meta-benchmark for evaluating cybersecurity ai agents (2025), <https://arxiv.org/abs/2510.24317>
18. Sparrenberg, L., Deußer, T., Berger, A., Sifa, R.: Small and fast llms on commodity hardware: Post-training quantization in llama. cpp. In: 2025 IEEE 12th

- International Conference on Data Science and Advanced Analytics (DSAA). pp. 1–10 (2025). <https://doi.org/10.1109/DSAA65442.2025.11247985>
19. Tian, S., Zhang, T., Liu, J., Wang, J., Wu, X., Zhu, X., Zhang, R., Zhang, W., Yuan, Z., Mao, S., Kim, D.I.: Exploring the role of large language models in cybersecurity: A systematic survey (2025), <https://arxiv.org/abs/2504.15622>
 20. Van Nguyen, C., Shen, X., Aponte, R., Xia, Y., Basu, S., Hu, Z., Chen, J., Parmar, M., Kunapuli, S., Barrow, J., et al.: A survey on small language models. In: Proceedings of the 15th International Conference on Recent Advances in Natural Language Processing-Natural Language Processing in the Generative AI Era. pp. 807–821 (2025)
 21. Wang, Z., Shi, T., He, J., Cai, M., Zhang, J., Song, D.: Cybergym: Evaluating ai agents’ real-world cybersecurity capabilities at scale (2026), <https://arxiv.org/abs/2506.02548>
 22. Xiao, G., Tian, Y., Chen, B., Han, S., Lewis, M.: Efficient streaming language models with attention sinks (2024), <https://arxiv.org/abs/2309.17453>
 23. Xie, Z., Ren, X., Zheng, T., Bai, J., Fan, W., Xu, B., Li, H., Jing, H., Song, Y.: A survey on AI Agent Harness. ResearchGate Preprint (5 2026), dOI:10.13140/RG.2.2.31393.57447
 24. Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., et al.: Qwen3 technical report. arXiv preprint arXiv:2505.09388 (2025)
 25. Yang, A., Kang, F., Bu, W.: Tinysecgpt: Small-parameter llms can outperform large-parameter llms in cybersecurity. Computers, Materials and Continua **87**(2) (2026). <https://doi.org/https://doi.org/10.32604/cmc.2025.073979>, <https://www.sciencedirect.com/science/article/pii/S1546221826002729>
 26. Yang, J., Prabhakar, A., Narasimhan, K., Yao, S.: Intercode: Standardizing and benchmarking interactive coding with execution feedback (2023), <https://arxiv.org/abs/2306.14898>
 27. Yang, M., Huang, E., Zhang, L., Surdeanu, M., Wang, W.Y., Pan, L.: How is LLM reasoning distracted by irrelevant context? an analysis using a controlled benchmark. In: Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing. pp. 13329–13347. Association for Computational Linguistics (Nov 2025). <https://doi.org/10.18653/v1/2025.emnlp-main.674>, <https://aclanthology.org/2025.emnlp-main.674/>
 28. Zhang, A.K., Perry, N., Dulepet, R., Ji, J., Menders, C., Lin, J.W., Jones, E., Hussein, G., Liu, S., Jasper, D., Peetathawatchai, P., Glenn, A., Sivashankar, V., Zamoshchin, D., Glikbarg, L., Askaryar, D., Yang, M., Zhang, T., Alluri, R., Tran, N., Sangpisit, R., Yiorkadjis, P., Osele, K., Raghupathi, G., Boneh, D., Ho, D.E., Liang, P.: Cybench: A framework for evaluating cybersecurity capabilities and risks of language models (2025), <https://arxiv.org/abs/2408.08926>
 29. Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., Wang, Z., Chen, B.: H₂o: Heavy-hitter oracle for efficient generative inference of large language models (2023), <https://arxiv.org/abs/2306.14048>

A Pseudo Python code

The following pseudo-Python code provides a clear view on our proposed implementation:

```

1 def give_up():
2     """
3     Final fallback tool. Use this exclusively to terminate
4     the search and indicate failure to find alternatives.
5     You must call this ONLY as an absolute last resort,
6     strictly after all other available tools, strategies,
7     and reasoning paths have been exhaustively attempted
8     and have conclusively failed.
9     """
10    return """Give-up acknowledged. Provide a full bullet
11    point recap of what you have done and why you stopped."""
12
13 def run_strategy(strategy: str) -> str:
14     """
15     Run a strategy to solve the problem,
16     using a smart remote agent.
17
18     Args:
19         strategy: the strategy to run, in clear,
20         natural language instructions
21
22     Returns:
23         the outcome of the execution of the strategy
24     """
25     # A new, clean agent spawns when a strategy is run
26     worker_agent = Agent(tools=tools+[give_up])
27     response = worker_agent.run(problem=user_prompt+strategy)
28     return response
29
30 def context_segmentation(user_prompt, tools):
31     # Explorer run as the main loop, and autonomous calls to
32     # run_strategy trigger other sub-agent loops
33     explorer_agent = Agent(tools=[run_strategy, give_up])
34     return explorer_agent.run(problem=user_prompt)

```

B Hardware and Software Infrastructure

All experiments were conducted on consumer-grade hardware. We employed two distinct setups: a primary workstation equipped with an AMD Radeon RX 7600 XT (16GB VRAM) and a secondary node featuring an AMD Radeon RX 6600 XT (8GB VRAM). On the software side, model inference was performed using the llama.cpp framework [18], leveraging the Vulkan API (version 1.4.305)⁶. Fur-

⁶ <https://www.vulkan.org/>

thermore, to ensure a secure and reproducible environment, task execution was isolated using Docker⁷ containers, with a dedicated sandbox deployed for each CTF challenge.

C Models

Throughout our experiments, we excluded certain candidate models, specifically Qwen-3.5-4B⁸, Qwen-3.5-9B⁹, Ling-3.0-Tiny¹⁰, G9v3-3B¹¹ and LFM2.5-8B-A1B¹² variants [15,3,24,2] from final benchmarking. During preliminary testing, these models, when quantized to 4 bit precision to fit the 6GB RAM envelope, exhibited runaway generation by rapidly expanding their internal reasoning processes, exhausting the context window after doom loops or excessive reasoning, making them unfeasible to run on consumer-grade hardware.

D Intercode

To evaluate our agents, we employ Intercode’s CTFEnv [26], which provides a Dockerized environment containing approximately one hundred picoCTF challenges organized into individual directories. The environment includes a Python SDK to initialize the container and reset challenges, along with two core functions for the agent: executing Bash commands within the container and submitting captured flags.

We integrate these functions as tools within our chosen agent framework, Pydantic AI [16], enabling the agent to autonomously navigate and solve challenges within a secure, reproducible environment. Upon submission, the agent’s output is evaluated against the ground-truth flag for that specific challenge. If successful, the agent receives positive feedback; otherwise, it may iteratively refine its approach or terminate the attempt.

The included picoCTF tasks range from basic file system exploration to reverse engineering obfuscated code and exploiting network vulnerabilities, with difficulty levels scaling up to those expected of junior cybersecurity professionals. Each task provides the agent with a brief problem description and full access to the isolated challenge directory.

E Strategies

The original idea of context segmentation came up when trying to develop an engine called agent-compose. When fed a manually written declarative YAML

⁷ <https://www.docker.com/>

⁸ <https://huggingface.co/unsloth/Qwen3.5-4B>

⁹ <https://huggingface.co/unsloth/Qwen3.5-9B>

¹⁰ <https://huggingface.co/bloomer010/Ling-3.0-tiny-GGUF>

¹¹ https://huggingface.co/bartowski/ai9stars_G9v3-3B-GGUF

¹² <https://huggingface.co/unsloth/LFM2.5-8B-A1B-GGUF>

schema containing subproblems, it would solve each subproblem in a separate context and merge the results to solve larger problems. The subproblems could depend on the results of other subproblems. Eventually, despite implementing LLM-driven generation of the YAMLs, performance was still underwhelming. The most successful execution strategy in agent-compose involved a dual agent system that separated high level planning from low level execution, so that part was extracted, the rigid YAML was removed, and it became the focus of this paper.

F Additional Results

In Table 4 we present mean reward and mean token usage for individual experiments runs. As discussed in section 5, each *plain+explorer* run achieves higher reward compared to each *plain* run, albeit at the cost of significantly higher token consumption.

Table 4. Mean reward and Mean token usage for individual experiments runs.

Model	Strategy	Run ID	Reward	Tokens
gemma-4-E2B-it	plain	0	0.14	2420.47
		1	0.17	2378.77
		2	0.12	2337.20
		3	0.15	2263.57
		4	0.14	2414.94
	plain+explorer	0	0.16	8173.25
		1	0.19	8247.53
		2	0.22	7672.27
gemma-4-E4B-it	plain	0	0.25	2525.10
		1	0.23	2736.98
		2	0.21	2678.86
		3	0.25	2933.69
		4	0.21	2598.05
	plain+explorer	0	0.41	8814.74
		1	0.36	8320.30
		2	0.40	8822.53