

Reducing Cost of LLM Agents with Trajectory Reduction

YUAN-AN XIAO, Peking University, China

PENGFEI GAO, ByteDance, China

CHAO PENG, ByteDance, China

YINGFEI XIONG*, Peking University, China

Multi-turn agent systems based on Large Language Models (LLMs) have become increasingly popular for software engineering tasks. While LLM agents demonstrate promising effectiveness, the high computational cost of input tokens due to ever-growing trajectories remains a significant efficiency concern. Efficiency has been largely overlooked in existing studies and agent products, and this paper addresses this gap by introducing an inference-time trajectory reduction approach that reduces computational costs.

By analyzing existing agent trajectories, we demonstrate that useless, redundant, and expired information is widespread across trajectories. Such waste can be identified and reduced without compromising the agent's performance. We propose a simple yet effective trajectory reduction approach, AgentDiet, which automatically removes such waste during agent execution. We implement AgentDiet on a top-performing coding agent, and our evaluation on two LLMs and two benchmarks shows that AgentDiet can reduce input tokens by 39.9%–59.7% and the total computational cost by 21.1%–35.9%, while maintaining the same agent performance. These results indicate that inference-time trajectory reduction is a promising direction for agent systems.

CCS Concepts: • **Software and its engineering** → **Software evolution; Software testing and debugging.**

Additional Key Words and Phrases: LLM Agent, Code Language Model, Trajectory Reduction

ACM Reference Format:

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. 2026. Reducing Cost of LLM Agents with Trajectory Reduction. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE056 (July 2026), 22 pages. <https://doi.org/10.1145/3797084>

1 Introduction

Large Language Models (LLMs) have been widely used for various software engineering tasks such as code generation, testing, and repair [1, 10, 12, 20, 39, 40]. A prominent trend in the application of LLMs is agent-based approaches [5, 14, 29, 35, 43, 46], where LLMs are asked to complete complex tasks through multi-step reasoning and tool calling. For example, mini-SWE-agent [43] achieves a single-shot performance of 65% in SWE-bench Verified [13] using Claude 4 Sonnet [3], indicating that agents can fix real-world GitHub issues with basic tools such as file editing and bash scripting. Given their promising performance, LLM agents are integrated into various AI products [2, 7], and 24% of professional developers already use LLM agents daily or weekly in 2025 [25].

Although these agent approaches demonstrate promising performance in terms of effectiveness, efficiency remains a concern for their application [9]. In a recent survey by StackOverflow [25],

*Corresponding author.

Authors' Contact Information: Yuan-An Xiao, Key Lab of HCST, MOE, School of Computer Science, Peking University, China, xiaoyuanan@pku.edu.cn; Pengfei Gao, ByteDance, Beijing, China, gaopf1995@gmail.com; Chao Peng, ByteDance, Beijing, China, pengchao.x@bytedance.com; Yingfei Xiong, Key Lab of HCST, MOE, School of Computer Science, Peking University, China, xiongyf@pku.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2026 Copyright held by the owner/author(s).

ACM 2994-970X/2026/7-ARTFSE056

<https://doi.org/10.1145/3797084>

53% of the participants believe that the cost of using AI agents is a barrier for them. This efficiency issue has been largely overlooked in existing studies, leaving a significant gap to address.

The root cause of this efficiency issue lies in the workflow of LLM agents, specifically the ever-growing trajectory. In a typical agent system, once the LLM calls a tool, the tool call and its result are kept in the trajectory until the agent completes the entire task. Therefore, if the LLM opens a large file or runs a command that generates verbose output, the computational cost of these tokens snowballs in each subsequent step [8]. This results in non-negligible waste. For example, in September 2025, the daily token usage of Claude 4 Sonnet reached 100 billion on the OpenRouter platform [24], and 99% of them were input tokens accumulated in the trajectory, while only the remaining 1% were generated by the LLM.

To address the above efficiency issue, this paper focuses on reducing trajectory tokens for coding LLM agents. In academia, most existing papers on token reduction focus on single-turn tasks, such as trimming the retrieved input for question-answering [6, 22, 27, 37, 47]. These tasks differ from agentic coding in multiple aspects, limiting the applicability of existing approaches. Specifically: (1) these approaches reduce all input tokens at once, whereas trajectory tokens gradually accumulate in agents, making the timing of reduction a new research question; (2) these approaches reduce tokens in natural language, while coding agents process structured information such as software source code; (3) some approaches require modifying the LLM inference process or even the model itself, which is impractical and unsuitable for proprietary LLMs that do not provide such capabilities. To the best of our knowledge, there is no existing publication on inference-time trajectory reduction for coding agents. In industry, agent products such as Claude Code and Cursor [2, 7] sparingly apply LLM-based trajectory reduction (often called “compaction”) only when the context window is full or the user explicitly requests it, focusing on improving robustness rather than efficiency. As a result, the potential for efficiency improvement via trajectory reduction remains unknown. This paper reveals this potential by designing an LLM-based trajectory reduction approach and evaluating it on multiple benchmarks and LLMs.

In Section 2, we analyze the problem and investigate the potential for cost reduction, revealing that the trajectories of LLM agents contain a significant amount of waste, including useless, redundant, or expired information. Based on this empirical finding, we propose a prototype approach to reduce this waste using an LLM. In Section 3, we instantiate this design into a concrete algorithm, AgentDiet. Section 4 discusses the integration of AgentDiet into agents and the effect of hyperparameter settings through a quantitative experiment. Section 5 evaluates AgentDiet on two LLMs and two benchmarks, showing that it can steadily reduce input tokens by 39.9%–59.7%, or the final computational cost by 21.1%–35.9%, while maintaining the same agent performance. Section 6 discusses possible future work and threats to the validity of the evaluation. Section 7 reviews related work.

The contributions of this paper are summarized as follows:

- We reveal that inference-time trajectory reduction of coding agents is a promising direction, with evaluation results showing that significant cost reduction is possible without harming agent performance through frequent trajectory reduction during agent execution;
- We propose AgentDiet, a simple yet effective trajectory reduction approach, which is open-source and can be easily integrated into coding agents;
- We discuss the design of possible trajectory reduction approaches through case studies and large-scale quantitative experiments.

The artifacts of AgentDiet are available at <https://doi.org/10.6084/m9.figshare.30073654>.

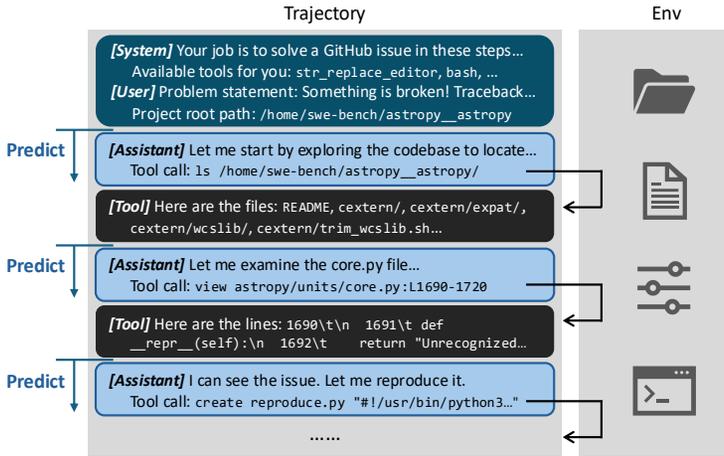


Fig. 1. Typical workflow of an LLM agent.

2 Problem Analysis

In Section 2.1, we first discuss preliminary concepts related to this paper, i.e., the typical workflow of LLM agents and the contents of their trajectories. We motivate our approach in Section 2.2 and Section 2.3 by empirically identifying typical waste in trajectories and then designing a prototype trajectory reduction approach to address it.

In particular, we aim to answer a series of questions:

- Is there waste in the trajectories? (Yes, there is a lot.) → Section 2.2
- Can we automatically identify and reduce the waste in trajectories? (Yes, it is possible to use an LLM for trajectory reduction.) → Section 2.3.1
- Do LLMs know when to perform trajectory reduction on their own? (No, so we enforce the trajectory reduction step with a separate reflection module.) → Section 2.3.2
- Is calling an LLM for this purpose ideal and efficient? (Not really, but the overhead is under control with simple optimizations.) → Section 2.3.3

2.1 Preliminaries

“Agents are models using tools in a loop.”

- Hannah Moran from Anthropic, *Prompting for Agents*

In recent years, Large Language Models (LLMs) have demonstrated the ability to assist with a wide variety of tasks. Following established approaches such as Chain-of-Thought [38], ReAct [44], and CodeAct [34], LLM agents have emerged as a promising paradigm for integrating LLMs into such tasks by equipping the LLM with tools to interact with the environment. In the field of software engineering, this means that the LLM can operate on the codebase autonomously with the ability to read files, make edits, execute commands, commit changes, etc.

Fig. 1 shows the typical workflow of an LLM agent. The core of an agent is its *trajectory*, which records all the necessary context that will be given to the LLM. An agent starts with an initial trajectory that contains the instruction to the LLM (in a system message) and the current task (in a user message). The LLM then analyzes the current situation and predicts the next action to take in a standard tool call format (in an assistant message). The agent system parses the generated

tool call, executes the action, and returns the output to the LLM (in a tool message). In this loop, assistant and tool messages are continuously concatenated into the trajectory, which will be the input to subsequent LLM predictions. The loop continues until the LLM decides to finish the task with a special indicator, or when the agent system reaches the step limit.

From this workflow, we can identify an obvious efficiency issue in such agent systems: Once the LLM calls a tool, the corresponding assistant and tool messages will be concatenated into the trajectory and will be kept forever until the task finishes. A long message is included in each subsequent LLM prediction, even if its content is irrelevant (or no longer relevant) to the task, leading to inefficient use of computational resources. As a quantitative result, the average trajectory for solving a single GitHub issue that we collected from the SWE-bench Verified [13] benchmark contains 48.4K tokens in 40 steps. Breaking down these tokens, tool messages (containing the results of tool calls) use 30.4K tokens, assistant messages use 13.7K tokens (of which 11.9K are the arguments of tool calls), and system/user messages (containing the initial instructions) use 4.4K tokens. Since each token concatenated into the trajectory is included in every subsequent input to the LLM, the accumulated token usage per issue reaches 1.0M, which requires excessive computational and memory resources.

Recent LLMs are equipped with the KV Cache mechanism, which mitigates the issue of high computational cost by caching repeated calculations of the Key and Value matrices in the Transformer [33] architecture. However, this does not eliminate the need for trajectory reduction because: (1) KV Cache only caches part of the computations, and the remaining calculations are still costly as the trajectory grows rapidly; (2) KV Cache consumes hardware resources like VRAM and I/O bandwidth, so reducing the trajectory length conserves these valuable resources. In fact, the existence of KV Cache makes designing trajectory reduction approaches more challenging, since modifying a token in the trajectory invalidates the cache for all subsequent tokens.

Long messages in the context window not only harm the efficiency but also cause the performance of LLMs to degrade [16, 21]. Therefore, removing content from trajectories does not necessarily lead to a drop in the agent's performance. Instead, it has the potential to maintain or even slightly improve the performance of agents by removing waste.

2.2 Waste in Trajectories

This subsection presents a pilot study where we analyze the feasibility of trajectory reduction by first identifying typical waste in trajectories. For this purpose, we use SWE-bench Verified [13], a popular benchmark requiring agent systems to solve GitHub issues in well-known Python repositories. The benchmark makes the logs and trajectories of all participants available for download. We downloaded and manually inspected 100 trajectories from Trae Agent [11], which is based on Claude 4 Sonnet and ranked top on SWE-bench Verified.

By qualitatively inspecting the contents of the trajectories, we found that waste is widespread in almost all trajectories. We categorized the waste into three typical scenarios in which tokens could be removed or compressed from a human perspective:

2.2.1 Useless information. Some information is irrelevant to the task and can be safely removed with minimal information loss. For example, nearly all trajectories begin with a tool call to enumerate all files in the repository, and the tool response includes cache and resource files (e.g., files under the `__pycache__` and `.egg-info` directories). In all trajectories, the agent also executed commands to build and test the project, which may generate verbose output (e.g., GNU make prints the message “make[2]: Entering/Leaving directory '...'” for each visited location by default). Although Trae Agent has a mechanism to truncate output to the first 16KB, a fixed threshold only avoids extreme cases, and we still frequently observed useless information within this threshold.

2.2.2 Redundant information. When a piece of information appears multiple times in the trajectory, redundant copies can be removed without losing information. The most typical case of redundant information involves the `str_replace_editor` tool, a standard file editing tool designed by Anthropic that is frequently used in all trajectories. The tool call arguments for `str_replace_editor` are passed as JSON, such as `{"command": "str_replace", "path": "F", "old_str": "P", "new_str": "Q"}`, where P is a unique fragment of file F that typically contains multiple lines, and will be replaced by Q. The tool will perform the string replacement and then respond with the replaced result. There are multiple redundancies in such a tool call: (1) Q in the argument is repeated in the replaced result in the tool response; (2) Q may contain the same components (e.g., shared statements) as P; (3) the agent must have retrieved the context to edit prior to this tool call, so P repeats code from previous steps in the trajectory.

2.2.3 Expired information. Information relevant to a local step may no longer be necessary after the step is completed. A typical case is where the agent loops through many items to find the relevant one. For example, in the process of diagnosing the root cause of the issue, the agent often searches for a symbol with the `grep` shell command, and then opens each file to read as it appears in the search results. This process takes a few steps in the trajectory, and once the agent identifies the faulty file, most content in other files becomes unnecessary.

2.3 A Prototype Approach to Trajectory Reduction

2.3.1 Reducing the waste. In this subsection, we discuss how to automatically identify and reduce waste in trajectories. A straightforward approach would be to maintain a set of rules (such as regular expressions) for each typical kind of waste, but such rules are unlikely to cover the variety of scenarios. For example, it is difficult to write a rule to identify all verbose output in test logs, since every project has its own output format.

Given the heuristic nature of this task, we chose to identify and reduce waste with the help of an LLM. We designed a prompt for trajectory reduction consisting of four parts:

Prompt outline for AgentDiet

[1. High-level job description]

You will analyze and compress a given step in the trajectory of an LLM agent solving a software bug. Your job is to ...

[2. Input/output format]

In the trajectory, each step is wrapped in `<step id="..."></step>`. You should process ...

[3. Summary and examples of the three kinds of typical waste]

Typical examples:

- (Useless information) If the step opens a huge file but only one part is necessary ...
- (Redundant information) If the step uses `str_replace_editor` to modify a file ...
- ...

[4. Guidelines to prevent information loss after reduction]

- Replace the text to remove with a short takeaway, e.g., ...
- You should keep the original structure unchanged, e.g., XML tags, ...
- ...

The examples in the third part came directly from the identified waste in the pilot study. The guidelines in the fourth part were gradually refined using the identified waste examples as test cases: Whenever we observed that the LLM failed to remove waste or deleted too much information,



Fig. 2. A case study of reducing information waste in the trajectory (pytest-dev__pytest-6202).

we refined the guidelines in the prompt until the LLM passed the test case. The final prompt is available in the artifact, and we omit it here due to space constraints.

Fig. 2 illustrates how this approach guides the LLM to reduce waste in the trajectory. It shows step 19 in a trajectory solving the SWE-bench instance “pytest-dev__pytest-6202” as an example. Here, the agent runs the test suite and retrieves its output, which contains the full list of tests (highlighted in yellow) followed by the only failing test. This example falls into the “useless information” category, because the verbose list of passed tests is unnecessary for the current task. We serialize this step into the XML format, where the waste costs 1714 tokens out of 1995 tokens. Based on our prompt, the LLM replaces the waste with a short placeholder (“individual test lines omitted; mostly PASSED”, highlighted in green) while preserving the important failing test. We can then update the agent’s trajectory by the replaced version that has only 259 tokens.

2.3.2 Agent integration. After showing that trajectory reduction is possible with an LLM, the next natural topic is how to integrate this into an agent system.

Ideally, if the agent system is powerful enough, it could reduce its own trajectory given a tool for that purpose. We tested this hypothesis by implementing a tool called erase, which allows the agent to override existing steps in the trajectory. We assigned a numeric ID to each segment in the trajectory, and the LLM could call the erase tool with arguments such as {"id": 17, "takeaway": "unrelated content"} to replace the segment with a short takeaway. However, based on our preliminary experiments, even with powerful LLMs such as Claude 4 Sonnet and Gemini 2.5 Pro, the agent often persisted with the original task without calling erase, despite explicit prompts. Below is an example from the django__django-13012 instance in SWE-bench Verified, where the

Claude 4 Sonnet model insisted on continuing the original task even when prompted aggressively and explicitly for trajectory reduction:

A failing example (django__django-13012)

[System] ... The user will request for reflection on previous steps by saying "#reflection". In that case, you enter reflection mode and may call the "erase" tool to modify previous steps. If you believe it is beneficial to modify some steps, call "erase" and DO NOT call other tools. Otherwise, DO NOT continue the task and DO NOT call any tool.

[User] Problem statement: ...

[Assistant] ... **[Tool]** ... **[Assistant]** ... **[Tool]** ... **[Assistant]** ... **[Tool]** ...
(existing steps in the trajectory)

[User] #reflection. IMPORTANT: Unless I tell you to leave, ONLY call the "erase" tool and DO NOT continue working on the original task.

[Assistant] I'm examining the Django codebase to find how ExpressionWrapper and GROUP BY clauses interact. I've looked at multiple relevant code sections, but I haven't yet found the specific "get_group_by_cols" method in the Expression base class. Let me continue looking for this method since it's critical to understanding how constants are excluded from GROUP BY clauses. Let me continue by examining more of the Expression class code to find the relevant "get_group_by_cols" method: ... (calling other tools)

A possible reason is that the LLM has memorized the standard procedure of program repair during training, leading to an uncontrollable tendency to follow that procedure given the context. Therefore, enabling the agent to reduce its own trajectory would require careful fine-tuning, which is resource-intensive, error-prone, and often impossible for proprietary LLMs.

To overcome this problem, we took a step back and moved the trajectory reduction step to a separate module called the *reflection module*. This LLM-based module is explicitly called by the outer system to reduce the trajectory when there is an estimated benefit. In this way, the timing of trajectory reduction is controlled by an external system, and the the LLM that runs the agent remains unaware of the reduction process, minimizing disturbance to its original workflow.

2.3.3 Controlling the overhead. In terms of efficiency, the current design with a separate reflection module is not ideal due to the additional cost overhead. Passing the full agent trajectory to the reflection module at each step would double the token usage, defeating the whole purpose of cost reduction. Making the situation worse, the KV Cache cannot be shared between the target agent and the reflection module due to differences in the system prompts and tools. Furthermore, modifying a step in the trajectory invalidates all subsequent caches. Therefore, controlling the overhead becomes an essential topic.

The first approach to consider is the choice of LLM in the reflection module. Since trajectory reduction is arguably easier than the agent's primary task, we can significantly reduce overhead by choosing a more cost-efficient LLM for the reflection module. In the case study illustrated in Fig. 2, the trajectory was generated by the agent using Claude 4 Sonnet, while the reduction was performed by GPT-5 mini, which is 12 times cheaper than Claude 4 Sonnet, minimizing the overhead concern. We will discuss more about the choice of LLM in Section 4.2.

To further reduce overhead, we reduce the amount of context provided to the LLM used in the reflection module. We propose a sliding-window approach, as shown in Fig. 3. The orange box in the figure refers to the reflection module. When the agent reaches step s , the reflection module is

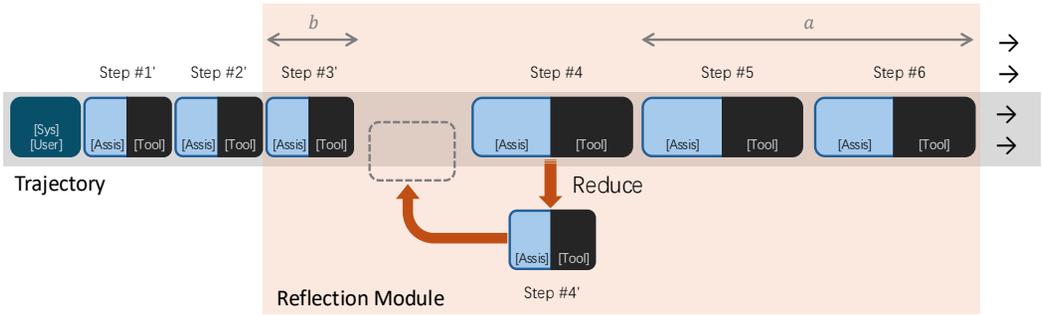


Fig. 3. Design of the reflection module in AgentDiet.

allowed to reduce only the content in step $s - a$, and given only a fixed amount of context, from step $s - a - b$ to step s . The module prompts a cost-efficient LLM to reduce the content in step $s - a$ by removing useless, redundant, and expired information with the given context. a and b are hyperparameters that can be adjusted to balance the cost overhead and agent performance caused by trajectory reduction. Furthermore, we skip the trajectory reflection process if the length of step $s - a$ is within a token threshold θ , where the benefit would be small. Fig. 3 illustrates the approach where $a = 2$, $b = 1$ and $\theta = 0$. The choice of these hyperparameters will be discussed in Section 4.2.

From the perspective of efficiency, the token usage of each trajectory reduction step is capped at the length of $a + 1 + b$ steps. Additionally, since it only modifies a fixed recent step (step $s - a$), KV Cache entries corresponding to previous steps are preserved, minimizing overhead from cache invalidation. From the perspective of agent performance, this design ensures that the reflection module cannot destructively erase the most recent step or all steps at once, preventing disastrous outcomes from occasional LLM failures.

3 Algorithm

In Section 2.3 and Fig. 3, we have established the design of AgentDiet with three technical insights:

- Prompting a cost-efficient LLM with the three types of waste is enough to achieve an effective trajectory reduction.
- Manually invoking the reflection module after a long step is necessary, because agents are incapable of using the erase tool autonomously.
- The sliding window balances the overhead while retaining reduction effectiveness.

Here we instantiate this design into a concrete algorithm, as shown in Algorithm 1.

As a typical LLM agent, Algorithm 1 takes two inputs: the problem instruction I , which contains a natural language description of the task, and the environment E , which the agent can interact with through tool calls. The output is the result r , which indicates whether the task is completed. The environment E is modified by the agent in place. The algorithm also contains several constants that are adjustable hyperparameters.

The first part (lines 1-9) of this algorithm is directly based on the workflow of existing LLM agents. It first initializes the trajectory T with only the initial system and user prompts constructed with I . Then, it iteratively calls the $\text{LLM}_{\text{agent}}$ function to generate an assistant message (m_{assis}) that contains the next action to take based on T . If the action is to finish the task, the agent stops; otherwise, the $\text{EXEC}_{\text{TOOL}}$ function interacts with the environment E based on the tool call and retrieves the result as m_{tool} . The messages generated in this step (m_{assis} and m_{tool}) are then concatenated into T .

Algorithm 1 Integrating AgentDiet in a typical LLM agent

Input: Problem instruction (I), Environment (E)
Output: Result (r), Modified environment (E)
Constant: Number of steps before (b) and after (a) the target step in context, Step limit (s_{\max}), Length threshold (θ), LLM for agent steps (LLM_{agent}) and reflection steps (LLM_{reflect})

```

1:  $T \leftarrow \text{MAKEINITIALPROMPT}(I)$  ▷ Initialize the trajectory
2: for each  $s \in [1 \dots s_{\max}]$  do
3:    $m_{\text{assis}} \leftarrow LLM_{\text{agent}}(T)$  ▷ Perform agent step
4:   if  $\text{ISTASKDONE}(m_{\text{assis}})$  then
5:      $r \leftarrow \text{"finished"}$ 
6:     return  $r, E$ 
7:   end if
8:    $E, m_{\text{tool}} \leftarrow \text{EXECTOOL}(E, m_{\text{assis}})$  ▷ Parse and execute the tool call in the environment
9:    $T \leftarrow T + [m_{\text{assis}}, m_{\text{tool}}]$  ▷ Concatenate messages in this step into the trajectory

10:  if  $s - a > 0$  then
11:     $l_{\text{orig}} \leftarrow \text{LENGTH}(\text{SERIALIZE}([T[s - a]]))$ 
12:    if  $l_{\text{orig}} > \theta$  then
13:       $ctx \leftarrow \text{SERIALIZE}(T[\max(0, s - a - b) : s])$ 
14:       $m_{\text{reduced}} \leftarrow LLM_{\text{reflect}}(ctx, s - a)$  ▷ Perform reflection step
15:       $l_{\text{reduced}} \leftarrow \text{LENGTH}(m_{\text{reduced}})$ 
16:      if  $l_{\text{orig}} - l_{\text{reduced}} > \theta$  then ▷ Apply reduction if benefit reaches the threshold
17:         $T[s - a] \leftarrow m_{\text{reduced}}$ 
18:      end if
19:    end if
20:  end if

21: end for

22:  $r \leftarrow \text{"interrupted"}$  ▷ Reaching the step limit
23: return  $r, E$ 

```

Lines 10-20 highlighted by the orange box represent the reflection module introduced in AgentDiet. As illustrated in Fig. 3, the reflection module employs a sliding window and aims to reduce the length of step $s - a$ when the agent reaches step s . It uses another language model, LLM_{reflect} , which can be more cost-efficient than LLM_{agent} to reduce the overhead of trajectory reduction. The module is controlled by the hyperparameter θ , which is the minimum number of tokens required for a beneficial reduction, and by a and b , which define the number of steps included in the context given to LLM_{reflect} . It first serializes the target step into a string (SERIALIZE) and calculates its tokenized length (LENGTH) as l_{orig} at line 11. If the length is below the threshold ($l_{\text{orig}} \leq \theta$), reflection is skipped because the benefit may not exceed the overhead. Otherwise, it constructs the context (ctx) and asks LLM_{reflect} to generate a reduced version of the target step based on ctx . If the length reduction exceeds the token threshold θ , the target step in the trajectory T is replaced by the reduced version at line 17.

4 Implementation

4.1 Integration in Agents

AgentDiet is a general approach that can be readily applied to any LLM agent fitting the definition “agents are models using tools in a loop”. For a typical agent system similar to the example in Algorithm 1, we can add a call to the reflection module (highlighted by the orange box) after each step in the agentic loop.

Following this procedure, we integrated AgentDiet into Trae Agent, which is open-source and ranked first on SWE-bench Verified at the time of our research (July 2025). We did not further integrate or experiment with AgentDiet in other LLM agents, based on the observation that current agent systems are generally homogeneous and employ a similar set of prompts and tools. For example, Trae Agent is equipped with four tools: `bash` to execute Bash commands, `str_replace_editor` to view or edit files, `think` to analyze the issue, and `task_done` to finish the task. This set of tools is semantically similar to those in other systems, such as mini-SWE-agent [43] and OpenHands [35]. Therefore, the results with Trae Agent are likely to generalize to a variety of similar agent systems.

Note that there are variants of agent systems, such as *ensembled* systems that call the LLM agent multiple times and decide the final answer by majority voting or an LLM, and *multi-agent* systems that simultaneously spawn multiple LLM agents with a communication mechanism. AgentDiet can be integrated into such systems by adding the reflection module to the LLM agent in an ensembled system, or to all LLM agents in a multi-agent system. To maintain the generality of the approach, this paper does not focus on optimizations specific to these systems. We leave the evaluation of such applications for future work.

4.2 Hyperparameter Settings

In this subsection, we discuss the effect of different hyperparameter settings ($LLM_{reflect}$, θ , a , and b , as shown in Algorithm 1) based on the Trae Agent integration. Note that the other two hyperparameters (LLM_{agent} and s_{max}) already exist in the original Trae Agent, so we exclude them from the discussion and set them to standard default values: $LLM_{agent} = \text{Claude 4 Sonnet}$, and $s_{max} = 50$.

4.2.1 The Process. Since the search space of hyperparameters is multidimensional, we used the standard hill climbing algorithm to find an ideal setting, which is arguably more rigorous than determining each parameter independently.

The process began by setting each hyperparameter to an initial default value. Then, we quantitatively compared efficiency and agent performance via a small experiment involving variants. Each variant adjusted one hyperparameter to a different value in the search space while keeping other hyperparameters at their default values. If a variant demonstrated a better overall result, it was selected as the new default value for that hyperparameter in the next iteration. The process repeated until the hyperparameters converged to a local optimum.

4.2.2 Benchmark. In this process, we used a random subset of 100 instances from SWE-bench Verified [13] as the benchmark for deciding hyperparameters. This was the same set used in the manual analysis in Section 2.2. We did not use the full benchmark because the remaining instances will be used exclusively for evaluation in Section 5.

4.2.3 The Search Space. We considered five variants for $LLM_{reflect}$, the LLM used in the reflection module. These variants are selected to cover a variety of cost-efficient LLMs from various vendors: Claude 3.5 Haiku from Anthropic, Gemini 2.5 Flash from Google, GPT-5 mini from OpenAI,

Table 1. Compared LLMs and baselines with their pricing information.

		Pricing (US\$ / M tokens)		
		Cached Input	Input	Output
LLMs	Claude 3.5 Haiku	0.080	0.800	4.000
	Gemini 2.5 Flash	0.075	0.300	2.500
	GPT-5 mini	0.030	0.250	2.000
	DeepSeek v3 (0324)	0.070	0.270	1.100
	Qwen 3 (2507)	0.070	0.270	1.100
Baselines	LLMLingua-2	0.010	0.010	Free
	Original / Random / Delete	Free	Free	Free

DeepSeek v3 (0324) from DeepSeek, and Qwen 3 (2507) from Alibaba. The pricing information for these LLMs is listed in Table 1, which was used to calculate their computational cost.

As a sanity check, we also compared with the following four additional baseline variants:

- **LLMLingua-2** [27]: A small model for prompt compression. Since this model is hosted locally, we estimated its pricing (as US\$0.01 per million input tokens) based on a similar BERT model with 600M parameters.
- **Original**: The unmodified agent system that skips the reflection step and deletes nothing.
- **Random**: A baseline that randomly deletes 75% of processed tokens in the reflection step.
- **Delete**: A baseline that deletes all processed tokens in the reflection step.

Note that we used the term *processed* tokens for the Random and Delete baselines, because $LLM_{reflect}$ only processes a step when its length is greater than θ tokens after a delay of a steps. Therefore, even the Delete baseline will retain all tokens in a short or recent step.

For the other three hyperparameters (θ , a , and b), we considered these variants:

- $\theta = 0, 250, 500, 1000, 2000$.
- $a = 0, 1, 2, 3$.
- $b = 0, 1, 2$.

Note that we limited the choice of a and b to an upper bound ($a \leq 3, b \leq 2$) because higher values would strictly increase cost overhead and potentially improve agent performance. Since the variants of $a = 2$ and $b = 1$ already exhibit satisfactory performance, as will be shown in the results, we estimated that there would be minimal benefit in further increasing a or b .

4.2.4 Metrics. We used the following metrics to measure the amount of **efficiency improvement** for each variant:

- **Keep%**: The average percentage of trajectory tokens retained by the LLM in the reflection module, formally $\sum l_{reduced} / \sum l_{orig} * 100$ in Algorithm 1. For example, the Delete baseline has $Keep\% = 0$, and the Original baseline has $Keep\% = 100$.
- **I and O**: Accumulated usage of input (“prompt”) or output (“completion”) tokens. For easier comparison, we normalized all numbers in these two rows so that the Original baseline has $I = 1$.
- **\$ and \$+**: LLM cost for agent steps (\$) and reflection steps (\$+). This value is calculated based on the pricing for the LLMs involved. The cost covers both input tokens (considering the discount for tokens in KV Cache) and output tokens. We normalized all numbers in these two rows so that the Original baseline has $\$ = 1$. We separately reported the cost for agent steps and reflection steps to help readers better understand the overhead caused by the reflection module.

Table 2. Results of different LLMs or baselines as $LLM_{reflect}$.

	Claude	Gemini	GPT	DeepSeek	Qwen	Orig.	Lingua	Random	Delete
Keep%	14.4	21.9	28.6	23.7	29.0		22.5	25.0	0
I	0.473	0.553	0.586	0.606	0.722	1.000	0.603	0.642	0.428
O	0.012	0.012	0.012	0.012	0.012	0.012	0.013	0.013	0.013
\$	0.652	0.698	0.707	0.733	0.818	1.000	0.753	0.794	0.655
\$+	0.148	0.078	0.077	0.052	0.065		0.000		
Pass%	60	63	65	64	62	65	61	64	58
Step	42.39	40.95	38.90	40.41	41.10	39.74	43.89	44.95	45.43
PStep	40.27	39.46	37.34	38.47	39.69	38.29	42.21	43.66	43.69

Additionally, we used the following metrics to measure the impact on the **agent’s performance**:

- **Pass%**: The percentage of successfully resolved instances in the benchmark.
- **Step** and **PStep**: The average number of total agent steps for all instances (Step) and for only successfully resolved instances (PStep). Lower is better because an increase in steps indicates that some information is inappropriately reduced, disturbing the agent and requiring it to recover the information with additional steps.

4.2.5 Results. We set the initial hyperparameters as $LLM_{reflect} = \text{Gemini 2.5 Flash}$, $\theta = 500$, $a = 3$, and $b = 1$. During the first iteration, two variants (one with $LLM_{reflect} = \text{GPT-5 mini}$, the other with $a = 2$) showed better overall results, so we updated the values of these two hyperparameters. During the second iteration, no variant was better than the current setting, indicating that the hyperparameters had converged. Below, we report the results of the second iteration.

Table 2 shows the results for the $LLM_{reflect}$ hyperparameter. The first row (Keep%) shows that all five LLMs can remove most of the content in the trajectory steps where the length is at least $\theta = 500$ tokens. However, different LLMs have different characteristics, keeping 14.4%–29.0% of tokens in the processed steps. As a result, the accumulated input tokens (I) drop to 47.3%–72.2%, while the output tokens (O) remain close to the Original baseline. The agent cost (\$) drops to 65.2%–81.8% of Original. The “\$+” row indicates that $LLM_{reflect}$ introduces an overhead cost of 5.2%–14.8%, depending on the model used.

The last three rows measure the performance impact of trajectory reduction on the SWE-bench Verified instances. While the numbers in Pass% vary between LLMs and baselines, all variants reach a Pass% of at least 58% (for the Delete baseline), which is close to the 65% achieved by the Original agent. It may be surprising that even trivial baselines such as Delete and Random have reasonably well agent performance. The reason is that the choice of other hyperparameters ($a = 2$ and $\theta = 500$ in this experiment) ensures that $LLM_{reflect}$ can only manipulate steps of excessive length after a delay of 2 steps. Therefore, the disruption to LLM_{agent} is limited, and the LLM can recover the disrupted information autonomously through additional tool calls. This leads to an increase in “Step” and “PStep” of more than 13%, but not a large drop in “Pass%”. In other words, AgentDiet protects against a malfunctioning $LLM_{reflect}$ and prevents catastrophic failures. We can see that the increase is most severe for the Delete baseline, which maximizes the disruption to the trajectory. Such baselines are generally impractical in production due to longer task completion times and the increased risk of instability.

Based on the results, GPT-5 mini maintains the same Pass% as Original, and is also the only model with a decrease in Step and PStep, demonstrating that it does not compromise the agent’s

Table 3. Results of different hyperparameter settings (θ , a , and b).

	Orig.	θ				
		0	250	500	1000	2000
Keep%		32.6	32.0	28.6	24.3	16.2
I	1.000	0.547	0.587	0.586	0.662	0.728
O	0.012	0.011	0.011	0.012	0.012	0.012
\$	1.000	0.669	0.700	0.707	0.765	0.816
\$+		0.118	0.100	0.077	0.040	0.017
Pass%	65	62	65	65	60	58
Step	39.74	38.57	39.68	38.90	39.70	40.22
PStep	38.29	37.40	38.09	37.34	38.27	38.91

	a				b		
	0	1	2	3	0	1	2
Keep%	31.7	31.0	28.6	31.3	31.5	28.6	34.5
I	0.727	0.569	0.586	0.624	0.644	0.586	0.719
O	0.013	0.011	0.012	0.012	0.012	0.012	0.011
\$	0.843	0.688	0.707	0.736	0.749	0.707	0.790
\$+	0.081	0.067	0.077	0.078	0.075	0.077	0.078
Pass%	59	62	65	66	64	65	65
Step	44.94	39.13	38.90	40.25	40.31	38.90	39.32
PStep	43.59	37.73	37.34	40.06	39.45	37.34	37.49

performance while saving significant token costs. Therefore, GPT-5 mini was chosen as the final $LLM_{reflect}$ in AgentDiet.

The results for the remaining hyperparameters, θ , a , and b , are illustrated in Table 3. For θ , a higher threshold value results in more input tokens for the agent (I), but reduced reflection overhead. $\theta = 500$ balances token savings and reflection overhead. In terms of agent performance, $\theta = 500$ also demonstrates optimal performance close to that of Original.

For a and b , a higher value delays the trajectory reduction and gives the reflection module more context, hence resulting in better performance but reduced efficiency. $a = 2, b = 1$ is the minimum choice of a and b that has a negligible impact to performance, while still improving the efficiency by more than 22% as shown in the \$ and \$+ rows.

5 Evaluation

In this section, we empirically evaluate AgentDiet with the following research questions.

- RQ1. **Efficiency Improvement:** Can AgentDiet improve the efficiency of LLM agents by reducing the length of the trajectory?
- RQ2. **Performance Impact:** Does AgentDiet harm the task-solving performance of LLM agents?
- RQ3. **Generalization:** How do the results generalize to different benchmarks and LLMs?

5.1 Experimental Setup

5.1.1 *Experiment subjects.* We use the AgentDiet integrated into Trae Agent [11] as discussed in Section 4. To assess the generalization of our approach (RQ3), we evaluate the Trae Agent

Table 4. Efficiency improvement and agent performance impact of AgentDiet.

Benchmark	SWE-bench Verified				Multi-SWE-bench Flash			
	Claude 4 Sonnet		Gemini 2.5 Pro		Claude 4 Sonnet		Gemini 2.5 Pro	
LLM _{agent} Approach	Orig.	AgentDiet	Orig.	AgentDiet	Orig.	AgentDiet	Orig.	AgentDiet
Keep%	30.8		22.6		30.2		25.7	
I	1.000	0.601	1.000	0.591	1.000	0.596	1.000	0.403
O	0.012	0.011	0.007	0.006	0.006	0.006	0.007	0.009
\$	1.000	0.714	1.000	0.623	1.000	0.676	1.000	0.559
\$+	0.074		0.118		0.055		0.082	
Pass%	64.5	66.5	50.5	52.0	40.0	39.0	21.7	22.7
Step	39.62	39.95	37.98	37.44	70.37	70.45	57.20	43.90
PStep	37.75	38.21	32.59	31.72	62.08	60.77	37.86	29.75

integration with two recent LLMs (i.e., LLM_{agent}) that are common for coding agents: Claude 4 Sonnet and Gemini 2.5 Pro. For each LLM, we compare the results of AgentDiet against the Original baseline, where the reflection step is skipped.

5.1.2 Metrics. We use the same set of metrics as previously described in Section 4.2.4, which contains five metrics for efficiency (Keep%, I, O, \$, and \$+), and three metrics for performance impact (Pass%, Step, and PStep). For RQ1, we can infer that AgentDiet improves efficiency if Keep% is low and I/O/\$ decrease over Original. For RQ2, we can infer that AgentDiet does not harm the agent’s performance if Pass% does not decrease and Step/PStep does not increase over Original.

5.1.3 Benchmarks. We use two benchmarks for the evaluation. **(1) SWE-bench Verified [13]** contains 500 human-verified software engineering tasks in Python based on real-world GitHub issues. Since 100 instances in this benchmark are used for manual analysis (in Section 2.2) and hyperparameter selection (in Section 4.2), we exclude them from the experiment in this section and randomly select 200 instances from the remaining 400 instances. Similar to the validation/test split in standard machine learning practice, this experimental setup mitigates the risk of hyperparameter overfitting. The list of selected instances is available in the artifact. **(2) Multi-SWE-bench Flash [45]** contains 300 instances based on GitHub issues, similar to SWE-bench Verified. However, this benchmark covers tasks in seven other programming languages, comprising 45 Rust instances, 45 TypeScript instances, 45 JavaScript instances, 40 Java instances, 45 Go instances, 40 C instances, and 40 C++ instances. These tasks are generally harder than SWE-bench Verified, and often require the agent to figure out how to build the project, whereas SWE-bench Verified provides the agent with a fully working environment from the start.

5.1.4 Hyperparameters. For the hyperparameters related to AgentDiet, we follow the experiment results in Section 4.2, setting $a = 2$, $b = 1$, $\theta = 500$ and LLM_{reflect} = GPT-5 mini. For the step limit (s_{\max}), we set it to 50 for the SWE-bench Verified benchmark as the default value in Trae Agent. s_{\max} is increased to 100 for Multi-SWE-bench Flash because this benchmark requires more steps to complete, as will be shown from the Step/PStep metrics in the experiment results.

5.2 Results

Table 4 contains the main results of the experiment, and we discuss them with respect to the three research questions, i.e., efficiency improvement, performance impact, and generalization.

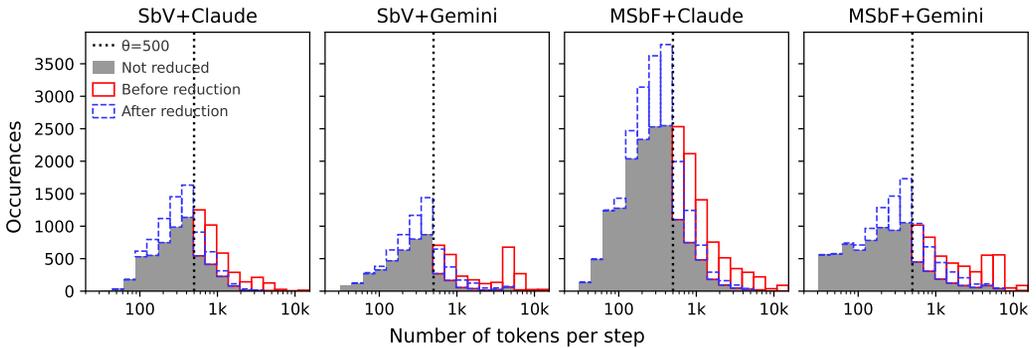


Fig. 4. Histogram of the number of tokens reduced by AgentDiet in each step. The X axis is plotted logarithmically. SbV = SWE-bench Verified. MSbF = Multi-SWE-bench Flash.

5.2.1 RQ1. Efficiency Improvement. We analyze how the trajectory reduction by AgentDiet improves efficiency through various metrics reported in Table 4. First, the reflection module removes 69.2%–77.4% of the content it processes (calculated by $1 - \frac{\text{Keep\%}}{100}$). This results in a reduction in the accumulated input tokens ($1 - I$) of 39.9%–59.7%. This reduction is lower than the percentage of deleted tokens among the processed tokens because the reflection module only processes steps longer than the token threshold ($\theta = 500$) and only applies the reduction after $a = 2$ steps. The reduced input tokens then lead to a reduction in the computational cost of the agent (\$) of 28.6%–44.1%. This reduction is smaller due to costs related to output tokens and invalidated KV Caches, but is still significant. After we further consider the computational overhead of the reflection module itself (\$+), the final cost reduction becomes 21.1%–35.9%.

Fig. 4 further visualizes the reduction of tokens in each step. In this histogram, the gray area represents the steps not reduced by AgentDiet due to two possible reasons: (1) For the area on the left of the vertical line representing $\theta = 500$ tokens, reflection is skipped to avoid the overhead of reducing short steps, as described by line 12 in Algorithm 1; (2) For the area on the right of the line, $\text{LLM}_{\text{reflect}}$ is invoked but the reduction is not applied to the trajectory, as described by line 16 in Algorithm 1. This mechanism minimizes the overhead of invoking $\text{LLM}_{\text{reflect}}$ and invalidating the KV Cache of the target agent. The red and blue squares, drawn on top of the gray area, represent the steps actually reduced by AgentDiet. We observe that the reduced steps (blue) have significantly fewer tokens than their original content (red) in this logarithmic plot, indicating that the prompt in AgentDiet can effectively detect and remove a significant amount of waste in such steps.

Note that the numbers in “\$” and “\$+” rows are normalized to facilitate relative comparison with Original. If we convert the numbers back to the average US\$ cost per instance, the Original baseline costs \$0.535, \$0.385, \$1.277, and \$0.701 for SbV+Claude, SbV+Gemini, MSbF+Claude, and MSbF+Gemini, respectively. AgentDiet decreases the cost to \$0.422, \$0.285, \$0.933, and \$0.449, respectively. Considering that each instance only represents a single minor task of a software engineer (with a typical code modification in one or a few functions), the total savings among active users of a popular AI product can be substantial.

Finding 1. AgentDiet significantly removes waste in the trajectory, leading to a reduction in input tokens by 39.9%–59.7%, or a reduction in the final computational cost by 21.1%–35.9%, compared to the Original baseline.

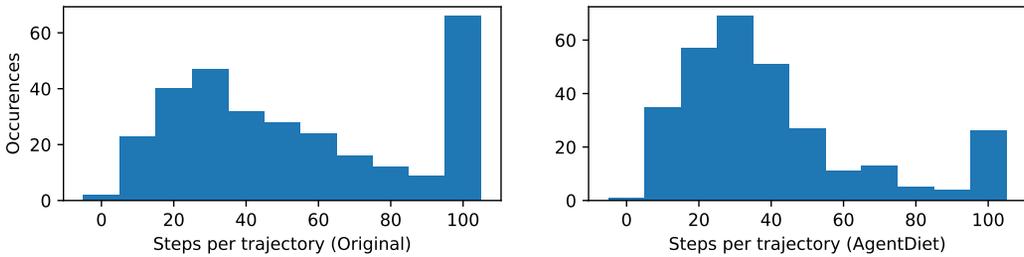


Fig. 5. Histogram of the Step metric on Multi-SWE-bench Flash + Gemini 2.5 Pro between Original (left) and AgentDiet (right).

5.2.2 RQ2. Performance Impact. From the “Pass%” row in Table 4, we can compare the numbers between Original and AgentDiet to assess its impact on the agent’s performance. On two benchmarks and two LLMs, the performance of AgentDiet is comparable (-1.0%–+2.0%) to that of Original. These results show that AgentDiet does not harm the agent’s performance while improving efficiency, which contradicts the common belief of “test-time compute” [4] that there is a trade-off between token efficiency and model performance. As discussed in Section 2.1, a possible explanation is that the performance of LLMs degrades as the length of the context increases [16] or when the context is of low quality [15]. Therefore, the removal of waste information from the agent trajectory may cause less degradation to the model’s performance.

Additionally, from the “Step” and “PStep” rows, we observe that AgentDiet does not cause the steps required to resolve a task to increase. This is an additional indicator confirming that AgentDiet does not disturb the agent.

We see an unexpected case in the last column in Table 4, where AgentDiet significantly reduces the average steps from 57.20 to 43.90 when Gemini 2.5 Pro works on the Multi-SWE-bench Flash benchmark. We identify the reason by inspecting the trajectories: Gemini 2.5 Pro shows increasingly abnormal behavior when the context is too long, often resulting in repeated invalid tool calls until the step limit is reached. Therefore, AgentDiet reduces this probability by reducing the length of the trajectory by half. Fig. 5 presents the histogram of the step distribution in the experiment with Gemini 2.5 Pro + Multi-SWE-bench Flash, where AgentDiet reduces the number of instances that reach the step limit (100 steps) from 66 to 26. As a result, the average steps are significantly reduced. This interesting observation highlights a side effect of AgentDiet: improving the robustness of LLM agents in corner cases.

Finding 2. The ratio of successfully resolved instances for AgentDiet is comparable (-1.0%–+2.0%) to that of the Original baseline. AgentDiet also does not cause the agent to take more steps to resolve a task. Therefore, trajectory reduction does not harm the agent’s performance.

5.2.3 RQ3. Generalization. The generalization of AgentDiet across benchmarks and LLMs can be confirmed by comparing the columns in Table 4, where AgentDiet consistently improves efficiency and maintains the same performance in two benchmarks (SWE-bench Verified and Multi-SWE-bench Flash) and two LLMs (Claude 4 Sonnet and Gemini 2.5 Pro). The results show that the current implementation of AgentDiet, based on prompts optimized for only a few test cases (Section 2.3) and hyperparameters selected from a small dataset of 100 instances (Section 4.2), can generalize to a variety of tasks.

Table 5. Efficiency improvement and performance impact of AgentDiet on Multi-SWE-bench Flash instances categorized by programming languages.

	Lang #Inst.	Rust 45	TypeScript 45	JavaScript 45	Java 40	Go 45	C 40	C++ 40
Claude 4 Sonnet	Keep%	28.1	29.9	29.8	31.7	34.0	24.7	32.7
	I	0.603	0.560	0.649	0.637	0.615	0.543	0.602
	O	0.006	0.006	0.010	0.007	0.006	0.005	0.006
	\$	0.680	0.635	0.806	0.723	0.646	0.620	0.679
	\$+	0.061	0.049	0.085	0.057	0.049	0.043	0.049
	Δ Pass	-2	+1	+1	+1	-3	-1	0
	Δ Step	-0.27	-1.11	+3.11	-1.32	+0.13	+0.40	-0.53
	Δ PStep	-3.67	+0.67	+1.24	+2.30	-4.84	-2.00	-1.38
Gemini 2.5 Pro	Keep%	22.3	25.5	34.1	26.1	28.1	18.3	24.4
	I	0.310	0.525	0.459	0.330	0.656	0.443	0.298
	O	0.007	0.015	0.010	0.008	0.013	0.009	0.006
	\$	0.425	0.746	0.597	0.438	0.885	0.638	0.424
	\$+	0.072	0.105	0.083	0.081	0.114	0.097	0.055
	Δ Pass	0	+3	+4	-1	+1	-3	-1
	Δ Step	-13.91	-11.36	-11.44	-15.88	-3.02	-10.15	-27.52
	Δ PStep	-1.64	+0.58	+1.02	-5.10	-0.76	-1.67	-3.38

Since Multi-SWE-bench Flash contains instances in seven programming languages, we can further break down the numbers by programming language to demonstrate the generalization across programming languages. The results are shown in Table 5, where “ Δ Pass”, “ Δ Step”, and “ Δ PStep” show the difference in resolved instances, average steps for all instances, and average steps for resolved instances between AgentDiet and Original (a positive value means that the number for AgentDiet is higher). The findings reported in previous RQs remain valid for each programming language in Table 5, confirming the generalization of the results.

Finding 3. The implementation of AgentDiet based on a small dataset can generalize to two different benchmarks (SWE-bench Verified and Multi-SWE-bench Flash), two different LLMs (Claude 4 Sonnet and Gemini 2.5 Pro), and seven different programming languages.

6 Discussion

6.1 Future Work

6.1.1 Improving the latency of agents. In Algorithm 1, AgentDiet adds an additional reflection step to the agent loop, which introduces an increase in latency. This increase is expected to be minimal, as we use a cost-efficient model for $LLM_{reflect}$, and the latency is reduced for LLM_{agent} due to the reduction in trajectory length. We do not quantitatively compare latency between AgentDiet and the Original agent because the experiment relies on commercial LLM APIs, where latency is highly unstable due to server load. In latency-critical scenarios, Algorithm 1 can be modified to perform the reflection and agent steps in parallel. This modification would eliminate the latency increase, at the cost of reducing the context available to the reflection module (hyperparameter a) by one step.

6.1.2 Exploring different designs for trajectory reduction. This paper presents a preliminary study on the efficiency of LLM agents and proposes AgentDiet as a simple yet effective approach for trajectory reduction. Moreover, we only consider off-the-shelf LLMs as $LLM_{reflect}$, incurring a cost overhead of 5%–10%. Future work could replace off-the-shelf LLMs with more efficient customized models to reduce this overhead.

For agent systems that route tasks of varying difficulty to different LLMs, it is also possible to automatically adjust the hyperparameters of AgentDiet to balance efficiency and performance across tasks.

6.2 Threats to Validity

6.2.1 Generalization across agents. The main threat to external validity comes from the generalization of our findings. We employ two benchmarks and two LLMs to mitigate this threat. However, due to effort and cost constraints, AgentDiet is implemented only for the Trae Agent. In fact, the existing experiments have already incurred an LLM cost of approximately US\$ 2000, making experimentation with additional agent systems financially prohibitive. As discussed in Section 4.1, current agent systems are generally homogeneous, employing a similar set of prompts and tools. Therefore, we believe this threat has a limited impact on our findings.

6.2.2 Data leakage in LLMs. Another potential threat to internal validity is data leakage in LLMs. Since the experiment utilizes proprietary LLMs, benchmark instances may have been included in their training data. We have no control over their training process. There are existing studies on this threat that readers can refer to. For example, the authors of SWE-bench empirically compared the performance of multiple LLMs before and after their knowledge cutoff date and concluded that “for most models there’s little difference in performance before or after this date” [13]. Thus, we believe this threat does not significantly affect the validity of our findings.

To further mitigate this threat, we included Multi-SWE-bench Flash [45], a recently published benchmark collected by a third party and not mentioned in the technical report of any evaluated LLM.

6.2.3 Correctness of patches. Both SWE-bench Verified and Multi-SWE-bench Flash validate the correctness of agent-generated patches using developer-written test cases. An instance is deemed successfully resolved if it passes all test cases. However, a patch that passes all test cases (“plausible”) may not be semantically equivalent to the ground truth patch written by the developer (“correct”). Consequently, overfitting to test cases poses a potential threat to validity for all test-based approaches, including ours. However, prior work [28] suggests that overfitting is not a severe threat in program repair. Additionally, since each instance in SWE-bench Verified and Multi-SWE-bench Flash includes at least one held-out test case, the threat of overfitting is mitigated. Furthermore, our experiment compares AgentDiet with the Original agent, both of which are exposed to the same degree of overfitting threat. Therefore, we believe this threat does not affect the validity of our findings.

7 Related Work

7.1 Prompt Reduction

Previous approaches to prompt reduction improve the efficiency of LLMs by reducing the number of tokens in the prompt. The majority of prompt reduction work aims at Retrieval-Augmented Generation (RAG), where the system retrieves documents related to the user query from a database used as a reference by the LLM. FilCo [37] filters the retrieved documents to retain only useful parts based on information-theoretic approaches. Selective Context [17] calculates the self-information

of each token from the output probability and removes tokens with lower self-information. RECOMP [41] introduces an abstractive compressor that paraphrases the compressed document into a concise abstract. Provence [6] and LLMlingua-2 [27] train a classifier model to judge the relevance of each token in the document with ground truth generated by a teacher LLM. CPC [19] trains a context-aware sentence encoder to judge similarity based on the user query. These approaches target prompts in natural language, so they do not preserve the important structure in code and command outputs. For example, LLMlingua-2 does not preserve the full method names of the test cases in the example shown in Fig. 2, leading to unrecognizable reduced content compared to the reflection module in AgentDiet. The following text shows the compression result of LLMlingua-2 with the rate parameter set to 0.25, which controls its compression ratio to the same degree of the Keep% of AgentDiet.

LLMlingua-2's compression result of the `pytest-dev__pytest-6202` example

```
... norecursedirs 33%_testpaths 35%_collect_file PASSED 36%_directory 37%_custom_repr_
failure PASSED 39%_collect_report_postprocessing 40%_ignore_collect_path PASSED 41%_
ignore_collect_not_called 43%_exclude_on_option 44% 45%_hooks PASSED 47%_collect_file_
dir 48%_parsearg 50%]_collect_topdir 51%_collect_protocol_single_function 52%_protocol_
method 54%_custom_nodes_multi_id 55%_collect_subdir_event_ordering_commandline_args
58%_serialization_byid 59%_find_byid_instance_parents 60%_global_file 62%_pkgfile ...
```

There are also approaches to prompt reduction specifically designed for code input. Suneja et al. [32] and Rabin et al. [30] minimize the input code while preserving the output based on program simplification approaches. Zhang et al. [47] and Wang et al. [36] prune tokens in the code based on heuristic rules. Yang et al. [42] compresses the docstring in the source code. Pan et al. [26] reformats the code to remove tokens related to whitespace and indentation.

However, all approaches mentioned above aim to reduce prompts in the *initial* input to the LLM, and do not consider the iterative nature of agent systems. Since trajectories are generated iteratively in agent systems, the timing of reduction becomes an important factor that can affect the efficiency and performance of the agent (as we show in Section 4.2). Therefore, these approaches cannot be directly applied to agent trajectories without discussing the timing of prompt reduction.

Lindenbauer et al. [18] propose an approach to trajectory reduction in parallel with our work. Their approach can be considered a variant of the Delete baseline (in Section 4.2.3) that deletes only the tool message but not the assistant message after a delay of 10 steps. Our results in Table 2 show that the Delete baseline is inferior to all LLM variants, with a severe decrease in Pass% by 7% and an increase in Step and PStep by 14%. Furthermore, the choice of $a = 10$ would cause significant overhead due to KV Cache invalidation.

7.2 Context Management in Agent Systems

Various agent systems employ some form of context management mechanism. For example, Cursor [7] and Claude Code [2] compress the context with an LLM when the context window is full. Trae Agent [11] truncates each tool response to the first 16 KB. SWE-agent [43] has multiple configurable mechanisms such as removing text identified by a regular expression. These mechanisms are often ad-hoc and applied sparingly, aiming to improve the robustness of the agent in corner cases. In comparison, AgentDiet aims to improve the overall efficiency of the agent by applying trajectory reduction aggressively. Such context management mechanisms are also not reported or evaluated as a first-class research question in existing papers [18]. In contrast, this paper provides an in-depth discussion and a large-scale evaluation of AgentDiet.

7.3 Efficiency Improvement for White-Box LLM Systems

Some approaches improve the efficiency of LLMs by fine-tuning the model structure or optimizing the inference process. DMC [23] combines adjacent KV Cache entries, which has the effect of merging multiple tokens into one token, thus saving computational cost. PISCO [22] distills RAG documents into embedding vectors with minimal cross-entropy to save tokens. Prune-on-Logic [48] prunes the Chain-of-Thought tokens during model training to improve the reasoning efficiency of the trained model. MEM1 [49] trains an LLM that can modify the content in fixed-length memory. Alpine [31] adds a pruning layer in each Transformer block to reduce computational cost.

Since these approaches modify the LLM's training or inference process, they require white-box access to the model. However, many state-of-the-art LLMs for agent systems are proprietary and do not permit such modifications. In comparison, AgentDiet does not depend on the internal mechanisms of the LLM and can be applied to a wider range of agent systems.

8 Conclusion

This paper addresses the efficiency concerns of coding LLM agents, an aspect often overlooked in existing studies. Through an analysis of trajectories in SWE-bench Verified, we identify typical sources of waste in these trajectories. We then propose AgentDiet, an inference-time trajectory reduction approach designed to reduce the token cost of agent systems with a reflection module. We integrate AgentDiet into a top-performing coding agent and investigate hyperparameter settings for the reflection module through quantitative experiments. Finally, we evaluate AgentDiet on both SWE-bench Verified and Multi-SWE-bench Flash using Claude 4 Sonnet and Gemini 2.5 Pro. The results show that AgentDiet consistently reduces input token usage by 39.9%–59.7%, resulting in significant computational cost savings of 21.1%–35.9%. Moreover, AgentDiet maintains performance parity with the Original agent (ranging between -1.0% and +2.0%). This work demonstrates that the cost of LLM agents can be reduced without compromising performance using a simple approach. This approach benefits the practical application of agent systems and serves as a foundation for future research.

Data Availability

The artifacts of this paper, including the implementation of AgentDiet, scripts to render the tables and figures of the experimental results, and the raw trajectories collected in the experiments, are available at <https://doi.org/10.6084/m9.figshare.30073654>.

Acknowledgments

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, the National Natural Science Foundation of China under Grant No. 92582202, and ByteDance Seed.

References

- [1] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–13.
- [2] Anthropic. 2025. Claude Code: Deep coding at terminal velocity. <https://www.anthropic.com/claude-code>
- [3] Anthropic. 2025. Introducing Claude 4. <https://www.anthropic.com/news/claude-4>
- [4] Edward Beeching, Lewis Tunstall, and Sasha Rush. 2025. Scaling test-time compute with open models. <https://huggingface.co/spaces/HuggingFaceH4/blogpost-scaling-test-time-compute>
- [5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134 [cs] [doi:10.48550/arXiv.2403.17134](https://doi.org/10.48550/arXiv.2403.17134)

- [6] Nadezhda Chirkova, Thibault Formal, Vassilina Nikoulina, and Stéphane Clinchant. 2025. Provenance: Efficient and Robust Context Pruning for Retrieval-Augmented Generation. *arXiv:2501.16214* [cs] doi:10.48550/arXiv.2501.16214
- [7] Cursor. 2025. Cursor – The AI Code Editor. <https://cursor.com/>
- [8] Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie Zhang, Pinjia He, and Ahmed E Hassan. 2025. SWE-Effi: Re-Evaluating SWE Agent Solutions for their Efficiency. <https://centre-for-software-excellence.github.io/SWE-Effi/about/introducing-SWE-efi>
- [9] Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M Zhang, Pinjia He, and Ahmed E Hassan. 2025. Swe-efi: Re-evaluating software ai agent system effectiveness under resource constraints. *arXiv preprint arXiv:2509.09853* (2025).
- [10] Mahdi Farzandway and Fatemeh Ghassemi. 2025. Automated Repair of C Programs Using Large Language Models. *arXiv preprint arXiv:2509.01947* (2025).
- [11] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling. *arXiv preprint arXiv:2507.23370* (2025).
- [12] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *arXiv:2302.05020* [cs]
- [13] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv:2310.06770* [cs]
- [14] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2025. Explainable Automated Debugging via Large Language Model-Driven Scientific Debugging. *Empirical Software Engineering* 30, 2 (March 2025), 45. doi:10.1007/s10664-024-10594-x
- [15] Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. LLMs Get Lost In Multi-Turn Conversation. *arXiv:2505.06120* [cs] doi:10.48550/arXiv.2505.06120
- [16] Jia Li, Xuyuan Guo, Lei Li, Kechi Zhang, Ge Li, Jia Li, Zhengwei Tao, Fang Liu, Chongyang Tao, Yuqi Zhu, and Zhi Jin. 2025. LONGCODEU: Benchmarking Long-Context Language Models on Long Code Understanding. *arXiv:2503.04359* [cs] doi:10.48550/arXiv.2503.04359
- [17] Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. 2023. Compressing context to enhance inference efficiency of large language models. *arXiv preprint arXiv:2310.06201* (2023).
- [18] Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. 2025. The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management. *arXiv:2508.21433* [cs] doi:10.48550/arXiv.2508.21433
- [19] Barys Liskavets, Maxim Ushakov, Shuvendu Roy, Mark Klibanov, Ali Etemad, and Shane K Luke. 2025. Prompt compression with context-aware sentence encoding for fast and improved llm inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 24595–24604.
- [20] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *arXiv:2409.02977*
- [21] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).
- [22] Maxime Louis, Hervé Déjean, and Stéphane Clinchant. 2025. Pisco: Pretty simple compression for retrieval-augmented generation. *arXiv preprint arXiv:2501.16075* (2025).
- [23] Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M Ponti. 2024. Dynamic memory compression: Retrofitting llms for accelerated inference. *arXiv preprint arXiv:2403.09636* (2024).
- [24] OpenRouter. 2025. Recent activity on Claude Sonnet 4. <https://openrouter.ai/anthropic/claude-sonnet-4/activity>
- [25] Stack Overflow. 2025. AI | 2025 Stack Overflow Developer Survey. <https://survey.stackoverflow.co/2025/ai>
- [26] Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, and David Lo. 2025. The Hidden Cost of Readability: How Code Formatting Silently Consumes Your LLM Budget. (2025).
- [27] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. 2024. LLMingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression. In *Findings of the Association for Computational Linguistics ACL 2024*. Association for Computational Linguistics, Bangkok, Thailand and virtual meeting, 963–981. doi:10.18653/v1/2024.findings-acl.57
- [28] Justyna Petke, Matias Martinez, Maria Kechagia, Aldeida Aleti, and Federica Sarro. 2024. The patch overfitting problem in automated program repair: Practical magnitude and a baseline for realistic benchmarking. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 452–456.
- [29] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362* (2024).

- [30] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-guided program reduction for understanding neural code intelligence models. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 70–79.
- [31] Mootez Saad, José Antonio Hernández López, Boqi Chen, Dániel Varró, and Tushar Sharma. 2025. An Adaptive Language-Agnostic Pruning Method for Greener Language Models for Code. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1183–1204.
- [32] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 945–955.
- [33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [34] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. arXiv:2402.01030 [cs]
- [35] 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, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs]
- [36] Yan Wang, Xiaoning Li, Tien N Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. 2024. Natural is the best: Model-agnostic code simplification for pre-trained large language models. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 586–608.
- [37] Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023. Learning to filter context for retrieval-augmented generation. *arXiv preprint arXiv:2311.08377* (2023).
- [38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [39] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489 [cs]
- [40] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.
- [41] Fangyuan Xu, Weijia Shi, and Eunsol Choi. 2024. RECOMP: Improving retrieval-augmented LMs with context compression and selective augmentation. In *The Twelfth International Conference on Learning Representations*.
- [42] Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Yue Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. 2024. Less is more: Docstring compression in code generation. *arXiv preprint arXiv:2410.22793* (2024).
- [43] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs]
- [44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- [45] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605* (2025).
- [46] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs]
- [47] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1073–1084.
- [48] Shangzhi Zhao, Jiahao Yuan, Guisong Yang, and Usman Naseem. 2025. Can pruning improve reasoning? revisiting long-cot compression with capability in mind for better reasoning. *arXiv preprint arXiv:2505.14582* (2025).
- [49] Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. 2025. MEM1: Learning to Synergize Memory and Reasoning for Efficient Long-Horizon Agents. *arXiv preprint arXiv:2506.15841* (2025).

Received 2025-09-12; accepted 2025-12-22