Improving the Efficiency of LLM Agent Systems through 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 been increasingly popular for software engineering tasks. While LLM agents show decent effectiveness, the high computational cost of input tokens due to the ever-growing trajectory remains an efficiency concern for their applications. Efficiency is largely neglected in existing studies and agent products, and this paper fills the gap by introducing an inference-time trajectory reduction approach to reduce the cost of agents.

Through analyzing existing agent trajectories, we demonstrate that useless, redundant, and expired information is widespread in all trajectories, which can be identified and reduced without harming the agent's performance. We then design a simple yet effective trajectory reduction approach, AgentDiet, which automatically removes such waste information. We implement AgentDiet on a top-performing coding agent, and the evaluation on two LLMs and two benchmarks shows that AgentDiet can reduce input tokens by 39.9% ~ 59.7%, or the final computational cost by 21.1% ~ 35.9%, while maintaining the same agent performance. This indicates that trajectory reduction is a promising direction for agent systems.

ACM Reference Format:
Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. 2018. Improving the Efficiency of LLM Agent Systems through Trajectory Reduction. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large Language Models (LLMs) have been widely used for various software engineering tasks such as code generation, testing, and repair [1, 9, 11, 19, 39, 40]. The most recent trend in the application of LLMs is agent-based approaches [5, 13, 28, 35, 43], where LLMs are asked to complete difficult 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 [12] using Claude 4 Sonnet [3], indicating that recent agents can fix real-world GitHub issues with basic tools such as file editing and bash scripting. Given their satisfactory performance, LLM agents are integrated in various AI products [2, 7], and 24% professional developers already use LLM agents daily or weekly [24].

Although these agent approaches demonstrate decent performance in terms of effectiveness, efficiency remains a concern for their application. In a recent survey by StackOverflow [24], 53% of

Authors' Contact Information: Yuan-An Xiao, xiaoyuanan@pku.edu.cn, Peking University, China; Pengfei Gao, gaopengfei.se@bytedance.com, ByteDance, China; Chao Peng, pengchao.x@bytedance.com, ByteDance, China; Yingfei Xiong, xiongyf@pku.edu.cn, Peking University, China.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

the participants believe that the cost of using AI agents is a barrier for them. The efficiency issue is largely neglected in existing studies, leaving a large gap to address.

The root cause of the efficiency issue comes from the workflow of LLM agents, i.e., 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 finishes the whole task. Therefore, if the LLM opens a large file or runs a command that generates verbose output, the computational cost of these trajectory tokens snowballs for each of the following steps [8], resulting in a large waste. In fact, daily usage of the Claude 4 Sonnet reaches 100B tokens on the OpenRouter platform [23], and 99% of them are input tokens accumulated in the trajectory, while only 1% tokens are generated by the LLM, showing great potential for improvement.

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, 21, 26, 37, 46], which are different with coding agents in multiple aspects: (1) these approaches reduce all input tokens at once, while trajectory tokens gradually build up in the agents, and the timing of reduction is 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 process of LLM inference, while the best performing LLMs on code are proprietary LLMs that do not provide such capabilities. To the best of our knowledge, there is no existing publication on the inference-time trajectory reduction for coding agents. In the industry, some agent products [2, 7] sparingly apply LLM-based compression only when the context window is full, focusing on robustness rather than efficiency. As a result, the potential for efficiency improvement by 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 possibility of cost reduction, revealing that the trajectories of LLM agents contain a large amount of waste, including useless, redundant, or expired information. We then propose the design of a prototype trajectory reduction approach. In Section 3, we instantiate the design into a concrete algorithm, AgentDiet. Section 4 discusses the integration of AgentDiet in 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 the input tokens by 39.9% $\sim$ 59.7%, or final computation cost by 21.1% $\sim$ 35.9%, while maintaining the same agent performance.

The contributions of this paper are summarized as follows:

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

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 ingredients in their trajectories. We then motivate our approach in Section 2.2 and Section 2.3 by gradually analyzing the efficiency problem in the trajectory. In particular, we will answer a series of questions:

- Is there waste in the trajectories? *(Yes, there is a lot.)*

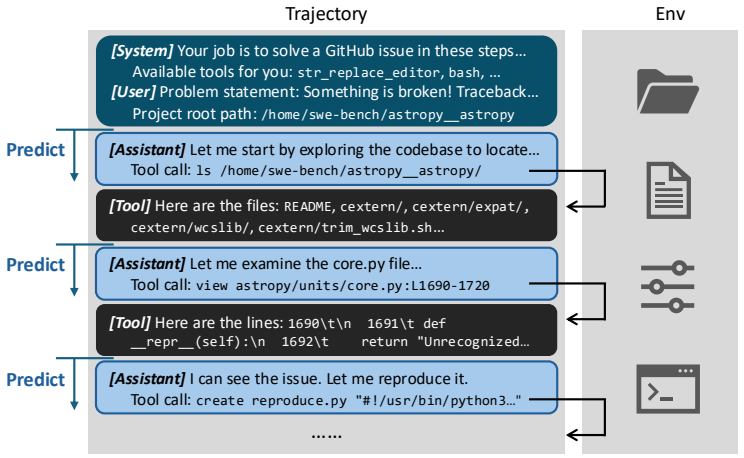

Fig. 1. Typical workflow of an LLM agent.

- Can we automatically identify and reduce the waste in trajectories? *(Yes, it is possible to use LLM for trajectory reduction.)*
- Do LLMs know when to perform trajectory reduction on their own? *(No, so we enforce the trajectory reduction step with a separate reflection module.)*
- Is it ideal and efficient to call an LLM for that purpose? *(Not really, but the overhead is under control with optimizations.)*

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 shown the ability to assist with various software engineering tasks. Following established approaches like Chain-of-Thought [38], ReAct [44], and CodeAct [34], the LLM agent has been 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 the LLM can operate on the codebase with the ability to read files, make changes, execute commands, 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). Then, the LLM 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, performs 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 of the subsequent LLM predictions. The loop continues until the LLM decides to finish the task with a special tool call, or when it reaches the step limit.

From this workflow, we can identify an obvious efficiency issue in such agent systems: Once the agent 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 will be included in each of the subsequent LLM predictions, even if the content is not (or no longer) relevant to the task,

causing an inefficient use of computational power. As a quantitative result, the average trajectory to solve a single GitHub issue, as we collected from the SWE-bench Verified [12] 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 (where 11.9K among them 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 resources.

Recent LLMs are equipped with the KV Cache mechanism, which mitigates the issue of high computational cost by caching repeated calculation 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 caches only part of the computations, but the remaining calculations are still costly as the trajectory grows; (2) KV Cache takes hardware resources like VRAM and I/O bandwidth, so reducing the length of the trajectory saves such valuable resources. Actually, the existence of KV Cache makes the design of trajectory reduction approaches a more challenging research question, since modifying a token in the trajectory invalidates the cache for all following tokens.

In addition to harming efficiency, long messages in the context window also cause the performance of LLMs to degrade [15, 20]. Therefore, reducing the length of the trajectories does not necessarily lead to a drop in the agent's performance. Instead, it has the potential to slightly improve the performance by removing waste from the trajectories.

2.2 Waste in Trajectories

In this subsection, we begin to analyze the possibility of trajectory reduction by first identifying typical waste in trajectories. For this purpose, we refer to SWE-bench Verified [12], a popular benchmark that requires agent systems to solve GitHub issues in popular Python repositories. The benchmark provides the logs and trajectories of all participants for public download. We downloaded and manually inspected the trajectories of Trae Agent [10], 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 the waste is widespread in almost all trajectories. We then categorized them into three typical scenarios in which tokens could be removed or compressed from a human perspective:

2.2.1 Useless information. Some information is not relevant 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). Also in all trajectories, the agent executes 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 the output to the first 16KB, a fixed threshold only avoids the extreme case, and we still frequently observe 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 is the `str_replace_editor` tool, which is a standard file editing tool designed by Anthropic and 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, typically containing multiple lines, which 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 repeats with the

Fig. 2. A case study of reducing information waste in the trajectory by GPT-5 mini and LLMLingua-2.

replaced result in the response; (2) Q may contain same ingredients (e.g., shared statements) in P; (3) the agent should have retrieved the fragment to edit prior to this tool call, so P repeats with some code in previous steps in the trajectory.

2.2.3 Expired information. Information relevant to a step may no longer be necessary after the step is completed. The most typical case is where the agent loops through many subjects 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 occurred in the search result. This process takes a few steps in the trajectory, and after the agent identifies the faulty file, most of the content in other files is no longer useful.

2.3 A Prototype Approach to Trajectory Reduction

2.3.1 Identifying the waste. In this subsection, we discuss how to automatically identify and reduce the amount of waste in trajectories. A straightforward idea will be to maintain a set of rules (such as regular expressions) for each typical kind of waste, but such rules are not likely to cover a variety of scenarios. For example, it is difficult to write a rule to identify all verbose output in test cases, since every project has its own output format.

Given the heuristic nature of this task, we decided to identify and reduce waste with the help of an LLM. As a case study, we picked a typical example of the "useless information" category,

where the test command shows the list of each test passed in a verbose way, as shown in Fig. 2. We designed a prompt based on LLMLingua-2 [26], a recent work on prompt compression, instructing GPT-5 mini to replace useless, redundant, or expired information with a placeholder of "... (a short description)". The full prompt is available in the artifact, and we omit it here due to space constraints. On the right side of Fig. 2, we can see that GPT-5 mini successfully removes these lines with a description ("individual test lines omitted; mostly PASSED"), while preserving the name of the failed test at the end, which is important for the task.

We also show the compression result of LLMLingua-2 in this case, at the bottom of Fig. 2. Its result is rather uninformative because the names of the test cases are corrupted in the middle. The main reason for the huge difference between GPT-5 mini and LLMLingua-2 is that LLMLingua-2 is a small model (based on `xlm-roberta-large` [30]) distilled from synthesized data on natural language. While such a model is trained to remove redundancy in the natural language at a token granularity, it may not have the reasoning ability at a larger granularity (e.g., the full list of passed tests is useless for the task). Therefore, LLMs such as GPT-5 mini seem like a promising choice for the trajectory reduction task.

2.3.2 The timing of trajectory reduction. After showing that trajectory reduction is possible with an LLM, the next natural topic is the timing of trajectory reduction.

Ideally, if the agent system is powerful enough, it can reduce its own trajectory given a tool for that purpose. We then experiment with this hypothesis by implementing a tool called `erase`, where the agent can override the existing steps in the trajectory. We assign a numeric ID to each segment in the trajectory, and the LLM can 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 observation, even with the most powerful models such as Claude 4 Sonnet, the agent often persists on the original task without calling `erase` under explicit prompts. Below is an example collected on the `django__django-13012` instance in SWE-bench Verified, where the agent continues on the task even if asked explicitly for trajectory reduction:

A failed 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)*

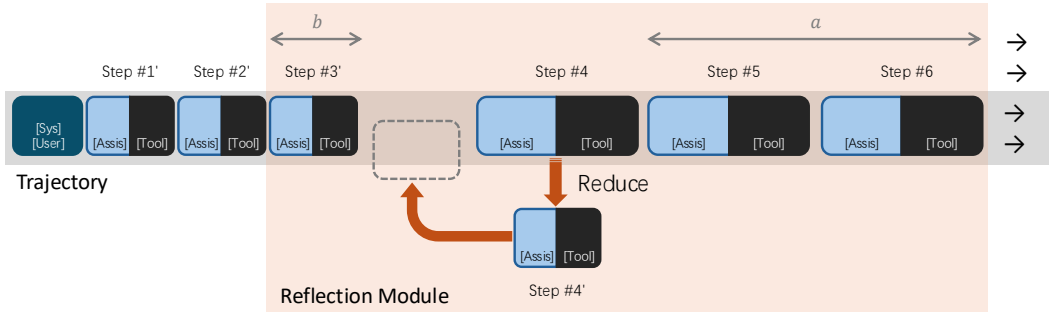

Fig. 3. Design of the reflection module in AgentDiet.

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

To overcome this problem, we take a step back and move the trajectory reduction step to a separate module called *reflection module*. This LLM-based module will be explicitly called by the outer system to reduce the trajectory when there is an estimated benefit. In this way, the timing of the trajectory reduction is controlled by an external system, and the LLM that runs the agent is unaware of the trajectory reduction, reducing the 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 because it has an additional cost overhead. If we pass the full trajectory to the reflection module for each step, the token usage will be doubled, defeating the purpose of cost reduction. Therefore, controlling the overhead becomes our next topic.

The first approach to consider is the choice of LLM in the reflection module. Since trajectory reduction is arguably easier than the original task of the agent, it is possible to significantly reduce the overhead by choosing a more cost-efficient LLM for the reflection module. In the previous case study illustrated in Fig. 2, the trajectory is generated by the agent with Claude 4 Sonnet, and the reduction is performed by GPT-5 mini, which is 12 times cheaper than Claude 4 Sonnet, minimizing overhead concerns. We will discuss more about the choice of LLM in Section 4.2.

To further reduce the overhead, we can reduce the amount of context for the LLM used in the reflection module. We propose a sliding-window approach, as shown in Fig. 3. The yellow box in the figure refers to the reflection module: When the agent reaches step s, the reflection module is now only allowed to reduce the content in step $s-a$, given only a fixed amount of context (from step $s-a-b$ to step s). The module asks a cost-efficient LLM to reduce the content in Step $s-a$ by removing useless, redundant, and expired information given the 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 as the trajectory grows. Also, since it may modify only a fixed recent step (Step $s-a$), entries in the KV Cache corresponding to previous steps will be preserved, minimizing the overhead introduced by invalidated KV Cache. From the perspective of agent performance, this

design of the reflection module has the benefit of having no chance to destructively erase the most recent step or all steps at once, which prevents disastrous outcomes in occasional LLM failures.

3 Algorithm

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) / after (a) the target step in context, Step limit ($s_{\max}$), Length threshold (θ), LLM for the agent ($\text{LLM}_{\text{agent}}$) / reflection ($\text{LLM}_{\text{reflect}}$) steps

1: $T \leftarrow \textsc{MakeInitialPrompt}(I)$ ▷ Initiate the trajectory
2: **for each** $s \in [1 \dots s_{\max}]$ **do**
3: $m_{\text{assis}} \leftarrow \text{LLM}_{\text{agent}}(T)$ ▷ Perform agent step
4: **if** $\textsc{IsTaskDone}(m_{\text{assis}})$ **then**
5: $r \leftarrow$ `"finished"`
6: **return** r, E
7: **end if**
8: $E, m_{\text{tool}} \leftarrow \textsc{ExecTool}(E, m_{\text{assis}})$ ▷ Parse and execute the tool call in the environment
9: $T \leftarrow T + [\langle m_{\text{assis}}, m_{\text{tool}} \rangle]$ ▷ Concatenate messages in this step into the trajectory

10: **if** $s - a > 0$ **then**
11: $l_{\text{orig}} \leftarrow \textsc{Length}(\textsc{Serialize}([T[s-a]]))$
12: **if** $l_{\text{orig}} > \theta$ **then**
13: $ctx \leftarrow \textsc{Serialize}(T[\max(0, s-a-b) : s])$
14: $m_{\text{reduced}} \leftarrow \text{LLM}_{\text{reflect}}(ctx, s-a)$ ▷ Perform reflection step
15: $l_{\text{reduced}} \leftarrow \textsc{Length}(m_{\text{reduced}})$
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: $r \leftarrow$ `"interrupted"` ▷ Reaching the step limit
23: **return** r, E

In this section, we instantiate the design of AgentDiet introduced in Section 2.3 and Fig. 3 into a concrete algorithm as shown in Algorithm 1. The orange box in Algorithm 1 shows how to integrate AgentDiet in a typical LLM agent.

As a typical LLM agent, Algorithm 1 takes two arguments as input: the problem instruction I, which contains a natural language description of the task to complete, and the environment E, which the agent can interact with through tool calls. The output is the result r that indicates whether the task is completed. The environment E will be 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 initiates 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 m_{assis} that contains the next action to take (in a tool call) based on T. If the action is to finish the task, the agent stops; otherwise, the ExecTool 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.

Lines 10-20 highlighted with the orange box represent the new reflection module introduced in AgentDiet. As illustrated in Fig. 3, the reflection module follows a sliding window that aims to reduce the length of step $s - a$ when the agent reaches step s. The reflection module uses another language model, $\text{LLM}_{\text{reflect}}$, which can be more cost-efficient than $\text{LLM}_{\text{agent}}$ to reduce the overhead of trajectory reduction. This module is controlled by the hyperparameter θ, which is the minimum number of tokens of a beneficial reduction, and a, b, which are the number of steps in the context given to $\text{LLM}_{\text{reflect}}$. 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 too short ($l_{\text{orig}} \leq \theta$), reflection is skipped for this step because the benefit may not exceed the overhead. Otherwise, it calculates the context (ctx) and asks $\text{LLM}_{\text{reflect}}$ 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 will be replaced by the reduced version at line 17.

4 Implementation

4.1 Integration in Agents

AgentDiet is a general approach that can be applied simply to any LLM agent under the "models using tools in a loop" definition. For a typical agent system similar to Algorithm 1, we can add a call to the reflection module (marked in the orange box) after each step in the loop.

Following this procedure, we integrated AgentDiet in Trae Agent, which is open-source and ranked first on SWE-bench Verified when this research was performed (in July 2025). We did not further integrate and experiment AgentDiet in other LLM agents based on the observation that current agent systems are generally homogeneous, including 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 reason about the issue, and `task_done` to finish the task. This set of tools is semantically equivalent to tools in other systems, such as mini-SWE-agent [43] and OpenHands [35]. Therefore, the results with Trae Agent have a high chance of generalizing 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 a final answer by majority voting, and *multi-agent* systems that simultaneously spawn multiple LLM agents with a communication mechanism. We can integrate AgentDiet in such systems by adding the reflection module in the LLM agent in an ensembled system, or in all LLM agents in a multi-agent system. As early research in the field, this paper does not focus on these applications. We leave the evaluation of such an application for future work.

4.2 Hyperparameter Settings

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

To set an ideal value for each of these hyperparameters, we first started with an initial setting ($\text{LLM}_{\text{reflect}}$ = Gemini 2.5 Flash, $\theta = 500$, $a = 3$, and $b = 1$) and quantitatively compared the efficiency and agent performance with an experiment against variants, each with a different setting of one hyperparameter. Two variants (one with $\text{LLM}_{\text{reflect}}$ = GPT-5 mini, the other with $a = 2$) showed a

better overall result than the initial one, so we iterated the hyperparameter setting to $\text{LLM}_{\text{reflect}}$ = GPT-5 mini, $\theta = 500$, $a = 2$, and $b = 1$. We reran the experiment, and no variant showed a better overall result in this iteration. As a result, this hyperparameter setting is chosen as the ideal one.

Below, we discuss the setup of this experiment and show the results in the second iteration.

4.2.1 Benchmark. For this experiment, we randomly choose 100 instances from SWE-bench Verified [12] as the benchmark. We do not use the full benchmark because the remaining instances will be used exclusively for further evaluation in Section 5.

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	0.80	4.0
	Gemini 2.5 Flash	0.075	0.30	2.5
	GPT-5 mini	0.030	0.25	2.0
	DeepSeek v3 (0324)	0.070	0.27	1.1
	Qwen 3 (2507)	0.070	0.27	1.1
Baselines	Original	N/A		
	LLMLingua-2	0.01		
	Random	N/A		

4.2.2 Variants for $\text{LLM}_{\text{reflect}}$. This experiment considers five variants, covering a variety of recent cost-efficient LLMs from various vendors: Claude 3.5 Haiku, Gemini 2.5 Flash, GPT-5 mini, DeepSeek v3 (0324), and Qwen 3 (2507). The pricing information for these LLMs is listed in Table 1, which will be used to calculate their computational cost.

As a sanity check, we also added the following three additional baseline variants:

- **Original**: The unmodified agent system that skips the reflection step (lines 10-20).
- **LLMLingua-2** [26]: A small model for prompt compression. Since this model runs on our own machine, we estimate its pricing (as US$0.01 / M tokens) based on a similar BERT model with 600M parameters.
- **Random**: A baseline that randomly deletes 75% processed tokens.
- **Delete**: A baseline that deletes all processed tokens.

Note that $\text{LLM}_{\text{reflect}}$ processes a step only when its length is greater than θ tokens (line 12 in Algorithm 1). As a result, all baseline variants (even the Delete baseline) will keep all tokens in a short step as-is.

4.2.3 Variants for θ, a, and b. The variants considered for these hyperparameters are shown in the list below:

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

Note that we limit the choice of a and b to an upper bound ($a \leq 3, b \leq 2$) because a higher value would strictly cause more cost overhead with the opportunity of better agent performance. Since the variants of $a = 2$ and $b = 1$ already have a decent performance, as will be shown in the results, we estimate that there would be little benefit in further increasing a or b.

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

- **Keep%**: The average ratio of trajectory tokens kept by the LLM in the reflection module, formally $\sum l_{\text{reduced}} / \sum l_{\text{orig}}$ in Algorithm 1.
- **I** and **O**: Normalized accumulated usage of input ("prompt") / output ("completion") tokens. We normalize all numbers in these two rows so that the Original baseline has $I = 1$.
- **$** and **$+**: Normalized 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 normalize all numbers in these two rows so that the Original baseline has $ = 1. We separately report the cost for agent steps and reflection steps to help readers better understand the overhead caused by the reflection module.

Additionally, we use the following metrics to measure the impact on the agent's performance:

- **Pass%**: The ratio 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 incorrectly reduced, disturbing the agent and requiring it to recover the information with extra steps.

Table 2. Results of different LLMs or baselines as $\text{LLM}_{\text{reflect}}$.

	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

4.2.5 Results for $\text{LLM}_{\text{reflect}}$. The LLM to use for the reflection step is an important hyperparameter in AgentDiet, since it should be both cost-efficient and capable enough to identify the waste.

The results are illustrated in Table 2. 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% tokens in the processed steps. As a result, the accumulated input tokens (I) drop to 47.3% ∼ 72.2%, where 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 $\text{LLM}_{\text{reflect}}$ introduces an overhead cost of 5.2% ∼ 14.8%, depending on the model used.

The last three rows measure the performance impact of the trajectory reduction on the SWE-Bench Verified instances. We can see that while the numbers in Pass% vary between LLMs and baselines, all of them reach a Pass% of at least 58% (for the Delete baseline), which is close to the 65% achieved by the Original agent. This is due to the default choice of other hyperparameters (especially $a = 2$ and $\theta = 500$), which ensures that $\text{LLM}_{\text{reflect}}$ can only manipulate steps of excessive length after a delay of 2 steps. Therefore, the disruption to $\text{LLM}_{\text{agent}}$ is limited, and LLM can recover

the disrupted information by performing on its own through tool calls, leading to an increase in "Step" and "PStep". We can see that the increase is most severe for the Delete baseline that maximized the disturbance to the trajectory.

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, showing that it does not harm the agent's performance while saving significant token costs. Therefore, GPT-5 mini is chosen as the final $\text{LLM}_{\text{reflect}}$ in AgentDiet.

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

		θ				
	Orig.	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

4.2.6 Results for θ, a, and b. The results for the remaining hyperparameters are illustrated in Table 3. For θ, a higher threshold value results in more input tokens for the agent (I), but fewer tokens for reflection overhead. $\theta = 500$ is a balance between token saving and reflection overhead. In terms of agent performance, $\theta = 500$ also shows optimal performance close to 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 worse efficiency. $a = 2, b = 1$ is the minimum choice of a and b that has a negligible performance harm, 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 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 integrated AgentDiet in Trae Agent [10], as discussed in Section 4. To learn more about the generalization of our approach (RQ3), we evaluated the Trae Agent integration with two recent LLMs (i.e., $\text{LLM}_{\text{agent}}$) showing decent agent capability: Claude 4 Sonnet and Gemini 2.5 Pro. For each LLM, we compare the results of AgentDiet over the Original baseline, where the reflection step is skipped.

5.1.2 Metrics. We used the same set of metrics as previously described in Section 4.2.4, which contains five metrics for efficiency (Keep%, I, O, $, and $+), and three metrics for performance impact (Pass%, Step, and PStep). In RQ1, we can infer that AgentDiet improves efficiency if Keep% is low and I/O/$ decreases over Original; In 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 used two benchmarks for the evaluation:

- **SWE-bench Verified** [12]: This benchmark contains 500 human-verified software engineering tasks in Python based on GitHub issues in the real world. Since Section 4.2 used 100 instances in this benchmark for hyperparameter selection, we excluded them and randomly selected 200 instances from the remaining 400 instances for the experiment in this section. Similar to the validation/test split in standard machine learning practice, this experimental setup addresses the overfitting threat of hyperparameters. The list of selected instances is available in the artifact.
- **Multi-SWE-bench Flash** [45]: Similar to SWE-bench Verified, this benchmark contains 300 instances based on GitHub issues. However, this benchmark covers tasks in seven other programming languages, containing 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 gives the agent a fully working environment in the beginning.

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 $\text{LLM}_{\text{reflect}}$ = 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 Step/PStep metrics in the experiment results.

5.2 Results

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

5.2.1 RQ1. Efficiency Improvement. We can understand how the trajectory reduction by AgentDiet ultimately improves efficiency through various metrics reported in Table 4. First, the reflection module removes 69.2% ∼ 77.4% content it processed (1 - Keep%). This results in a reduction in the accumulated input tokens (1 - I) at 39.9% ∼ 59.7%. This reduction is less than the percentage of deleted processed tokens because the reflection module only processes steps longer than the token threshold ($\theta = 500$) and only applies the reduction after several ($a = 2$) steps. The reduced input tokens then lead to a reduced computational cost of the agent ($) at 28.6% ∼ 44.1%. This reduction is again smaller due to costs related to output tokens and invalidated KV Caches, but still significant

Table 4. Efficiency improvement and performance impact of AgentDiet.

Benchmark	SWE-bench Verified				Multi-SWE-bench Flash			
LLM_{agent}	Claude 4 Sonnet		Gemini 2.5 Pro		Claude 4 Sonnet		Gemini 2.5 Pro	
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

enough. After we further consider the computational overhead of the reflection module itself ($+), the final cost reduction becomes 21.1% ∼ 35.9%.

Note that the numbers in "$" and "$+" rows are normalized to make the relative comparison with Original easier. If we convert the numbers back to the format of average US$ per instance, the Original baseline costs $0.535, $0.385, $1.277, and $0.701, respectively, for the SbV+Claude, SbV+Gemini, MSbF+Claude, and MSbF+Gemini columns. AgentDiet decreased 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 within one or a few functions), the total savings among active users of a popular AI product can be a large number.

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.

5.2.2 RQ2. Performance Impact. From the "Pass%" row in Table 4, we can compare the numbers between Original and AgentDiet to learn its impact to the agent's performance. On two benchmarks and two LLMs, the performance of AgentDiet is comparable (-1.0% ∼ +2.0%) to Original. The result shows 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 LLM degrades as the length of the context increases [15] or when the context is of low quality [14]. Therefore, the removal of waste information from the agent trajectory will cause less degradation to the model's performance.

Additionally, from the "Step" and "PStep" rows, we can learn 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. An interesting case occurs in the last column, where AgentDiet significantly reduced the average steps from 57.20 to 43.90 when Gemini 2.5 Pro is working on the Multi-SWE-bench Flash benchmark. We found the reason through inspecting the trajectories: Gemini 2.5 Pro shows increasingly abnormal behavior when the context is long, often resulting in repeating the same tool call until the step limit is reached. Therefore, AgentDiet reduces this probability by reducing the length of the trajectory by half. Fig. 4 draws the histogram of the step

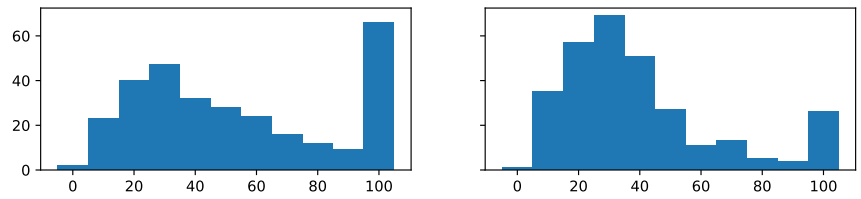

Fig. 4. Distribution of the Step metric on Multi-SWE-bench Flash with Gemini 2.5 Pro between Original (left) and AgentDiet (right).

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 (at 100 steps) from 66 to 26. As a result, the average steps are significantly reduced.

Finding 2. The ratio of successfully resolved instances for AgentDiet is comparable (-1.0% ∼ +2.0%) to 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 steadily 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).

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

	Lang	Rust	TypeScript	JavaScript	Java	Go	C	C++
	#Inst.	45	45	45	40	45	40	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
	ΔPass	-2	+1	+1	+1	-3	-1	0
	ΔStep	-0.27	-1.11	+3.11	-1.32	+0.13	+0.40	-0.53
	ΔPStep	-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
	ΔPass	0	+3	+4	-1	+1	-3	-1
	ΔStep	-13.91	-11.36	-11.44	-15.88	-3.02	-10.15	-27.52
	ΔPStep	-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 further show the generalization across programming languages. The results are shown in Table 5, where "ΔPass", "ΔStep", and "ΔPStep" 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 results 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. The overall latency increase should be slight, since we use a cost-efficient model as $\text{LLM}_{\text{reflect}}$, and the latency is reduced for $\text{LLM}_{\text{agent}}$ due to the reduction in trajectory length. The evaluation does not quantitatively compare the latency between AgentDiet and Original, because the experiment calls LLMs through commercial APIs, and the latency is highly unstable due to server load. In latency-critical scenarios, it is possible to modify Algorithm 1 so that the reflection step and the agent step are performed in parallel, with the trade-off of reducing the context after the target step (a) by one step.

6.1.2 Exploring more possible designs of trajectory reduction. This paper is an early study on the efficiency aspect of LLM agents, and presents AgentDiet as a simple yet effective approach to trajectory reduction. Furthermore, this paper only considers off-the-shelf LLMs as $\text{LLM}_{\text{reflect}}$, with an overhead of 5% ∼ 10% in terms of cost. Future work may explore more possible designs of a trajectory reduction approach, and may fine-tune a more efficient LLM to reduce the cost overhead even more.

6.2 Threats to Validity

6.2.1 Generalization across agents. The main threat to external validity comes from the generalization concern. The current experiment setup considers two benchmarks and two LLMs to mitigate this threat. However, due to effort and cost constraints, AgentDiet is only implemented for the Trae Agent but not other agent systems. In fact, existing experiments already lead to a total LLM cost of approximately US$ 2000, so experimenting with even more agent systems would be challenging. Section 4.1 argues that current agent systems are generally homogeneous, including a similar set of prompts and tools. Therefore, this threat should have a limited chance of affecting the findings.

6.2.2 Data leakage in LLMs. A possible threat to internal validity comes from data leakage in LLMs. Since the experiment uses proprietary LLMs, it is possible that some information about the experiment subjects is used for their training, and we cannot control the process. There are existing studies on this threat, and the readers could refer to them. 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" [12]. Therefore, we believe that the threat should not affect the validity of the findings. To further mitigate this threat, we also added Multi-SWE-bench Flash [45] to the benchmark, which

was collected by a third party, published recently, and not mentioned in the tech reports of any LLM used in the evaluation.

6.2.3 The correctness of patches. Both SWE-bench Verified and Multi-SWE-bench Flash validate the correctness of the patches generated by the agent through the test cases written by the developer. An instance is considered successfully resolved if all test cases pass. A patch passing all test cases ("plausible") may not be semantically equivalent to the ground truth patch written by the developer ("correct"). As a result, overfitting the test cases becomes a possible threat to internal validity for all test-based approaches, including this paper. However, an existing study [27] revealed that the threat of overfitting is not severe in the field of program repair. Additionally, since each instance in SWE-bench Verified and Multi-SWE-bench Flash has at least one additional test case that is not given to the agent, the possibility of overfitting is limited. Furthermore, our experiment compares the ratio of resolved instances between AgentDiet and Original, which are exposed to the same overfit threat. Therefore, the findings should not be affected by this threat.

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 that will be used for reference by the LLM. FilCo [37] filters the retrieved documents to only useful parts based on information-theoretic approaches. Selective Context [16] calculates the self-information of each token from the output probability and removes tokens with less self-information. RECOMP [41] introduces an abstractive compressor that paraphrases the compressed document into a concise abstract. Provence [6] and LLMLingua-2 [26] train a classifier model to judge the relevance of each token in the document with ground truth generated by a teacher LLM. CPC [18] 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 name of failed tests (as shown in Fig. 2), leading to suboptimal performance compared to the reflection module in AgentDiet (as shown in Table 2).

There are also approaches to prompt reduction specifically designed for code input. Suneja et al. [32] and Rabin et al. [29] minimize the input code while preserving the output based on program simplification techniques. Zhang et al. [46] and Wang et al. [36] prune tokens in the code based on heuristic rules. Yang et al. [42] compress the docstring in the source code. Pan et al. [25] reformat the code to remove tokens related to whitespaces and indentations.

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 applied to agent trajectories without discussing the timing of prompt reduction.

Lindenbauer et al. [17] proposed an approach to trajectory reduction in parallel with us, where their paper was just posted on arXiv in early September. Their approach can be considered as a variant of the Delete baseline (in Section 4.2.2) 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%. Also, the choice of $a = 10$ would cause too much cost overhead due to invalidated KV Caches.

7.2 Context Management Mechanism in Agent Systems

Various agent systems employ some kind of context management mechanism. For example, Cursor [7] and Claude Code [2] compress the context with LLM when the context window is full, Trae Agent [10] truncates each tool response to the first 16KB, and SWE-agent [43] has multiple configurable mechanisms such as removing texts identified by a regular expression. These mechanisms are often ad-hoc and aim to improve the robustness of the agent over corner cases, while this paper aims to improve the overall efficiency. Such mechanisms are also not reported or evaluated as a first-class research question [17], while this paper contains an in-depth discussion and large-scale evaluation of AgentDiet.

7.3 Efficiency Improvement for Whitebox LLM Systems

Some approaches improve the efficiency of LLM by fine-tuning the model or optimizing the inference process. DMC [22] combines adjacent KV Cache entries, which has the effect of merging multiple tokens into one token, thus saving computational cost. PISCO [21] distills RAG documents into embedding vectors with minimal cross-entropy to save tokens. Prune-on-Logic [47] prunes the Chain-of-Thought tokens during model training to improve the reasoning efficiency of the trained model. MEM1 [48] 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 training or inference process of LLM, they require the LLM to be a white box. Unfortunately, many state-of-the-art LLMs for agent systems are proprietary and do not allow such modification. In comparison, AgentDiet does not depend on the internal factors of LLM and can be applied to a wider range of agent systems.

8 Conclusion

This paper focuses on the efficiency concern of coding LLM agents, which is neglected by existing studies. Through an analysis of trajectories in SWE-bench Verified, we identify typical waste in the trajectories. We then design an inference-time trajectory reduction approach, AgentDiet, to reduce the token cost of agent systems with a reflection module. We instantiate and integrate AgentDiet in a top-performing coding agent, and discuss hyperparameter settings in the reflection module through quantitative experiments. Finally, we evaluate AgentDiet on both SWE-bench Verified and Multi-SWE-bench Flash, with both Claude 4 Sonnet and Gemini 2.5 Pro. The results show that AgentDiet steadily reduces the amount of input tokens by 39.9% ∼ 59.7%, leading to significant savings in the final computational cost by 21.1% ∼ 35.9%. Meanwhile, the performance of AgentDiet is on par with the original agent (at -1.0% ∼ +2.0%). This paper highlights the possibility of reducing the cost of LLM agents without harming performance through a simple approach, which benefits the 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 experiment results, and the raw trajectories collected in the experiments, are available at https://figshare.com/s/5a8cce18f2eccb5db761.

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. https://doi.org/10.48550/arXiv.2403.17134 arXiv:2403.17134 [cs]

[6] Nadezhda Chirkova, Thibault Formal, Vassilina Nikoulina, and Stéphane Clinchant. 2025. Provence: Efficient and Robust Context Pruning for Retrieval-Augmented Generation. https://doi.org/10.48550/arXiv.2501.16214 arXiv:2501.16214 [cs]

[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-effi

[9] Mahdi Farzandway and Fatemeh Ghassemi. 2025. Automated Repair of C Programs Using Large Language Models. *arXiv preprint arXiv:2509.01947* (2025).

[10] 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).

[11] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. arXiv:2302.05020 [cs]

[12] 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]

[13] 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. https://doi.org/10.1007/s10664-024-10594-x

[14] Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. LLMs Get Lost In Multi-Turn Conversation. https://doi.org/10.48550/arXiv.2505.06120 arXiv:2505.06120 [cs]

[15] 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. https://doi.org/10.48550/arXiv.2503.04359 arXiv:2503.04359 [cs]

[16] 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).

[17] 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. https://doi.org/10.48550/arXiv.2508.21433 arXiv:2508.21433 [cs]

[18] 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.

[19] 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

[20] 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).

[21] Maxime Louis, Hervé Déjean, and Stéphane Clinchant. 2025. Pisco: Pretty simple compression for retrieval-augmented generation. *arXiv preprint arXiv:2501.16075* (2025).

[22] 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).

[23] OpenRouter. 2025. Recent activity on Claude Sonnet 4. https://openrouter.ai/anthropic/claude-sonnet-4/activity

[24] Stack Overflow. 2025. AI | 2025 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2025/ai

[25] Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, and David Lo. 2025. The Hidden Cost of Readability: How Code Formatting Silently Consumes Your LLM Budget. (2025).

[26] 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. LLMLingua-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. https://doi.org/10.18653/v1/2024.findings-acl.57

[27] 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.

[28] 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).

[29] 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.

[30] Sebastian Ruder, Anders Søgaard, and Ivan Vulić. 2019. Unsupervised cross-lingual representation learning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*. 31–38.

[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] 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.

[47] Shangziqi 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).

[48] 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).