

Appendix

Here, we provide detail about the following topics:

1. Code Link
2. Choice of Baselines
3. Limitations of GenCache
4. Prompts for the LLMs used
5. Data for FLASH
6. Structural Modification to the Data
7. LLM token usage pattern over time

Code Link

Code link for GenCache is available at <https://github.com/sarthak-chakraborty/GenCache>

Choice of Baselines

GenCache stores the input prompt along with the generated program in its cache store. The program can then be executed locally via a runtime like Python interpreter to generate the correct response for the input prompt. Thus, the obvious candidate for a baseline is the exact prompt matching (ExactCache), which returns the same response verbatim when the input prompt is identical.

Semantic caching is a widely used technique for LLMs where input prompts are matched to stored prompts based on embedding similarity, and a cache hit returns the exact response associated with the matched prompt. There are multiple semantic caching techniques in the literature. We chose GPTCache [14] as the representative approach due to its wide popularity and well-maintained library (over 7500 stars on Github [5]). Other approaches, such as Mean Cache and the method proposed by Gill et al. [20], improve semantic caching by customizing the embedding model via fine-tuning on domain- and user-specific data. However, these methods are not open-sourced and, like GPTCache, suffer from a key limitation that in datasets without prompt repetition, any cache hit results in a negative hit. InstCache [64] is a recent predictive caching technique that predicts tokens likely to appear in an input prompt and precomputes responses for different token combinations using an LLM. Since InstCache does not have an available open-source code, reproducing its functionality is infeasible without knowledge of the exact prompting strategy used.

Zhu et.al. [63] introduce caching with model multiplexing to improve LLM inference. However, their caching technique is primitive, they check whether the request id is present in the cache or not. If so, it uses the same response verbatim, otherwise, it uses LLM to generate the response and chooses to add the request to the cache based on a novel cache replacement policy. Since GenCache’s main technique is generating an appropriate reusable cache, rather than a cache replacement policy, [63] is not an ideal baseline.

Prompt prefix matching is a common caching technique where, if a new prompt shares a prefix with a stored prompt, the key-value (KV) caches of the stored prompt are reused to accelerate inference. Many works leverage this idea [22, 10, 26, 18, 55]. Other works like Cache Blend [52] and CacheCraft [13] extend this idea to adapt within the RAG framework, while also selectively recomputing attention states when prompt prefixes differ slightly. However, these techniques operate at the level of KV cache reuse and represent low-level decoding optimizations, making them orthogonal to our approach. While such methods reduce inference latency, GenCache focuses on reducing the frequency of LLM calls altogether. That said, prompt prefix matching can be integrated into GenCache as an optimization during cache misses, when LLM invocations are required anyway.

Limitations

We scope GenCache’s applicability to AI agents that employ structurally similar prompts for repetitive tasks, such as fault diagnosis, web navigation, and computer-using agents. We find that GenCache is particularly effective for agents following ReAct-style prompting [54], especially when LLM responses correspond to actions rather than rationales, as these exhibit consistent structural patterns across invocations. GenCache also performs well when variable user instructions are generated automatically (e.g., by alert monitors in SRE agents), since such prompts naturally maintain high structural similarity.

Since GenCache relies on LLMs to identify consistent patterns and store them as reusable caches, GenCache may identify an incorrect cache for a few prompts and consequently generate erroneous responses. Therefore, we recommend using GenCache with strict guardrails and mechanisms to detect such errors. When an agent detects an incorrect cache-generated response through downstream validation, it can provide feedback to GenCache, which then deletes the corresponding cache entry (GenCache-feedback). Hence, agent reliability and the ability to identify or roll back incorrect executions are critical for ensuring GenCache’s robustness in practice.

Although CodeGenLLM can identify branching patterns and generate programs with `if-else` clauses when provided with sufficient exemplars, GenCache may fail when the number of branches is large, as too few exemplars exist to capture decisions for all branches. Hence, an adaptive strategy is needed to modify the cached program dynamically whenever a new branch is detected, either through agent feedback upon failure or when a cache remains unused.

Future works include expanding GenCache to more general prompting structure and agent designs. Prompting strategy for CodeGenLLM and ValidLLM could be improved to reduce the number of LLM token usage for generating cache. Some common strategies that can be used to improve the prompting is Reflexion [42], Chain-of-Thought [48], Plan-and-Solve [46], self-critic [23], etc.

Prompts for the LLMs used

Note that the prompts are not optimized, and there will be multiple repetitions in the instructions provided. A more optimized prompt conveying the same message will improve the LLM token usage for CodeGenLLM and ValidLLM.

CodeGenLLM Prompt (for Web-Navigation Agent and with WebShop dataset; clients use OpenAI Chat Completion API to interact with the LLM)

You are an expert who can analyze a few example prompts and their corresponding responses and find a common intent from which the responses were generated by the prompts. Once you find the common intent, your task is to generate a Python program for it.

The example prompts will have a ‘system’ role, a ‘user’ role and might have a ‘function’ description. Your task is to analyze the ‘text’ part within the ‘content’ subfield of the ‘user’ role to find a common pattern across the examples. The part that is to be analyzed will contain some static phrases and some variable phrases. You need to understand how these parts relate to the response. If the responses across all examples are similar (e.g., searching a product), the common pattern must be able to identify how to structure the response based on the static and the variable parts of the phrase.

Your job is to:

1. Identify a consistent pattern across examples
2. Generate a Python program that extracts key variables (e.g., item name, attributes, price) using regex and constructs the response accordingly.
3. Ensure the program works for arbitrary prompts that follow the same overall structure, even with minor variations or synonyms.

Here are some guidelines for pattern extraction:

1. Write a regular expression that can extract the variable parts of the phrase from the user instruction
2. Find reusable structures in the prompts, e.g., "buy {item} from Amazon". Identify and divide the human prompt into verb, item and price phrases.
3. For each part of the phrase, write regex containing synonyms using the examples provided (e.g., if the prompt says "find me", synonyms are "i want to buy" or "i am looking for"). Use up to a MAXIMUM of 5 synonyms per phrase (no more), or the code may raise EOL errors.
4. Use the synonyms to write a regex to identify the item along with its attributes, and the price of the item
5. Generate regex that captures most prompts and is general enough to apply to new prompts, not just the seen ones (e.g. for keyword extraction, identify where the keyword appears in the sentence and extract them from the similar part of the sentence for any arbitrary user

instruction, but following the same prompt template). Please go through all the examples provided before coming up with the regex pattern.

6. Use `re.DOTALL` if needed (e.g., multi-line string matching).

Example Input Prompts and their Corresponding Responses

=====

Guidelines for the code output format and other generic guidelines:

1. Analyze only the 'text' portion in the 'content' subfield of the 'user' role.
2. The code must produce an output in the exact format shown in the example responses. For responses in the format of a dictionary, there should be no additional key-value pairs, otherwise, there will be errors in the downstream task that uses this output.
3. Put escape characters for single and double quotes wherever necessary to avoid syntax errors.
4. Replace `\\n` with `\n` in the final code to handle newline characters correctly from the command line input of the prompt.
5. Put ample `try/except` blocks to catch errors. The code should not crash for any input prompt. If any error occurs, print 'None' and the error.
6. Every `if/elif` must be followed by an `else` to handle unmatched conditions. If no conditions are satisfied, the `else` block should print 'None' along with the reason.
7. The code should be complete with no EOL or syntax errors.

Guidelines for Code Execution Format:

1. The code generated will be saved as `'runnable_code.py'`.
2. It will be run as `'python3 runnable_code.py <input-prompt>'`
3. The code must execute without any manual intervention and take the entire prompt (save it in the variable name 'prompt') as command-line input `<input-prompt>` which includes both the fixed and the variable parts

Final Instructions (strict):

1. Output only the complete Python code.
2. Do not print any explanation, description, or English text apart from the code
3. The output format should exactly match the format of the example responses.

Now Begin!!

CodeGenLLM Prompt (for Cloud-Operations Agent; clients use Langchain API to interact with the LLM)

You are an expert who can analyze a few example prompts and their corresponding responses and find a common intent from which the responses were generated by the prompts. Once you find the common intent, your task is to generate a Python program for it.

The example input prompts provided below will have a prompt template, describing what the LLM was asked to do. This is the static part. It will also contain a dictionary of inputs where each key-value pair can be represented as `'{abc:def}'`. To reconstruct the actual prompt that was used to query the LLM, replace each key (`'abc'`) in the template with its corresponding value (`'def'`). The `'def'` part in the full prompt is the variable part. Your goal is to analyze how this transformed prompt maps to the given output and find a generalizable pattern that applies across all examples. If the responses across all examples are similar (e.g., calling the same API with some parameters), the common pattern must be able to identify how to structure the response based on the static and the variable parts.

Your job is to:

1. Identify a consistent pattern across examples
2. Generate a Python program that extracts key variables using regex and constructs the response accordingly.

3. Ensure the program works for arbitrary prompts that follow the same overall structure.

Here are some guidelines for pattern extraction:

1. The common pattern can be a code to perform a task (extracting a substring from a string) or even a text sentence if all the example responses are sentences with minor changes that do not alter the semantics.
2. For extracting a substring from a string, strip leading/trailing whitespace from the string before matching.
3. Generate regex that captures most prompts and is general enough to apply to new prompts, not just the seen ones (e.g. for keyword extraction, identify where the keyword appears in the sentence and extract them from the similar part of the sentence for any arbitrary user instruction, but following the same prompt template). Please go through all the examples provided before coming up with the regex pattern.
4. Use `re.DOTALL` if needed (e.g., multi-line string matching).

In the examples below, the input dictionary is written in the form:

```
~~~  
KEY -> ABC  
VALUE -> def  
~~~
```

Example Input Prompts and their Corresponding Responses

=====

Guidelines for the code output format and other generic guidelines:

1. The code must produce an output in the exact format shown in the example responses without 'Thought'. For responses in the format of a dictionary, there should be no additional key-value pairs, otherwise, there will be errors in the downstream task that uses this output.
2. Do NOT include 'Thought' in the code-generated output, even if it appears in the examples.
3. Put ample `try/except` blocks to catch errors. The code should not crash for any input prompt. If any error occurs, print 'None' and the error.
4. Every `if/elif` must be followed by an `else` to handle unmatched conditions. If no conditions are satisfied, the `else` block should print 'None' along with the reason.
5. The code should be complete with no EOL or syntax errors.

Guidelines for Code Execution Format:

1. The code generated will be saved as `'runnable_code.py'`.
2. It will be run as `'python3 runnable_code.py <input-dict>'`
3. The code must execute without any manual intervention and take the input dictionary in the form `'{abc:def}'` passed as a string as command-line input `<input-dict>`.

Final Instructions (strict):

1. Output only the complete Python code.
2. Do not print any explanation, description, or English text apart from the code
3. The output format should exactly match the format of the example responses.

Now Begin!!

ValidLLM Prompt

You are an expert evaluator tasked with comparing multiple LLM-generated outputs to their corresponding ground-truth answers. Each answer may be a JSON object, a string, or a code snippet. You should validate only the JSON or code portions; ignore any general English descriptions.

Some of the comparisons may be about an API call searching for a product description that users want to buy. In those cases, consider a match valid if the key attributes are preserved, even if phrased differently (e.g., "a blue headphone with active noise cancellation" and "blue headphone, active noise cancellation"). Often, the LLM-generated answer will be more verbose (former in the example) than the ground-truth answer (latter in the example).

Some important validation rules are:

1. If the ground truth response is in the JSON format, all keys must be present in the LLM-generated response as well. Extra keys in the JSON mean the result is invalid.
2. If some values within the JSON contain English sentences, check semantic equivalence between the ground truth and the LLM-generated response, not exact wording (e.g. "The product is available in the store" and "The store has the product available" are semantically equivalent).
3. For verbose (in LLM-generated response) vs. concise (in ground-truth response) sentences when comparing for certain keys in the JSON, ensure keywords from the concise form appear in the verbose one.
4. For short phrases or code blocks in the response (e.g., "Buy Item", "Search"), check for exact matches.
5. If the LLM-generated response contains 'null' for some keys in the response, while the ground-truth response contains 'None', treat them as equivalent.
6. Ignore punctuation or numeric formatting (e.g., 10 and 10.00 are equal) when comparing.
7. Ignore quote style (single vs. double quotes) (e.g., "content" and 'content' are valid).

Expected Output Format:

```
...
{
  "valid": [0 or 1],
  "reason": "The output is correct/incorrect because ..."
}
...
```

"valid" is a list of 0s and 1s (length of the list = number of comparisons done), where 1 at i'th position means a correct match for comparison 'i', and 0 means a mismatch.

"reason" should give a single combined explanation for why any outputs were incorrect (e.g., extra keys, wrong structure, mismatched values). Do not provide individual explanations per comparison, nor include the comparison number. If an LLM-generated response for any comparison includes an error/exception (e.g., "None: <error>"), include that reason. The objective of the reason field is to easily identify mistakes and rectify, hence be concise.

Do not output anything except the specified JSON.

Strictly follow the format and rules above. Now validate the given examples.

=====

GPT-4.1 Prompt (used to measure negative hits in §4.1)

You are an expert at checking the correctness of two phrases. You will be given an instruction, and two phrases extracted from that instruction. One of the phrases is the ground truth answer, while the other is an answer from our algorithm. The instruction is a user request to buy an item within a price limit. Both phrases will contain (item name, price limit) tuples.

Your task is to verify whether the extracted phrase from our algorithm matches the item description in the instruction. If the description contains some important attributes that will be required if it needs to be searched in an e-commerce website, but those attributes are missing in the extracted phrase, then it is not a match.

The ground truth may have a different phrasing of the item description, which is fine, but the phrase from our algorithm should contain all the necessary information about the item.

You will be given the information in the following format:

Instruction: {{instruction}}

Ground Truth Phrase: {{ground_truth}}

Algorithm Phrase: {{algorithm}}

Your task is to answer with "yes" if the algorithm's response is correct and "no" otherwise. Do not include any other text.

Data for FLASH

For our experiments with the Cloud-Operations Agent FLASH, we show the distribution of the incidents that map to each troubleshooting scenario in Figure 7a. We see that the first troubleshooting scenario (TS-1) covers 69% of the incidents. Hence most incidents' diagnosis strategy remains the same since they map to the same troubleshooting strategy document. We plot the number of repetitions for each incident in Figure 7b. Of the 298 incidents, 253 were unique. We see that 229 incidents never re-occurred, 15 unique incidents recurred twice, while one incident re-occurred 7 times. This shows that using ExactCache cannot produce a high cache hit rate.

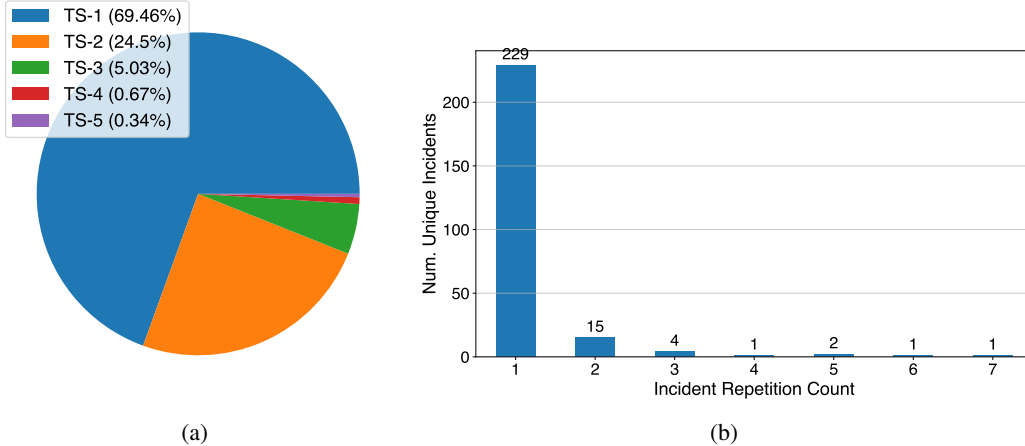


Figure 7: (a) Percentage of incidents that map to each troubleshooting scenario, (b) Number of repetitions for each incident

Structural Modification to the Data

To complete our experiments in §4.1, we also evaluated on a third dataset with prompt characteristics different from *Param-Only* and *Param-w-Synonym*. We call this variation in the prompt set *structural*, where we expressed each user instruction in 10 different ways with structural variations but semantically identical. For example, "I want to buy Bluetooth headphones, under the price of 150 dollars" is expressed as "For under 150 dollars, I want a Bluetooth headphone".

As shown in Table 7, GenCache experiences a drop in both cache hit rate and precision when there are structural changes in the user instructions, compared to the results in Table 1. The reason for this is that most cached regular expressions fail to generalize when the user instruction structure differs.

Method	Structural		
	Hit %	+ve Hit	-ve Hit
ExactCache	0 (\pm 0.0)	N/A	N/A
GPTCache [14]	96.28 (\pm 0.01)	20.72 (\pm 0.01)	79.28 (\pm 0.01)
GenCache	4.23 (\pm 0.21)	72.4 (\pm 0.12)	27.6 (\pm 0.12)

Table 7: Baseline comparison of Hit Rate and its correctness when Prompts had Structural changes

ExactCache has 0% hit rate since no prompts were repeated. Thus, we argue that in domains with structurally diverse prompts, GenCache is less effective (which it is not designed for), and reverting to ExactCache is preferable (hoping that prompts repeat). While GPTCache’s negative hit rate continues to be high, it is not 100%, as it occasionally returns correct responses for semantically similar prompts with structural variations. We observe that when GPTCache shows a positive cache hit for one user instruction, it tends to return positive hits for all structural variants of that instruction. However, as the number of user instructions increases, its reliance on approximate nearest neighbor search for semantic similarity often yields incorrect matches, leading to inaccurate cache hits. Since GenCache already experiences a high negative hits, we do not experiment GenCache-feedback on *Structural* prompt set.

LLM token usage pattern over Time

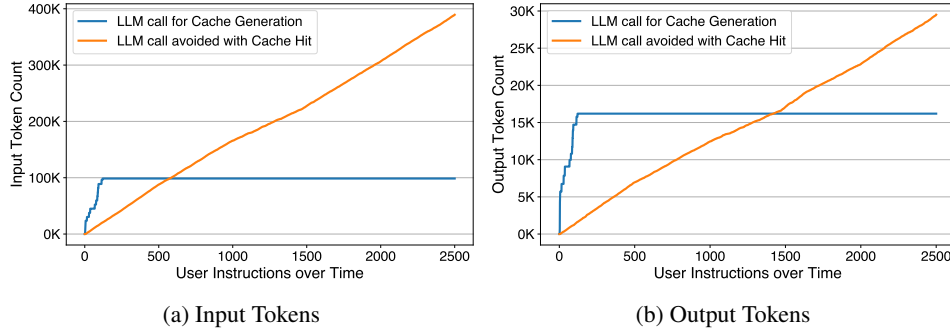


Figure 8: LLM tokens used for cache creation vs LLM tokens avoided due to cache hit for $\nu = 4$

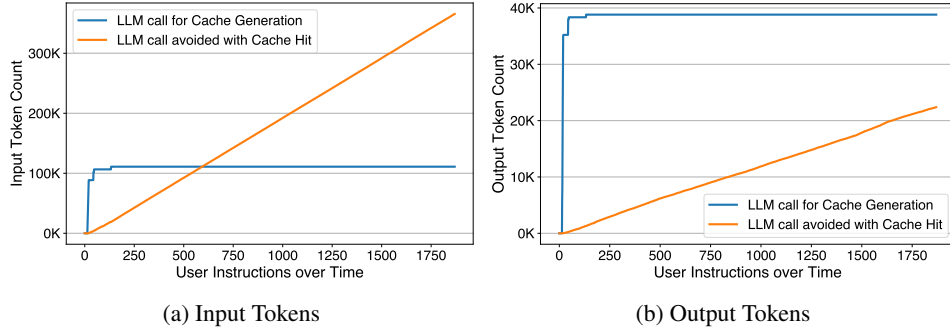


Figure 9: LLM tokens used for cache creation vs LLM tokens avoided due to cache hit for $\nu = 15$

In Figure 5, we showed at least 35% token savings per request on using GenCache. We now show how the LLM token usage varies over time. Figure 8 and Figure 9 plots how the number of input and output tokens varies for $\nu = 4$ and $\nu = 15$ respectively. For all the plots, the ‘blue’ line plots the token usage (input or output) when LLM was called for cache generation, while the ‘orange’ line plots the tokens (input or output) that were saved as a result of avoiding LLM call due to cache hit. We observe that the LLM token usage during cache generation plateaus after initially being high, while the token savings continue to increase as cache hits become more frequent than cache creations over time. While the benefits due to cache hit in input token usage surpass that of cache generation for both ν , the output tokens used for creating the cache with $\nu = 15$ are still higher than the savings due to cache hit after around 1800 user instructions (even though there is an upward trend).