Generative Caching for Structurally Similar Prompts and Responses

Sarthak Chakraborty[1]* **Xuchao Zhang**[2] **Chetan Bansal**[2] **Indranil Gupta**[1]
Suman Nath[2]
[1]University of Illinois at Urbana-Champaign, [2]Microsoft Research

Abstract

Large Language Models (LLMs) are increasingly being used to plan, reason, and execute tasks across various scenarios. Use cases like repeatable workflows, chatbots, and AI agents often involve recurring tasks and tend to reuse similar prompts when interacting with the LLM. This opens up opportunities for caching. With structurally similar prompts that differ in subtle yet important ways, which are also reflected in their corresponding responses, exact prompt matching fails, while semantic caching techniques may return cached responses that are incorrect since they ignore these variations. To address this, we introduce `GenCache`, a generative cache that produces variation-aware responses for structurally similar prompts. It identifies and reuses the pattern in which responses are generated for structurally similar prompts for new requests. We show that `GenCache` achieves an 83% cache hit rate with minimal negative hits on datasets without prompt repetition. In agentic workflows, it improves cache hit rate by $\sim$20% and reduces end-to-end execution latency by $\sim$34% for one workflow compared to standard prompt matching.

1 Introduction

AI applications (consisting of AI agents) augment Large Language Models (LLMs) with external tools [25, 39, 32]. This allows LLMs to interact with the environment and solve intricate tasks [44, 48, 40, 2, 29, 31, 56]. Many such applications themselves tackle similar recurring tasks, and this naturally leads to constructing similar prompts across tasks [18]. For instance, customer support agents frequently encounter repetitive queries with minor variations, or cloud operations agents frequently diagnose recurring system faults [46, 37, 55]. Furthermore, prompt engineering techniques often format prompts with reusable templates (called *system messages* for AI agents), leading to similar prompts. Previous approaches have shown that caching and reusing prompts and associated responses on the application side can significantly reduce latency and cost for such recurring tasks [11, 60, 61, 42, 19, 18, 17].

Existing client-side LLM caches are typically designed as key-value stores. Traditional exact request (or prompt) matching returns a cached value if the new prompt exactly matches a stored prompt key [24, 38, 30], while semantic caches return a cached value based on semantical similarity of the new request with a stored prompt key [11, 17, 18]. Similarity is computed using vector embeddings, where two keys are considered similar if their cosine distance surpasses a predefined threshold.

However, these caches fail for applications where prompt-response pairs show two key properties: (a) the prompts are not exact matches or semantically similar, but instead *structurally similar* with minimal yet important variations (formally defined in §2), and (b) their responses also follow a consistent format, but are not identical to each other; they are novel and *correlated* to the variations in prompts. Consider a web-shopping agent [29, 56, 21] that queries an LLM to determine its next action based on the current page content and the user instruction. Consider two user instructions that together satisfy both the properties mentioned above: (1) *buy 12 AAA batteries from Amazon*, and (2) *buy a USB-C cable from Amazon*. While exact prompt matching (`ExactCache` in short) treats them as

*Work done during internship at Microsoft Research.

Preprint. Under review.

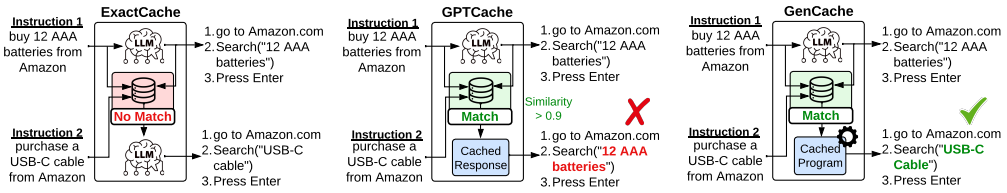


Figure 1: Comparison of GenCache with existing caching techniques. treats both instructions as distinct and results in cache misses, hence uses LLM to generate responses for both. GPTCache encounters cache hit for Instruction 2, but incorrectly returns the already saved response for a similar prompt. GenCache on the other hand, executes the cached program locally on cache hit to generate the correct response tailored to the input

distinct and hence both suffer cache miss, a semantic caching technique like GPTCache [11] considers them as semantically similar, resulting in a cache miss for instruction 1 and a cache hit for instruction 2. However, this is incorrect since GPTCache's response for instruction 2 is identical to the response for instruction 1, as shown in Figure 1. Exact prompt matching strictly prohibits any variations for key matching, thereby ensuring correctness and sacrificing performance. Semantic caching, on the other hand, ignores minimal but potentially important variations, thus risking occasional incorrect results, which may be dangerous if agents perform a non-reversible action.

In this work, we propose *Generative Cache* (GenCache in short), a new caching technique that finds a balance between the performance-correctness trade-off. Unlike traditional caches that return stored LLM responses verbatim on a cache hit, GenCache generates custom responses tailored to the prompt, enabling higher cache hit rates without compromising accuracy. This requires prompt-response pairs to satisfy two key properties described previously, common in AI agents, repetitive workflows, and chatbots. In the example above, instructions 1 and 2 have a similar structure (verb-item-phrase) with minor variations, in verb synonyms (*buy* and *purchase*) and the item name. GenCache automatically discovers such structural similarity and generates a cache hit. On cache hit for instruction 2, GenCache *synthesizes* a custom response (similar in structure to the previous response) with the important variation in item name—Figure 1 depicts this workflow.

GenCache clusters structurally similar prompt-response pairs based on their embedding similarity and generates a program that takes the prompts as input and synthesizes the correctly formatted responses as output. This program captures a common pattern of how the prompt maps to the response across all the prompt-response pairs in the cluster, and saves it as a cache. A new prompt request is matched with an existing cluster based on its embedding similarity, and the stored program is executed locally via a runtime like Python interpreter to generate a response. Unlike existing caches that store prompts and their LLM responses, GenCache stores prompts and the program that can generate these responses. Our evaluations on the Webshop dataset [51] demonstrate over 83% cache hit rate and at least 35% cost savings. On a synthetic dataset with higher structural similarity, cache hit rate is above 98%. When integrated with existing AI agents, it reduced the end-to-end execution latency and achieved 20% higher hit rate. In summary our contributions are:

- We propose a new caching technique, GenCache, for structurally similar prompts that can generate novel variation-aware responses for new prompts.
- GenCache identifies common patterns of generating response from structurally similar prompts within a cluster, and encodes the pattern in the form a program. GenCache then validates the program for correctness and stores it as cache.
- We compare our method with synthetic and benchmark datasets, and report our results by integrating it with two agent frameworks that perform repetitive tasks.

2 Problem Definition

Repetitive tasks or tasks following a similar template [59] are ideal for GenCache to extract the common pattern between multiple prompt-response pairs. Some common use cases are as follows.

Key Use Cases: Apart from the example in §1, other use cases are: (1) SRE agents [46, 37, 55] on receiving system alerts execute remediation step by following relevant troubleshooting documents. Alerts are repetitive and have templates, for example, *"NSM to RNM connection is lost in useast1"* and *"NSM to RNM connection is lost in uswest1"* differ only in the region name. Such alerts map to the same documentation, and SRE agents follow the same steps by only modifying the affected

component. (2) Web agents managing Google Maps may have repeated queries with minor variations *How long does it take to walk from Univ of Pittsburgh to starbucks on Craig Street?* and *How long does it take to walk from Carnegie Mellon University to Univ of Pittsburgh?*. Such agents will need to create a similar API call by varying parameters to serve the structurally similar queries.

`GenCache` is particularly effective for ReAct-style prompting [52] when the expected response is a structured action. In this work, we target use cases involving reversible actions, which are common across many applications like system fault identification (only read queries are issued to databases), adding items to a cart in an e-commerce website (can be checked and removed), checking distances between two endpoints in map (endpoints can be rectified), etc.

Formal definition of Prompt Characteristics: Let x be an individual request (or, user instruction) from a set of repetitive requests $\mathcal{T}$. Let $\alpha(x)$ be the constructed prompt passed to the LLM by a client or an agent, and $\beta(x)$ be the corresponding response. Different characteristics of $\alpha(.)$ and $\beta(.)$ for two requests $x_1, x_2 \in \mathcal{T}$ provide distinct caching opportunities.

1. (Identical Prompts) If $\alpha(x_1) = \alpha(x_2)$ and $\beta(x_1) = \beta(x_2)$, it implies identical input prompts generate the same LLM responses, ideal for exact prompt matching.
2. (Semantically Similar Prompts) If $\alpha(x_1) \sim \alpha(x_2)$, but $\beta(x_1) = \beta(x_2)$ implies *Semantic Caching* techniques like GPTCache [11], InstCache [61] and others [17, 18] are ideal.
3. (Structurally Similar Prompts) If $\alpha(x) = a_1.x.a_2$ and $\beta(x) = b_1.f(x).b_2$ where a_1, a_2, b_1, b_2 are prompt phrases (e.g., phrases like *"buy"* and *"from Amazon"* in the prompt *"buy item X from Amazon"*) and $f(x)$ is a transformation on x (e.g., extracting keywords like *"item X"* from the entire prompt using a regular expression) which are common over multiple requests, `GenCache` can automatically identify a_1, a_2, b_1, b_2 and $f(.)$ from multiple examples using regex matching over $\alpha(x)$ and $\beta(x)$. This allows it to locally generate $\beta(x)$ for any new request x. Unlike `ExactCache` and *semantic caching*, `GenCache` does not require that $\beta(x_1) = \beta(x_2)$. Furthermore, since short prompts by nature exhibit only a handful of deviations in their way to write, `GenCache` can also capture such minor deviations (typically synonyms) in a_1, a_2 (for example, "buy" and "purchase"), by using an "OR" operator in the regex pattern.
4. (Structurally Dissimillar Prompts) When neither $\alpha(x)$, nor $\beta(x)$ show commonalities in their structure or phrasing, `GenCache` does not generalize well

3 `GenCache` Design

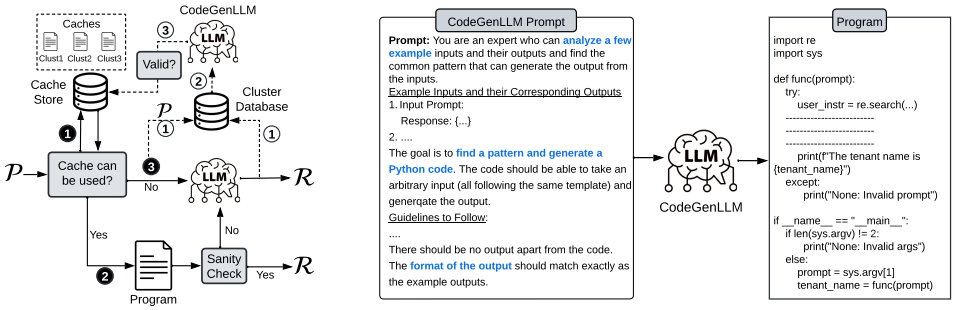
3.1 Overview

The overall workflow of `GenCache` (Figure 2) follows the general cache pattern, i.e., for each incoming input prompt $\mathcal{P}$, identify the correct cache and attempt to use it to return the result. If successful, we directly return the response generated by the cached program. If not, the processing path is same as if the cache did not exist, i.e., to use an LLM. Before returning the LLM-generated response, it is stored in a database along with the prompt so that a cache generation attempt can be made. The primary tasks for cache generation are–(I) cluster *similar* input prompts together such that there is a consistent pattern in the way an LLM generate responses for each of those prompts, (II) for each cluster, identify the consistent pattern (we leverage an LLM) and convert it to a program, and (III) validate whether the generated program is error-free and cache it for further use. For step II, we use in-context learning [12, 33] to help the LLM identify how responses relate to their input prompts within a cluster, and then prompt it to generate the pattern that mirrors this relationship.

Along with the cached program, we store a regular expression representing the structure of clustered prompts, which aids in cache selection at runtime. A new prompt $\mathcal{P}$ is matched to the closest cluster based on cosine similarity of its text embeddings to the cluster center. The stored regex verifies whether the clustered group of prompts is structurally similar to the new request. If so, and a cached program exists for the matched cluster, `GenCache` executes it using a runtime like Python interpreter and returns the generated response after sanity checks. The cached programs are generated with enough exception handling blocks to catch an exception when it does not apply to $\mathcal{P}$.

3.2 Prompt Clustering

Consider an LLM generates the response $\mathcal{R}$ for an input prompt $\mathcal{P}$ on a cache miss. Similar sets of $(\mathcal{P}, \mathcal{R})$ pairs are clustered together and stored. For each prompt $\mathcal{P}$, we first convert it to an n-dimensional embedding e_p using `SentenceTransformer` [36]. If no clusters exist, we initialize a



(a) Cache Reuse process (b) Generated Program and its prompt

Figure 2: `GenCache` workflow (solid lines: cache reuse, dotted lines: cache generation). Figure 2a illustrates the cache reuse process–For a new prompt $\mathcal{P}$, the system finds the nearest cluster based on a similarity threshold and checks for an available cache for reuse ❶. If a suitable cache is found, it generates response $\mathcal{R}$ after passing sanity checks ❷. If not, an LLM generates $\mathcal{R}$ ❸. In this case, we store $(\mathcal{P}, \mathcal{R})$ in the cluster database ①. Once enough example pairs accumulate in a cluster, `CodeGenLLM` attempts to generate a program ② and store it in the cache store ③ after validation. Figure 2b shows the prompt for `CodeGenLLM` and the generated program.

new cluster $\mathcal{C}_0$ with the pair $(\mathcal{P}, \mathcal{R})$. However, in LLM systems, prompts are often wrapped in large common templates (*system messages*), which can dominate the embedding and cause semantically unrelated prompts to appear similar. To mitigate this, we also use response similarity along with prompt similarity for clustering. Since $\mathcal{R}$ often contains multiple key-value pairs, sentence-level embeddings may be insufficient. Therefore, we create an embedding array $[e_r]$ by computing n-dimensional embeddings for each value of the key-value pair of $\mathcal{R}$.

We now compute two similarities of $\mathcal{P}$ with each cluster $\mathcal{C}_i$, one based on prompt embeddings and the other on response embeddings. The prompt similarity score s^p is the cosine similarity between e_p and the cluster prompt centroid c_i^p (the mean of all e_p embeddings in $\mathcal{C}_i$). The response similarity score s^r is the average cosine similarity between $[e_r]_j$ and the cluster response centroid $[c_i^r]_j$ (mean of all $[e_r]_j$ embeddings in $\mathcal{C}_i$ at index j) across all indices j. We additionally verify that the number of key-value pairs matches between $\mathcal{R}$ and the response centroid.

To assign the input prompt to a cluster $\mathcal{C}$, both s^p and s^r must exceed a threshold, and their sum should be the highest among all candidate clusters. Let $\{\mathcal{C}^p\}$ be the set of all clusters whose s^p is greater than a similarity threshold T^p, while $\{\mathcal{C}^r\}$ is the set of clusters whose s^r is greater than another threshold T^r. $\{\mathcal{C}^{int}\} = \{\mathcal{C}^p\} \cap \{\mathcal{C}^r\}$ captures the set of clusters where both the similarities are over the threshold. The most similar cluster $\mathcal{C}$ for $(\mathcal{P}, \mathcal{R})$ is chosen such that the similarity is the maximum, i.e., $\mathcal{C} = \text{argmax}_{\mathcal{C}^{int}}(s^p + s^r)$.

`GenCache` performs online clustering without any constraint on the number of clusters. Each cluster is stored as a key-value store where the cluster-ID forms the key and the group of prompts and their corresponding LLM responses along with their embeddings form the values.

3.3 Program Generation for Cache

For each cluster $\mathcal{C}$, we use an LLM (`CodeGenLLM`) to generate a Python program to be stored as cache. Similar ideas have been explored previously [13, 16], but their solution is tailored towards solving mathematical reasoning tasks. The program we generate takes as input the prompt $\mathcal{P}$ and outputs the response $\mathcal{R}$ in the correct format. To generate the program, we provide `CodeGenLLM` with a sufficient number of prompt-response pairs from $\mathcal{C}$ as in-context examples, and prompt it to identify the underlying pattern that maps most input prompts to their responses. The underlying pattern must identify the variations in the structurally similar prompts and use these to generate the response. We noticed that such variations are often keywords from the input prompt (e.g., in Figure 1). Since the prompts for our use case have a specific structure, a regular expression can typically be used to extract those keywords. Thus, if a consistent pattern is detected, `CodeGenLLM` generates a program that performs regular expression search to extract the necessary keywords from the prompt and synthesize the desired response in the correct format. Returning to our running examples from §1, `CodeGenLLM` finds and encodes the consistent pattern with a regular expression `r'buy|purchase|get (item_name) from Amazon'`. Note that the regex contains possible synonyms for terms or phrases. It takes a user instruction *"buy item from Amazon"* as input and produces the response `actions:(1) go to`

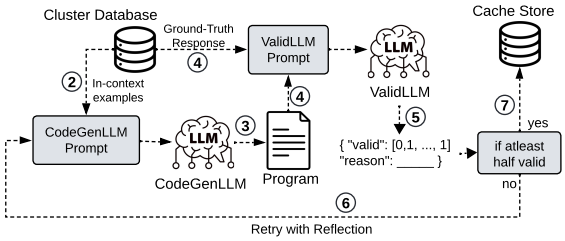


Figure 3: Program validation process before storing it as cache. Step numbering remain consistent with Figure 2a, and we only show from (2) onwards here. After program generation (3), `ValidLLM` validates that the program-generated responses match expected outputs by using in-context examples in the prompt (4). It produces a boolean-array for each example (5). If less than half matches, the system retries cache generation using a reflection-based prompt (6). Otherwise, it stores the validated program as cache (7).

`Amazon.com, (2) Search("match.groups(0)"), and (3) press enter'`, by replacing the item name in `match.groups(0)`. Figure 2b shows the prompt snippet used for `CodeGenLLM`.

The prompt for `CodeGenLLM` includes both the common template and the user instruction for each in-context example. Providing the template enables the LLM to infer the way responses were derived from the input prompts. These multiple examples guide `CodeGenLLM` to identify a consistent pattern, write a program with regex search computations, and capture the necessary variations. The `CodeGenLLM` prompt also provides guidelines for appropriate error handling, format adherence, and default `{null}` response when regex matching fails. This ensures that the generated program produces reliable outputs and avoids malformed responses, that could otherwise disrupt an agent execution. The full prompt used for `CodeGenLLM` is included in the supplementary material.

A minimum number of exemplars, maintained as a hyperparameter ν, is needed before `CodeGenLLM` is invoked. A higher ν enables `CodeGenLLM` to easily identify the patterns, but also increases the risk of introducing noise to the in-context examples if the clusters contain outliers. We evaluated against varying ν to analyze this trade-off. Once a cluster contains at least ν exemplars, program generation is invoked in the background without affecting the critical path of client response.

3.4 Program Validation

Before a generated program is stored as a cache, it must be validated for correctness, specifically, whether it can process the input prompt, extract relevant keywords from the prompt, and generate the response with correct variations in the required format. This validation step ensures that only error-free code or code with appropriate exception handling (to indicate incorrect results) is stored.

We use a separate LLM, `ValidLLM`, to validate the generated program. Using the same exemplars from §3.3, we execute the program locally via a Python interpreter to obtain *program-generated responses*. `ValidLLM` assesses whether each response matches its corresponding exemplar response, both in content and format. For the text sentence portions, `ValidLLM` checks for semantic correctness and the presence of 'key information' instead of strict matching. E.g., *"The User wants to purchase Item X"* and *"user wants to buy item X"* will be evaluated as a match. `ValidLLM` returns a boolean (0 or 1) indicating correctness for each example along with a justification.

If fewer than half of the program-generated responses match their exemplars, we retry program generation with *reflection*, which incorporates `ValidLLM`'s justification feedback into the `CodeGenLLM` prompt. However, if at least half of the responses match, the program is accepted as a cache and stored in the cache store. While one alternative is to filter out non-matching examples and reassign them to a new cluster, this often results in numerous fragmented clusters highly similar to each other, so we avoid this approach. Figure 3 illustrates the overall program validation process. `GenCache` limits the number of cache generation retries for a cluster to ρ to prevent increasing the cost of cache generation. The prompt used for `ValidLLM` is included in the supplementary material.

3.5 Cache Management

Each cached Python program is typically under ~5KB, so we cap the total cache size to a few tens of megabytes. Once this limit is reached, the eviction policy removes the least-recently used cache entry while retaining its associated cluster. To prevent clusters from growing indefinitely, `GenCache` limits each cluster to at most 3ν prompt-response pairs. If the prompt clustering technique identifies an

already full cluster $\mathcal{C}$ as the closest match, it simply does not append it to the cluster. Once a program is stored as a cache, it can still fail to provide a response for a new request. In such cases, `GenCache` may regenerate the program for the corresponding cluster, but the total retries will be capped to ρ.

4 Experiments

Setup: We implement `GenCache` as a library and expose an API interface to which clients issue input prompts. It either returns a cache-generated response or uses an LLM to return the response on cache misses. We evaluated `GenCache` in two scenarios: (1) issue standalone user prompts (§4.1, §4.2)–we create synthetic data as discussed below, and (2) integrate with two AI agents (§4.3). While `GenCache` can be used with agents built using any framework, we evaluate with simple prompt-based agents and leave the incorporation with a more performant agent for future work. The first agent is an open-source web navigation agent Laser [29, 7], built on the OpenAI Chat Completion [1] framework. It plans and executes webpage actions like "Search" and "Buy" to purchase items/add appropriate items to the cart based on user instructions. The second agent, `AgentX`, is a cloud-operations agent built using LangChain [6] that diagnose recurring system faults. Given an incident description, it retrieves relevant troubleshooting documents via RAG [25], and follows the relevant steps until the root cause is identified. However, we restrict it from conducting any mitigation. Both agents use ReAct-style prompting [52]. We use `GPT-4o` for `CodeGenLLM` and `ValidLLM`, and set a default $\rho = 30$ for each cluster based on empirical decisions. Small prompt modifications are applied to `CodeGenLLM` and `ValidLLM` prompts to accommodate task-specific characteristics. On cache misses, `GenCache` use `GPT-4` to generate responses for `AgentX`, and `GPT-4o` for all other experiments. We run our experiments on cloud servers with 8-core Intel Xeon CPU and 64 GB memory.

Datasets: Since `GenCache` excels at repetitive tasks with structurally similar prompts, popular language understanding benchmarks [43, 49, 57, 27, 54] that evaluate an LLM and Agent's comprehensiveness on diverse tasks are not suitable. We thus evaluate on prompts from WebShop [51], a simulated e-commerce environment featuring 12000+ diverse crowd-sourced user instructions. While the original prompts do not include item price, we adopt Laser's [7] augmentation, which adds a maximum price to each instruction. As a result, each instruction follows a consistent structure: a verb indicating purchase request (e.g., "I want to buy"), then an item description with attributes, and a price limit. The item and price (i.e., the parameters) vary across prompts, while the overall structure remains consistent with minor variations in the verb phrasing, making this dataset ideal for evaluating our method. To evaluate the setup scenario 1, we use WebShop's user instructions as seed data and generate two synthetic prompt sets with different characteristics: (i) *Param-Only*–only the parameters vary while other phrasing remains identical, and (ii) *Param-w-Synonym*–parameters and the verb phrasing while maintaining the `verb-item-price` structure. For *Param-Only*, we use GPT-4o to extract the item and price and reformat all prompts as *"I want to buy* `{item}`*, under the price range of* `{price}`. For *Param-w-Synonym*, optional words or synonyms are introduced into verb phrasings across prompts. Some instructions are split into two sentences, the first describing the verb and the item, and the second detailing attributes and price. These prompts are issued to `GenCache`'s API interface with the expected response of receiving just the (item name, price limit) tuple.

Baselines: We compare `GenCache` to `ExactCache` and *GPTCache* [11, 4], a widely popular representative among semantic caching techniques. For `ExactCache`, we leverage hash-based indexing of prompts. GPTCache, however, uses embeddings similarity for prompt matching. We use the Sentence Bert embedding `all-MiniLM-L6-v2` [36] for computing prompt embeddings and the FAISS [22] indexing strategy for similarity search with a similarity threshold of 0.95. We do not compare with prompt-prefix matching techniques [10, 19] as they focus on low-level decoding operations and are orthogonal to `GenCache`. Our baseline choices are well detailed in the supplementary material.

We evaluate `GenCache` in the following categories:

1. What is the cache hit rate, and how many times does the hit result in an incorrect response?
2. What is the overall cost in terms of the number of LLM calls and the token sizes to create a reusable cache? Does our cache generation cost exceed the savings we get?
3. How well does `GenCache` perform within repeatable agentic workflows?

4.1 Hit-Rate Measurements

We measure the cache hit rate of each method over 10,000 input prompts from the synthetic prompt sets, and report the results in Table 1. Each experiment starts with an empty Cache Store and Cluster

Method	Param-Only			Param-w-Synonym		
	Hit %	+ve Hit	-ve Hit	Hit %	+ve Hit	-ve Hit
`ExactCache`	0 (± 0.0)	N/A	N/A	0 (± 0.0)	N/A	N/A
GPTCache [11]	90.92 (± 0.02)	0 (± 0.0)	100 (± 0.0)	88.71 (± 0.01)	0 (± 0.0)	100 (± 0.0)
`GenCache`	**97.81 (± 0.96)**	98.03 (± 0.05)	1.97 (± 0.05)	**83.66 (± 6.37)**	92.16 (± 0.12)	7.84 (± 0.12)
`GenCache-feedback`	82.35 (± 1.57)	**99.63 (± 0.02)**	**0.37 (± 0.02)**	68.32 (± 4.85)	**95.58 (± 0.08)**	**4.4 (± 0.08)**

Table 1: Baseline comparison of Hit Rate and its correctness with different prompt datasets

Database. Thus, the first ν (default $\nu = 4$) input prompts were used to populate the cluster database before caching can even be attempted. Beyond measuring cache hit rate, we evaluated the correctness of the responses generated by `GenCache` using `GPT-4.1` with occasional human feedback. `GPT-4.1` verifies whether the item name and price extracted by the cache semantically align with the ground truth and whether the item remains searchable in an e-Commerce setting. We report the percentage of positive and negative hits, i.e., among cache hits, how many responses had semantically correct item name with all attributes (positive) versus those that did not (negative). We also extend `GenCache` to reduce negative hits by incorporating feedback from `GPT-4.1` that acts as an oracle correctness verifier. We name this method `GenCache`-feedback. When a cache-generated response is identified as a negative hit, `GenCache` deletes the corresponding cached program, but retains the associated cluster so that it can retry generating a new program.

We observe that `GenCache` achieves over 97% cache hit rate with ∼2% negative hits for *Param-Only* dataset. Since only the parameters (i.e. item name and price) change, `CodeGenLLM` is able to reliably detect patterns that can be cached as programs. The few negative hits are attributed to long, complex item descriptions that might result in unsatisfactory e-Commerce search results for the specific item, as flagged by `GPT-4.1` and human feedbacks. For *Param-w-Synonym* where verbs phrases may deviate, the cache hit rate drops to 84%, with a higher false positive rate. In addition to lengthy item descriptions, false positives also arise due to cases where the item attributes follow a period. For example, a cached pattern like `r'buy|need|purchase|get (item name)'`, misfires on the user instruction *"want a wireless headphone. need it in black"*, returning *"it in black"* as the item. However, using feedback lowers cache hit rate for `GenCache`-feedback since there are cache deletions, but the negative hits also reduce. This shows that if a client (or an agent) can provide a feedback that the response from cache use resulted in an error, `GenCache` can adapt and modify the cache.

`ExactCache` exhibits no cache hits, while all GPTCache's cache hits are negative. Since there are no prompt repetitions, no responses should be repeated as well. However, GPTCache returns a cached response corresponding to a prompt that shows high similarity to the new request (example in §1).

4.2 Cache Generation Cost vs Savings

`GenCache` incurs a cost for using `CodeGenLLM` and `ValidLLM` to generate programs and store them. To be effective, this cost must not exceed the savings we get from cache hits. To evaluate this, we measure the ratio of the number of LLM calls used for creating cache (cost) to the number of cache hits (savings, or avoided LLM calls). We run `GenCache` with the default $\nu = 4$ over 5000 input prompts for each dataset type and plot this ratio over time as more prompts are processed in Figure 4. For the first few prompts, `GenCache` incurs cost to create caches, hence the ratio exceeds 1. As prompt count exceeds ν, the program is generated, validated, and cached, leading to a sharp rise in cache hits, decreasing the ratio. *Param-Only* dataset exhibits a higher cache hit rate, hence a lower ratio than *Param-w-Synonym*.

While Figure 4 elicits the cost with a fixed ν, varying ν changes the number of exemplars used as in-context examples for `CodeGenLLM`, which will impact the number of LLM calls required to generate a reusable cache. As shown in Figure 2 with *Param-w-Synonym* prompts across 5 runs, increasing ν consistently reduces the number of LLM calls. With more in-context exemplars, `CodeGenLLM` can better infer the regex patterns and reliably write it as a program. Although the exact number of LLM calls may vary across different domains, the general trend is expected to be consistent.

In addition to optimizing the number of LLM calls, `GenCache` must also minimize token usage, as LLM pricing is often based on the number of input and output tokens [8]. To evaluate token usage cost, we vary ν and compute the average tokens (input + output) used per user request for cache creation across 5 runs (each run with 2000 prompt requests), comparing it with the cost incurred during cache misses (Figure 5). As a baseline, we plot the token usage without using a cache. For every ν, we observe that caching yields at least 35% token savings per request. For $\nu = 2$, the savings are as high as 73%. As ν increases, the average cost of cache construction increases due to

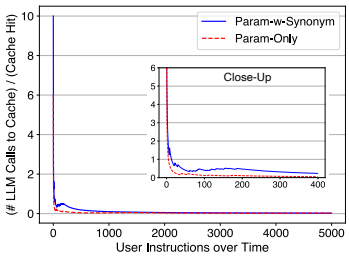


Figure 4: Ratio of no. of LLM calls used for creating cache to number of cache hits plotted against incoming prompts in time

ν	LLM calls
2	32.4
4	29.6
6	21.2
10	17.2
15	18.8

Table 2: Total no. of LLM calls to create reusable caches

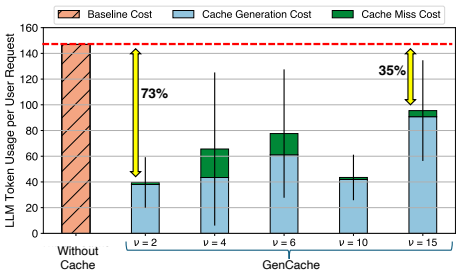


Figure 5: LLM token usage for caching compared against the baseline of 'Without Cache'. With no prompt repetitions, `ExactCache` incurs same cost as the baseline

Agent	Performance Metrics	`GenCache+ExactCache`	`ExactCache`
`AgentX`	No. of API calls	4725	4960
	No. of LLM calls for Cache Creation	536	N/A
	% Cache Hits (`ExactCache` contribution)	**54.7%** (31.1%)	34.4%
	Execution Time	**39.91s**	53.45s
Laser	No. of API calls	1315	1308
	No. of LLM calls for Cache Creation	100	N/A
	% Cache Hits (`ExactCache` contribution)	**37.2%** (12.8%)	5.7%
	Execution Time	**5.02s**	5.16s

Table 3: Evaluating the benefits of using `GenCache` in regular agentic workflows

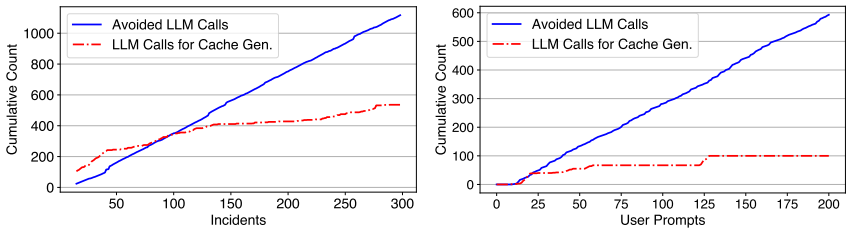
more in-context exemplars in the `CodeGenLLM` prompt. However, for each ν, this cost plateaus after initially being high, while savings continue to increase as cache hits become more frequent than cache creations over time. The cost dip at $\nu = 10$ is because a highly reused cache was stored with very few retries. Figure 5 shows large standard deviations in cache construction and cache miss cost, primarily due to `GPT-4o` being unreliable across runs, and hence cache construction required multiple retires for some runs. Suitable prompt engineering like optimized prompting, fewer repetitions of guardrails, specifying the system message once, etc. can lower the cache construction cost. Alternatively, a stronger model like `GPT-4` can generate reliable programs with fewer retries.

4.3 Impact on Agentic Workflows

We evaluate end-to-end performance impact of `GenCache` on two agentic workflows. Since such workflows already leverage `ExactCache` by default, we deploy `GenCache` alongside it to measure additional benefits. Table 3 reports improvements in cache hit rate and execution time, while Figure 6a compares the cost against savings.

Cloud-Operations Agent: We run `AgentX` using incident diagnosis data from Company-X, collected over 2 months. The data includes a total of 298 incidents covering five different troubleshooting scenarios. The top-1 troubleshooting scenario covers 69% of the incidents, while the top-2 covers 93%. This indicates that incidents are recurring, which will make caching particularly effective instead of repeated LLM calls. To diagnose an incident, `AgentX` took 16 LLM calls on average, including identifying the correct documents, generating a coarse and fine-grained plan, and conducting the diagnosis. `GenCache` improved cache hit rate by 23% achieving 54% overall, compared to ∼34% with `ExactCache` alone. Using `GenCache` also reduced the average diagnosis time of the agents by ∼25%. Figure 6a illustrates that as more and more incidents are handled by `AgentX`, the number of LLM calls avoided due to cache hits outweighs the LLM calls incurred to create cache which plateaus. The growing gap demonstrates `GenCache`'s long-term benefits. Out of 536 LLM calls made for cache creation, only 10.4% (56/536) resulted in reusable caches within the workflow; the remaining attempts were flagged as invalid by `ValidLLM`. There were also failure cases; across 298 incidents, `AgentX` failed in only 3 cases (1%) due to mapping to an incorrect troubleshooting document as a result of cache hit, but it was non-detrimental since we focused only on reversible workflows.

Web-Navigation Agent: We run Laser [29] on 200 requests from the WebShop dataset [51], which iteratively queries the LLM, first for a *rationale*, then for the corresponding action type. The workflow is typically to search for an item, compare item descriptions to user requirements, and ultimately add it to the cart. Since rationales involve prompt-specific reasoning and are structurally diverse, we disable `GenCache` for those steps. Instead, we enable `GenCache` only when the LLM selects



(a) Cloud Operations Agent (b) Web Navigation Agent

Figure 6: Comparison of LLM calls avoided due to cache hit (savings) with the LLM calls incurred for cache creation (cost). As more inputs are processed, the gap between savings and cost widens.

an action type and its parameters. On average, Laser required 12 LLM calls per request, of which ∼50% (1315 total) used `GenCache`'s API interface. Using `GenCache`, we observed 37.2% cache hits (489/1315), with 12.8% of them (63/489) due to exact prompt matching. Similar to `AgentX`, we see that the savings due to cache hit grow with more prompts while LLM calls for cache generation plateaus in Figure 6b. `GenCache` incurred an additional 100 LLM calls for cache creation. Out of these 100 LLM calls, 34% resulted in creation of reusable caches, while 66% failed (number of retries capped at $\rho = 30$) since the exemplars that were clustered together were too different. Failure cases were: (i) 5.5% of prompts failed due to the cached program repeatedly returning the wrong item from the catalog, and (ii) 11% failed due to formatting errors in cached response–issues that can be addressed through improved prompting, output guardrails, or providing feedback to `GenCache`. Long and complex item descriptions, which were flagged as negative hits in Table 1, did not lead to failures here, since the multi-step workflow included intermediate LLM calls for generating rationales, which mitigates the impact of complex item descriptions.

5 Related Works

Semantic Caching: While GPTCache [11] is a popular semantic caching approach, few works have focused on implementing a caching architecture to make semantic caching usable in real-world settings [14, 26, 34, 60]. SCALM [26] identifies requests frequently visited by users and selectively caches those requests, while [60] improves LLM inference by introducing model multiplexing along with semantic caching. However, [60] relies on the existence of some semantic caching oracle that can group prompts without false positives. LangCache [17] innovates on the embedding layer by domain-adaptive fine-tuning of the embedding models. MeanCache [18] introduces a user-centric semantic caching system that preserves user privacy, and hence employs federated learning to build different embedding models locally at each user device. InstCache [61] introduces predictive caching for short user prompts by predicting user instructions using LLMs and pre-populating the cache.

Prompt Caching: Reusing attention states using Key-Value (KV) Cache is a popular LLM inference optimization for a single prompt during autoregressive token generation [35]. Prompt caching [19, 50, 10, 58, 53, 28] extends this idea to multiple prompts, where KV caches across multiple prompts are reused based on prompt prefix matching. Prompt Cache [19] designs an explicit structure for writing prompts to enable seamless detection of prompt prefixes. SGLang [58] and ChunkAttention [53] build efficient data structures for KV cache reuse, while works like Cache-Craft [10] and CacheBlend [50] implement prompt caching for RAG systems by efficiently reusing and recomputing only the necessary cached chunks. Prompt caching is also used in popular LLM services [9, 3] to reduce costs. However, these works are orthogonal to `GenCache`, and they can be used in parallel to reduce costs when `GenCache` incurs a cache miss.

6 Summary and Discussion

We proposed `GenCache`, a novel caching technique for structurally similar prompts that uses LLM to identify a common pattern to generate responses from similar prompts, and caches the patterns as programs after validation. On a cache hit, a stored program is executed to generate variation-aware responses. Future works include supporting structurally diverse prompts and non-reversible agent workflows, a limitation of the current work. Furthermore, modifying AI agents to identify negative hits and relaying back the feedback to the caching layer can help improve cache accuracy over time. Designing structured human-LLM interaction schemas, like in [19] will enable better caching.

References

[1] Chat completions. https://platform.openai.com/docs/guides/chat-completions, 2025.

[2] Computer-using agents. https://openai.com/index/computer-using-agent/, 2025.

[3] Effectively use prompt caching on amazon bedrock. https://aws.amazon.com/blogs/machine-learning/effectively-use-prompt-caching-on-amazon-bedrock/, 2025.

[4] Gptcache : A library for creating semantic cache for llm queries. https://gptcache.readthedocs.io/en/latest/, 2025.

[5] Gptcache : A library for creating semantic cache for llm queries. https://github.com/zilliztech/gptcache, 2025.

[6] Langchain. https://www.langchain.com/, 2025.

[7] Laser: Llm agent with state-space exploration for web navigation. https://github.com/Mayer123/LASER, 2025.

[8] Openai api pricing. https://openai.com/api/pricing/, 2025.

[9] Prompt caching with openai, anthropic, and google models. https://www.prompthub.us/blog/prompt-caching-with-openai-anthropic-and-google-models, 2025.

[10] Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. Cache-craft: Managing chunk-caches for efficient retrieval-augmented generation. *arXiv preprint arXiv:2502.15734*, 2025.

[11] Fu Bang. GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. In Liling Tan, Dmitrijs Milajevs, Geeticka Chauhan, Jeremy Gwinnup, and Elijah Rippeth, editors, *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pages 212–218, Singapore, December 2023. Association for Computational Linguistics.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[13] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023.

[14] Soumik Dasgupta, Anurag Wagh, Lalitdutt Parsai, Binay Gupta, Geet Vudata, Shally Sangal, Sohom Majumdar, Hema Rajesh, Kunal Banerjee, and Anirban Chatterjee. wallmartcache: A distributed, multi-tenant and enhanced semantic caching system for llms. In *Pattern Recognition: 27th International Conference, ICPR 2024, Kolkata, India, December 1–5, 2024, Proceedings, Part VII*, page 232–248, Berlin, Heidelberg, 2024. Springer-Verlag.

[15] Yuan Feng, Hyeran Jeon, Filip Blagojevic, Cyril Guyot, Qing Li, and Dong Li. Attmemo : Accelerating transformers with memoization on big memory systems, 2023.

[16] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, 23–29 Jul 2023.

[17] Waris Gill, Justin Cechmanek, Tyler Hutcherson, Srijith Rajamohan, Jen Agarwal, Muhammad Ali Gulzar, Manvinder Singh, and Benoit Dion. Advancing semantic caching for llms with domain-specific embeddings and synthetic data, 2025.

[18] Waris Gill, Mohamed Elidrisi, Pallavi Kalapatapu, Ammar Ahmed, Ali Anwar, and Muhammad Ali Gulzar. Meancache: User-centric semantic cache for large language model based web services. *arXiv preprint arXiv:2403.02694*, 2024.

[19] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 325–338, 2024.

[20] Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*, 2024.

[21] Izzeddin Gur, Hiroki Furuta, Austin V Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. In *The Twelfth International Conference on Learning Representations*, 2024.

[22] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, page 19–28, New York, NY, USA, 2003. Association for Computing Machinery.

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

[26] Jiaxing Li, Chi Xu, Feng Wang, Isaac M von Riedemann, Cong Zhang, and Jiangchuan Liu. Scalm: Towards semantic caching for automated chat services with large language models. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2024.

[27] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In Nicoletta Calzolari, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Koiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asuncion Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga, editors, *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA).

[28] Yuhan Liu, Yuyang Huang, Jiayi Yao, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. Droidspeak: Kv cache sharing for cross-llm communication and multi-llm serving, 2024.

[29] Kaixin Ma, Hongming Zhang, Hongwei Wang, Xiaoman Pan, Wenhao Yu, and Dong Yu. Laser: Llm agent with state-space exploration for web navigation. *arXiv preprint arXiv:2309.08172*, 2023.

[30] Mauricio Marin, Veronica Gil-Costa, and Carlos Gomez-Pantoja. New caching techniques for web search engines. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, page 215–226, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Sami Marreed, Alon Oved, Avi Yaeli, Segev Shlomov, Ido Levy, Aviad Sela, Asaf Adi, and Nir Mashkif. Towards enterprise-ready computer using generalist agent. *arXiv preprint arXiv:2503.01861*, 2025.

[32] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

[33] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[34] Ramaswami Mohandoss. Context-based semantic caching for llm applications. In *2024 IEEE Conference on Artificial Intelligence (CAI)*, pages 371–376, 2024.

[35] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 606–624. Curan, 2023.

[36] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China, November 2019. Association for Computational Linguistics.

[37] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 208–219, New York, NY, USA, 2024. Association for Computing Machinery.

[38] Patricia Correia Saraiva, Edleno Silva de Moura, Nivio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, page 51–58, New York, NY, USA, 2001. Association for Computing Machinery.

[39] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[40] Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. Building ai agents for autonomous clouds: Challenges and design principles. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 99–110, New York, NY, USA, 2024. Association for Computing Machinery.

[41] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc., 2023.

[42] Simranjit Singh, Michael Fore, Andreas Karatzas, Chaehong Lee, Yanan Jian, Longfei Shangguan, Fuxun Yu, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. Llm-dcache: Improving tool-augmented llms with gpt-driven localized data caching. In *2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2024.

[43] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. *SuperGLUE: a stickier benchmark for general-purpose language understanding systems*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[44] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[45] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

[46] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, page 4966–4974, New York, NY, USA, 2024. Association for Computing Machinery.

[47] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[48] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652. Curran Associates, Inc., 2024.

[49] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[50] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 94–109, New York, NY, USA, 2025. Association for Computing Machinery.

[51] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 20744–20757. Curran Associates, Inc., 2022.

[52] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[53] Lu Ye, Ze Tao, Yong Huang, and Yang Li. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11608–11620, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[54] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[55] Xuchao Zhang, Tanish Mittal, Chetan Bansal, Rujia Wang, Minghua Ma, Zhixin Ren, Hao Huang, and Saravan Rajmohan. Flash: A workflow automation agent for diagnosing recurring incidents. October 2024.

[56] Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. Webpilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23378–23386, 2025.

[57] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. LMSYS-chat-1m: A large-scale real-world LLM conversation dataset. In *The Twelfth International Conference on Learning Representations*, 2024.

[58] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 62557–62583. Curran Associates, Inc., 2024.

[59] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024.

[60] Banghua Zhu, Ying Sheng, Lianmin Zheng, Clark Barrett, Michael Jordan, and Jiantao Jiao. Towards optimal caching and model selection for large model inference. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 59062–59094. Curran Associates, Inc., 2023.

[61] Longwei Zou, Tingfeng Liu, Kai Chen, Jiangang Kong, and Yangdong Deng. Instcache: A predictive cache for llm serving. *arXiv preprint arXiv:2411.13820*, 2024.

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 `AgentX`
6. Structural Modification to the Data
7. LLM token usage pattern over time

Code Link

Code link for `GenCache` is available at https://anonymous.4open.science/r/GenCache_Artifact-ED54/

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 [11] 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.[17], 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 [61] 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. [60] 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, [60] 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 [19, 9, 23, 15, 53]. Other works like Cache Blend [50] and CacheCraft [10] 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 use structurally similar prompts for repetitive tasks, such as fault diagnosis, web navigation, and computer-using agents. We also observe improved performance for agents that follow ReAct-style prompting [52], particularly when LLM responses are actions, as they tend to follow consistent patterns across different agent invocations. Thus, 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 [41], Chain-of-Thought [47], Plan-and-Solve [45], self-critic [20], 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.

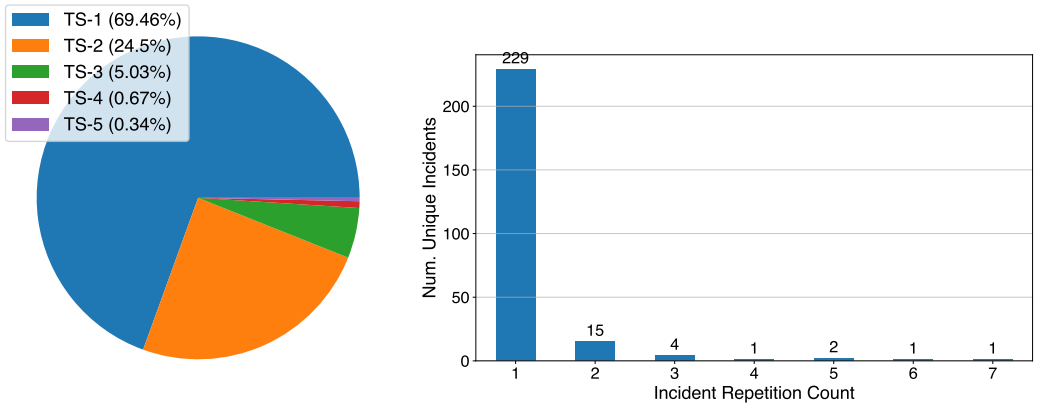
The ground truth may have a different phrasing of the item description, which is fine, but the phrase from our algorithm should contain all the necessary information about the item.

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 `AgentX`

For our experiments with the Cloud-Operations Agent `AgentX`, 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.



(a) (b)

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".

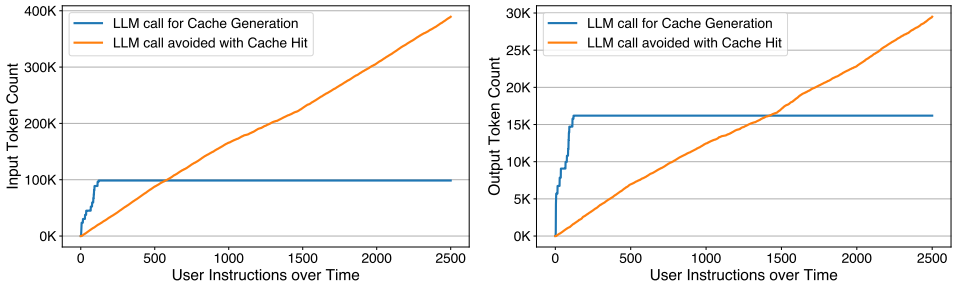
As shown in Table 4, `GenCache` experiences a drop in both cache hit rate and precision when there are structural changes in the user instructions, compared to the results in Table 1. The reason for this is that most cached regular expressions fail to generalize when the user instruction structure differs.

Method	Structural		
	Hit %	+ve Hit	-ve Hit
`ExactCache`	0 ($\pm$ 0.0)	N/A	N/A
GPTCache [11]	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 4: Baseline comparison of Hit Rate and its correctness when Prompts had Structural changes

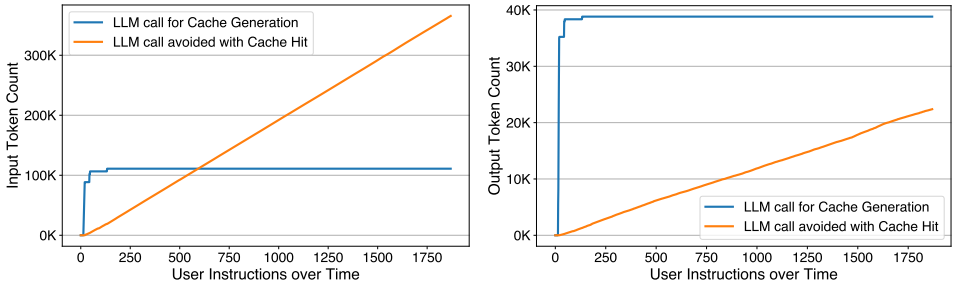
`ExactCache` has 0% hit rate since no prompts were repeated. Thus, we argue that in domains with structurally diverse prompts, `GenCache` is less effective (which it is not designed for), and reverting to `ExactCache` is preferable (hoping that prompts repeat). While GPTCache's negative hit rate continues 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



(a) Input Tokens (b) Output Tokens

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



(a) Input Tokens (b) Output Tokens

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).