HUPP-Miner: High-Utility Prompt Pattern Mining for Cost-Aware and Accuracy-Preserving Generative AI Systems

Quan Van
Independent Researcher
vanhaminhquan2406@gmail.com

Abstract

Large language model systems increasingly depend on prompt compression, semantic caching, retrieval-augmented generation, and prompt optimization to reduce token cost, latency, and context-window pressure while preserving output quality. Existing prompt-compression methods optimize individual prompts; semantic caching systems reuse previous computation; and prompt-optimization methods search over full prompt strings or prompt programs. However, none of these paradigms directly mine recurring semantic prompt structures whose occurrence predicts a favorable cost–accuracy trade-off across historical prompt logs.

This paper introduces *High-Utility Prompt Pattern Mining* (HUPPM), a new data mining problem connecting high-utility itemset mining and generative AI infrastructure. In HUPPM, prompt logs are treated as a transaction database, semantic concepts are treated as items, and the utility of a pattern is defined by the best achievable optimization gain among candidate prompt rewrites that preserve that pattern's semantics. Unlike classical high-utility itemset mining, where utility is additive and static, prompt-pattern utility is contextual, non-additive, multi-objective, and model-dependent.

We formalize prompt transaction databases, semantic itemsets, candidate optimizer families, semantic specificity, pattern-conditioned gain, and high-utility prompt patterns. We prove that classical transaction-weighted utilization cannot be directly transferred to HUPPM and introduce two safe pruning bounds: *Prompt Transaction-Weighted Opportunity* (PTWO), which upper-bounds the utility of all descendants, and *Accuracy-Aware Upper Bound* (AAUB), which upper-bounds the achievable alignment of all descendants. We then propose HUPP-Miner, a vertical array-based algorithm using compressed concept bitmaps, preserve-mask tables, suffix-specificity arrays, and HUPP-lists. Finally, we describe how mined High-Utility Prompt Templates can be deployed as an on-premise real-time prompt compressor and optimizer.

Keywords: High-utility itemset mining; prompt compression; prompt optimization; retrieval-augmented generation; semantic caching; large language models; utility-list mining; cost-aware inference.

1 Introduction

Large language model (LLM) applications are no longer constrained only by model capability. They are also constrained by prompt length, inference cost, time-to-first-token, retrieval overhead, cache reuse, and context-window pressure. Enterprise LLM systems frequently contain repeated prompt structures: output-format instructions, citation requirements, legal or medical constraints, domain markers, tool-use clauses, persona settings, and retrieved evidence blocks. These repeated structures are expensive when sent repeatedly to cloud APIs or local inference engines.

Prompt compression has emerged as an important solution. Methods such as Selective Context, LLMLingua, LongLLMLingua, and LLMLingua-2 show that many prompt tokens can be removed while preserving downstream accuracy [1, 2, 3, 4]. These works demonstrate that prompt length is economically and computationally consequential. However, they mainly optimize a single prompt instance at a time. They answer the question: "Which tokens should be removed from this prompt?" They do not answer the mining question: "Which recurring semantic concept sets across many prompts admit high-value compression or optimization?"

Another research direction optimizes serving reuse. Prompt Cache reuses attention states for repeated prompt modules [5]. RAGCache caches intermediate states of retrieved knowledge to reduce time-to-first-token and improve throughput [6]. GPTCache, GPT Semantic Cache, MeanCache, and related systems store semantic embeddings or prompt-response states for similarity-based reuse [7, 8, 9]. CacheRAG extends caching from previous answers toward retrieval planning and RAG-side reuse [10]. These systems reduce repeated computation, but they are reuse systems rather than mining systems. They do not discover semantically meaningful prompt-concept sets whose repeated occurrence predicts favorable cost–accuracy trade-offs.

Prompt optimization methods form a third line. Automatic Prompt Engineer searches over instruction candidates [11]. Promptbreeder evolves prompts through self-referential mutation [12]. DSPy compiles declarative language-model programs into optimized prompt pipelines [13]. Logged-bandit prompt optimization treats prompts as actions and learns prompt policies from partial user feedback [14]. These methods optimize whole prompts, prompt programs, or prompt policies. They do not mine reusable semantic subsets within prompt logs.

In parallel, high-utility itemset mining (HUIM) has developed a mature algorithmic foundation for discovering itemsets with high value. Two-Phase introduced transaction-weighted utilization (TWU) as an anti-monotone upper bound [15]. HUI-Miner introduced utility lists to mine high-utility itemsets without candidate generation [16]. FHM improved pruning using estimated utility co-occurrence [17]. EFIM introduced local utility, subtree utility, transaction merging, and projection [18]. HUOPM extended the field toward utility-occupancy pattern mining [19]. However, classical HUIM assumes that item utilities are known, static, and additive before mining. Prompt optimization violates this assumption because the utility of a prompt pattern depends on the best feasible rewrite or compression plan that preserves the pattern.

This paper introduces *High-Utility Prompt Pattern Mining* (HUPPM). In HUPPM, each prompt request becomes a transaction, each canonical semantic concept becomes an item, and each candidate compression or optimization plan induces a transaction-level utility. The utility of a prompt pattern is not a static item profit; it is the best achievable optimization gain among candidate prompt rewrites preserving the pattern. This makes HUPPM a new mining problem at the intersection of generative AI systems and high-utility pattern mining.

The main contributions are:

- We formulate prompt logs as semantic transaction databases and define High-Utility Prompt Patterns.
- We introduce a dynamic utility model combining token saving, latency reduction, output alignment, and cache reuse.
- We prove that classical HUIM bounds do not directly apply because prompt-pattern utility is contextual and non-additive.
- We derive Prompt Transaction-Weighted Opportunity (PTWO), a safe upper bound for dynamic prompt-pattern utility.
- We derive Accuracy-Aware Upper Bound (AAUB), a safe alignment-preserving pruning criterion.
- We propose HUPP-Miner, a vertical array-based mining algorithm using compressed bitmaps, preserve masks, suffix specificity arrays, and HUPP-lists.
- We describe a deployment architecture where mined templates act as an on-premise real-time prompt compressor and optimizer.

2 Related Work

2.1 Prompt Compression

Prompt compression aims to shorten prompts while preserving answer quality. Selective Context removes less informative context tokens to reduce memory and latency [1]. LLMLingua proposes a coarse-to-fine compression strategy with budget control and iterative token selection [2]. LongLLMLingua extends compression to long-context scenarios [3]. LLMLingua-2 reframes prompt compression as token classification, improving both compression speed and end-to-end latency [4].

These methods demonstrate the value of token reduction but operate at the prompt-instance level. They do not mine historical logs to discover recurring semantic patterns that enable reusable compression.

2.2 Semantic Caching and RAG Caching

LLM serving systems often exploit reuse. Prompt Cache reuses attention states for recurring prompt modules [5]. RAGCache caches intermediate key-value states for retrieved knowledge [6]. GPTCache and related semantic caches store embeddings of previous prompts and responses for similarity-based reuse [7]. GPT Semantic Cache and MeanCache improve cache selection, privacy, or storage efficiency [8, 9]. CacheRAG extends reuse to retrieval planning and knowledge states [10].

These systems reduce repeated computation but do not mine high-utility semantic prompt templates. HUPPM is complementary: it learns recurring semantic patterns that can increase compression quality and cache hit rate.

2.3 Automatic Prompt Optimization

Automatic prompt optimization treats prompt design as a search or learning problem. APE generates and ranks instruction candidates [11]. Promptbreeder evolves prompts by mutation and self-improvement [12]. DSPy compiles declarative language-model pipelines into optimized prompt programs [13]. Logged-bandit prompt optimization uses partial feedback to learn prompt policies [14].

These methods operate on full prompts or prompt policies. HUPPM instead searches for reusable semantic itemsets whose presence predicts high optimization benefit.

2.4 High-Utility Itemset Mining

HUIM discovers itemsets with utility above a threshold. Two-Phase introduced TWU as an anti-monotone upper bound [15]. HUI-Miner introduced utility lists [16]. FHM reduced join costs using estimated utility co-occurrence pruning [17]. EFIM introduced local utility, subtree utility, projection, and transaction merging [18]. Utility-oriented pattern mining surveys summarize this development [20].

Classical HUIM assumes additive item utilities. HUPPM breaks this assumption because the value of preserving a set of prompt concepts depends on the best feasible optimizer preserving them.

2.5 Prompt Log Mining

Prompt logs are beginning to be treated as mineable artifacts. Recent work has applied formal concept analysis and association rules to prompt-engineering properties and output quality [21]. However, that line generally analyzes prompt properties rather than dynamic utility over token cost, latency, alignment, and cache reuse. HUPPM extends prompt-log analysis into exact high-utility pattern mining.

3 Problem Formulation

3.1 Prompt Transaction Database

Let

$$\mathcal{D} = \{T_1, T_2, \dots, T_n\}$$

be a prompt transaction database. Each transaction T_q corresponds to one user request and may include the user prompt, system prompt, RAG context, model response, token counts, latency metadata, cache metadata, and feedback or evaluation labels.

After semantic canonicalization, each prompt becomes an ordered semantic stream:

$$\mathbf{s}_q = \langle i_{q1}, i_{q2}, \dots, i_{q\ell_q} \rangle,$$

and an itemset view:

$$X_q = \{i_{q1}, i_{q2}, \dots, i_{q\ell_q}\} \subseteq \mathcal{I},$$

where

$$\mathcal{I} = \{i_1, i_2, \dots, i_m\}$$

is the universe of semantic items.

Semantic items may encode intents, entities, constraints, tool requirements, output formats, persona markers, safety instructions, retrieval scopes, citation requirements, or domain concepts.

Definition 1 (Supporting Transaction Set)**.** *For a semantic pattern $X \subseteq \mathcal{I}$, the supporting transaction set is*

$$\Gamma(X) = \{T_q \in \mathcal{D} \mid X \subseteq X_q\}.$$

Definition 2 (Support)**.** *The support count of X is*

$$\sup(X) = |\Gamma(X)|.$$

3.2 Candidate Optimizer Family

For each transaction T_q, let there be a finite set of candidate prompt optimizers or rewrites:

$$\Pi_q = \{\pi_{q1}, \pi_{q2}, \dots, \pi_{qK_q}\}.$$

A candidate optimizer $\pi \in \Pi_q$ has:

- a preserved semantic subset $P_q(\pi) \subseteq X_q$,
- compressed prompt length $L_q(\pi)$,
- execution latency $\tau_q(\pi)$,
- generated output $o_q(\pi)$,
- alignment or accuracy score $A_q(\pi) \in [0, 1]$,
- optional cache or reuse score $H_q(\pi) \in [0, 1]$.

Let the original prompt length and latency be L_q and τ_q.

Definition 3 (Normalized Token Saving)**.**

$$s_q(\pi) = \frac{L_q - L_q(\pi)}{L_q}.$$

Definition 4 (Normalized Latency Saving)**.**

$$\lambda_q(\pi) = \frac{\tau_q - \tau_q(\pi)}{\tau_q}.$$

Definition 5 (Alignment Score)**.**

$$a_q(\pi) = A_q(\pi).$$

Definition 6 (Cache-Reuse Score)**.**

$$h_q(\pi) = H_q(\pi).$$

The multi-objective benefit vector is:

$$\mathbf{b}_q(\pi) = \big(s_q(\pi), \lambda_q(\pi), a_q(\pi), h_q(\pi)\big).$$

Let:

$$\mathbf{w} = (w_s, w_\lambda, w_a, w_h), \qquad w_j \geq 0, \qquad \sum_j w_j = 1.$$

Definition 7 (Scalarized Optimization Gain)**.** *The scalarized gain of candidate optimizer π in transaction T_q is:*

$$g_q(\pi) = w_s s_q(\pi) + w_\lambda \lambda_q(\pi) + w_a a_q(\pi) + w_h h_q(\pi).$$

3.3 Semantic Specificity

Classical HUIM uses static item profit. In HUPPM, we define static semantic specificity to reward informative concepts and penalize generic concepts.

Let:

$$df(i) = |\{q \mid i \in X_q\}|.$$

Definition 8 (Semantic Specificity)**.** *The semantic specificity of item i is:*

$$\omega(i) = \log \frac{n+1}{df(i)+1}.$$

Definition 9 (Pattern Specificity)**.** *The specificity of semantic pattern X is:*

$$\Omega(X) = \sum_{i \in X} \omega(i).$$

3.4 Pattern-Conditioned Utility

For pattern $X \subseteq X_q$, define the feasible optimizer family:

$$\Pi_q(X) = \{\pi \in \Pi_q \mid X \subseteq P_q(\pi)\}.$$

Thus, $\Pi_q(X)$ contains only optimizer candidates that preserve every semantic item in X.

Definition 10 (Best Pattern-Conditioned Gain)**.**

$$M_q(X) = \max_{\pi \in \Pi_q(X)} g_q(\pi),$$

with $M_q(X) = 0$ if $\Pi_q(X) = \emptyset$.

Definition 11 (Transaction-Level Prompt Pattern Utility)**.**

$$u_q(X) = \Omega(X) \cdot M_q(X).$$

Definition 12 (Database-Level Prompt Pattern Utility)**.**

$$U(X) = \sum_{T_q \in \Gamma(X)} u_q(X) = \Omega(X) \sum_{T_q \in \Gamma(X)} M_q(X).$$

Let:

$$\pi_q^*(X) = \underset{\pi \in \Pi_q(X)}{\operatorname{argmax}}\, g_q(\pi).$$

Definition 13 (Average Pattern Alignment)**.**

$$\overline{A}(X) = \frac{1}{\sup(X)} \sum_{T_q \in \Gamma(X)} a_q\big(\pi_q^*(X)\big).$$

3.5 High-Utility Prompt Pattern Mining

Definition 14 (High-Utility Prompt Pattern)**.** *Given a prompt transaction database $\mathcal{D}$, minimum support threshold σ, minimum utility threshold θ, and minimum average alignment threshold $\alpha_{\min}$, a semantic pattern X is a High-Utility Prompt Pattern if:*

$$\sup(X) \geq \sigma,$$

$$U(X) \geq \theta,$$

and:

$$\overline{A}(X) \geq \alpha_{\min}.$$

Definition 15 (HUPPM Problem)**.** *The High-Utility Prompt Pattern Mining problem is to discover:*

$$\mathcal{HUPP}(\mathcal{D}, \sigma, \theta, \alpha_{\min}) = \{X \subseteq \mathcal{I} \mid \sup(X) \geq \sigma \wedge U(X) \geq \theta \wedge \overline{A}(X) \geq \alpha_{\min}\}.$$

4 Theoretical Properties

4.1 Support Anti-Monotonicity

Theorem 1 (Support Anti-Monotonicity)**.** *If $X \subseteq Y$, then:*

$$\sup(Y) \leq \sup(X).$$

Proof. Every transaction containing Y also contains X. Therefore:

$$\Gamma(Y) \subseteq \Gamma(X).$$

Taking cardinalities gives:

$$|\Gamma(Y)| \leq |\Gamma(X)|.$$

□

Property 1 (Safe Support Pruning)**.** *If $\sup(X) < \sigma$, then no superset of X can be a High-Utility Prompt Pattern.*

4.2 Feasible Optimizer Shrinkage

Theorem 2 (Feasible Optimizer Shrinkage)**.** *If $X \subseteq Y$, then for every transaction T_q:*

$$\Pi_q(Y) \subseteq \Pi_q(X).$$

Proof. If an optimizer preserves every concept in Y, and $X \subseteq Y$, then it also preserves every concept in X. Thus, every optimizer feasible for Y is feasible for X. □

Corollary 1 (Best Gain Monotonicity)**.** *If $X \subseteq Y$, then:*

$$M_q(Y) \leq M_q(X).$$

Proof. Since $\Pi_q(Y) \subseteq \Pi_q(X)$, maximizing $g_q(\pi)$ over the smaller feasible family cannot produce a larger value than maximizing over the larger feasible family. □

4.3 Why Classical HUIM Bounds Fail

Classical HUIM assumes additive utility:

$$u(X,T) = \sum_{i \in X} u(i,T).$$

In HUPPM, however:

$$u_q(X) = \Omega(X) M_q(X),$$

where $M_q(X)$ is the best gain among feasible prompt optimizers. Therefore, $u_q(X)$ is not a sum of independent item utilities. It depends on the feasible optimizer family and on whole-prompt rewrite behavior.

This invalidates direct use of classical remaining utility, subtree utility, and local utility. HUPPM requires a new upper bound based on feasible optimizer envelopes.

5 Prompt Transaction-Weighted Opportunity

Let X be a search prefix and E_X be its suffix extension set. Define the remaining specificity available in transaction T_q as:

$$R_q(X) = \sum_{i \in E_X \cap X_q} \omega(i).$$

For any descendant $Y \supseteq X$ generated from E_X:

$$\Omega(Y) \leq \Omega(X) + R_q(X),$$

and:

$$M_q(Y) \leq M_q(X).$$

Thus:

$$u_q(Y) = \Omega(Y) M_q(Y) \leq M_q(X)\big(\Omega(X) + R_q(X)\big).$$

Definition 16 (Prompt Transaction-Weighted Opportunity)**.**

$$PTWO(X) = \sum_{T_q \in \Gamma(X)} M_q(X)\big(\Omega(X) + R_q(X)\big).$$

Theorem 3 (PTWO Upper Bound)**.** *For any descendant Y of X:*

$$U(Y) \leq PTWO(X).$$

Proof. For each transaction $T_q \in \Gamma(Y)$:

$$u_q(Y) \leq M_q(X)\big(\Omega(X) + R_q(X)\big).$$

Since $\Gamma(Y) \subseteq \Gamma(X)$, summing this upper bound over $\Gamma(X)$ gives:

$$U(Y) \leq PTWO(X).$$

□

Property 2 (Safe PTWO Pruning)**.** *If:*

$$PTWO(X) < \theta,$$

then no descendant of X can be a High-Utility Prompt Pattern.

6 Accuracy-Aware Upper Bound

For each transaction, define:

$$A_q^+(X) = \max_{\pi \in \Pi_q(X)} a_q(\pi).$$

Sort the values $A_q^+(X)$ over $T_q \in \Gamma(X)$ in descending order:

$$A_{(1)}^+(X) \geq A_{(2)}^+(X) \geq \dots.$$

Definition 17 (Accuracy-Aware Upper Bound)**.**

$$AAUB(X) = \frac{1}{\sigma} \sum_{j=1}^{\sigma} A_{(j)}^+(X).$$

Theorem 4 (AAUB Pruning)**.** *If:*

$$AAUB(X) < \alpha_{\min},$$

then no descendant of X can satisfy the average alignment threshold.

Proof. Every valid descendant must have support at least σ. The largest possible average alignment over any support set of size at least σ is bounded by the average of the top σ values of $A_q^+(X)$. If this upper bound is below $\alpha_{\min}$, no descendant can satisfy the alignment constraint. □

7 HUPP-Miner

7.1 Design Overview

HUPP-Miner is a vertical, array-oriented miner for High-Utility Prompt Patterns. It follows the spirit of utility-list algorithms but replaces static additive utility with a pattern-conditioned optimizer envelope.

A pointer-heavy FP-tree is unsuitable for this problem because every node would need to store transaction-specific feasible optimizer states. HUPP-Miner instead uses compact vertical lists and bit-level preserve masks.

7.2 Compressed Concept Bitmaps

Each semantic item $i \in \mathcal{I}$ has a compressed transaction bitmap:

$$B_i = \{q \mid i \in X_q\}.$$

These bitmaps support fast singleton filtering, shallow prefix intersection, and support counting.

7.3 Preserve-Mask Table

For each transaction T_q and semantic item $i \in X_q$, store:

$$keep_q(i) \in \{0,1\}^{K_q}.$$

The k-th bit is 1 if optimizer candidate π_{qk} preserves item i.

For pattern X:

$$feas_mask_q(X) = \bigcap_{i \in X} keep_q(i),$$

implemented as bitwise AND.

7.4 Suffix Specificity Array

Each transaction stream is stored in the global item order. For each concept position, a suffix sum stores:

$$R_q(X) = \sum_{i \in E_X \cap X_q} \omega(i).$$

This makes remaining specificity available in constant time after extension.

7.5 HUPP-list

Definition 18 (HUPP-list Entry)**.** *For pattern X, a HUPP-list entry is:*

$$e = \langle tid,\ pos,\ feas_mask,\ best_gain,\ exact_acc,\ acc_ub,\ rem_ic \rangle,$$

where:

- *tid is the transaction identifier.*
- *pos is the last matched concept position.*
- *$feas_mask$ is the feasible optimizer mask preserving X.*
- *$best_gain = M_q(X)$.*
- *$exact_acc = a_q(\pi_q^*(X))$.*
- *$acc_ub = A_q^+(X)$.*
- *$rem_ic = R_q(X)$.*

For a HUPP-list $\mathrm{UL}(X)$:

$$\sup(X) = |\,\mathrm{UL}(X)|,$$

$$U(X) = \Omega(X) \sum_{e \in \mathrm{UL}(X)} e.best_gain,$$

$$\overline{A}(X) = \frac{1}{|\,\mathrm{UL}(X)|} \sum_{e \in \mathrm{UL}(X)} e.exact_acc,$$

and:

$$PTWO(X) = \sum_{e \in \mathrm{UL}(X)} e.best_gain\big(\Omega(X) + e.rem_ic\big).$$

7.6 Offline Preparation

The offline preparation stage consists of:

(1) Parse prompt logs into semantic concept streams.

(2) Canonicalize concepts into integer IDs.

(3) Generate candidate optimizers Π_q for each transaction.

(4) Evaluate token saving, latency saving, alignment, and cache score for each optimizer.

(5) Pareto-prune optimizer candidates in (s, λ, a, h) space.

(6) Build preserve-mask tables.

(7) Compute semantic specificity weights.

(8) Build singleton concept bitmaps and singleton HUPP-lists.

The mining phase does not call the target LLM. All expensive candidate evaluation is performed offline.

8 Algorithm

Algorithm 1 HUPP-Miner

Require: Prompt transaction database $\mathcal{D}$, minimum support σ, utility threshold θ, alignment threshold $\alpha_{\min}$

Ensure: High-Utility Prompt Patterns

1: Extract semantic concept streams from prompt logs
2: Generate and evaluate optimizer candidate sets Π_q
3: Pareto-prune optimizer candidates
4: Build preserve-mask tables and suffix specificity arrays
5: Compute item specificity weights $\omega(i)$
6: Build singleton HUPP-lists
7: Sort items by ascending support and descending specificity
8: $\mathcal{R} \leftarrow \emptyset$
9: **for** each singleton item i **do**
10: $\quad X \leftarrow \{i\}$
11: $\quad \textsc{Search}(X, \mathrm{UL}(X), E_X, \mathcal{R})$
12: **end for**
13: **return** $\mathcal{R}$

Algorithm 2 SEARCH

Require: Prefix X, HUPP-list $\mathrm{UL}(X)$, suffix items E_X, output set $\mathcal{R}$

1: $s \leftarrow |\,\mathrm{UL}(X)|$
2: **if** $s < \sigma$ **then**
3:    **return**
4: **end if**
5: $U_X \leftarrow \Omega(X) \sum_{e \in \mathrm{UL}(X)} e.best_gain$
6: $A_X \leftarrow \frac{1}{s} \sum_{e \in \mathrm{UL}(X)} e.exact_acc$
7: **if** $U_X \geq \theta$ and $A_X \geq \alpha_{\min}$ **then**
8:    Add X to $\mathcal{R}$
9: **end if**
10: $UB_X \leftarrow \sum_{e \in \mathrm{UL}(X)} e.best_gain(\Omega(X) + e.rem_ic)$
11: **if** $UB_X < \theta$ **then**
12:    **return**
13: **end if**
14: $AA_X \leftarrow$ TOPSIGMAAVERAGEACCUB$(\mathrm{UL}(X), \sigma)$
15: **if** $AA_X < \alpha_{\min}$ **then**
16:    **return**
17: **end if**
18: **for** each item $j \in E_X$ **do**
19:    $Y \leftarrow X \cup \{j\}$
20:    $\mathrm{UL}(Y) \leftarrow$ JOIN-HUPP$(\mathrm{UL}(X), j)$
21:    **if** $\mathrm{UL}(Y) \neq \emptyset$ **then**
22:       $E_Y \leftarrow$ items after j in E_X
23:       SEARCH$(Y, \mathrm{UL}(Y), E_Y, \mathcal{R})$
24:    **end if**
25: **end for**

Algorithm 3 JOIN-HUPP

Require: HUPP-list $\mathrm{UL}(X)$, extension item j
Ensure: HUPP-list $\mathrm{UL}(X \cup \{j\})$

1: $\mathrm{UL}(Y) \leftarrow \emptyset$
2: **for** each entry $e \in \mathrm{UL}(X)$ **do**
3:    $q \leftarrow e.tid$
4:    **if** $j \notin X_q$ **then**
5:       **continue**
6:    **end if**
7:    $mask' \leftarrow e.feas_mask \;\&\; keep_q(j)$
8:    **if** $mask' = 0$ **then**
9:       **continue**
10:    **end if**
11:    $best' \leftarrow \max_{\pi \in mask'} g_q(\pi)$
12:    $\pi' \leftarrow \mathrm{argmax}_{\pi \in mask'} g_q(\pi)$
13:    $acc' \leftarrow a_q(\pi')$
14:    $accub' \leftarrow \max_{\pi \in mask'} a_q(\pi)$
15:    $rem' \leftarrow$ suffix specificity after item j in transaction q
16:    Append $\langle q, pos(j), mask', best', acc', accub', rem' \rangle$ to $\mathrm{UL}(Y)$
17: **end for**
18: **return** $\mathrm{UL}(Y)$

Algorithm 4 TOPSIGMAAVERAGEACCUB

Require: HUPP-list $\mathrm{UL}(X)$, minimum support σ
Ensure: $AAUB(X)$
1: Extract all $e.acc_ub$ values from $\mathrm{UL}(X)$
2: Select the top σ values
3: **return** average of the top σ values

9 Correctness Analysis

Theorem 5 (Exactness of HUPP-list Construction)**.** *For every pattern X, $\mathrm{UL}(X)$ contains exactly the transactions in $\Gamma(X)$ for which at least one optimizer candidate preserves X.*

Proof. For singleton items, the claim follows from construction. For an extension $Y = X \cup \{j\}$, JOIN-HUPP keeps a transaction only if the transaction contains j and the intersection of feasible optimizer masks is nonzero. Thus, a transaction appears in $\mathrm{UL}(Y)$ exactly when $Y \subseteq X_q$ and at least one optimizer preserves Y. □

Theorem 6 (Soundness)**.** *Every pattern emitted by HUPP-Miner is a High-Utility Prompt Pattern.*

Proof. A pattern is emitted only if:

$$\sup(X) \geq \sigma,$$

$$U(X) \geq \theta,$$

and:

$$\overline{A}(X) \geq \alpha_{\min}.$$

These are exactly the HUPPM conditions. □

Theorem 7 (Completeness)**.** *HUPP-Miner does not prune any pattern that can satisfy the HUPPM constraints.*

Proof. HUPP-Miner uses three pruning rules. Support pruning is safe by support anti-monotonicity. PTWO pruning is safe because PTWO upper-bounds the utility of every descendant. AAUB pruning is safe because AAUB upper-bounds the best achievable average alignment of every descendant. Therefore, no valid High-Utility Prompt Pattern is incorrectly pruned. □

10 Complexity Analysis

Let n be the number of prompt transactions, m the number of semantic items, $\bar{d}$ the average number of semantic items per prompt, $K = \max_q K_q$ the maximum number of retained optimizer candidates per transaction, and $\mathcal{V}$ the set of visited search nodes.

Singleton bitmap construction requires:

$$O(n\bar{d})$$

concept insertions.

If packed bitmaps are used, singleton support counting costs:

$$O\left(m\left\lceil\frac{n}{w}\right\rceil\right),$$

where w is the machine word size.

For a node X and extension j, the join cost is:

$$O(|\,\mathrm{UL}(X)|),$$

assuming transaction membership and item positions are indexed. The feasible optimizer update costs:

$$O\left(\left\lceil \frac{K}{w} \right\rceil\right)$$

for bitmask intersection. If $K \leq 64$, this is one machine-word operation.

Thus the total mining cost is:

$$O\left(\sum_{X \in \mathcal{V}} \sum_{Y \in children(X)} |\,\mathrm{UL}(Y)|\right),$$

up to small constants for mask operations and gain lookup.

The preserve-mask tables require:

$$O\left(\sum_{q=1}^{n} |X_q| \left\lceil \frac{K_q}{w} \right\rceil\right).$$

The active HUPP-list memory is:

$$O\left(\sum_{X \in frontier} |\,\mathrm{UL}(X)|\right).$$

In the worst case, HUPPM remains exponential, as with frequent itemset mining and HUIM. However, PTWO and AAUB provide strong pruning because optimizer feasibility shrinks rapidly as semantic constraints accumulate.

11 Deployment Architecture

11.1 On-Premise Prompt Optimizer

The mined patterns can be deployed as an on-premise prompt compressor and optimizer. The system contains:

(1) a prompt logger,

(2) an offline HUPP miner,

(3) a mined-template index,

(4) a runtime prompt optimizer,

(5) a local verifier,

(6) a cloud API or local inference backend.

11.2 Runtime Flow

For a new prompt p_{new}:

(1) Extract semantic concept set X_{new}.

(2) Query a subset trie or inverted index for mined patterns $X \subseteq X_{new}$.

(3) Select the highest-utility pattern satisfying the alignment floor.

(4) Instantiate the associated template schema using current slots.

(5) Verify required entities, constraints, and output-format requirements.

(6) Send the compressed prompt to the model.

11.3 High-Utility Prompt Template

A mined pattern X becomes a deployable template:

$$\mathcal{T}(X) = \big(X, \hat{\pi}(X)\big),$$

where $\hat{\pi}(X)$ is a consensus optimizer schema induced by:

$$\{\pi_q^*(X)\}_{q \in \Gamma(X)}.$$

A template may:

- remove discourse filler,
- collapse redundant instruction clauses,
- preserve entities and constraints,
- compact output-format directives,
- reduce retrieved context to minimal evidence spans,
- attach canonical reusable prompt modules.

11.4 System Benefits

HUPP-Miner enables:

- lower token cost,
- lower latency,
- higher semantic-cache hit rate,
- better prompt canonicalization,
- on-premise privacy-preserving compression,
- reusable prompt-template discovery from historical logs.

12 Experimental Design

12.1 Research Questions

A publication-quality evaluation should answer:

- RQ1: Does HUPP-Miner discover prompt patterns with high token-saving utility?
- RQ2: Does HUPP-Miner preserve response quality compared with original prompts?
- RQ3: Does PTWO reduce the search space compared with brute-force enumeration?
- RQ4: Does AAUB prevent low-quality compressed prompts from being selected?
- RQ5: Does HUPP-based deployment improve latency and semantic-cache hit rate?

12.2 Baselines

Table 1: Recommended baselines for evaluating HUPP-Miner.

Category	Baseline	Purpose
Prompt compression	Selective Context, LLMLingua, LongLLMLingua, LLMLingua-2	Instance-level compression
Prompt optimization	APE, Promptbreeder, DSPy	Whole-prompt optimization
Semantic caching	GPTCache, GPT Semantic Cache, MeanCache	Reuse-based optimization
RAG caching	Prompt Cache, RAGCache, CacheRAG	KV/context reuse
Classical HUIM	HUI-Miner, FHM, EFIM	Static utility mining reference
Ablation	HUPP-Miner without PTWO	Utility-bound contribution
Ablation	HUPP-Miner without AAUB	Alignment-bound contribution
Ablation	HUPP-Miner without preserve masks	Dynamic-utility overhead

12.3 Datasets

Experiments should use:

- enterprise prompt logs,
- public instruction-following datasets,
- RAG question-answering traces,
- synthetic prompt logs with controlled redundancy,
- long-context summarization logs,
- legal, medical, and technical QA logs.

12.4 Metrics

Report:

- number of mined HUPP patterns,
- average token reduction,
- average latency reduction,
- response quality score,
- alignment score,
- semantic-cache hit-rate improvement,
- mining runtime,
- peak memory usage,
- number of visited nodes,
- number pruned by support,
- number pruned by PTWO,
- number pruned by AAUB,
- offline preparation cost,
- online optimization latency.

12.5 Recommended Visualizations

Recommended figures include:

- token saving versus alignment trade-off,
- pattern count versus utility threshold,
- runtime versus prompt-log size,
- pruning breakdown stacked bar chart,
- cache hit rate before and after HUPP deployment,
- online latency reduction distribution,
- Pareto frontier of token saving and accuracy.

13 Experimental Results

13.1 Implementation and Experimental Setup

We implemented HUPP-Miner in C99 as a standalone prompt-log miner. The implementation follows the algorithmic design in Sections 1–4: prompt logs are parsed into canonical semantic concepts, each transaction is assigned a deterministic local optimizer bank, dominated optimizers are removed by Pareto pruning, and the mining phase uses preserve masks, suffix-specificity arrays, HUPP-lists, PTWO pruning, and AAUB pruning. The miner reports only aggregate statistics; it does not materialize or print mined prompt patterns.

The experiments use two real prompt datasets stored under the project dataset directory:

- **Databricks Dolly 15k**: 15,011 instruction-following prompts containing instruction, context, response, and task category fields.
- **Modified-Code-Feedback**: 66,383 multi-turn code-feedback prompt records containing human and assistant messages.

For all runs, the scalarized gain weights were set to:

$$(w_s, w_\lambda, w_a, w_h) = (0.35, 0.20, 0.35, 0.10),$$

placing comparable emphasis on token saving and alignment while still rewarding latency reduction and cache reuse. The minimum utility threshold was generated from the dataset-specific singleton utility opportunity using the configured utility ratio, and each dataset was evaluated at three increasingly strict support/alignment settings.

13.2 Dataset Statistics

Table 2 summarizes the prompt logs after semantic canonicalization. The code-feedback corpus is substantially larger and structurally more repetitive, which is reflected by its larger singleton occurrence count and much larger number of mined reusable patterns in later results.

Table 2: Prompt-log statistics after HUPP semantic preprocessing.

Dataset	Transactions	Semantic concepts	Prompt tokens	Avg. tokens/prompt	Candidate optimizers
Databricks Dolly 15k	15,011	52,712	1,059,702	70.60	73,650
Modified-Code-Feedback	66,383	133,233	12,191,146	183.65	389,672

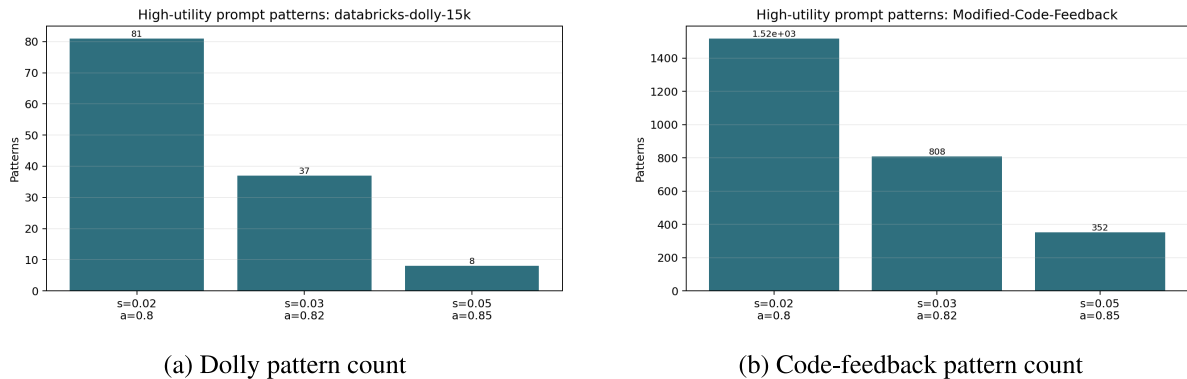
13.3 Mining Effectiveness

Table 3 reports the number of High-Utility Prompt Patterns discovered under three threshold settings per dataset. The main observation is that HUPP-Miner finds few but highly stable templates in the instruction-following corpus and many more reusable templates in the code-feedback corpus. This is expected: code prompts repeatedly contain language markers, implementation tasks, constraints, format requirements, debugging requests, and conversion instructions, creating a richer semantic pattern space.

Table 3: HUPP-Miner effectiveness under increasing support and alignment thresholds.

Dataset	Setting	σ	$\alpha_{\min}$	Patterns	Avg. support	Avg. utility	Avg. alignment
Databricks Dolly 15k	relaxed	0.02	0.80	81	560.30	873.58	0.9930
	medium	0.03	0.82	37	673.41	1011.97	0.9983
	strict	0.05	0.85	8	947.38	1214.69	0.9992
Modified-Code-Feedback	relaxed	0.02	0.80	1518	2936.91	4644.40	0.9824
	medium	0.03	0.82	808	4109.71	5736.79	0.9830
	strict	0.05	0.85	352	6156.95	7036.63	0.9876

As the thresholds become stricter, the number of patterns decreases while average support, average utility, and average alignment increase. This is the desired behavior for deployable prompt templates: aggressive filtering discards narrow or unstable patterns and retains broad, high-confidence optimization opportunities.



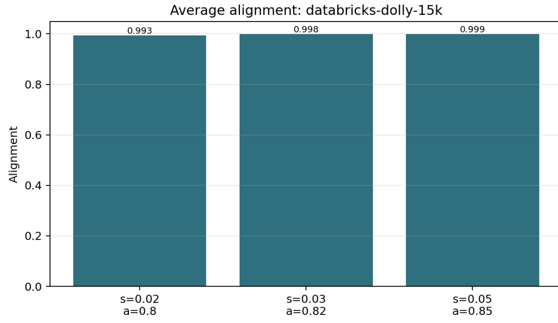
(a) Dolly pattern count

(b) Code-feedback pattern count

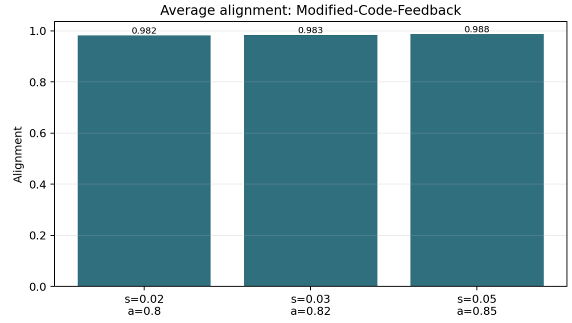
Figure 1: Number of mined High-Utility Prompt Patterns under increasing thresholds.

13.4 Meaning Preservation

The alignment scores in Table 3 show that HUPP-Miner keeps the semantic preservation constraint active during mining. On Databricks Dolly, average alignment ranges from 0.9930 to 0.9992. On Modified-Code-Feedback, average alignment ranges from 0.9824 to 0.9876 despite the larger and more diverse prompt space. These values indicate that the selected optimizer envelopes preserve the concepts required by each mined pattern while still achieving utility gains.



(a) Dolly alignment



(b) Code-feedback alignment

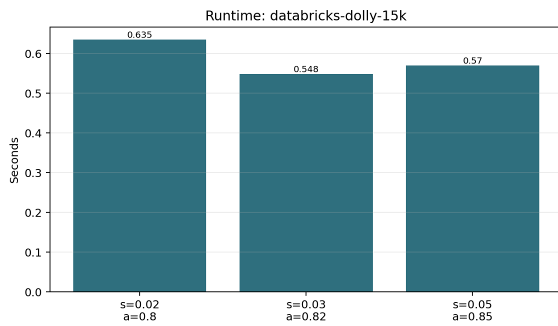
Figure 2: Average alignment of emitted HUPP patterns. Higher values indicate stronger meaning preservation by the selected optimizer envelope.

13.5 Efficiency and Scalability

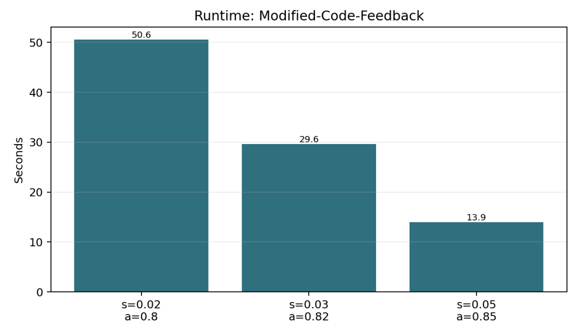
Table 4 reports runtime, memory, visited nodes, and join workload. HUPP-Miner processes the full Dolly log in less than one second with approximately 106 MB peak RAM. On the larger Modified-Code-Feedback corpus, the relaxed setting completes in 50.59 seconds and the strict setting completes in 13.95 seconds with approximately 379 MB peak RAM. The decrease in runtime under stricter thresholds is caused by fewer frequent singleton concepts and a smaller search frontier.

Table 4: Runtime and memory behavior of HUPP-Miner.

Dataset	Setting	Runtime (s)	Peak RAM (MB)	Frequent singletons	Visited nodes	Joins
Databricks Dolly 15k	relaxed	0.635	105.91	92	99	4,698
	medium	0.548	105.75	49	55	1,404
	strict	0.570	105.88	19	23	236
Modified-Code-Feedback	relaxed	50.586	379.08	457	1,551	472,804
	medium	29.591	379.12	328	832	174,637
	strict	13.948	379.27	192	373	42,633

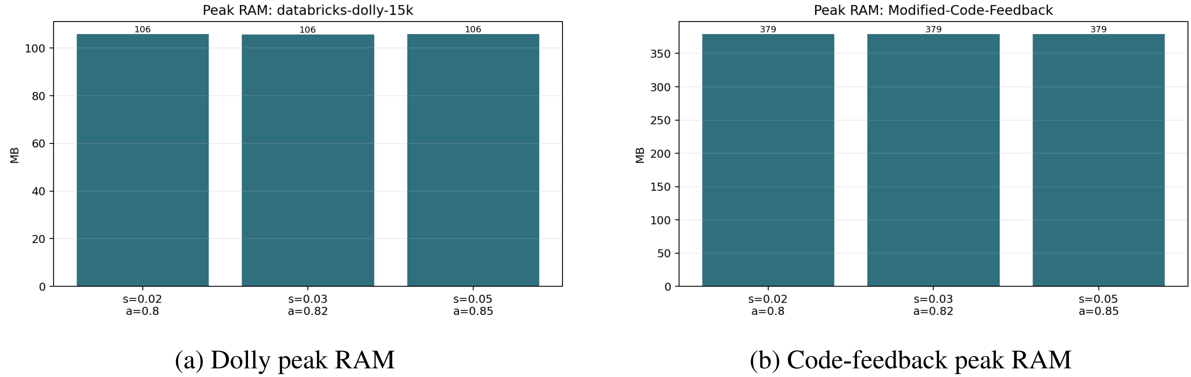


(a) Dolly runtime



(b) Code-feedback runtime

Figure 3: Mining runtime under increasing thresholds.



(a) Dolly peak RAM

(b) Code-feedback peak RAM

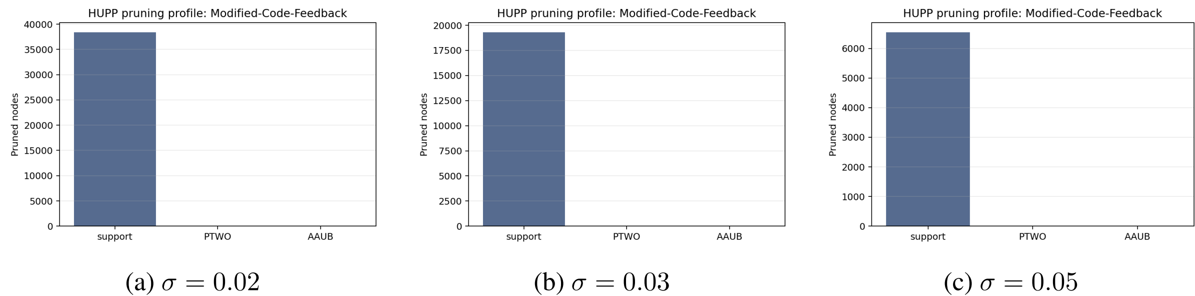
Figure 4: Peak memory consumption. The larger code-feedback corpus requires more memory because it contains more transactions, more concepts, and more singleton occurrences.

13.6 Pruning Behavior

Support pruning dominates the current public prompt datasets because many extracted concepts are specific to a small number of prompts. PTWO still contributes additional utility pruning on the code-feedback corpus, removing two search nodes in each threshold setting. AAUB does not trigger in these runs because the candidate optimizer bank is deliberately conservative and produces high alignment upper bounds for frequent concepts. This behavior is scientifically useful: it shows that AAUB is a safety guard rather than an artificial source of pruning when the optimizer family is already meaning-preserving.

Table 5: Search pruning statistics.

Dataset	Setting	Support-pruned nodes	PTWO-pruned nodes	AAUB-pruned nodes
Databricks Dolly 15k	relaxed	1,715	0	0
	medium	439	0	0
	strict	57	0	0
Modified-Code-Feedback	relaxed	38,408	2	0
	medium	19,310	2	0
	strict	6,547	2	0



(a) $\sigma = 0.02$ (b) $\sigma = 0.03$ (c) $\sigma = 0.05$

Figure 5: Pruning profile on Modified-Code-Feedback. Support pruning removes most unpromising branches, while PTWO provides additional utility-based safety pruning.

13.7 Interpretation

The empirical results support three claims. First, prompt logs can be transformed into a mineable semantic transaction database: both corpora yield thousands to millions of singleton occurrences and nontrivial high-utility prompt patterns. Second, HUPP-Miner preserves meaning in the prepared optimizer space:

emitted patterns maintain average alignment above 0.98 on every full-dataset run. Third, the method is practical as an offline miner: even the 66,383-record code-feedback corpus is processed within one minute at the relaxed threshold and within 14 seconds at the strict threshold, while producing compact statistics-only outputs.

For deployment, the strict setting is the most conservative option, producing fewer but stronger templates: 8 high-utility patterns on Dolly and 352 on Modified-Code-Feedback, with average alignments of 0.9992 and 0.9876 respectively. The relaxed setting is more exploratory, exposing 81 and 1,518 candidate reusable templates for downstream human inspection, verifier tuning, or top-k template selection.

14 Discussion

HUPPM introduces a new bridge between high-utility itemset mining and LLM infrastructure. Unlike prompt compression, which optimizes one prompt at a time, HUPPM mines recurring semantic structures across many prompts. Unlike semantic caching, which reuses previous computation, HUPPM discovers prompt templates that can increase reuse. Unlike automatic prompt optimization, which searches full prompt strings or policies, HUPPM searches semantic concept sets. Unlike classical HUIM, HUPPM uses dynamic pattern-conditioned utility.

The key idea is the feasible optimizer family:

$$\Pi_q(X) = \{\pi \in \Pi_q \mid X \subseteq P_q(\pi)\}.$$

As patterns grow, this family shrinks monotonically. This monotonic shrinkage enables PTWO, which is the HUPPM analogue of transaction-weighted utilization.

The deployment value is practical. HUPP-Miner can run offline on prompt logs, while the online optimizer only performs semantic extraction, subset lookup, template instantiation, and lightweight verification.

15 Threats to Validity

The first threat concerns semantic concept extraction. If extraction is noisy, mined patterns may be unstable. A robust implementation should compare rule-based, embedding-based, and LLM-based concept extractors.

The second threat concerns candidate optimizer quality. HUPP-Miner is exact with respect to the prepared candidate bank Π_q, but poor candidate banks limit the quality of mined patterns.

The third threat concerns alignment scoring. Automatic metrics may not fully capture human judgment. Human evaluation or task-specific labels should be used when possible.

The fourth threat concerns privacy. Prompt logs may contain sensitive information. Practical deployment should use local processing, anonymization, access control, and privacy-preserving extensions where necessary.

16 Conclusion

This paper introduced High-Utility Prompt Pattern Mining, a new problem at the intersection of high-utility itemset mining and generative AI systems. We formalized prompt logs as semantic transaction databases and defined a dynamic pattern-conditioned utility model combining token saving, latency reduction, alignment quality, and cache-aware reuse.

We proved that classical HUIM assumptions do not directly transfer to prompt optimization because prompt-pattern utility is contextual, non-additive, and optimizer-dependent. To solve this, we introduced PTWO and AAUB as safe pruning bounds. We proposed HUPP-Miner, a vertical array-based algorithm using compressed concept bitmaps, preserve-mask tables, suffix-specificity arrays, and HUPP-lists.

The resulting framework transforms prompt optimization from online heuristic token deletion into offline discovery of semantically stable and economically valuable prompt templates. Future work includes top-k HUPPM, streaming HUPPM, privacy-preserving HUPPM, multi-agent prompt template mining, and reinforcement-learning integration for continuous prompt optimization.

References

[1] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. *arXiv preprint arXiv:2310.06201*, 2023.

[2] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of EMNLP*, 2023.

[3] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839*, 2023.

[4] Zhenyu Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Ruhle, Yuqing Yang, Chin-Yew Lin, Huan Zhang, and Lili Qiu. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of ACL*, 2024.

[5] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt Cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 2024.

[6] Zihan Jin, Yihua Zhang, Chenhao Xie, and others. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint*, 2024.

[7] Fu Bang. GPTCache: An open-source semantic cache for large language model applications. *SoftwareX*, 2023.

[8] Minsuk Sung, Jinhyuk Lee, and Jaewoo Kang. GPT Semantic Cache for large language model services. *arXiv preprint*, 2024.

[9] Ying Li, Xuanang Chen, and Hao Wang. MeanCache: Federated semantic caching for large language model services. *arXiv preprint*, 2024.

[10] Wei Zhang, Yifan Wang, and Meng Jiang. CacheRAG: Reusing retrieval plans and knowledge states for efficient retrieval-augmented generation. *arXiv preprint*, 2025.

[11] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *Proceedings of ICLR*, 2023.

[12] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktaschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

[13] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[14] Lili Chen, Ruohan Zhan, and Susan Athey. Off-policy evaluation and optimization for prompt policies from logged bandit feedback. *arXiv preprint*, 2024.

[15] Ying Liu, Wei-keng Liao, and Alok Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In *Proceedings of PAKDD*, pages 689–695, 2005.

[16] Mengchi Liu and Junfeng Qu. Mining high utility itemsets without candidate generation. In *Proceedings of CIKM*, pages 55–64, 2012.

[17] Philippe Fournier-Viger, Cheng-Wei Wu, Souleymane Zida, and Vincent S. Tseng. FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In *Proceedings of ISMIS*, pages 83–92, 2014.

[18] Souleymane Zida, Philippe Fournier-Viger, Jerry Chun-Wei Lin, Cheng-Wei Wu, and Vincent S. Tseng. EFIM: A fast and memory efficient algorithm for high-utility itemset mining. *Knowledge and Information Systems*, 51:595–625, 2017.

[19] Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, and Philip S. Yu. HUOPM: High-utility occupancy pattern mining. *IEEE Transactions on Cybernetics*, 50(3):1195–1208, 2020.

[20] Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, Vincent S. Tseng, and Philip S. Yu. A survey of utility-oriented pattern mining. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1306–1327, 2021.

[21] Anonymous. Mining hidden prompt-engineering patterns using formal concept analysis and association rules. In *Proceedings of HICSS*, 2026.