ADAPTIVE CO-SERVING LLM WATERMARKING ON MODERN INFERENCE ENGINES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) watermarking is important for ownership verification and intellectual property protection. However, existing approaches focus on algorithmic design while treating LLM inference engines as separate components. This separation often introduces auxiliary models or external tools, increasing latency and memory overhead while limiting the use of modern inference optimizations. As a result, a deployment gap remains: practical watermarking must preserve utility, detectability, and robustness while minimizing serving overhead.

To bridge this gap, we propose SWIFT, a framework that co-designs LLM watermarking with modern inference infrastructure. SWIFT (i) integrates text generation and watermark construction within a shared LLM backend to reduce recomputation, improve cache reuse, and simplify system complexity; (ii) uses instruction-guided candidate generation with semantic protection to preserve factual consistency and meaning; (iii) executes watermarking concurrently with text generation through asynchronous co-serving and adaptive scheduling to reduce latency and improve resource utilization; and (iv) leverages vLLM optimizations to enhance serving efficiency. Extensive experiments across domains and tasks show that SWIFT achieves strong utility, detectability, robustness, and downstream performance while substantially improving serving efficiency. Specifically, it achieves the highest utility score (4.87), 99.65% detection accuracy compared with 76.65% for the low-latency baseline SynthID, and up to 97.7% detection under watermark removal attacks, with $5.9\times$ lower latency than the highest-utility baseline, SafeSeal. These results demonstrate the benefits of co-designing watermarking with inference infrastructure for practical LLM serving. Code and artifacts are available at: `https://anonymous.4open.science/r/SWIFT-0597`.

1 INTRODUCTION

As large language models (LLMs) (OpenAI, 2026; Anthropic, 2026; Google, 2026) become widely deployed across domains, their outputs can be copied, transformed, or redistributed without evidence of origin. This provenance gap makes it difficult to verify whether text was generated by a model or service, especially after it leaves the provider. LLM watermarking (Kirchenbauer et al., 2023; Dathathri et al., 2024; Zhu et al., 2024; Munyer et al., 2024; Yang et al., 2023) provides a practical solution by embedding imperceptible but detectable watermark signals in generated text, supporting origin verification and ownership attribution. For watermarking to be practical in LLM services, however, it must integrate efficiently into the inference engines that generate these outputs at scale.

Existing watermarking methods largely focus on watermark construction and detection algorithms (Kirchenbauer et al., 2023; Kuditipudi et al., 2023; Dathathri et al., 2024; Yoo et al., 2024), while LLM inference engines have evolved independently to optimize latency, memory, and throughput (Li et al., 2023; Zhong et al., 2024; Hu et al., 2025). This separation creates two deployment gaps. First, watermarking is often implemented as an auxiliary pipeline, relying on separate modules or external natural language processing (NLP) tools such as spaCy (Honnibal et al., 2020), Stanza (Qi et al., 2020), and NLTK (Bird et al., 2009) for named entity recognition (NER) and part-of-speech (POS) tagging, rather than sharing an engine with text generation (Wouters, 2024; Wang et al., 2025; Huo et al., 2024; Huang & Wan, 2025). Such fragmentation increases latency, memory usage, and system complexity. Second, serving optimizations such as asynchronous generation, key-value cache (KV-cache) reuse, prefix sharing, and scheduling remain largely unexplored for watermarking.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

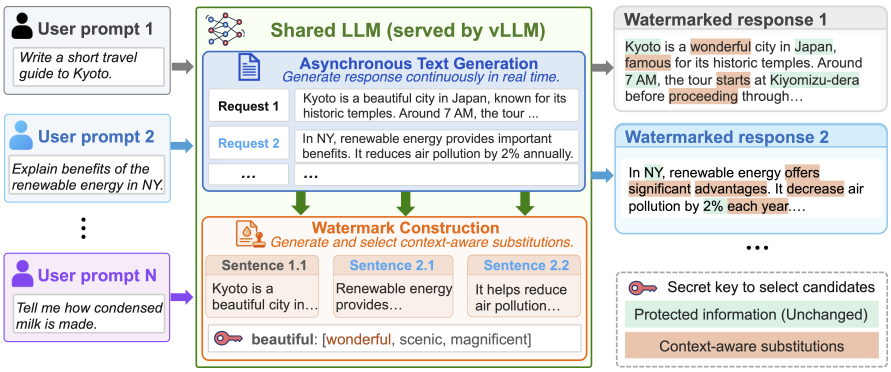


Figure 1: **SWIFT example.** Multiple prompts are processed concurrently through a shared vLLM backend. For a Kyoto guide, SWIFT generates context-aware substitutions (e.g., "beautiful" → "wonderful") while preserving key information such as locations and times. Watermark construction runs asynchronously alongside text generation, enabling streaming with minimal latency overhead.

These gaps motivate a question: *Can we design watermarking as part of LLM inference engines, rather than as an auxiliary pipeline?* To address this question, we introduce SWIFT (Fig. 1), to our knowledge, the first high-utility **S**erving-aware **W**atermarking for **I**nference-**F**riendly **T**ext generation framework that co-designs watermarking with modern inference engines. Built on vLLM (Kwon et al., 2023), a high-throughput LLM serving engine with efficient KV-cache management and request scheduling, SWIFT addresses the gaps through a unified co-serving architecture. First, it ***co-serves text generation and watermark construction*** on the same LLM, inference engine, and GPU, eliminating separate modules or NLP tools while reducing duplicated computation, memory overhead, and system complexity. For watermark construction, the shared LLM performs instruction-guided candidate generation, identifying watermarkable words, protecting named entities and sensitive words, and generating context-aware substitutions under semantic and factual constraints. Second, SWIFT introduces an ***adaptive co-serving scheduler*** that dynamically admits watermark requests based on queue length and waiting time pressure, preventing backlogs while preserving resources for text generation. This unified design further enables both workloads to share serving optimizations, including asynchronous execution, prefix caching, continuous batching, structured decoding, KV-cache management, and chunked prefill, allowing watermark computation to overlap with ongoing generation while reducing redundant computation and resource overhead.

Experiments across tasks and LLMs demonstrate that SWIFT improves watermark quality and serving efficiency over baselines, including KGW (Kirchenbauer et al., 2023), SynthID (Dathathri et al., 2024), and SafeSeal (Dang et al., 2026). It achieves a utility score of 4.87 and 99.65% detection accuracy with 0.905 seconds (s) latency, $5.9\times$ lower latency than the highest-utility baseline. SWIFT retains up to 97.7% detection accuracy under removal attacks and generalizes to downstream tasks, including summarization, short reasoning, and technical medical text, without performance degradation. Qualitative analysis shows its substitutions better preserve factual consistency. Its key-conditioned detection reaches 99.7% with the correct key and near-zero detection with incorrect keys. From a serving perspective, SWIFT achieves high throughput of 528.7 tokens/s, while reducing KV-cache usage and attaining a 96.8% prefix-cache hit rate. Overall, our approach combines high utility, robust detectability, and efficient serving, making LLM watermarking more practical.

2 Background and Related Work

LLM Watermarking. LLMs generate text autoregressively by decoding one token at a time. LLM watermarking modifies this process or generated text to embed imperceptible but detectable watermark signals for provenance verification, with a generator embedding the watermark and a detector verifying its presence. Existing methods broadly fall into two categories. ***In-processing watermarks*** modify token selection during decoding (Zhao et al., 2024; Liu & Bu, 2024; Hu et al., 2024). Integrated into text generation, these methods naturally support streaming but may not fully exploit modern serving optimizations (Fernandez et al., 2025). Their per-token modifications may accumulate utility degradation over long outputs. ***Post-processing watermarks*** modify generated text, often

using auxiliary NLP tools or language models (LMs) (Topkara et al., 2006; Chang et al., 2024). They can better preserve utility by selecting watermark locations and substitutions under semantic and contextual constraints (Yang et al., 2022; Dang et al., 2026), but add latency and memory overhead and are less suited to streaming. These trade-offs suggest that watermarking performance depends on both the embedding algorithm and its integration with the underlying inference system.

LLM Inference Engines and Serving Systems. Modern LLM inference engines improve serving efficiency through KV-cache management, continuous batching, and request scheduling (Miao et al., 2025; Sun et al., 2024). As KV-cache memory grows with the number and length of active requests, it can limit serving concurrency (Jiang et al., 2026; Mitzenmacher & Shahout, 2025). vLLM (vLLM Team, 2026a) addresses this bottleneck with PagedAttention (Kwon et al., 2023), which manages KV states in fixed-size blocks to reduce memory fragmentation and improve utilization. It also supports continuous batching, prefix caching, asynchronous generation, and structured decoding. Other systems optimize complementary serving components: Orca introduces iteration-level scheduling (Yu et al., 2022), SGLang improves cross-request KV reuse through RadixAttention (Zheng et al., 2024), and Sarathi-Serve uses chunked prefill to mitigate prefill-decode interference (Agrawal et al., 2024). DistServe and Splitwise disaggregate prefill and decode execution (Zhong et al., 2024; Patel et al., 2024), while others explore preemptive and dynamic request scheduling (Wu et al., 2023).

Despite these advances, watermarking has largely evolved independently of inference engine design. Most methods use frameworks such as Hugging Face Transformers (Kirchenbauer et al., 2023; Munyer et al., 2024; Dang et al., 2026), with limited cache reuse, asynchronous co-execution, and adaptive scheduling. Existing vLLM integrations mainly target in-processing watermarking (Verma, 2025; vLLM Team, 2026b), while post-processing methods with serving optimizations remain underexplored. This gap limits watermarking from fully leveraging LLM inference mechanisms.

Challenges of Practical Watermarking. Bridging this gap requires addressing two interconnected challenges. First, watermark construction is often implemented as a separate processing pipeline. In-processing methods introduce operations along the decoding path, whereas post-processing methods may rely on external NLP tools or auxiliary LMs. These designs increase computation and deployment complexity. Second, watermarking introduces an additional workload that must coexist with latency-sensitive text generation. Static or uncoordinated execution can create watermark backlogs, compete for resources, and delay streaming outputs. Thus, practical watermarking is both an algorithmic and systems problem: it must preserve utility, detectability, and robustness while sharing inference resources efficiently. These challenges motivate SWIFT, which integrates watermark construction into a shared LLM backend and adaptively co-schedules it with ongoing text generation.

Threat Model. We consider a proprietary LLM service provider where users can submit prompts and receive outputs via an API. A black-box adversary can query the API and observe outputs but cannot access model parameters, architecture, or internal states. The adversary seeks to weaken watermark detection while preserving the original content through watermark removal attacks such as paraphrasing, insertion, synonym substitution, and rewriting (Zhang et al., 2024; Lin et al., 2021).

3 SWIFT: Adaptive Co-Serving LLM Watermarking

3.1 Design Objectives and System Overview

The design of SWIFT follows four objectives: *(i) Unified design*, minimizing external dependencies through a shared LLM backend; *(ii) Utility preservation*, maintaining semantic and factual consistency through instruction-guided watermarking; *(iii) Adaptive co-serving*, coordinating concurrent workloads to control watermark backlogs and maintain streaming responsiveness; and *(iv) Serving efficiency*, leveraging inference optimizations to reduce computation, memory, and latency.

Fig. 2 and Alg. 1 (Appx. A) illustrate the architecture of SWIFT, which integrates text and watermark generation streams within a single vLLM engine. As both streams share a backend, SWIFT loads the LLM once, enabling both workloads to leverage serving optimizations and reducing external dependencies. The workloads execute asynchronously, overlapping watermark processing with ongoing text generation, while an adaptive co-serving scheduler coordinates their execution under varying demand. Together, these components balance watermark quality, detectability, and serving efficiency. The following subsections describe co-serving, scheduling, and watermark construction.

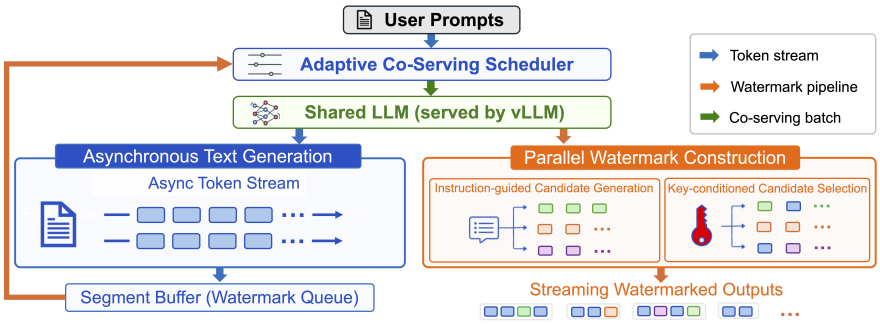


Figure 2: Overview of SWIFT.

3.2 Asynchronous Co-Serving of Text and Watermark Generation

SWIFT co-serves text generation and watermark construction as two asynchronous inference streams sharing the same vLLM engine, LLM, and GPU. Given a prompt x, the text generation stream autoregressively decodes a response $y = (y_1, \ldots, y_T)$ $(T > 0)$, producing tokens incrementally rather than waiting for the complete response. This enables streaming generation while allowing watermark processing to begin before text generation finishes, reducing the latency of conventional generate-then-watermark pipelines (He et al., 2022a; Hao et al., 2025; Yang et al., 2022). By reusing the same inference backend for both workloads, SWIFT eliminates separate watermarking engines, external NLP tools, and auxiliary LMs, reducing complexity and memory overhead.

To enable asynchronous execution, SWIFT maintains a segment buffer that collects generated tokens and identifies completed segments (typically sentences). At decoding step t, token y_t is appended to the buffer. Upon detecting a sentence boundary, SWIFT extracts the completed segment s_i and submits it as a watermarking request to the shared LLM. Meanwhile, text generation continues decoding subsequent tokens $y_{t+1}, y_{t+2}, \ldots$ without waiting for watermarking to finish (Fig. 2). The adaptive co-serving scheduler coordinates watermark request admission alongside ongoing generation to balance both workloads, as detailed in Section 3.3. Once processed, watermarked segments are streamed to users in their original order, maintaining incremental output delivery.

Because both streams share the same LLM backend, SWIFT loads the model once and leverages vLLM serving optimizations to improve efficiency (vLLM Team, 2026a). Prefix caching reuses KV states for shared watermark instructions, while continuous batching incorporates newly admitted watermark requests. Request prioritization manages latency-sensitive workloads, and chunked prefill mitigates prefill-decode interference. Shared KV-cache management and consistent model precision further reduce memory overhead and avoid redundant model representations or conversion costs.

3.3 Adaptive Co-Serving Scheduler

To coordinate text generation and watermark construction, SWIFT introduces an ***adaptive co-serving scheduler*** that adjusts watermark request admission based on runtime workload pressure (Fig. 3). The inference engine co-serves text generation and watermark requests, each occupying a sequence in the mixed serving batch (*mixed-batch*). Rather than reserving a fixed portion of the batch for watermarking, the scheduler adapts watermark admission based on queue length and waiting time, balancing workloads while maintaining streaming responsiveness and resource efficiency.

Scheduler Update and Adjustment. To balance the two request types, the adaptive scheduler updates the watermark admission ratio r_τ every K scheduling steps, where τ is a scheduling step:

$$r_\tau = r_{\min} + (r_{\max} - r_{\min})\, p_\tau, \tag{1}$$

where $r_{\min}$ and $r_{\max}$ define the lower and upper bounds of the watermark admission ratio, and $p_\tau \in [0, 1]$ represents normalized watermark pressure. This formulation adjusts watermark admission according to demand. Under low pressure ($p_\tau \approx 0$), r_τ remains close to $r_{\min}$, preserving more capacity for text generation. As pressure increases ($p_\tau \to 1$), r_τ rises toward $r_{\max}$ to admit more watermark requests and limit backlog accumulation. Once pressure decreases, subsequent updates reduce r_τ, returning capacity to text generation and maintaining streaming responsiveness.

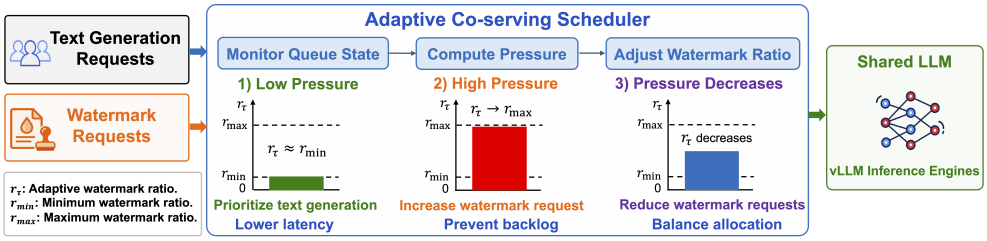


Figure 3: Adaptive co-serving scheduler behaviors.

Watermark Pressure Estimation. To obtain the normalized watermark pressure p_τ, SWIFT monitors the first-in, first-out (FIFO) queue of completed segments awaiting watermarking. Let q_τ denote the watermark queue length (i.e., the number of pending watermark requests) and w_τ denote the waiting time of the oldest request. SWIFT combines q_τ and w_τ to estimate watermark pressure:

$$p_\tau = \text{clip}\left(\max\left(\frac{q_\tau}{B_{\max}}, \frac{w_\tau}{W_{\max}}\right), 0, 1\right). \tag{2}$$

Here, $B_{\max}$ and $W_{\max}$ denote the queue length and waiting time thresholds, respectively. The term $\frac{q_\tau}{B_{\max}}$ captures *backlog pressure*, measuring pending requests relative to the queue threshold, while $\frac{w_\tau}{W_{\max}}$ captures *waiting time pressure*, measuring the oldest request's delay relative to its threshold. These terms capture complementary conditions: a large queue indicates accumulating demand, while a long waiting time reveals delays even with few pending requests. Taking the maximum prioritizes the more urgent condition, while clipping p_τ to $[0, 1]$ keeps r_τ within its prescribed limits.

Adaptive Mixed-Batch Admission. The scheduler translates r_τ into the target watermark capacity within the shared batch. Let B denote the maximum number of concurrent sequences supported by the shared vLLM. The target watermark capacity is $B_\tau^{\text{wm}} = \lfloor r_\tau B \rfloor$, with the remaining capacity available for text generation. Thus, r_τ controls watermark request admission without modifying the decoding process. Both request types execute within the shared vLLM engine, leveraging serving optimizations while adapting concurrency to limit watermark backlogs and processing delays.

3.4 Watermark Construction and Detection

To reduce complexity and external dependencies, SWIFT reuses the same LLM for candidate generation and employs key-conditioned selection to embed detectable watermarks while preserving semantic and factual consistency. We next describe these mechanisms and watermark detection.

Instruction-guided Candidate Generation. To preserve output utility without auxiliary NLP tools or LMs, SWIFT uses instruction-guided candidate generation to identify watermarkable words and generate substitutions using the same LLM. A key insight is that replacing words with context-aware synonyms can preserve meaning while protecting key information (e.g., named entities or sensitive spans) to ensure factual accuracy. Accordingly, the instructions guide the LLM to identify protected information, replaceable words, and generate substitutions under semantic and factual constraints (Fig. 6). Once admitted by the adaptive scheduler, each completed segment $s_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,n_i})$, where $w_{i,j}$ is its j-th word, is processed through this pipeline. The LLM identifies watermarkable positions $\mathcal{I}_i \subseteq \{1, \ldots, n_i\}$, typically replaceable nouns, verbs, adjectives, and adverbs that can be substituted without altering the meaning or factual content of s_i.

For each segment s_i, SWIFT constructs a watermarking prompt $P_i = [P_{\text{shared}}; s_i]$, where P_{shared} is a shared instruction prefix and the completed segment s_i is the segment-specific suffix. The shared prefix specifies protected information rules, semantic and factual constraints, candidate-generation instructions, and few-shot examples (Appx. C). Since P_{shared} is identical across watermark requests, vLLM prefix caching reuses its KV states, reducing repeated prefill computation. This is particularly beneficial because the shared prefix is substantially longer than the segment-specific suffix.

Watermark requests are processed via ***vLLM continuous batching*** (vLLM Team, 2026a), joining the shared serving batch as slots become available. This enables dynamic scheduling and concurrent execution of text generation and watermark requests with varying sequence lengths (Yu et al., 2022).

To ensure consistent candidate generation, SWIFT leverages ***vLLM structured output decoding*** to constrain responses to a JSON schema. For each watermarkable position $j \in \mathcal{I}_i$, let $\mathcal{C}_{i,j}$ denote the set of context-aware candidate substitutions for $w_{i,j}$ (e.g., {*"beautiful": ["wonderful", "scenic", "magnificent"]*}). The structured output for segment s_i is represented as $\mathcal{C}_i = \{w_{i,j} : \mathcal{C}_{i,j} \mid j \in \mathcal{I}_i\}$. This reduces malformed outputs and provides consistent candidate mapping for watermark selection.

Key-conditioned Candidate Selection. For each completed segment s_i with its context-aware candidate set $\mathcal{C}_i$, SWIFT embeds the watermark using key-conditioned Tournament selection (Dathathri et al., 2024). The selected substitutions are defined as: $\mathcal{R}_i = \mathbb{T}(k, \mathcal{C}_i, s_i) = \{c^*_{i,j}\}_{j \in \mathcal{I}_i}$, where $\mathbb{T}$ is the Tournament selection function parameterized by secret key k, and $c^*_{i,j} \in \mathcal{C}_{i,j}$ is the selected candidate for word $w_{i,j}$. Using key-derived pseudorandomness, $\mathbb{T}$ selects candidates reproducibly under the same key but unpredictably without it. The watermarked segment $\tilde{s}_i = \mathrm{Replace}(s_i, \mathcal{R}_i)$ is obtained by applying these substitutions $\mathcal{R}_i$ to $\mathcal{I}_i$ while leaving other words in s_i unchanged. The segment is then streamed as described in Section 3.2. Since candidate generation enforces semantic, factual, and grammatical constraints, SWIFT avoids additional semantic reranking with external scoring models or similarity filters, reducing model loading, memory overhead, and complexity.

Watermark Detection. SWIFT verifies watermarked text using a key-conditioned contrastive detector, which determines whether a text is watermarked under a specific secret key (Dang et al., 2026). Given a text y and key k, the detector predicts $\mathcal{D}(y, k) \in \{0, 1\}$, where $\mathcal{D}(y, k) = 1$ indicates that y is watermarked under k, and $\mathcal{D}(y, k) = 0$ otherwise. The detector encodes y with a bidirectional Transformer encoder (RoBERTa (Liu et al., 2019)), maps k to a deterministic key embedding, and combines the text and key representations (element-wise interaction and absolute difference) for classification. During training, matched text-key pairs are positives, while mismatched ones are negatives. The detector is optimized with a detection loss for binary classification and a contrastive loss that aligns matched pairs while separating mismatched ones, enabling provider-specific detection.

Additional detailed time and space complexity analyses of SWIFT *are provided in Appx. A.*

4 Experiments

Our extensive evaluation covers **(1)** watermark effectiveness and efficiency in terms of utility, detectability, and latency compared with representative baselines; **(2)** robustness against watermark removal attacks; **(3)** generalization across domains, tasks, and LLMs; **(4)** adaptive scheduling analysis; **(5)** key-conditioned watermark detection; and **(6)** qualitative analysis and ablation studies.

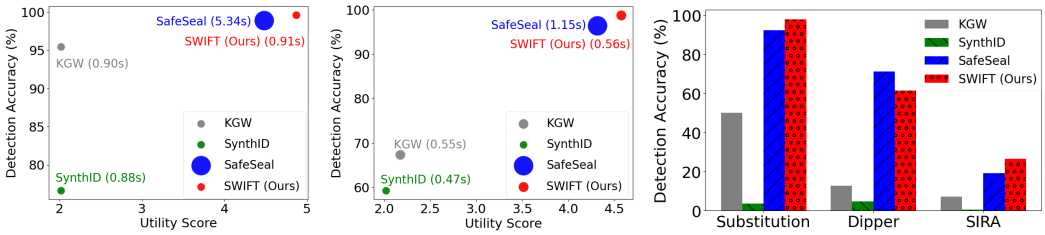
4.1 Experimental Settings

Baselines. We compare against representative LLM watermarks: *KGW* (Kirchenbauer et al., 2023), a logit-bias method that steers decoding toward preferred tokens; *SynthID* (Dathathri et al., 2024), a sampling-based method using pseudorandom keys; and *SafeSeal* (Dang et al., 2026), a lexical substitution method with named-entity protection. These baselines cover two watermarking categories.

Watermark Removal Attacks. We evaluate robustness against three attacks: *Dipper*, a sentence-level paraphrasing attack (Krishna et al., 2024); *Substitution*, a token-level synonym substitution (Pan et al., 2024); and *SIRA*, an LLM-based self-information rewriting attack (Cheng et al., 2025).

Tasks, Datasets, and LLMs. We evaluate SWIFT across four tasks: text generation on C4 dataset (Dodge et al., 2021), summarization on CNN/Daily Mail dataset (Hermann et al., 2015), mathematical reasoning and short text watermarking on GSM8K dataset (Cobbe et al., 2021), and technical medical question answering on MedMCQA dataset (Pal et al., 2022). For C4, we use the first 50 tokens as prompts and generate up to 200 tokens. Other datasets follow their respective summarization or question-answering formats. Gemma-3-27B-IT (Team et al., 2025) serves as the shared LLM, while Qwen2.5-32B (Yang et al., 2024), LLaMA-3.1-8B-Instruct (Grattafiori et al., 2024), and Ministral-8B-Instruct (AI, 2024) support cross-LLM and key-conditioned detection experiments. We sample $20,000$ examples for detector training and $1,000$ held-out examples for evaluation.

Evaluation Metrics. We evaluate SWIFT along four key dimensions: utility, detectability, robustness, and serving efficiency. ***Utility*** measures preservation of semantic meaning and factual consistency, assessed by GPT-5.5 (OpenAI, 2026) as an LLM-as-a-judge on a 0-5 scale. ***Detectability*** measures the detector's ability to distinguish watermarked from non-watermarked outputs using



(a) Text generation (b) Text summarization (c) Attack robustness

Figure 4: Watermark effectiveness and efficiency on (a) text generation and (b) summarization, with circle area proportional to latency, and (c) robustness against watermark removal attacks.

balanced detection accuracy averaged across both output classes. ***Robustness*** measures detection accuracy after watermark removal attacks, reported as the fraction of attacked watermarked outputs correctly classified as watermarked. ***Serving efficiency*** is measured primarily by end-to-end latency.

Additional metrics and results on serving efficiency, statistical significance, ROC analysis, and scalability, along with detailed definitions, calculations, and implementation details, are in Appx. B.

4.2 Experimental Results and Analysis

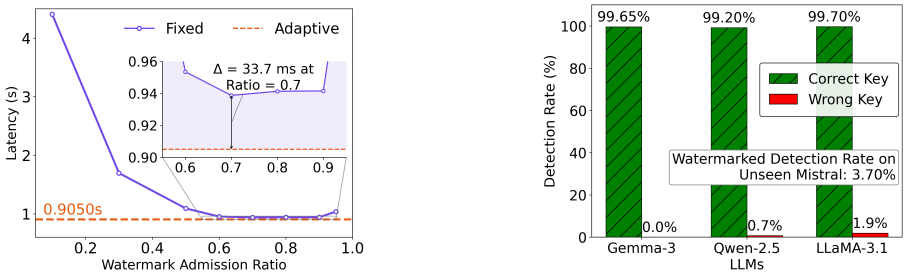
Watermark Effectiveness and Efficiency. Fig. 4 demonstrates that SWIFT achieves the best overall trade-off among utility, detectability, and serving latency. ***On text generation***, SWIFT attains the highest utility (4.87) and detection accuracy (99.65%) with only 0.905s latency. KGW and SynthID are slightly faster (0.88-0.90s) but achieve substantially lower utility (2.01-2.02) and detection accuracy (76.65-95.50%). SafeSeal achieves strong watermark quality but incurs 5.34s latency, making SWIFT approximately $5.9\times$ faster. ***On text summarization*** (Fig. 4(b)), SWIFT similarly achieves the highest utility (4.57) and detection accuracy (98.8%) with only 0.559s latency. Although KGW and SynthID are slightly faster, their detection accuracies drop to 67.35% and 59.25%, respectively. SafeSeal reaches 96.45% detection but with lower utility and approximately $2\times$ higher latency. Similar trends are observed on Qwen for both text generation and summarization (Fig. 7), demonstrating consistent performance across LLMs. These results highlight the benefit of co-designing watermarking with inference infrastructure. Instruction-guided candidate generation preserves semantic and factual quality, while key-conditioned selection provides strong detectability. Meanwhile, asynchronous co-serving, adaptive scheduling, and vLLM optimizations reduce watermarking overhead, allowing SWIFT to approach the latency of in-processing methods while substantially improving utility and detectability, and to remain consistently much faster than post-processing SafeSeal.

Robustness Against Attacks. Fig. 4(c) shows that SWIFT remains highly robust to watermark removal attacks, retaining 97.7% detection rate under Substitution and 61.3% under Dipper. SWIFT and SafeSeal substantially outperform KGW and SynthID. This robustness is partly enabled by the key-conditioned detector, which learns watermark patterns from text-key pairs rather than predefined rules or statistical scores that may be disrupted by attacks. SIRA reduces SWIFT's detection rate to 26.4%, but at substantial cost. Its attacked outputs achieve only a 0.145 pairwise win rate against the original watermarked outputs using the same LLM judge, indicating considerable utility degradation. SIRA also requires 8.933s, about $9.87\times$ SWIFT's 0.905s latency. Therefore, while SIRA weakens detectability, it incurs substantial utility loss and runtime overhead.

Domain and Task Generalization. Beyond general text generation and summarization, practical watermarking should remain effective on short outputs and technical domains where constrained token choices can reduce watermark capacity. ***On short mathematical reasoning*** (Table 1), SWIFT preserves the original task accuracy (without watermark) of 93.2%, achieves the highest utility (4.93), 90.70% detection accuracy, and the lowest watermarking latency (0.352s). KGW and SynthID slightly reduce task accuracy and achieve only 53.35% and 54.95% detection, respectively. SafeSeal preserves task accuracy with comparable detection (90.75%) but requires 3.253s, making SWIFT over $9\times$ faster. ***On highly technical medical question answering*** with constrained watermarking choices (Table 6), SWIFT maintains the baseline accuracy of 62.6% without degradation while achieving the highest utility (4.70), 96.70% detection accuracy, and the lowest latency

Table 1: Performance on GSM8K. ΔAcc. is the relative accuracy change from the No watermark.

Setting	Accuracy (%) ↑	ΔAcc. (%) ↑	Utility ↑	Detection (%) ↑	Latency (s) ↓
No watermark	93.2	–	–	–	0.208
KGW	92.3	-0.97	4.66	53.35	0.445
SynthID	92.7	-0.54	4.45	54.95	0.437
SafeSeal	**93.2**	**0.00**	4.61	**90.75**	3.253
SWIFT (Ours)	**93.2**	**0.00**	**4.93**	90.70	**0.352**



(a) Scheduler Latency Performance

(b) Key-conditioned Detection

Figure 5: Core serving evaluation and key-conditioned detection performance.

(0.401s). Together, these results show that SWIFT generalizes across domains and tasks while maintaining strong utility, detectability, and serving efficiency. Its instruction-guided watermarking selectively modifies suitable words under semantic and factual constraints, enabling detectable signals without altering task answers. In contrast, KGW and SynthID make watermarking decisions at every decoding step; limited token choices causes fewer substitutions, weakening watermark strength and detectability. SWIFT's lower latency on reasoning and medical tasks is further explained by fewer watermark requests and lower watermark density, reducing workload, whereas KGW and SynthID operate at every decoding step. Detailed task characteristics are in Table 7.

Adaptive Co-serving Scheduler. Fig. 5(a) compares SWIFT's adaptive scheduler with fixed admission ratios in $[0.1, 0.95]$. The adaptive scheduler updates every five scheduling steps and achieves the lowest latency of 0.905s. Low fixed ratios allocate insufficient capacity to watermarking, causing request backlogs and higher latency, while an overly high ratio ($r = 0.95$) interferes with text generation and increases latency. In contrast, SWIFT adapts watermark admission to runtime workload pressure, balancing both streams while maintaining low latency and streaming responsiveness.

Key-conditioned Detection. Fig. 5(b) evaluates SWIFT's universal detector across LLMs using correct and incorrect keys. With the correct key, detection remains high at 99.2-99.7% across Gemma-3, Qwen-2.5, and LLaMA-3.1, while incorrect-key detection remains near zero (0-1.9%), demonstrating strong key specificity and reducing false attribution. The key-conditioned objective aligns matched text-key pairs while separating mismatched pairs, enabling key-specific verification. On the unseen Ministral model, only 3.7% of outputs are detected using the Gemma-3 key, further indicating that the detector does not associate unseen models with known keys. A single detector can therefore generalize across LLMs without per-model retraining, simplifying system integration.

Qualitative Evaluation. To further examine text quality, Table 2 compares the original response with outputs from different watermarking methods. SWIFT best preserves factual consistency with mostly meaning-preserving substitutions, whereas KGW and SynthID exhibit substantial factual drift and SafeSeal introduces less appropriate substitutions. KGW changes the duration from 20 to 25 years and adds unsupported roles and personnel. SynthID changes it to 23 years and adds unsupported policing details. These examples show how token-level perturbations during autoregressive decoding can propagate into broader deviations. SafeSeal preserves the main facts but changes "a lot going for it" to "a lot riding for it." In contrast, SWIFT preserves key facts while making context-aware changes such as "a lot happening for it." By generating context-aware candidates with the shared LLM, SWIFT eliminates external dependencies while improving semantic control and limiting factual drift. Additional reasoning and medical examples (Appx. B) show similar trends.

Table 2: Text utility comparison of original and watermarked outputs. Green marks factual consistency, blue marks meaning-preserving substitutions, and red marks semantic or factual errors.

Prompt	...Chief inspector Neil Kentish joined Thames Valley Police less than a month ago after serving in ...
No watermark	the Metropolitan Police for 20 years. He's taken over from chief inspector Olly Wright, who has moved to a new role within the force. Chief Insp Kentish said he's been spending his time getting to know the area, meeting local officers and community leaders. "It's been a really busy few weeks, but a really positive few weeks" he said. "Aylesbury is a really interesting place, it's got a lot going for it." ...
KGW	the Met Police for 25 years. He heads up the Wycombe and Aylesbury areas as deputy area commander, reporting to area commander for Buckinghamshire and Milton Keynes, Chief Superintendent Rob Harding. Chief Insp Kentish said: "It's very different coming from London. Aylesbury is a lovely market town, the pace of life is noticeably slower but people are really friendly". "I've been out and about meeting community groups," ...
SynthID	Metropolitan Police for 23 years. He is taking over from chief inspector Rick Carr, who recently transferred to the Roads Policing Unit. Chief Inspector Kentish has made it a priority to meet local communities and businesses and key partners. "I've been really impressed with the commitment and dedication of the Aylesbury local policing area team. They are really enthusiastic and want to do the best for the community." ...
SafeSeal	the Metropolitan Police for 20 years. He's taken over from chief inspection Olly Wright, who has bolted to a new work within the commission. Chief Insp Kentish explained he's been spent his work starting to know the community, meeting local commanders and community representatives. "It's been a pretty busy few weeks, but a totally positive few weeks," he added. "Aylesbury is a seriously interesting town, it's got a lot riding for it." ...
SWIFT (Ours)	the Metropolitan Police for 20 years. He's stepped in over from chief inspector Olly Wright, who has transferred to another role within the force. Chief Insp Kentish said he's been devoting his time learning to know the area, encountering local officers and community leaders. "It's been a really hectic few weeks, but a really nice few weeks" he said. "Aylesbury is a really engaging place, it's got a lot happening for it." ...

Table 3: Ablation Study: Comparison of SWIFT Variants.

Version	**Latency (s)** ↓	**Δ Latency (%)** ↓	**Utility** ↑	**Δ Utility (%)** ↑
SWIFT	0.905	–	4.871	–
SWIFT-FB	1.0921	+20.67	4.871	0.00
SWIFT-NER	1.7005	+87.90	4.610	-5.36
SWIFT-EN	0.900	-0.55	2.389	-50.95

Ablation Studies. Table 3 evaluates key SWIFT components. SWIFT-FB disables continuous batching and request prioritization, increasing latency by 20.67% without affecting utility. SWIFT-NER replaces LLM-based entity handling with an external NER tool (spaCy (Honnibal et al., 2020)), increasing latency by 87.90% and reducing utility by 5.36%. SWIFT-EN removes entity-protection instructions, improving latency by 0.55% while reducing utility by 50.95%. Overall, serving optimizations, a shared LLM, and entity protection jointly improve SWIFT's latency-utility trade-off.

Limitations. Despite its effectiveness, SWIFT has several limitations. First, *watermark quality depends on the capability of the shared LLM*. Since the same model performs text generation, watermarkable-word identification, and candidate substitution, weaker models may produce less reliable substitutions or fail to preserve protected information, reducing output quality and utility in specialized domains. Second, SWIFT *does not fully leverage KV-cache reuse*. While shared instructions benefit from prefix caching, reusing KV states for segment-specific suffixes remains non-trivial because text generation uses the original prompt and history, while candidate generation uses a separate watermarking context. Finally, although SWIFT achieves competitive serving efficiency (Table 12), it *currently streams outputs at the sentence level* rather than token by token.

5 Conclusion

We presented SWIFT, a serving-aware LLM watermarking framework that co-designs watermark construction with modern inference infrastructure. SWIFT unifies text generation and watermark construction within a shared LLM backend, reducing reliance on auxiliary models and external NLP tools, while instruction-guided candidate generation and entity protection preserve semantic and factual utility. Its asynchronous co-serving architecture and adaptive scheduler coordinate text generation and watermark workloads under varying demand while leveraging inference-engine optimizations. Experiments across domains, tasks, and LLMs demonstrate strong utility, detectability, robustness, downstream task preservation, and serving efficiency. These results highlight the benefits of jointly considering watermark quality and inference-system design. Future work will explore finer-grained streaming through adaptive token release and improved KV-cache reuse across generation and watermarking contexts, further improving responsiveness and computational efficiency.

AI USE STATEMENT

In this work, generative AI tools were used to assist with research brainstorming and refinement of experimental ideas, code development and debugging, literature exploration, and improving the clarity of the manuscript. In addition, an OpenAI language model was used as an LLM-as-a-judge for utility evaluation and pairwise win rate analysis, as described in the evaluation metrics. Generative AI was not used to fabricate experimental data or results, and all experimental measurements and reported results were obtained from our implemented systems and evaluation pipelines. All AI-assisted research ideas, code, and text revision were reviewed and verified by the authors. The authors made all final decisions regarding the methodology, experimental design, interpretation of results, and claims presented in the paper. We take responsibility for the final content of this work, including any text, code, claims, or other artifacts produced with the aid of generative AI.

ETHICS STATEMENT

SWIFT supports provenance and attribution of LLM-generated text, but watermark detection is probabilistic and should not be treated as sole evidence of authorship, misuse, or policy violations, especially in high-stakes settings. Real-world deployment also requires secure key management and consideration of adversarial removal or forgery, privacy, fairness, distribution shift, and misuse beyond validated conditions. Our experiments use existing benchmarks only; medical QA data is used solely to evaluate watermark utility and detectability and not for clinical decision making. We report utility and robustness results to characterize both the capabilities and limitations of SWIFT.

REPRODUCIBILITY STATEMENT

We provide sufficient information to reproduce SWIFT and its evaluation throughout the main paper and appendix. The main paper describes the system architecture, asynchronous co-serving, adaptive scheduling, watermark construction, and key-conditioned detection. The appendix provides the end-to-end algorithm, implementation and hardware details, evaluation settings and metrics, detector training configuration, serving measurements, additional analyses, and the watermark-generation and LLM-as-a-judge prompts. Code and artifacts are available at: `https://anonymous.4open.science/r/SWIFT-0597`

REFERENCES

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv*, 2023.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *OSDI 24*, pp. 117–134, 2024.

Mistral AI. Ministral-8b-instruct-2410. `https://huggingface.co/mistralai/Ministral-8B-Instruct-2410`, 2024.

Anthropic. Claude. `https://claude.ai/`, 2026. Accessed: 2026-09-16.

Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. 2009.

Yapei Chang, Kalpesh Krishna, Amir Houmansadr, John Wieting, and M. Iyyer. Postmark: A robust blackbox watermark for LLMs. In *EMNLP*, 2024.

Yixin Cheng, Hongcheng Guo, Yangming Li, and Leonid Sigal. Revealing weaknesses in text watermarking through self-information rewrite attacks. In *ICML*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, et al. Training verifiers to solve math word problems. *arXiv*, 2021.

Kieu Dang, Phung Lai, NhatHai Phan, Yelong Shen, and Ruoming Jin. Robust llm watermarking with minimal semantic distortion for ip protection. *arXiv*, 2026.

Sumanth Dathathri, Abigail See, Sumedh Ghaisas, Po-Sen Huang, et al. Scalable watermarking for identifying LLM outputs. *Nature*, 634, 2024.

Jesse Dodge, Maarten Sap, et al. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. In *Proceedings of EMNLP*, 2021.

Pierre Fernandez, Tom Sander, Hady Elsahar, Hongyan Chang, Tomáš Souček, Valeriu Lacatusu, et al. How good is post-hoc watermarking with language model rephrasing? *arXiv*, 2025.

Google. Gemini. `https://gemini.google.com/`, 2026. Accessed: 2026-09-16.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, et al. The llama 3 herd of models. *arXiv*, 2024.

Jifei Hao, Jipeng Qiang, Yi Zhu, Yun Li, Yunhao Yuan, and Xiaoye Ouyang. Post-hoc watermarking for robust detection in text generated by large language models. In *COLING*, 2025.

Haiyun He, Yepeng Liu, Ziqiao Wang, Yongyi Mao, and Yuheng Bu. Theoretically grounded framework for llm watermarking: A distribution-adaptive approach. *NeurIPS*, 2026.

Xuanli He, Qiongkai Xu, Lingjuan Lyu, Fangzhao Wu, and C. Wang. Protecting IP of language generation apis with lexical watermark. In *Proceedings of AAAI*, 2022a.

Xuanli He, Qiongkai Xu, Yi Zeng, Lingjuan Lyu, et al. Cater: IP protection on text generation apis via conditional watermarks. *Advances in Neural Information Processing Systems*, 2022b.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28, 2015.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spacy: Industrial-strength natural language processing in python. 2020.

Abe Hou, Jingyu Zhang, Tianxing He, Yichen Wang, Yung-Sung Chuang, Hongwei Wang, Lingfeng Shen, Benjamin Van Durme, Daniel Khashabi, and Yulia Tsvetkov. SemStamp: A semantic watermark with paraphrastic robustness for text generation. In *Proceedings of the ACL*, 2024.

Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, et al. DEEPSERVE: Serverless large language model serving at scale. In *USENIX ATC 25*, 2025.

Zhengmian Hu, Lichang Chen, Xidong Wu, Yihan Wu, Hongyang Zhang, and Heng Huang. Unbiased watermark for large language models. In *ICLR*, 2024.

Baizhou Huang and X. Wan. WaterPool: A LM watermark mitigating trade-offs among imperceptibility, efficacy and robustness. In *NAACL: HLT*, 2025.

Mingjia Huo, Sai Ashish Somayajula, Youwei Liang, Ruisi Zhang, Farinaz Koushanfar, and Pengtao Xie. Token-specific watermarking with enhanced detectability and semantic coherence for large language models. In *Proceedings of ICML*, 2024.

Jiantong Jiang, Peiyu Yang, Rui Zhang, and Feng Liu. Towards efficient large language model serving: A survey on system-aware KV cache optimization. In *Findings of ACL*, 2026.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *ICML*, pp. 17061–17084. PMLR, 2023.

Kalpesh Krishna, Yixiao Song, et al. Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense. *NeurIPS*, 36, 2024.

Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. Robust distortion-free watermarks for language models. *TMLR*, 2023.

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 SOSP*, pp. 611–626, 2023.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for DL serving. In *OSDI*, 2023.

Zhe Lin, Yitao Cai, and Xiaojun Wan. Towards document-level paraphrase generation with sentence rewriting and reordering. In *EMNLP*, 2021.

Aiwei Liu, Leyi Pan, Xuming Hu, Shiao Meng, and Lijie Wen. A semantic invariant robust watermark for large language models. In *ICLR*, 2024.

Yepeng Liu and Yuheng Bu. Adaptive text watermark for large language models. In *ICML*, 2024.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv*, 2019.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *ACM Computing Surveys*, 58(1):1–37, 2025.

Michael Mitzenmacher and Rana Shahout. Queueing, predictions, and LLMs: Challenges and open problems. *Stochastic Systems*, 15(3):195–219, 2025.

Travis Munyer, Abdullah All Tanvir, Arjon Das, and Xin Zhong. Deeptextmark: A deep learning-driven text watermarking approach for identifying LLM generated text. *IEEE Access*, 2024.

OpenAI, 2026. `https://openai.com/index/openai-api/` [Accessed: 2026-09-15].

OpenAI. Introducing gpt-5.5. `https://openai.com/index/introducing-gpt-5-5/`, April 2026. Accessed: 2026-09-15.

Ankit Pal, Logesh Kumar Umapathi, and Malaikannan Sankarasubbu. Medmcqa: A large-scale multi-subject multi-choice dataset for medical domain question answering. In *CHIL*, 2022.

Leyi Pan, Aiwei Liu, Zhiwei He, Zitian Gao, Xuandong Zhao, Yijian Lu, Binglin Zhou, et al. Markllm: An open-source toolkit for llm watermarking. In *EMNLP*, pp. 61–71, 2024.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, I. Goiri, S. Maleki, and R. Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ACM/IEEE ISCA*, 2024.

Peng Qi, Yuhao Zhang, Yuhui Zhang, et al. Stanza: A python natural language processing toolkit for many human languages. In *ACL: System Demonstrations*, 2020.

Jie Ren, Han Xu, Yiding Liu, Yingqian Cui, Shuaiqiang Wang, Dawei Yin, and Jiliang Tang. A robust semantics-based watermark for LLM against paraphrasing. In *NAACL*, pp. 613–625, 2024.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for LLM serving. In *OSDI*, 2024.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, et al. Gemma 3 technical report. *arXiv*, 2025.

Mercan Topkara, Umut Topkara, and Mikhail J Atallah. Words are not enough: sentence level natural language watermarking. In *MCPS*, 2006.

Apurv Verma. vllm-watermark: A tiny, hackable research framework for llm watermarking experiments, 2025. URL https://github.com/dapurv5/vLLM-Watermark.

vLLM Team. Engine arguments. https://docs.vllm.ai/en/v0.10.1/configuration/engine_args.html, 2025. vLLM v0.10.1 documentation, accessed September 15, 2026.

vLLM Team. vLLM Documentation. https://docs.vllm.ai/en/latest/, 2026a.

vLLM Team. Text watermarking. https://docs.vllm.ai/en/latest/features/watermarking/, 2026b. vLLM documentation, accessed September 16, 2026.

Zongqi Wang, Tianle Gu, Baoyuan Wu, and Yujiu Yang. Morphmark: Flexible adaptive watermarking for large language models. In *ACL*, 2025.

Bram Wouters. Optimizing watermarks for large language models. In *Proceedings of ICML*, 2024.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, et al. Fast distributed inference serving for large language models. *arXiv*, 2023.

Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, et al. Qwen2.5 technical report. *ArXiv*, 2024.

Xi Yang, Jie Zhang, Kejiang Chen, Weiming Zhang, Zehua Ma, Feng Wang, and Nenghai Yu. Tracing text provenance via context-aware lexical substitution. In *AAAI*, 2022.

Xi Yang, Kejiang Chen, Weiming Zhang, Chang Liu, Yuang Qi, Jie Zhang, Han Fang, and Nenghai Yu. Watermarking text generated by black-box language models. *arXiv*, 2023.

KiYoon Yoo, Wonhyuk Ahn, and Nojun Kwak. Advancing beyond identification: Multi-bit watermark for large language models. In *NAACL*, 2024.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, et al. Orca: A distributed serving system for transformer-based generative models. In *OSDI*, 2022.

Hanlin Zhang, Benjamin L Edelman, Danilo Francati, et al. Watermarks in the sand: Impossibility of strong watermarking for generative models. In *ICML*, 2024.

Xuandong Zhao, Prabhanjan Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for AI-generated text. In *International Conference on Learning Representations*, 2024.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 2024.

Yinmin Zhong, Sheng-Yu Liu, Junda Chen, Jianbo Hu, Yi-Bo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *USENIX Symposium on Operating Systems Design and Implementation*, 2024.

Chaoyi Zhu, Jeroen Galjaard, Pin-Yu Chen, and Lydia Y Chen. Duwak: Dual watermarks in large language models. In *ACL*, pp. 11416–11436, August 2024.

A Additional Technical Details and Analysis

Related Work. Table 4 compares SWIFT with existing watermarking methods across key characteristics relevant to practical implementation. In-processing methods naturally support streaming but generally lack explicit named-entity preservation, while post-processing methods can protect entities but typically require complete outputs and auxiliary LMs or NLP tools for watermark construction. Most existing methods are developed using frameworks such as Hugging Face Transformers, with limited integration of serving optimizations. In contrast, SWIFT combines named-entity preservation, streaming generation, and watermark construction without auxiliary models or NLP tools within a shared vLLM backend, enabling efficient watermarking through optimized inference and asynchronous co-serving.

Table 4: Comparison of SWIFT with representative techniques across key characteristics.

Method	Generation	Named Entity Preservation	Streaming Generation	No Auxiliary Models	Serving System
KGW (Kirchenbauer et al., 2023)	In-processing	✗	✓	✓	Hugging Face
SynthID (Dathathri et al., 2024)	In-processing	✗	✓	✓	Hugging Face
EXP (Kuditipudi et al., 2023)	In-processing	✗	✓	✓	Hugging Face
SIR (Liu et al., 2024)	In-processing	✗	✓	✗	Hugging Face
SemaMark (Ren et al., 2024)	In-processing	✗	✓	✗	Hugging Face
SemStamp (Hou et al., 2024)	In-processing	✗	✓	✗	Hugging Face
DAWA (He et al., 2026)	In-processing	✗	✓	✓	Hugging Face
CATER (He et al., 2022b)	Post-processing	✓	✗	✓	Hugging Face
TW (Yang et al., 2023)	Post-processing	✓	✗	✗	Hugging Face
DTM (Munyer et al., 2024)	Post-processing	✓	✗	✗	Hugging Face
PostMark (Chang et al., 2024)	Post-processing	✓	✗	✗	Hugging Face
SafeSeal (Dang et al., 2026)	Post-processing	✓	✗	✗	Hugging Face
SWIFT (Ours)	Post-processing	✓	✓	✓	vLLM

Algorithm. Algorithm 1 summarizes the end-to-end SWIFT pipeline, including asynchronous generation, segment buffering, watermark insertion, and key-conditioned detection.

Algorithm 1 SWIFT Algorithm

Inputs: user prompts $\mathcal{X} = \{x_1, \ldots, x_N\}$, LLM θ, secret key k, queue length threshold $B_{\max}$, waiting time threshold $W_{\max}$, admission bounds $r_{\min}, r_{\max}$, scheduling steps K, watermark queue $\mathcal{Q}^{wm}$, batch size B, shared instruction P_{shared}
Outputs: streaming watermarked text $\{y_1^{wm}, \ldots, y_N^{wm}\}$ and detector prediction $\mathcal{D}(y, k)$
Asynchronous Text Generation:
  for each user prompt $x_u \in \mathcal{X}$ **do**
    Generate async text via vLLM: $y_u \leftarrow \theta(x_u)$
    for each streamed token $y_{u,t}$ from y_u **do**
      Append $y_{u,t}$ to the partial output buffer for user u
      if a complete sentence segment s_i is identified **then**
        Add s_i to the watermark queue $\mathcal{Q}^{wm}$
Adaptive Co-Serving Scheduler:
  At scheduling step τ, obtain watermark queue length $q_\tau = |\mathcal{Q}_\tau^{wm}|$ and oldest-request waiting time w_τ
  Every K scheduling steps, update pressure p_τ and watermark admission ratio r_τ
  Compute watermark capacity $B_\tau^{wm} = \lfloor r_\tau B \rfloor$ and admit B_τ^{wm} requests to the shared serving batch
Asynchronous LLM Watermarking:
  while $\mathcal{Q}^{wm}$ is not empty **do**
    Collect admitted completed segments: $\mathcal{S}_\tau = \{s_1, \ldots, s_m\}$
    Form instruction-guided candidate-generation prompts: $P_i = [P_{\text{shared}}; s_i], \quad s_i \in \mathcal{S}_\tau$
    Identify positions $\mathcal{I}_i$ and generate candidates: $\mathcal{C}_i = \{w_{i,j} : \{c_{i,j,1}, \ldots, c_{i,j,M_{i,j}}\} \mid j \in \mathcal{I}_i\}$
    for each sentence segment $s_i \in \mathcal{S}_\tau$ **do**
      Select key-conditioned substitutions: $\mathcal{R}_i = \mathcal{T}(k, \mathcal{C}_i, s_i) = \{c_{i,j}^* \mid j \in \mathcal{I}_i\}$
      Apply $\mathcal{R}_i$ to obtain $\tilde{s}_i = \text{Replace}(s_i, \mathcal{R}_i)$
      Emit the watermarked segment $\tilde{s}_i$
  Return: streaming watermarked outputs $\{y_1^{wm}, \ldots, y_N^{wm}\}$
Watermark Detection:
  Given text y and secret key k, apply the key-conditioned detector: $\mathcal{D}(y, k) \in \{0, 1\}$
  where $\mathcal{D}(y, k) = 1$ indicates that y is watermarked under k, and $\mathcal{D}(y, k) = 0$ otherwise
  Return: detector prediction $\mathcal{D}(y, k)$

Instruction-guided Candidate Generation. Fig. 6 illustrates the instruction-guided candidate generation process for the watermark construction stream. Each watermark request combines a reusable shared instruction prefix - covering named-entity protection, watermarkable-word detection, synonym-generation constraints, and few-shot examples -with a sentence-specific suffix. The shared prefix enables prefix-cache reuse across continuously batched requests, while the unified LLM generates candidates under vLLM structured decoding to produce a consistent JSON mapping from watermarkable words to context-aware substitutions. In addition, Fig. 9 provides the detailed template of the shared instruction prefix used in SWIFT for candidate generation.

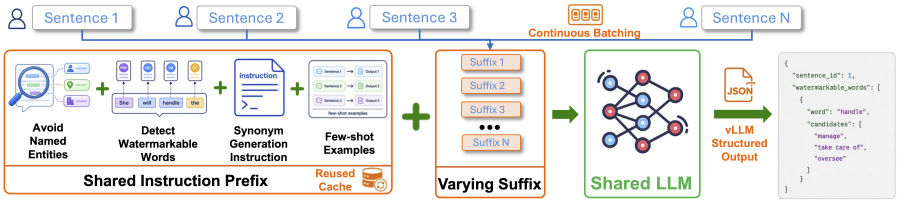


Figure 6: Detailed Instruction-guided Candidate Generation with optimization techniques.

System-level Optimizations. Beyond the scheduling mechanism and serving techniques, SWIFT incorporates additional system-level optimizations for efficient shared inference. Both text generation and watermark construction use the same LLM with a **unified precision** `bf16` configuration, avoiding duplicate model weights, separate runtimes, and additional memory overhead. SWIFT also leverages **chunked prefill** (vLLM Team, 2026a), which interleaves long watermark-prefill computation with ongoing decoding to reduce interference with latency-sensitive generation (Agrawal et al., 2023). Together, these optimizations improve resource efficiency under concurrent workloads.

Time Complexity Analysis. Let P and T denote the prompt and output lengths, respectively, and let N_s denote the number of completed segments, where $|s_i|$ is the length of segment s_i. Let G_i denote the number of output tokens generated for the structured candidate mapping of segment s_i.

SWIFT has two main computational costs: text generation and watermarking. The generation cost is $\mathcal{C}_g(P,T) = \mathcal{C}_{\text{prefill}}(P) + \sum_{t=1}^{T} \mathcal{C}_{\text{decode}}(P+t)$, including prompt prefill and autoregressive decoding of T tokens. Maintaining the segment buffer and detecting segment boundaries adds $\mathcal{O}(T)$ lightweight overhead. Thus, $\mathcal{C}_{\text{gen}} = \mathcal{O}\left(\mathcal{C}_g(P,T) + T\right)$.

For watermarking, each segment s_i is processed using $P_i = [P_{\text{shared}}; s_i]$. Without prefix caching, candidate generation repeatedly processes the shared instruction and generates the structured candidate mapping: $\sum_{i=1}^{N_s} \left[\mathcal{C}_{\text{prefill}}\left(|P_{\text{shared}}| + |s_i|\right) + \sum_{g=1}^{G_i} \mathcal{C}_{\text{decode}}\left(|P_{\text{shared}}| + |s_i| + g\right)\right]$.

With prefix caching, the shared prefix is computed once and reused across watermark requests, reducing repeated prefill computation. If $\mathcal{I}_i$ denotes the watermarkable positions in s_i, each with at most M candidates, key-conditioned selection adds $\mathcal{O}\left(\sum_{i=1}^{N_s} |\mathcal{I}_i| M\right)$. Therefore, the watermarking complexity can be expressed as

$$\mathcal{C}_{\text{wm}} = \mathcal{O}\left(\mathcal{C}_{\text{prefill}}(|P_{\text{shared}}|) + \sum_{i=1}^{N_s} \mathcal{C}_{\text{prefill}}(|s_i|) + \sum_{i=1}^{N_s}\sum_{g=1}^{G_i} \mathcal{C}_{\text{decode}}\left(|P_{\text{shared}}| + |s_i| + g\right) + \sum_{i=1}^{N_s} |\mathcal{I}_i| M\right). \tag{3}$$

The adaptive scheduler introduces only constant-time queue-pressure and admission computations at each update and therefore does not change the asymptotic complexity. Overall computational work remains $\mathcal{O}(\mathcal{C}_{\text{gen}} + \mathcal{C}_{\text{wm}})$. However, generation and watermark requests are asynchronously co-served within the same inference engine, allowing their execution to overlap through mixed continuous batching. Consequently, observed wall-clock latency depends on the adaptive admission ratio and runtime workload rather than following the strictly sequential $\mathcal{C}_{\text{gen}} + \mathcal{C}_{\text{wm}}$ execution of generate-then-watermark pipelines (Dang et al., 2026).

Space Complexity. SWIFT uses one unified LLM and shared inference engine for both generation and watermarking. Its dominant GPU memory cost is therefore $\mathcal{O}\left(M_{\text{LLM}} + M_{\text{KV}}\right)$, where M_{LLM}

denotes model-weight memory and M_{KV} denotes the shared KV cache for active generation and watermark requests, including cached prefix states.

Additional state consists of partial generation segments, the global watermark queue, and active candidate mappings. Let $\mathcal{A}_{\mathrm{gen}}$ and $\mathcal{A}_{\mathrm{wm}}$ denote the active generation and watermark requests, respectively. The total space complexity is

$$\mathcal{O}\left(M_{\mathrm{LLM}} + M_{\mathrm{KV}} + \sum_{u \in \mathcal{A}_{\mathrm{gen}}} |\mathcal{B}_u| + \sum_{s_i \in \mathcal{Q}^{\mathrm{wm}}} |s_i| + \sum_{i \in \mathcal{A}_{\mathrm{wm}}} |\mathcal{I}_i| M\right), \tag{4}$$

where $\mathcal{B}_u$ is the partial segment buffer for generation request u and $\mathcal{Q}^{\mathrm{wm}}$ is the global watermark queue. Candidate mappings are released after key-conditioned selection. Unlike pipelines that require separate auxiliary models or watermarking engines, SWIFT maintains only one copy of the LLM weights on the shared GPU.

B Additional Experimental Details and Results

Implementation Backends and Hardware. All experiments use a single NVIDIA A100 80GB GPU, batch size 32, and KV-cache block size 16. For fair comparison, all baselines use the same vLLM backend, model, and generation settings. KGW (Kirchenbauer et al., 2023) uses vLLM's logits-processing interface, while SynthID (Dathathri et al., 2024) adapts the vLLM sampler following vLLM-Watermark (Verma, 2025). For SafeSeal (Dang et al., 2026), text is generated with vLLM before applying its post-processing watermarking procedure. This minimizes backend differences and enables consistent comparison of watermarking overhead. For SWIFT, the adaptive scheduler updates the watermark admission ratio every $K = 5$ scheduling steps, with $r_{\min} = 0$, $r_{\max} = 1$, queue threshold $B_{\max} = 32$, and oldest-request waiting time threshold $W_{\max} = 1.5$s. These settings are fixed unless otherwise noted.

Evaluation Metrics. We use diverse evaluation metrics to assess watermark utility, detectability, robustness, and serving efficiency, covering both output quality and system-level performance.

Utility. Utility measures preservation of meaning and factual content. We use GPT-5.5 (OpenAI, 2026) as an LLM-as-a-judge $J(\cdot)$, assigning each original output y and watermarked output y^{wm} a score from 0-5, with higher scores indicating better preservation. The same judge performs pairwise attack analysis by comparing attacked outputs with their original watermarked versions and reporting the attacked outputs' win rate. The mean utility score over N samples is:

$$\text{UtilityScore} = \frac{1}{N}\sum_{n=1}^{N} J(y_n, y_n^{wm}). \tag{5}$$

Detectability. Detectability measures whether a detector can distinguish watermarked outputs from non-watermarked outputs. To account equally for both classes, we define detection accuracy as the average of the detector's accuracy on watermarked (y_i^{wm}) and non-watermarked (y_j) samples:

$$\text{DetectionAccuracy} = \frac{1}{2}\left[\frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(\mathcal{D}(y_i^{wm}, k) = 1\right) + \frac{1}{H}\sum_{j=1}^{H}\mathbb{I}\left(\mathcal{D}(y_j, k) = 0\right)\right], \tag{6}$$

where $\mathcal{D}(\cdot)$ is the detector with key k, $\mathbb{I}(\cdot)$ is the indicator function, N and H denote the numbers of watermarked and non-watermarked samples, respectively. Higher detection accuracy is better.

Robustness. Robustness evaluates whether a watermark remains detectable after its output is modified by an attack. For an attack $\mathcal{A}$, we compute the post-attack detection accuracy as

$$\text{DetectionAccuracy}_{\mathcal{A}} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(\mathcal{D}(\mathcal{A}(y_i^{wm}), k) = 1\right), \tag{7}$$

where $\mathcal{A}(y_i^{wm})$ is the attacked watermarked output. Higher accuracy indicates greater robustness.

Latency. We report amortized end-to-end serving time per request, computed as the total wall-clock time to process the serving workload divided by the number of requests. This measure accounts

for concurrent batched execution and includes text generation, watermark construction, and output emission. Lower values indicate higher serving efficiency:

$$L = \frac{t^{\text{end}} - t^{\text{begin}}}{N_{\text{req}}}, \tag{8}$$

where t^{begin} and t^{end} denote the start and completion times of the serving workload, respectively, and N_{req} is the number of requests.

Throughput. Throughput measures the number of generated output tokens processed per second over the serving workload, with higher throughput indicating better serving efficiency:

$$TP = \frac{\sum_{n=1}^{N_{\text{req}}} T_n}{t^{\text{end}} - t^{\text{begin}}}, \tag{9}$$

where T_n is the number of generated tokens for request n.

Peak GPU Memory. Peak GPU memory measures the maximum GPU memory occupied during generation and watermarking. Lower peak memory indicates a smaller deployment footprint.

GPU KV-Cache Usage. GPU KV-cache usage measures the fraction of the engine's KV-cache capacity occupied by active requests. Unlike peak GPU memory, it reflects only the KV-cache pool. Lower usage indicates greater headroom for additional concurrent requests.

Prefix-Cache Hit Rate. Prefix-cache hit rate measures the fraction of reusable prompt-prefix computation served from the prefix cache rather than recomputed during prefill. A higher hit rate indicates greater reuse of previously computed prefix states and reduced redundant computation.

Service TTFT. Service time-to-first-token (TTFT) measures the time from request admission to the first output token, excluding pre-admission waiting. Lower Service TTFT is preferred.

E2E TTFT. End-to-end time-to-first-token (E2E TTFT) measures the time from request submission to the first output token. Unlike Service TTFT, it includes queueing and scheduling delays, providing a workload-level measure of responsiveness. Lower E2E TTFT is preferred.

TPOT. Time per output token (TPOT) measures the average time required to produce each output token after the first token, with lower TPOT indicating faster autoregressive decoding:

$$TPOT = \frac{t^{\text{final}} - t^{\text{first}}}{T - 1}, \tag{10}$$

where t^{first} and t^{final} denote the timestamps of the first and final output tokens, respectively, and T is the number of generated tokens.

Additional Evaluation Setup. Unless otherwise specified, all results are computed over $1,000$ samples. In the C4 text generation, we use the first 50 tokens of each C4 sample as the prompt and generate up to 200 output tokens. In the CNN/Daily Mail Summarization, we use a fixed few-shot summarization prompt and instruct the model to generate a summary of at most 60 words. In the GSM8K reasoning task, the model is prompted to generate a reasoning trace of at most 50 tokens followed by a final answer. Task accuracy is computed by comparing the extracted final answer with the ground-truth answer. In the MedMCQA, we use the same reasoning-and-final-answer format as GSM8K but allow explanations of up to 100 tokens. We retain only single-answer questions with a valid correct option and a non-empty explanation longer than 50 words to ensure sufficient text for watermarking and evaluation.

For watermark detection, we train the RoBERTa-large (356M) (Liu et al., 2019) key-conditioned detector once on $20,000$ C4 samples and evaluate it on $1,000$ held-out samples. Training and evaluation prompts come from the official C4 training and validation splits, ensuring no overlap. The same detector is reused without retraining for summarization, mathematical reasoning, and medical QA, enabling cross-domain and cross-task evaluation under distribution shift.

For utility evaluation, we use GPT-5.5 (OpenAI, 2026) as an LLM-as-a-judge to evaluate semantic and factual preservation between the original and watermarked outputs using the fixed prompt shown in Fig. 10. The reported utility is the average score over all evaluated samples.

For serving evaluation, each method is evaluated over $1,000$ watermarked requests under the same workload. Throughput, GPU KV-cache usage, and prefix-cache hit rate are collected from standard

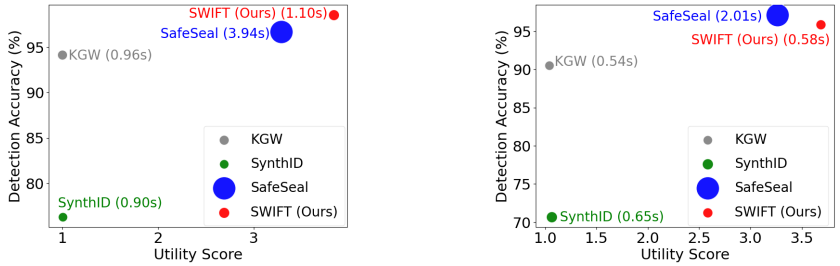
vLLM runtime statistics with log statistics enabled, while TTFT and TPOT are measured using the standard vLLM benchmark (`vllm bench serve`). For asynchronous co-serving systems such as SWIFT, these engine-level metrics may reflect both text and watermark workloads; we therefore treat throughput and E2E TTFT mainly as supplementary serving metrics.

Statistical Significance. To assess runtime stability and repeatability, we repeat the SWIFT experiment over 30 independent runs under the same setting (Table 5). SWIFT achieves an average latency of 0.9039 ± 0.0291s, with a narrow 95% confidence interval of $[0.8930\text{s}, 0.9148\text{s}]$. A one-sample t-test against the reported latency of 0.905s gives $p = 0.834$, indicating no statistically significant difference from the reported value and confirming stable runtime performance.

Table 5: Repeatability of SWIFT latency over 30 independent runs. Utility, detectability, and peak memory remain unchanged under the deterministic setting; latency varies due to system runtime.

Metric	Latency (s)
Number of runs	30
Mean $\pm$ Std.	0.9039 ± 0.0291
95% CI	$[0.8930, 0.9148]$
One-sample t-test vs. 0.905s	$p = 0.834$

Watermark Effectiveness across LLMs. We further evaluate SWIFT on Qwen2.5-32B (Yang et al., 2024). As shown in Fig. 7, a similar trend is observed for both text generation and summarization. On text generation, SWIFT achieves the highest utility (3.837) and detectability (98.56%) with competitive latency of 1.102s. On summarization, it again attains the highest utility (3.685) with 95.9% detectability at low latency of 0.58s, substantially faster than SafeSeal. These results indicate that SWIFT's effectiveness and efficiency generalize beyond the primary LLM.



(a) Text generation (b) Text summarization

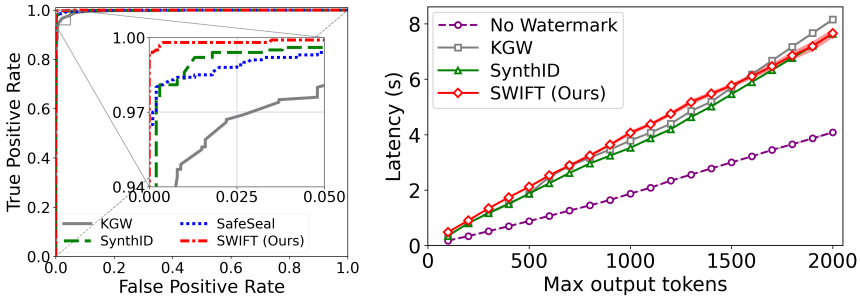
Figure 7: Watermark effectiveness on text generation and summarization on Qwen model. Circle area is proportional to latency, where smaller circles indicate lower latency.

Performance on Medical Text. Table 6 provides detailed results on MedMCQA. SWIFT preserves the non-watermarked accuracy of 62.6% (ΔAcc.=0) while achieving 4.70 utility and 96.70% detection at 0.401 s latency. SynthID and SafeSeal also preserve accuracy but achieve lower utility, while KGW slightly reduces accuracy. Although SafeSeal reaches the highest detection rate (97.90%), it incurs substantially higher latency and lower utility. These results show that SWIFT maintains task correctness while balancing utility, detectability, and serving efficiency on constrained medical text.

Table 6: Performance on medical MedMCQA dataset. Accuracy is measured against the ground-truth answer, while ΔAcc. denotes the relative accuracy change to non-watermarked baseline.

Setting	Accuracy (%) ↑	ΔAcc. (%) ↑	Utility ↑	Detection (%) ↑	Latency (s) ↓
No Watermark	62.6	–	–	–	0.205
KGW	62.3	-0.48	2.76	60.80	0.597
SynthID	**62.6**	**0.00**	2.97	66.25	0.416
SafeSeal	**62.6**	**0.00**	3.14	**97.90**	1.732
SWIFT (Ours)	**62.6**	**0.00**	**4.70**	96.70	**0.401**

ROC Analysis. Fig. 8(a) further confirms the strong detection performance of SWIFT. It achieves an AUC of 0.9999, compared with 0.9960 for KGW, 0.9987 for SynthID, and 0.9989 for SafeSeal. At a strict 1% FPR, SWIFT maintains a 99.8% TPR, increasing to 99.9% at 5% FPR. These results indicate strong separation between watermarked and non-watermarked outputs, even at low false-positive operating points. Notably, SynthID achieves a high AUC despite its lower detection accuracy in Section 4.2. This difference arises because AUC is threshold-independent and measures the overall separation between the two classes, whereas detection accuracy depends on the selected decision threshold. Overall, all methods perform strongly in the attack-free setting, with SWIFT achieving the highest AUC and maintaining high TPR at low FPR.



(a) ROC Analysis (b) Scalability

Figure 8: ROC analysis and latency scalability over output length.

Scalability over Text Length. Fig. 8(b) evaluates end-to-end latency as the maximum output length increases from 100 to 2,000 tokens, with least-squares fits showing strong linearity ($R^2 \geq 0.994$). Over this $20\times$ increase in length, SWIFT's latency increases by only $15.9\times$, indicating less-than-proportional growth over the evaluated range. Its fitted slope is 3.697 ms/token, comparable to SynthID (3.728) and KGW (3.953), demonstrating competitive scaling with output length. SafeSeal is omitted from the plot because its substantially higher baseline latency obscures comparison among the remaining methods. SWIFT's scaling behavior is supported by asynchronous co-serving, reuse of the fixed watermark instruction prefix, and selective watermarking over only a subset of watermarkable words within each completed segment. In contrast, in-processing methods such as KGW and SynthID apply watermarking decisions at every autoregressive decoding step. Since SWIFT's cost depends on content-specific factors such as sentence structure and the number of watermarkable positions, we interpret the results as showing favorable scaling comparable to token-level in-processing methods rather than universally superior scaling.

Task Characteristics and Watermarking Opportunities. Table 7 summarizes the output characteristics and watermarking opportunities across tasks. We define watermark density as the fraction of output words selected for watermarking. C4 text generation produces substantially longer outputs with more sentences and watermarkable words, resulting in more segment-level watermark requests and greater watermarking workload. In contrast, GSM8K reasoning and Medical QA outputs are shorter and provide more constrained substitution choices, resulting in fewer watermark requests and lower watermark density.

Table 7: Output characteristics and watermarking opportunities across evaluation tasks. Watermark density denotes the percentage of output words selected for watermarking.

Dataset	Average Tokens	Average Words	Average Sentences	Watermarked Words	Watermark Density (%)	Detectability (%)
Text Generation	197.74	143.89	10.37	58.29	40.51	99.65
Summarization	61.30	46.59	3.11	16.77	35.99	98.80
Reasoning	48.03	38.51	3.76	8.56	22.23	90.70
Medical QA	83.58	59.24	3.23	17.02	28.73	96.70

These characteristics help explain the task-dependent latency of SWIFT. Because SWIFT issues watermark requests only for completed sentence segments and selectively modifies watermarkable words, short and constrained outputs generate fewer watermark requests and consume less capacity in the shared serving batch. In contrast, in-processing methods like KGW and SynthID perform watermarking during autoregressive decoding for every token. This helps explain why SWIFT achieves lower latency than KGW and SynthID on reasoning and medical text (Tables 1 and 6), with the largest advantage on reasoning, which has the lowest watermark density. The same characteristics also help explain detectability. Lower watermark density provides fewer opportunities to embed the watermark signal, leading to lower detection rates on the more constrained tasks. As shown in Table 7, detectability generally increases with watermark density, from 90.70% on reasoning text with 22.23% density to 99.65% on C4 with 40.51% density.

Additional Qualitative Examples with Utility Evaluation.

On Medical Question-Answering Text. Table 8 presents the LLM-as-a-judge utility evaluation on a MedMCQA example. Although all methods preserve the correct answer, KGW, SynthID, and SafeSeal introduce explanatory, semantic, or factual distortions, receiving utility scores of 3, 2, and 2, respectively. In contrast, SWIFT preserves key medical facts, entities, and relations through meaning-preserving substitutions, achieving the maximum utility score of 5.

Table 8: Qualitative example of the LLM-as-a-judge utility evaluation on MedMCQA. Green marks preserved facts, light blue marks meaning-preserving changes, and red marks semantic or factual distortions. Utility is scored from 0–5 based on semantic and factual preservation.

Setting	Content	Final Answer	Utility	Judge Explanation
Question	Absorption of which of the following drugs is increased after a fatty meal? **A.** Amphotericin B, **B.** Griseofulvin, **C.** Ampicillin, **D.** Aspirin	N/A	N/A	N/A
Ground Truth	**Answer: B. Griseofulvin.** The oral administration of a 0.5 g dose of griseofulvin produces peak plasma concentrations of 1 μg/mL in 4 hours. Its absorption is improved when administered with a fatty meal.	**B**	N/A	N/A
No watermark	**Answer: B.** Griseofulvin is a lipophilic antifungal drug. Its absorption from the gastrointestinal tract is poor and significantly enhanced by concurrent intake of a fatty meal, which increases bile secretion and improves its solubility.	**B**	N/A	N/A
KGW	**Answer: B.** Griseofulvin is a lipophilic antifungal medication. Its absorption from the gastrointestinal tract is significantly enhanced by concurrent intake of a fatty meal, as it increases bile secretion, which aids in its dissolution and absorption. Amphotericin B has poor oral absorption regardless of food intake, ampicillin absorption is generally unaffected, and aspirin absorption might be slightly decreased with food due to potential gastric irritation.	**B**	3	The medical answer is preserved, but some explanatory details differ or are omitted.
SynthID	**Answer: B.** Griseofulvin is a lipophilic antifungal drug with poor water solubility. Its absorption from the gastrointestinal tract is significantly enhanced by concurrent ingestion with a fatty meal, as the fat facilitates its dissolution and transport across the intestinal membrane. Amphotericin B absorption is actually decreased with food, while ampicillin and aspirin absorption are generally not affected by dietary fat content.	**B**	2	The answer matches, but important explanatory medical content is substantially changed or missing.
SafeSeal	**Answer: B.** Griseofulvin is a lipophilic antifungal pill. Its emission from the gastrointestinal tract is severe and severely strengthened by concurrent ingest of a fatty plate, which alters bile removal and improves its solubility.	**B**	2	Core meaning is broadly preserved, but several substitutions alter important medical wording and relations.
SWIFT (Ours)	**Answer: B.** Griseofulvin is a lipophilic antifungal drug. Its absorption from the gastrointestinal tract is inadequate and significantly improved by simultaneous intake of a fatty meal, which raises bile secretion and facilitates its solubility.	**B**	**5**	Semantically equivalent; key facts and entities are preserved.

On Short Reasoning Text. Table 9 provides a GSM8K example illustrating the effect of watermarking on reasoning quality and the corresponding LLM-as-a-judge utility assessment. KGW and SynthID alter the arithmetic reasoning and produce incorrect final answers, receiving utility scores of 1. SafeSeal preserves the correct answer but introduces semantically inappropriate substitutions such as "lights" and "seats," resulting in a utility score of 3. In contrast, SWIFT preserves the reasoning process, quantities, relations, and final answer while applying meaning-preserving lexical substitutions, receiving the maximum utility score of 5. This example further demonstrates SWIFT's ability to maintain task correctness and semantic fidelity under short reasoning text.

Table 9: Math reasoning utility example on GSM8K comparing watermarked outputs with the original response. Green marks correct reasoning and factual consistency, light blue marks semantic-preserving word changes, and red marks reasoning errors or meaning-altering modifications.

Question: Hannah has a mental breakdown while studying for finals and starts smashing windows in the high school's parking lot. She smashes a quarter of the students' cars' windows and 3/4ths of the teachers' cars' windows. If there are 64 students' cars with four windows each and 32 teachers' cars with two windows each, how many windows does Hannah smash?

Method	Reasoning Answer	Final Answer	Utility	Judge Explanation
No Watermark	Hannah smashes $64/4 * 4 = 16 * 4 = 64$ student car windows. Hannah smashes $32/4 * 2 * 3 = 8 * 2 * 3 = 48$ teacher car windows. Hannah smashes $64 + 48 = 112$ windows. Answer: 112	**112**	N/A	N/A
KGW	Hannah smashes $64/4 * 4 = 16$ students' car windows. Hannah smashes $32/4 * 2 * 3 = 24$ teachers' car windows. In total, Hannah smashes $16 + 24 = 40$ windows. Answer: 40	**40**	1	The final answer changes, so the conclusion is contradictory.
SynthID	Hannah smashes $64/4 * 4 = 16 * 4 = 64$ student car windows. Hannah smashes $32/4 * 2 = 8 * 2 = 16$ teacher car windows. Hannah smashes $64 + 16 = 80$ windows. Answer: 80	**80**	1	The final conclusion changes, creating a contradiction in an important fact.
SafeSeal	Hannah smashes $64/4 * 4 = 16 * 4 = 64$ student car lights. Hannah knocks $32/4 * 2 * 3 = 8 * 2 * 3 = 48$ teacher car seats. Hannah smashes $64 + 48 = 112$ windows. Answer: 112	**112**	3	The quantities and final answer are preserved, but key objects are changed, altering important semantic content.
SWIFT (Ours)	Hannah breaks $64/4 * 4 = 16 * 4 = 64$ pupil car windows. Hannah breaks $32/4 * 2 * 3 = 8 * 2 * 3 = 48$ instructor car windows. Hannah breaks $64 + 48 = 112$ windows. Answer: 112	**112**	**5**	Meaning, quantities, relations, and key facts are preserved.

Additional Key Serving Evaluation.

Serving Performance on Text Generation. Table 10 demonstrates SWIFT's serving efficiency. SWIFT achieves the highest throughput at 528.7 tokens/s, corresponding to $2.64\times$ KGW, $2.26\times$ SynthID, and $14.1\times$ SafeSeal. It also reduces GPU KV-cache usage to 12.8%, about 19.5% lower than KGW and SynthID, while achieving a 96.8% prefix-cache hit rate compared with nearly 0% for the baselines. This high hit rate results from SWIFT's repeated watermark instruction prefix, allowing vLLM to reuse matching KV-cache blocks across watermark requests. In contrast, baseline C4 prompts share few identical full cache blocks (16 tokens by default), yielding little prefix reuse (vLLM Team, 2025). Prefix reuse both avoids redundant prefill computation and reduces duplicate KV-cache allocation; additionally, watermark requests have short variable suffixes, limiting their incremental KV-cache footprint. Despite its substantially higher throughput, SWIFT matches the peak memory usage of KGW and SynthID and uses approximately 8.5% less memory than SafeSeal by sharing a single unified model across text generation and watermark construction.

Table 10: Serving efficiency comparison across watermarking methods.

Methods	Throughput (Tokens/s) ↑	GPU KV Cache (%) ↓	Prefix Cache Hit Rate (%) ↑	Peak Memory (MB) ↓
KGW	200.3	15.90	0.00	**74169**
SynthID	234.2	15.90	0.00	**74169**
SafeSeal	37.4	15.47	0.00	81100
SWIFT (Ours)	**528.7**	**12.80**	**96.80**	**74169**

Serving Performance on Short Reasoning Text. Table 11 shows that SWIFT maintains strong serving efficiency on short reasoning workloads. SWIFT achieves the highest throughput at 481.8 tokens/s, approximately $2.56\times$ SynthID and $13.43\times$ SafeSeal, while also obtaining the lowest GPU KV-cache usage (12.7%) and the highest prefix-cache hit rate (96.68%). Unlike C4, the baselines also achieve high prefix-cache hit rates (about 87%) because GSM8K requests share a fixed few-shot reasoning prompt to instruct the model to generate short reasoning text with final answer at the end, creating repeated prefixes that span reusable 16-token cache blocks (vLLM Team, 2025). SWIFT further benefits from its repeated watermark instruction prefix, yielding additional KV-state reuse beyond the shared task prompt. These results support that SWIFT's prefix-caching advantage persists even when the underlying task already provides substantial prefix reuse.

Table 11: Serving performance on GSM8K short reasoning text. Higher throughput and prefix-cache hit rate are preferred, while lower GPU KV-cache usage is better.

Method	Throughput (Tokens/s) ↑	Running Requests ↑	GPU KV Cache (%) ↓	Prefix Cache Hit Rate (%) ↑
KGW	162.35	31.09	13.46	87.59
SynthID	187.99	**31.56**	12.94	87.31
SafeSeal	35.88	31.00	16.70	87.40
SWIFT (Ours)	**481.80**	31.10	**12.70**	**96.68**

Additional Serving Evaluation. Table 12 reports Service TTFT, E2E TTFT, and TPOT as supplementary serving measurements. These metrics are primarily defined for token-streaming autoregressive serving and are not fully architecture-independent across watermarking methods. In SWIFT, text-generation and watermark-generation requests are co-served by the same engine; consequently, aggregate benchmark statistics may reflect both request classes rather than isolating user-facing text-generation latency alone. Moreover, SWIFT emits completed watermarked segments, whereas KGW and SynthID operate directly in the token-generation path. We therefore report these values for reference rather than as the primary serving comparison. For SafeSeal, only Service TTFT is reported: its generate-then-post-process design does not expose the token-level timestamps required to obtain comparable E2E TTFT and TPOT from the vLLM serving benchmark.

Table 12: Serving latency comparison. Service TTFT measures time to first token after request admission, while E2E TTFT includes end-to-end request waiting and serving time. TTFT and TPOT are reported as mean values, where lower is better.

Method	Service TTFT (ms) ↓	E2E TTFT (ms) ↓	TPOT (ms) ↓
No Watermark	128.25	140,015.59	46.63
KGW	495.70	441,509.15	172.79
SynthID	**412.50**	**284,021.78**	91.52
SafeSeal	5,342.00	N/A	N/A
SWIFT	483.36	359,221.32	**59.65**

C TEMPLATES

Instruction Prefix. Fig. 9 provides a template of the shared instruction prefix used in SWIFT for candidate generation. The template includes semantic preservation constraints, entity-aware guidance, general generation rules, and few-shot examples that steer the LLM toward producing context-aware watermark candidates while maintaining consistency with the original text.

Instruction-Guided Candidate Generation Prompt

Shared Instruction Prefix

The shared prefix defines the reusable candidate-generation policy for all sentences in the batch.

Target rule	Select only watermarkable words, which can be naturally replaced, such as nouns, verbs, adjectives, and adverbs.
Named entity protection	Exclude named entities, including names, organizations, products, titles, dates, quantities, identifiers, and model numbers. Avoid capitalized words.
General Instruction	Generate 2 - 8 direct synonym candidates that preserve meaning, factuality, tone, tense, number, degree, and register. Candidates can be a single word or a phrase. Do not generate awkward, duplicate, loose, broader, narrower, or antonym candidates.
Few-shot examples	**Example 1:** ***Sentence:*** On Tuesday, Dr. Sarah Chen from Stanford University delivered a clear presentation about climate policy, and the audience responded warmly to her practical recommendations. ***Output:*** {"clear":["concise","understandable","lucid"], "presentation":["talk", "lecture","briefing"], "responded":["reacted","replied"], "warmly":["enthusiastically","favorably"], "practical":["useful","realistic","workable"], "recommendations":["suggestions","proposals","advice"]} **Example 2:** *(additional examples omitted for brevity)* **Example 3:** *(additional examples omitted for brevity)*

Varying Sentence Suffix

Sentence: {`completed sentence from asynchronous text generation stream`}

Structured Output

```
{
  "handle": ["manage", "oversee", "take care of"],
  "detailed": ["thorough", "comprehensive"]
}
```

Output only a valid JSON object; no markdown, explanation, or extra text.

Figure 9: Instruction prompt format for candidate generation in SWIFT. The shared instruction prefix contains the watermarkable word detection, named entity rules, general instructions, and few-shot examples. The varying suffix contains the current sentence, enabling many sentence-level requests to reuse the same shared prefix during continuous batching.

LLM-as-a-Judge Utility. Fig. 10 shows the utility evaluation prompt. The judge compares each watermarked output with its original response and assigns a score from 0 to 5 based on semantic and factual preservation, including named entities, numbers, relations, actions, and key details. The prompt prioritizes content over stylistic similarity and requires structured JSON output with a score and brief justification. We report the average score across evaluated samples.

LLM-as-a-Judge Utility Evaluation Prompt

```
SYSTEM_PROMPT = """
You are an expert evaluator of semantic and factual consistency.

Given an original text (Text A) and a watermarked text (Text B),
evaluate how well Text B maintains the core meaning and factual
content represented in Text A.

Text B does not need to reproduce the exact wording, sentence structure,
or level of detail of Text A. Different but semantically compatible
expressions or formulations should not be penalized.

Consider:
1. Overall semantic meaning and consistency.
2. Named entities, including people, organizations, locations,
   products, titles, and technical terms.
3. Whether important information is contradicted, omitted, or
   materially changed.
4. Numbers, dates, quantities, units, relations, actions, roles,
   and polarity.

Scoring rubric:
0 = Completely unrelated, contradictory, or the core meaning is not preserved.
1 = Same general topic, but most important meaning or facts differ.
2 = Partial semantic consistency, with substantial important information
    changed, contradicted, or missing.
3 = Mostly maintains the core meaning, but some important details,
    entities, or facts differ.
4 = Maintains the core meaning and factual content, with only minor
    semantic or factual differences.
5 = Semantically and factually consistent; key entities, numbers,
    relations, and important information are maintained, even if the
    wording, structure, or presentation differs.

Important:
- If Text B directly contradicts important information in Text A,
  assign a score of at most 1.
- Do not penalize Text B simply for using different wording, sentence
  structure, phrasing, or level of detail.
- Do not require lexical or stylistic similarity between the two texts.
- Focus on substantive semantic and factual differences rather than
  surface-form differences.
- Treat Text A as a reference for semantic and factual consistency,
  not as a template that Text B must closely reproduce.

Return only a JSON object in the following format:
{
  "score": <integer from 0 to 5>,
  "reason": "<brief justification>"
}
"""

USER_PROMPT = """
Text A:
{original_text}

Text B:
{watermarked_text}
"""
```

Figure 10: Prompt template used for LLM-as-a-judge utility evaluation.