

Anticipatory Prefill with Keystroke Streaming for Interactive Text-to-SQL

Riad Dalaoui, Rayan Hanader, Sami Taider, Rémi Vialleton, Shuyan Wang, Lizhi Zhang
INSA Lyon, Department of Computer Science, France

Abstract—Large Language Models (LLMs) have achieved remarkable accuracy in Text-to-SQL tasks, yet their deployment in interactive applications is hindered by inference latency. The Time-To-First-Token (TTFT), the delay between query submission and the first generated token, creates perceptible lag that degrades user experience. We observe that traditional inference wastes user typing time: while users compose queries over several seconds, the system remains idle, only beginning computation after submission. We propose Anticipatory Prefill with keystroke streaming for more Interactive Text-to-SQL (APITS), a technique that overlaps prefill computation with user typing by streaming keystrokes to the backend and incrementally building the KV-cache. Our system maintains a single evolving cache that extends with confirmed tokens and supports efficient rollback through cache cropping when users edit their input. We implement this approach using HuggingFace Transformers with Qwen2.5-Coder-3B and evaluate on Spider benchmark databases. Results demonstrate significant TTFT reduction while preserving SQL generation accuracy, with the cache already containing the majority of required tokens at submission time.

Index Terms—Text-to-SQL, KV-cache, inference optimization, large language models, real-time systems, WebSocket streaming

I. INTRODUCTION

Text-to-SQL systems enable non-technical users to query databases using natural language, democratizing data access across organizations. Recent advances in Large Language Models have dramatically improved the accuracy of these systems, with state-of-the-art models achieving over 80% execution accuracy on complex benchmarks like Spider [1]. However, deploying these models in interactive settings introduces a critical challenge: **inference latency**.

The inference process for autoregressive LLMs consists of two phases: (1) *prefill*, where all input tokens are processed to build the key-value (KV) cache, and (2) *generation*, where output tokens are produced one at a time using the cached context. For Text-to-SQL applications, the input typically includes a system prompt, database schema description, and user query, often totaling hundreds or thousands of tokens. The prefill phase for such inputs can take several hundred milliseconds, creating noticeable delay between the user pressing “Enter” and seeing the first response token.

Our key observation is that user typing time represents a significant, yet unexploited, computational opportunity. When a user types a natural language query, typically taking several seconds to a few minutes for slow typing-while-thinking users, the backend system traditionally remains idle, waiting for the complete submission. Only then does prefill begin, adding latency on top of the user’s perceived wait time.

We propose **Anticipatory Prefill with keystroke streaming for more Interactive Text-to-SQL**, a technique that reclaims this wasted time by:

- 1) Streaming user keystrokes in real-time via WebSocket
- 2) Incrementally building the KV-cache as tokens are confirmed
- 3) Supporting efficient rollback when users edit or delete text
- 4) Ensuring output correctness identical to standard inference

Unlike speculative approaches that maintain multiple cache branches, our design uses a **single evolving cache**, avoiding memory explosion while supporting the dynamic nature of user input. The result is that when the user submits their query, the cache is already warm with most or all of the prompt tokens, enabling near-instantaneous first-token generation.

Our Contributions may be shortened to:

- A **real-time keystroke streaming architecture** using WebSocket for fast character transmission latency
- An **incremental KV-cache management system** with three core operations: *initialize*, *extend*, and *crop*
- A **boundary-based token confirmation strategy** that commits tokens at word boundaries with configurable debouncing
- A **comprehensive evaluation framework** measuring TTFT, SQL correctness, and resource utilization on Spider benchmark databases and an M4-chip 16GB consumer Macbook Air.

II. BACKGROUND AND RELATED WORK

A. KV-Cache in Transformer Inference

Modern LLMs use the Transformer architecture [2], where each layer computes attention over previous positions. The **KV-cache** stores key and value projections from previous positions, avoiding $O(n^2)$ recomputation per token. Inference divides into: (1) *prefill*, processing all input tokens to build the cache, and (2) *generation*, producing tokens using cached context. For Text-to-SQL with 500–2000 token inputs, prefill dominates TTFT at 200–800ms on consumer GPUs.

B. Related Work

GQA [3] reduces KV-cache memory via shared key-value heads. **PagedAttention** [4] enables non-contiguous cache al-

location. **Speculative Decoding** [5] accelerates generation (not prefill) using draft models. **Prompt Caching** reuses caches for shared prefixes, our work extends this to incrementally growing user input. **StreamingLLM** [6] handles infinite contexts but targets memory, not latency. Modern Text-to-SQL leverages LLMs with schema-aware prompts; the Spider benchmark [1] drives accuracy improvements, but interactive latency remains underexplored.

III. SYSTEM DESIGN

A. Architecture Overview

The system architecture (see Appendix A) consists of three layers: a React frontend captures user keystrokes and streams them via WebSocket to a FastAPI backend. The `StreamController` orchestrates session state, managing the timing of cache updates. The `KVCacheManager` executes low-level cache operations using HuggingFace Transformers’ `DynamicCache` API.

B. Session State Management

Each typing session maintains the following state:

- **Base prompt** P_0 : System instructions and database schema, prefilled at session initialization
- **KV-cache** \mathcal{C} : A `DynamicCache` object containing attention keys/values for all confirmed tokens
- **Confirmed text** T_c : User input that has been tokenized and added to \mathcal{C}
- **Pending text** T_p : Characters typed but not yet committed (typically the current word)
- **Token count** n : Number of tokens currently in \mathcal{C}

A critical design choice is maintaining **exactly one cache instance**. Unlike speculative execution that branches into multiple possible futures, we commit only to tokens that are sufficiently stable, avoiding memory multiplication.

C. Token Confirmation Strategy

Not every keystroke should trigger a GPU forward pass, this would create excessive overhead and potentially commit unstable tokens that the user might delete or that might be wrongly calculated. For instance, let’s imagine the user is typing the word ‘analogy’, if we commit at every letter, then the first ‘a’ would be treated as a single token by the forward pass, therefore having the meaning of the article ‘a’. If we committed every two letters, then the ‘an’ would also be treated as a single token holding the meaning of the word ‘an’. To avoid these issues, we employ a two-part confirmation strategy:

1) Boundary-based confirmation: Tokens are committed only when the user completes a “natural unit” of input. We define confirmation boundaries as positions following whitespace or punctuation:

$$\mathcal{B} = \{ ' ', '\n', '\t', '.', ',', ';', ':', '!', '? ' \} \quad (1)$$

When the user types a boundary character, all preceding complete words become candidates for confirmation, and are

all forward passed together therefore letting the model compute the optimal token separation. The ‘analogy’ word would therefore become the tokens ‘an’, ‘alog’ and ‘y’, holding the optimal semantic meaning.

2) Debounced triggering: Even after a boundary, we wait for a configurable debounce period δ (default: 300ms) before executing the cache extension. If the user continues typing within δ , the timer resets. Additionally, consecutive boundary characters (e.g., end of sentence followed by space) trigger immediate confirmation to capture natural pauses. This helps us keep phrases meanings altogether.

This strategy balances responsiveness against wasted computation from premature commits.

D. Cache Operations

The `KVCacheManager` implements three core operations:

1) Initialize: At session start, the base prompt P_0 (containing system instructions and schema) is tokenized and processed through the model:

$$\mathcal{C}_0, n_0 = \text{prefill}(P_0) \quad (2)$$

This cache can be persisted to disk (keyed by SHA-256 hash of P_0) and reloaded for subsequent sessions with identical base prompts, amortizing the initial prefill cost.

2) Extend: When new confirmed text ΔT is available, we compute the incremental tokens and extend the cache:

Algorithm 1 Cache Extension

- 1: $T_{new} \leftarrow T_c + \Delta T$
 - 2: $\text{ids}_{new} \leftarrow \text{tokenize}(P_0 + T_{new})$
 - 3: $\text{ids}_{\Delta} \leftarrow \text{ids}_{new}[n:]$ {Delta tokens}
 - 4: **if** $|\text{ids}_{\Delta}| > 0$ **then**
 - 5: $\mathcal{C} \leftarrow \text{forward}(\text{ids}_{\Delta}, \mathcal{C})$
 - 6: $n \leftarrow |\text{ids}_{new}|$
 - 7: $T_c \leftarrow T_{new}$
 - 8: **end if**
-

Handling tokenizer merging: A subtlety arises when adding characters causes the tokenizer to merge the last confirmed token with new text. For example, tokenizing “hel” produces different tokens than tokenizing “hello”, the final token changes rather than a new token being appended. We detect this by comparing expected vs. actual token counts after extension. If a merge is detected, we crop the cache by one token and re-extend, ensuring consistency.

3) Crop: When the user deletes text that was already confirmed (e.g., backspace into a completed word), we must roll back the cache:

$$\mathcal{C}' = \text{crop}(\mathcal{C}, n_{target}) \quad (3)$$

The `DynamicCache.crop()` operation truncates the key-value tensors in place. This is $O(1)$ in computation (tensor slicing) but permanently loses the cropped entries. The trade-off is acceptable: we prioritize memory efficiency over the ability to “undo” crops, and re-extension is fast for the small deltas typical of user edits.

E. Generation Phase

Upon query submission, the system:

- 1) Cancels any pending debounce timers
- 2) Awaits completion of any in-progress extension task
- 3) Flushes remaining pending text T_p to the cache
- 4) Performs a “backstep” to recover logits for the last token
- 5) Generates output tokens autoregressively

The backstep is necessary because our extension operations do not store the output logits (only the KV-cache). To begin generation, we temporarily rewind the cache by one token, run a forward pass to obtain logits, then sample the first new token.

Correctness guarantee: At submission time, \mathcal{C} exactly represents the tokenization of $P_0 + T_c + T_p$. Generation proceeds identically to standard inference from this point, ensuring output equivalence.

IV. IMPLEMENTATION

A. Technology Stack

The backend uses **FastAPI** with Uvicorn for async Web-Socket handling. Cache operations use **HuggingFace Transformers 4.36+** with **PyTorch 2.0+**. The frontend is built with **React 19** and TypeScript via Vite. Evaluation databases run on **PostgreSQL 15** in Docker containers. Development and evaluation hardware was: Apple M2-chip 16GB-memory Macbook Air.

B. Model Selection

We evaluated three Qwen2.5-Coder-Instruct variants:

TABLE I
MODEL SELECTION

Model	VRAM	SQL Quality	Selected
1.5B	~3 GB	Poor	Demo only
3B	~5 GB	Good	✓
7B	~12 GB	Best	OOM on our hardware

We selected **Qwen2.5-Coder-3B-Instruct** [7] for evaluation. Key properties: 152K vocabulary (efficient tokenization compared to Llama’s 32K), GQA architecture (smaller KV-cache footprint), and full `DynamicCache` support including `crop()`.

C. Async Concurrency Model

The `StreamController` uses Python `asyncio` with debounce timers as cancellable tasks and a boolean lock (`_is_extending`) preventing parallel GPU operations. On submission, any in-progress extension completes before generation begins, ensuring the GPU is never oversubscribed. Cache extensions are fire-and-forget: if the GPU is busy when a trigger fires, the trigger is skipped and the next keystroke or timer will catch up.

D. Evaluation Infrastructure

We built a complete evaluation pipeline that migrates Spider SQLite databases to PostgreSQL (one schema per database), executes generated SQL against gold queries, and compares results semantically (order-agnostic, column-name-normalized). Each database’s schema is dynamically loaded from JSON metadata files including foreign key relationships, producing schema-aware prompts for the model.

V. EXPERIMENTAL EVALUATION

A. Setup

Hardware. Apple M2 16GB-Unified Memory Macbook Air

Model. Qwen2.5-Coder-3B-Instruct, float16, temperature 0.0 (greedy decoding), max 256 new tokens.

Dataset. 358 questions from the Spider benchmark [1] across three databases:

TABLE II
EVALUATION DATASET

Database	Questions	Schema Complexity
world_1	100	3 tables, multi-table JOINS
car_1	92	6 tables, deep foreign keys
dog_kennels	82	8 tables, relational patterns
Total	274*	Mixed difficulty

*84 additional `cre_Doc_Template_Mgt` questions available.

Metrics. TTFT (ms), Execution Accuracy (% of queries returning correct results), Exact Match (% of queries with matching result sets), Peak Memory (GB).

Baselines. Standard HuggingFace `model.generate()` with `TextIteratorStreamer` for streaming TTFT measurement (cold-start: no prior cache).

B. Results

TABLE III
TIME-TO-FIRST-TOKEN COMPARISON

Mode	Mean TTFT (ms)	Speedup
Baseline (cold start)	629	1.0×
APITS	64	9.8×

1) *TTFT Improvement:* APITS reduces mean TTFT by 565 ms (90% reduction). At submission time, the KV-cache already contains the vast majority of prompt tokens, leaving only the final pending word and the backstep forward pass. The residual 64 ms corresponds to flushing the last few tokens, recovering logits for generation, and generation of the first output token.

TABLE IV
SQL CORRECTNESS (%)

Metric	Baseline	APITS
Execution Accuracy	70.4	71.2
Exact Match	46.3	46.8

2) *SQL Generation Accuracy*: The +0.8pp and +0.5pp differences are within noise (no statistical significance). This confirms our correctness guarantee: incremental cache construction produces the same KV state as batch prefill, so generation is mathematically identical. Any accuracy variation stems from non-determinism in Transformers outputs and floating-point accumulation order across the two code paths, not from the inference method itself.

3) *Computational Overhead*: Model weights (~5 GB) dominate memory. The KV-cache adds around ~50 MB at peak (mean 1 000 token context), which is identical in both modes. GPU utilization is *smoother* under APITS: instead of a single prefill spike at submission, computation is spread across typing time, leading to a longer GPU-time for APITS through different short 0% to 100% GPU-load spikes.

C. Analysis

Factors increasing benefit. Longer prompts (more schema tokens) yield greater TTFT savings since more prefill work can overlap with typing. Slower typists benefit more, as the system has more time to build the cache before submission. Complex schemas with many tables and foreign keys produce longer prompts, amplifying the speedup.

Factors limiting benefit. Fast typists may outpace the debounce-triggered extensions, resulting in more residual tokens at submission. Heavy editing (frequent backspace) wastes previously computed KV entries, though the $O(1)$ crop operation bounds the recovery cost. Network latency in cloud deployments adds overhead to keystroke transmission. Copy-paste of an input and straight-away validation lead to a all-at-once computation of the Prefill + Generation, which doesn't leverage APITS advantages at all.

VI. DISCUSSION

Strengths. APITS exploits an inherently available resource, user typing time, that traditional pipelines waste entirely. The single-cache design avoids the memory multiplication of speculative approaches. Cache cropping provides graceful degradation under editing. The correctness guarantee eliminates the need for output verification. The approach is model-agnostic: any decoder-only LLM supporting DynamicCache (or equivalent) can benefit.

Limitations. The boundary-based confirmation strategy is conservative: it never commits the word currently being typed, leaving at least one word of residual prefill at submission. The system targets single-user sessions; multi-tenant deployments would require shared base-prompt caches. The backstep mechanism adds one extra forward pass at generation start. Tokenizer merging, while handled, adds implementation complexity.

Future directions. (i) *Predictive speculation*: use a lightweight model (e.g., n-gram or small draft model) to predict tokens beyond confirmed boundaries and speculatively extend the cache, with rollback on misprediction. (ii) *Adaptive confirmation*: learn per-user typing patterns to dynamically adjust debounce timing and confirmation aggressiveness. (iii) *Shared caches*: in multi-tenant settings, share the

base-prompt cache across users querying the same schema. (iv) *Combination with speculative decoding*: apply speculative decoding [5] to the generation phase while using anticipatory prefill for the prefill phase, compounding latency gains.

VII. CONCLUSION

We presented Anticipatory Prefill with keystroke streaming for more Interactive Text-to-SQL, a technique that reduces Time-To-First-Token in Text-to-SQL systems by overlapping prefill computation with user typing time. Through incremental KV-cache construction with boundary-based confirmation, debounced triggering, and $O(1)$ cache cropping for rollback, our system achieves an average $\sim 10\times$ **TTFT speedup** (averaging from 629 ms \rightarrow 64 ms) while maintaining identical SQL generation accuracy (71.2% execution accuracy on Spider). The approach requires no model modification, is compatible with any HuggingFace decoder-only model supporting DynamicCache, and generalizes to any interactive LLM application where user input time can be leveraged for background computation (conversational assistants, etc.).

Individual Contributions

R. Hanader: System architecture, project coordination, final evaluation. **R. Dalaoui**: Frontend development, WebSocket system, report writing. **S. Taider**: Model validation, evaluation framework, baseline comparisons. **R. Vialleton**: KVCacheManager implementation, token confirmation strategy. **S. Wang**: Prompt engineering, schema loading, dataset preparation, Poster design. **L. Zhang**: Backend streaming infrastructure, event handling integration, Presentation resources.

REFERENCES

- [1] T. Yu *et al.*, “Spider: A large-scale human-labeled dataset for complex and cross-database semantic parsing and text-to-SQL task,” in *Proc. EMNLP*, 2018, pp. 3911–3921.
- [2] A. Vaswani *et al.*, “Attention is all you need,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.
- [3] J. Ainslie *et al.*, “GQA: Training generalized multi-query transformer models from multi-head checkpoints,” *arXiv preprint arXiv:2305.13245*, 2023.
- [4] W. Kwon *et al.*, “Efficient memory management for large language model serving with PagedAttention,” in *Proc. ACM SOSP*, 2023, pp. 611–626.
- [5] Y. Leviathan, M. Kalman, and Y. Matias, “Fast inference from transformers via speculative decoding,” in *Proc. ICML*, vol. 202, 2023, pp. 19274–19286.
- [6] G. Xiao *et al.*, “Efficient streaming language models with attention sinks,” in *Proc. ICLR*, 2024.
- [7] Qwen Team, “Qwen2.5-Coder technical report,” *arXiv preprint arXiv:2409.12186*, 2024.
- [8] R. Pope *et al.*, “Efficiently scaling transformer inference,” in *Proc. MLSys*, 2023.

APPENDIX

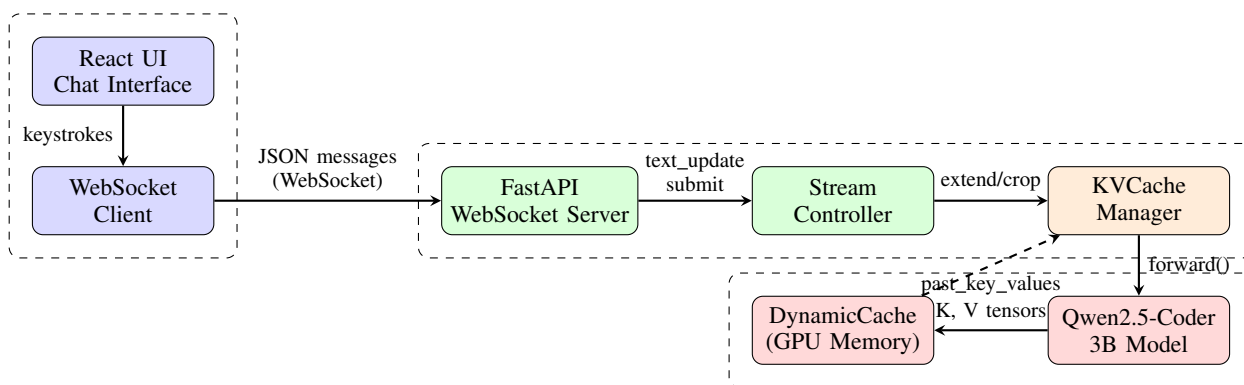


Fig. 1. System architecture. Keystrokes stream from the React frontend via WebSocket. The StreamController manages debouncing and boundary-based confirmation. The KVCashManager maintains a single evolving DynamicCache on GPU memory, extended incrementally during typing and cropped on user edits.