SCHEMAFILL: EFFICIENT LLM TOOL CALLING VIA SLOT-PARALLEL SPECULATIVE DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents interact with external systems by generating structured tool calls. Given a user request, conversational context, and a catalog of tool schemas, a tool-calling model must select tools and generate their arguments, potentially producing multiple calls in a single response. Standard autoregressive decoding generates these calls token by token, incurring substantial latency for requests involving multiple calls or many argument fields. The explicit argument structure offers opportunities for parallel generation, but later argument values may depend on preceding fields and calls, so independently generated values can differ from the target model's output. We present SchemaFill, a framework for efficient LLM tool calling through slot-parallel speculative decoding. SchemaFill generates future slot values concurrently as candidates, without requiring advance knowledge of the actual call sequence or argument values. Candidates spanning multiple fields and calls are concatenated for verification by the target model under the actual output prefix. Only verified tokens are committed, and the target supplies corrections when candidates disagree. This applies target verification while exploiting parallelism across slots and calls. On Glaive and BFCL, SchemaFill achieves up to a 4.37× improvement in end-to-end throughput over autoregressive decoding.

1 INTRODUCTION

LLM agents extend language models from producing text to acting on external systems (Yao et al., 2023; Schick et al., 2023). Given a user request, conversation history, and a catalog of available tools, an agent may need to query a database, invoke a software service, or coordinate several actions. Tool calling connects the model's decision to such operations: the model selects an available function and supplies the information required to invoke it (Liu et al., 2024). Function-calling evaluations therefore test whether an agent can choose and parameterize actions from natural-language requests (Patil et al., 2025; Li et al., 2023).

To make those actions executable, the agent cannot communicate them only as free-form text. A tool runtime expects a machine-readable object whose tool names, argument keys, value types, and nesting satisfy predefined interface schemas (Willard & Louf, 2023; Dong et al., 2025; Geng et al., 2025). This structured object is the contract between the model and the runtime: it can be parsed, validated, routed to the selected tools, and executed without another semantic interpretation step. Generating it requires the model to decide which tools to invoke, how many calls to emit, their order, and the exact argument values; one response may contain different tools or repeat the same tool several times (Qin et al., 2024; Patil et al., 2025).

Current tool-calling models usually generate the entire response autoregressively, one token at a time (Xia et al., 2026). Much of that response has explicit structure, including call wrappers, argument keys, and typed value slots, yet AR decoding treats it as one flat token sequence (Zheng et al., 2024). The model must finish earlier fields and calls before reaching later slots, so requests with many fields or multiple calls incur substantial decoding latency (Chen et al., 2023; Xia et al., 2026). Using the structure to draft values for possible future slots concurrently can shorten this serial path and achieve substantially higher throughput than AR decoding.

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.

Direct slot-parallel generation faces two problems: the realized tool-call sequence and argument slots are unknown when drafting begins, and independently drafted values may differ from what the target would generate under the actual preceding output. Figure 2 shows the second problem even with an oracle skeleton: deviations from target AR accumulate as more fields and calls are generated independently. Keeping the speedup while committing only target-consistent tokens is the central challenge.

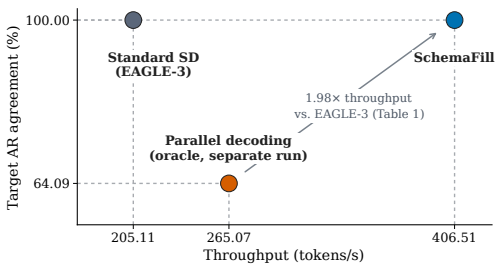


Figure 1: SD denotes speculative decoding. Direct slot-parallel generation can raise throughput but may change the target response. SchemaFill combines parallel proposals with target verification. The direct-parallel baseline follows the skeleton-then-parallel-fill idea of Skeleton-of-Thought (Ning et al., 2024), with the correct skeleton supplied in advance.

These disagreements become output errors if independently drafted slot values are committed without checking them against the target under the preceding fields and calls. To retain parallel generation while preserving the target response, the drafted values must remain candidates until the target decides which tokens to accept. Speculative decoding provides this accept-or-correct mechanism: the target verifies proposed tokens under the actual output prefix, accepts matches, and supplies a correction at the first mismatch (Leviathan et al., 2023; Chen et al., 2023). Conventional speculative decoders draft a continuation of the current prefix (Li et al., 2025); applying the same verification rule to future slots would let their values be drafted concurrently without committing disagreements to the response.

We introduce SCHEMAFILL, which combines parallel drafting of future slots with target verification across fields and calls (Figure 1). From the request and tool catalog, it forecasts possible later call positions and slots without assuming the realized calls or values. Isolated branches then draft slot values in parallel while the committed output advances. The runtime packs ready values with proposed tool names, field keys, and structural text for shared target verification. All packed tokens remain speculative until accepted by the target under the actual output prefix. Our contributions are threefold:

- We present SchemaFill, a framework for slot-parallel speculative decoding of structured tool calls. It drafts future argument values concurrently and packs them with structural text across fields and calls, while the target verifies every token under the actual output prefix.
- We design a shared verification runtime that evaluates concurrent drafts in one target forward pass while isolating branch attention and reusing verified KV states.
- We achieve up to a 4.37× mean end-to-end speedup over autoregressive decoding and outperform existing speculative decoding baselines on GLAIVE and BFCL.

2 RELATED WORK

Tool Use, Function Calling, and Structured Generation. Tool-use research studies how language models select APIs, decide when to invoke them, and produce executable arguments: Toolformer learns these decisions from self-supervised API annotations (Schick et al., 2023), Gorilla adds retrieval-aware training for large and changing API collections (Patil et al., 2024), and ToolLLM/ToolBench scales to thousands of real-world APIs and multi-tool solution paths (Qin et al., 2024). API-Bank and BFCL evaluate tool selection and structured arguments in tool-augmented and single-/multi-call settings (Li et al., 2023; Patil et al., 2025), while grammar-constrained methods enforce valid output languages or schemas (Geng et al., 2023; Dong et al., 2025). These works improve tool-use capability or structural validity; SchemaFill addresses the inference cost of generating an already structured response, using public schema metadata to propose slots while retaining target verification and target control over tools, fields, ordering, and values.

Speculative Decoding. Early methods use a lightweight drafter to propose future tokens for target verification (Leviathan et al., 2023; Chen et al., 2023). SpecInfer verifies draft trees (Miao et al., 2024), EAGLE and EAGLE-3 draft from target features (Li et al., 2024; 2025), and Medusa uses

multiple decoding heads to propose candidate continuations (Cai et al., 2024). Lookahead Decoding generates and verifies parallel token continuations without an auxiliary drafter (Fu et al., 2024). Despite different proposal mechanisms, these methods work on continuations of the current serialized prefix. SchemaFill applies target verification to a different unit of parallel work: values for possible fields and tool calls beyond that prefix.

Schema-Aware Tool-Call Acceleration. ToolSpec is the closest prior work to our setting: it combines schema proposals, retrieval, and token recycling to accelerate tool calls (Xia et al., 2026). SchemaFill instead prepares argument values for future fields and forecast call positions, and packs those candidates with proposed structural syntax across call boundaries. Branches can prepare candidates early, but their local acceptance never commits text to the final response; every proposed value is checked again under the actual output prefix. This distinction lets us evaluate slot-parallel drafting while preserving the target's freedom to select tools, fields, ordering, and values.

Together, these lines of work leave open how to apply target-verified speculation when the candidate units are noncontiguous slots in a structured multi-call response. SchemaFill addresses this gap by drafting possible argument slots across fields and calls, packing them for shared verification, and extending the response only with target-consistent tokens. The next section defines the tool-call setting and target verification rule, then examines why independently generated slot values cannot be committed directly.

3 Setting and Motivation

3.1 Structured Multi-Call Generation

An LLM agent processes a user request x in conversational context c, which may include earlier messages and tool results. Some requests can be handled directly; others require information from, or actions by, an external system. For the latter, the agent uses tool calling: it expresses the needed operation as a structured call that a runtime can execute. The runtime supplies a catalog $\mathcal{T}$ of available tools, each with a name, description, and argument schema S_t defining how it can be called. The model receives the request, context, and full catalog in the prompt $P = P(x, c, \mathcal{T})$.

Given P, the model generates an ordered call sequence. For each call, it chooses a tool, selects its argument fields, and supplies their values from x and c. The sequence may contain multiple calls or repeated use of one tool:

$$(x, c, \mathcal{T}) \longmapsto C = ((t_1, a_1), \ldots, (t_K, a_K)). \tag{1}$$

Here $t_j \in \mathcal{T}$ is the selected tool and a_j is its generated argument object. The model determines the number, order, and contents of the calls. Only after C is complete does the runtime dispatch each (t_j, a_j) to its tool for execution.

Each tool schema declares possible argument fields, their types, and whether they are required. For a realized call (t_j, a_j), we can view its arguments as field–value pairs $a_j = \{f_{j,k} : v_{j,k}\}_{k=1}^{M_j}$ in the order they are emitted. The target must decide which optional fields appear and generate the value $v_{j,k}$ of every included field using the request and context. Thus the schema exposes possible value positions, but neither the realized field set nor its values. In a multi-call response, different calls may use different schemas, and later values may depend on fields or calls generated earlier.

The target emits C as a serialized token sequence $y_{1:L}$, interleaving tool names, argument keys, values, punctuation, and enclosing structure. Standard autoregressive (AR) decoding generates all of these tokens in order under the full prompt P, even when the schema makes parts of the structure predictable. Generation stops at the first closed outer structured object or end-of-sequence token.

3.2 Speculative Decoding on Serialized Tool Calls

Speculative decoding proposes several tokens before committing them, then uses the target to verify the proposal (Leviathan et al., 2023; Chen et al., 2023). Let m be the committed output prefix and $z = (z_1, \ldots, z_n)$ a proposed continuation, and let $\|$ denote sequence concatenation. At candidate position i, the verifier evaluates $p_\theta(\cdot \mid P, m \| z_{<i})$: causal target evaluation can score the proposed positions together, while each score depends only on the actual prefix and preceding candidate to-

kens. The prefix or proposal may cross a tool name, argument key, value, or structural delimiter; schema constraints do not make later fields or calls committed decisions.

For the greedy target policy used in our main experiments at temperature 0, define $g_\theta(P, u) = \arg\max_{v \in \mathcal{V}} p_\theta(v \mid P, u)$. Serial AR decoding chooses $y_i^{\mathrm{AR}} = g_\theta(P, y_{<i}^{\mathrm{AR}})$ after the complete preceding output. Whether the drafter selects z deterministically or stochastically, greedy verification accepts the longest candidate prefix that follows the same target decisions:

$$r^\star(m, z) = \max\{r \in \{0, \ldots, n\} : z_i = g_\theta(P, m \| z_{<i}) \text{ for every } 1 \le i \le r\}. \tag{2}$$

If $r^\star < n$, the verifier commits $z_{\le r^\star}$ and the target correction $g_\theta(P, m \| z_{\le r^\star})$, discards the rejected suffix, and resumes from the corrected prefix. If $r^\star = n$, it commits all of z and continues from the next target position. This checks structural and value tokens under the same serialized prefix and can accept several tokens per target pass. Under a stochastic target policy, standard speculative sampling instead accepts a proposed token v with probability $\min\{1, p_\theta(v \mid P, u)/q_u(v)\}$, where q_u is its proposal distribution, and samples a correction from the residual distribution on rejection. At temperature 0, our runtime uses the g_θ verification rule. Appendix B proves consistency with greedy AR for this rule and gives the full stochastic rule and its distributional-consistency proof.

3.3 Accumulated AR Deviation in Direct Parallel Slot Generation

We isolate the effect of missing inter-slot context in direct parallel generation with a controlled BFCL diagnostic. For each response, we take the call and field layout from an offline greedy autoregressive (AR) output and replace every argument value with a distinct slot marker. The target model then generates each value independently from the public prompt, the complete marked layout, and that slot's identity, without seeing values generated for other slots. We compare each value with greedy AR at the corresponding position, where AR conditions on the full preceding output. This oracle layout controls for tool and field selection in the diagnostic; SchemaFill does not receive it at inference.

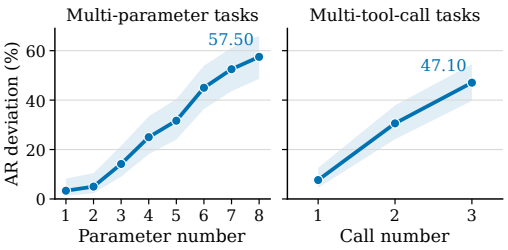


Figure 2: AR deviation from independently generated slot values. Left: BFCL responses with at least eight parameters; right: at least three calls. Each curve follows a fixed cohort as more slots or calls are considered. Shading: 95% Wilson intervals (Wilson, 1927).

Figure 2 reports the fraction of responses with at least one independently generated value differing from AR. Within the fixed parameter cohort, this fraction rises from 3.3% after the first slot to 57.5% after eight slots. Within the fixed multi-call cohort, it rises from 7.6% after the first call to 47.1% after the third. Thus, a small deviation rate at the first slot can accumulate across the values needed for a complete tool-call response. Direct parallel generation can therefore change the target's serialized output. The curves measure cumulative AR deviation, not downstream execution success or the difficulty of each individual slot. These results motivate a method that treats parallel slot values as provisional candidates and verifies them under the target's actual serialized prefix before committing them to the output.

4 Method: SchemaFill

4.1 Method Overview

SchemaFill follows a forecast–draft–verify loop. From the request, dialogue context, and full tool catalog, the method identifies possible argument slots and begins drafting later values before their serialized positions are reached. These drafts remain provisional. When a candidate becomes relevant, SchemaFill packs it with the required structural text after the current output prefix and lets the target accept or correct the resulting sequence in the same order as ordinary autoregressive decoding.

We call the authoritative output path the *committed stream* and the provisional value paths *speculative branches*; the implementation refers to them as MAIN and SIDE, respectively. The full schema always remains in the target prompt, and the target still determines the realized tools, fields, order,

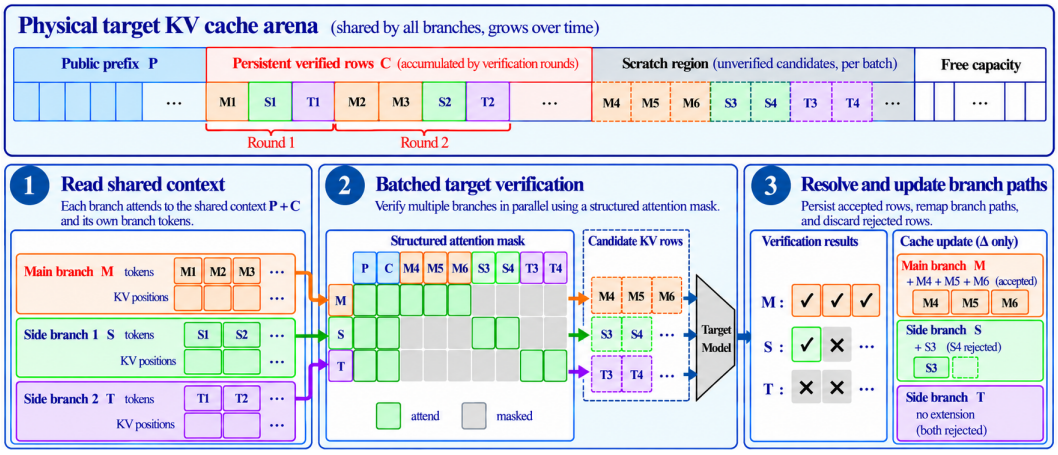


Figure 3: Target-KV cache lifecycle in SchemaFill. The public prefix and verified rows occupy a shared arena, while attention masks restrict branches to valid context and their own candidate paths. One target pass verifies branch scratch rows. Accepted rows extend their cache paths; rejected suffixes are discarded. Only verified tokens on the committed branch become output; a side-branch value must be verified again under the actual output prefix before it can be committed.

and values. Forecasting changes only where draft computation is spent, while target verification determines every emitted token. Figure 3 previews the target-KV paths, and Algorithm 1 summarizes the complete forecast–draft–verify loop. The following subsections explain the forecast, slot drafting and verification, cross-call packing, and target-KV management.

4.2 Slot-Parallel Speculative Decoding

During initialization, candidate key prefixes are verified and an admissible MAIN path is selected by accepted key-prefix length, then total accepted tokens and forecast order. For each branch b, a_b counts matching proposal tokens, c_b is the first target correction (ϵ if none), and K'_b contains its target-cache rows. Trimming bounds a draft to a candidate value literal; it does not accept that literal. The stopping test uses only verified MAIN tokens, including closing text.

Metadata-Guided Tool and Slot Forecasting. A forecasted slot $s = (j, t, f)$ specifies a possible call position j, candidate tool $t \in \mathcal{T}$, and field f from its public schema. The slot's type and optionality guide drafting, but neither the forecasted call nor field is a commitment. Given the request and dialogue context, our default runtime forms a lexical query and ranks the available tools with BM25 (Robertson & Zaragoza, 2009). Let q denote the tokenized request and context, and let d_t denote the tokenized public metadata of tool t (its name, description, and schema field names and descriptions). The ranking is

$$\pi = \mathrm{rank}_{t \in \mathcal{T}} \, \mathrm{BM25}(q, d_t), \tag{3}$$

where tools with equal scores keep their original catalog order. We cycle this ranked list and truncate it to the first $B = 3$ call groups,

$$\mathcal{F}_B = \mathrm{prefix}_B(\mathrm{cycle}(\pi)) \, , \tag{4}$$

then expand each forecasted tool into slots in rendered-schema key order, recording its declared type and requiredness. The scheduler activates these slots near the current committed position. If verified structure rules out a forecast slot, the scheduler retires it; an unused producer group may be rebound when its tool matches a verified call header. Forecasts allocate draft computation, while the target determines the realized tools, fields, and values (Algorithm 1, lines 2–4).

SIDE speculative drafting. For an active slot $s_i = (j_i, t_i, f_i)$, let H_i contain its call-position hint, tool/field prefix, and branch-private draft history. We instantiate the drafter D_ϕ with EAGLE-3 (Li et al., 2025). It produces a provisional value from the common prompt and this local context,

$$\hat{v}_i = D_\phi(P \oplus H_i) \, , \tag{5}$$

Algorithm 1 SchemaFill: slot-parallel speculative decoding

Require: Request x, context c, public tool catalog $\mathcal{T}$, forecast budget B; target f_θ, drafter D_ϕ
Ensure: Target-verified structured output y

1: $P \leftarrow \mathrm{BuildPrompt}(x, c, \mathcal{T})$
2: $\pi \leftarrow \mathrm{RankMetadata}(x, c, \mathcal{T})$; $G \leftarrow \mathrm{ForecastGroups}(\pi, B)$
3: $\mathcal{S} \leftarrow \mathrm{ExpandSlots}(G)$
4: $(y, K, \mathrm{MAIN}, \{H_i\}) \leftarrow \mathrm{InitVerify}_\theta(P, \mathcal{S})$ ▷ verified MAIN prefix, provisional SIDE states
5: **while** $\neg\,\mathrm{Stop}(y)$ **do**
6: $\quad A \leftarrow \{\mathrm{MAIN}\} \cup \mathrm{ActiveSIDE}(\mathrm{MAIN}, \mathcal{S})$
7: $\quad L \leftarrow \mathrm{PreparedSIDE}(\mathcal{S})$ ▷ snapshot before this round
8: $\quad H_{\mathrm{MAIN}} \leftarrow y$
9: $\quad$ **for all** $b \in A$ in parallel **do**
10: $\quad\quad R_b \leftarrow \mathrm{TrimToLiteralBoundary}(D_\phi(P \oplus H_b))$
11: $\quad$ **end for**
12: $\quad Z \leftarrow \mathrm{PackMAIN}(y, R_{\mathrm{MAIN}}, L, \mathcal{S})$ ▷ proposes structural text
13: $\quad \mathcal{B} \leftarrow \{(\mathrm{MAIN}, Z)\} \cup \{(i, R_i) : i \in A \setminus \{\mathrm{MAIN}\}\}$
14: $\quad (a_b, c_b, K'_b)_{b\in\mathcal{B}} \leftarrow \mathrm{VerifyTree}_\theta(\mathcal{B}, K)$
15: $\quad y \leftarrow y \oplus Z_{1:a_{\mathrm{MAIN}}} \oplus c_{\mathrm{MAIN}}$
16: $\quad$ **for all** $i \in A \setminus \{\mathrm{MAIN}\}$ **do**
17: $\quad\quad H_i \leftarrow H_i \oplus (R_i)_{1:a_i} \oplus c_i$ ▷ SIDE remains provisional
18: $\quad$ **end for**
19: $\quad K \leftarrow \mathrm{RetainAcceptedRows}(K, \{K'_b, a_b\}_{b\in\mathcal{B}})$ ▷ discard rejected scratch rows
20: $\quad (\mathrm{MAIN}, \mathcal{S}) \leftarrow \mathrm{AlignForecast}(\mathrm{MAIN}, \mathcal{S}, y)$
21: **end while**
22: **return** y

so branches share the public prompt, while the attention mask excludes every other branch's private history H_ℓ from branch i for $\ell \neq i$. MAIN continues along the committed stream. The target checks $\hat{v}_i$ under the local context $P \oplus H_i$; accepted tokens and corrections advance only that provisional SIDE branch. On a local mismatch, the rejected suffix is discarded, while the locally accepted prefix and correction remain provisional. A packed MAIN proposal is checked separately under the actual prefix $P \oplus C_t$. These paths can share one target tree forward, but a SIDE value completed during that forward can enter a packed MAIN proposal only in a later round. Local acceptance never appends text to the response (Algorithm 1, lines 6–10 and 14–17).

Cross-call packing. When the committed stream reaches a field for which a SIDE branch has prepared a completed or partial value, the packer places that value after the actual MAIN prefix. It also proposes the intervening tool names, field keys, punctuation, and closing text, so that one proposal can continue across fields and tool-call boundaries. If C_t is the target-verified MAIN prefix, $\hat{v}_i$ is a prepared SIDE value, and S_i is a proposed structural bridge, the packed suffix is

$$Z_t = S_0 \oplus \hat{v}_1 \oplus S_1 \oplus \cdots \oplus \hat{v}_k \oplus S_k.$$

The target checks Z_t as one causal MAIN path after $P \oplus C_t$: each token attends to earlier proposed tokens on that path, including across field and call boundaries, but never to a SIDE branch's private history. For greedy target decoding, Equation 2 accepts only the longest prefix matching the target's decisions. At a mismatch, MAIN commits that prefix and the target correction, discarding the remaining packed suffix and its target-KV rows. A prepared SIDE value can remain a private candidate if its forecast slot is still possible, but it can be used only after the realized call and field match; its local KV is never transferred to MAIN, and the value must be checked again under the corrected MAIN prefix (Algorithm 1, lines 12–15).

4.3 KV-Cache Management

Building on tree-based speculative verification (Miao et al., 2024), SchemaFill manages the target-KV lifecycle shown in Figure 3. The physical arena is shared by MAIN and SIDE branches, but logical ownership and attention visibility remain branch-specific. The public-prefix region stores the common prompt context; the persistent verified region stores the authoritative target-accepted history accumulated across verification rounds; and the scratch region holds the current batch of

unverified candidate rows. Free capacity is managed separately from these logical regions so that temporary tree paths can be created without making them part of the committed stream.

The lifecycle follows the stages in the figure. (1) Read shared context: each branch attends to the public prefix and verified history with only its visible ancestors and private tokens; shared physical storage does not merge branch histories. (2) Batched target verification: MAIN and active SIDE candidates are placed in one structured verification tree. The MAIN path is causal across its packed fields and calls, while the visibility mask prevents it from attending to private SIDE paths and masks sibling SIDE paths from one another. Candidate rows remain scratch until the target accepts them. (3) Resolve and update branch paths: accepted rows extend only their owning path. A MAIN correction extends the committed output, whereas a SIDE correction remains provisional; rejected suffix rows are discarded. A SIDE value is never grafted directly into MAIN state: when used as a MAIN proposal, it is verified again under the true prefix. This reuse of verified history avoids repeated re-prefill while preserving the target's serialized acceptance rule. Drafter KV and captured target features remain separate branch-owned state rather than additional authoritative target history.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Datasets. Our Glaive test set consists of 650 examples sampled from the full GLAIVE Function Calling dataset (Glaive AI, 2023). For the Berkeley Function Calling Leaderboard (BFCL) v3 (Patil et al., 2025), we select the three categories most suited to evaluating concurrent tool calling from the 25 categories in the release used here. Our BFCL test set contains all 450 requests from these three categories. Preprocessing normalizes tool definitions and extracts the user request and available preceding context at the selected decision point. All evaluations use these same fixed test sets; results characterize these examples and categories.

Task and input. All datasets use the same plain-text instruction template, and all methods receive an identical target prompt for each request, containing its prior context and full tool catalog. The target generates one JSON object with a `tool_calls` list; each entry contains a tool `name` and an `arguments` object. We use Qwen2.5-7B-Instruct (Yang et al., 2024) as the target model. Appendix A gives the model, training, runtime, and hardware settings.

Metrics. For request i, let L_i count committed output tokens, including values, tool names, keys, punctuation, and target corrections; let T_i be elapsed time in seconds and V_i its actual target verification calls, including initialization. We report *End-to-end throughput* as $\mathrm{TPS}_{\mathrm{E2E}} = \sum_i L_i / \sum_i T_i$ and *Acceptance length* as $\mathrm{AL} = \sum_i L_i / \sum_i V_i$. A shared tree forward counts as one verification even when it advances MAIN and SIDE branches. Timing uses CUDA synchronization at request boundaries and covers input preparation, initialization, drafting, target verification, and stopping. Model loading, CUDA graph construction, and warmup precede measurement; trace serialization follows it. Failed requests retain their tokens, target calls, and elapsed time in pooled metrics.

5.2 MAIN RESULTS

We compare decoding methods that represent different sources of acceleration. Autoregressive decoding (Leviathan et al., 2023) provides the target's one-token reference rate, while EAGLE-3 (Li et al., 2025) measures learned token-tree speculation along the emitted prefix without SchemaFill's slot forecasting or cross-call packing. ToolSpec (Xia et al., 2026) represents schema-aware acceleration based on proposed and retrieved continuations. SchemaFill prepares argument slots from public tool metadata and verifies slot candidates together with structural text across calls. Table 1 summarizes matched launches under the same target prompt and target decoding policy; all methods use temperature 0. Results for AR, EAGLE-3, and SchemaFill at temperature 1 appear in Appendix C.5. The component studies in Sections 5.3 and 5.4 use temperature 0.

SchemaFill achieves the highest throughput on both datasets: 315.24 ± 0.59 TPS on GLAIVE and 406.51 ± 0.53 TPS on BFCL. These correspond to $3.45\times$ and $4.37\times$ speedups over AR, $1.39\times$ and $1.98\times$ over EAGLE-3, and $1.56\times$ and $1.58\times$ over ToolSpec. For workload context, reference annotations in the broader pool of 27,278 automatically eligible Glaive tool-call decisions average 1.00 call and 1.88 top-level argument fields per decision; this pool is distinct from the Glaive test

Table 1: End-to-end results on the Glaive (Glaive AI, 2023) and BFCL v3 (Patil et al., 2025) test sets with Qwen2.5-7B-Instruct. All methods use temperature 0. Acceptance length (AL) is committed output tokens per target verification call; E2E TPS is committed output tokens per second, shown as mean ± sample standard deviation over three launches. Relative TPS is normalized to AR within each dataset.

Dataset	Method	Accept Length	E2E TPS	Relative TPS
GLAIVE	AR	1.00	91.35 ± 0.06	1.00×
	EAGLE-3 (Li et al., 2025)	3.31	227.44 ± 0.39	2.49×
	ToolSpec (Xia et al., 2026)	2.79	202.15 ± 0.32	2.21×
	SchemaFill	**5.82**	$\mathbf{315.24 \pm 0.59}$	**3.45×**
BFCL v3	AR	1.00	92.98 ± 0.16	1.00×
	EAGLE-3	2.87	205.11 ± 0.70	2.21×
	ToolSpec	3.42	257.32 ± 1.55	2.77×
	SchemaFill	**6.97**	$\mathbf{406.51 \pm 0.53}$	**4.37×**

Table 2: Cumulative ablation on the BFCL test set. Each stage retains all mechanisms from the preceding stage. A is the EAGLE-3 baseline. B uses SchemaFill's schema-guided MAIN path to forecast tool and slot positions and concatenate structural text with drafted values, without SIDE branches. C adds SIDE drafting of future slot values. D adds cross-call packing of prepared values and intervening structural text into one target-verification proposal, yielding full SchemaFill. Relative TPS uses A as the baseline.

Stage	Configuration	Accept Length	E2E TPS	Relative TPS
A	EAGLE-3	2.87	205.11	1.00×
B	A + schema-guided MAIN path	4.82	313.45	1.53×
C	B + SIDE speculative drafting	5.85	349.88	1.71×
D	C + cross-call packing (full SchemaFill)	6.97	406.51	1.98×

set. Across the BFCL test requests, the averages are 2.96 calls and 7.46 fields. These reference statistics are not inputs to SchemaFill; Appendix C.1 gives the counting scope. The larger gain on BFCL is consistent with its multi-call structure. On the single-call Glaive cases, SchemaFill drafts argument values concurrently within a call; on BFCL, it can also prepare arguments for later calls while the current call is generated, then verify candidates across call boundaries. This creates additional opportunities for acceleration in complex multi-call requests. SchemaFill more than doubles ToolSpec's AL on both datasets by verifying prepared values together with structural text.

Additional Experiments. We report a BFCL runtime breakdown including BM25 forecasting but excluding request initialization (Appendix C.3), KV-cache overhead (Appendix C.2), and run-to-run variance (Appendix C.4). Temperature-one results and slot-order diagnostics appear in Appendices C.5 and C.6, respectively; cross-target evaluations on Qwen3 (Yang et al., 2025), Mistral (Jiang et al., 2023), and Hermes (Teknium et al., 2024) appear in Appendix D.

5.3 ABLATION STUDY

We use four cumulative stages in Table 2. Stage A is the EAGLE-3 baseline. Stage B uses SchemaFill's schema-guided MAIN path, including tool and slot forecasting and structural/slot concatenation, without SIDE branches. Stage C enables SIDE speculative drafting for future slots. Stage D enables cross-call packing, which combines prepared candidates and intervening structural text into one proposal for target verification under the current MAIN prefix. Stages B–D share the same SchemaFill runtime.

Relative to EAGLE-3, Stage B raises TPS by 52.82% and AL by 68.09%. This comparison measures the combined effect of SchemaFill's MAIN runtime, metadata-guided forecasting, and structural/slot concatenation. Within the shared SchemaFill runtime, enabling SIDE drafting in Stage C adds 11.62% TPS and 21.52% AL by preparing later values before MAIN reaches them. Enabling cross-call packing in Stage D adds another 16.19% TPS and 19.06% AL over Stage C: prepared values

and structural bridges can be proposed across call boundaries and verified together by the target. The C–D comparison thus measures the additional benefit of packing on top of candidate preparation.

5.4 Scaling with Call Count

We study how SchemaFill's performance varies with task size, measured by reference call count. For this separate diagnostic, we randomly sample 100 natural requests from the broader BFCL v3 release (Patil et al., 2025) for each count from one to four (400 requests in total). All groups use the common target prompt. In this count-informed diagnostic, SchemaFill receives the reference call count C to set its forecast budget $B = C$, with other production settings unchanged. The groups are independently sampled rather than paired prefixes. We measure mean verification-tree size as $\overline{N} = \sum_r \sum_{v=1}^{V_r} N_{r,v} / \sum_r V_r$, where V_r counts actual target verification calls, including initialization, and $N_{r,v}$ counts distinct candidate-token nodes processed in that call. MAIN and SIDE nodes count whether accepted or rejected; shared nodes count once per verification, and prompt prefill rows are excluded.

Table 3: Separate BFCL v3 call-count diagnostic (100 requests/group), using $B = C$. AL is committed tokens per verification; E2E TPS is tokens/s; Nodes/Verify is pooled tree size. All requests and initialization are included.

Ref. Calls (C)	Accept Length	E2E TPS	Nodes/ Verify
1	5.14	265.95	23.43
2	7.98	438.62	35.76
3	8.06	443.54	39.29
4	7.54	410.72	42.71

Figure 4 shows how SchemaFill prepares more slots as the number of tasks within a request increases. With $B = C$, mean initialization-tree size grows from 61.78 nodes at one call to 151.98 at four. These additional nodes initialize more speculative slot contexts together, allowing their candidate values to be generated concurrently in the early decoding rounds. Slot preparation is therefore concentrated at the beginning of generation, so candidates can become available early for packing. The curves drop sharply after initialization, with subsequent rounds using much smaller verification trees.

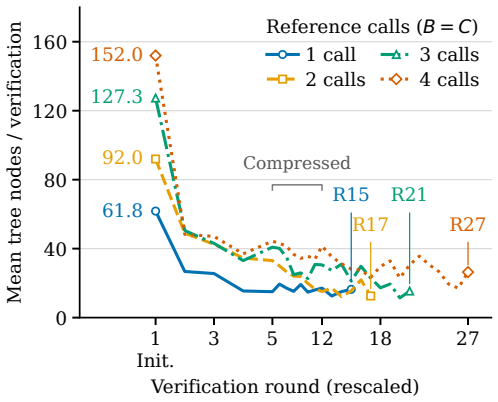


Figure 4: Target-verification workload during SchemaFill decoding. Candidate-node counts peak at slot initialization and drop sharply in subsequent rounds. Curves average requests reaching each round in the Table 3 diagnostic, excluding prompt tokens. Rounds 6–11 are compressed; endpoint labels mark the last rounds with at least 10 contributing requests.

Once slot candidates are ready, further acceleration depends on their quality: the target must accept them under the committed prefix to increase AL. Table 3 shows that performance reaches its peak around two to three calls, with 438.62–443.54 TPS and AL 7.98–8.06, up from 265.95 TPS and AL 5.14 at one call. At four calls, mean verification-tree size grows to 42.71 nodes, while AL drops to 7.54 and throughput to 410.72 TPS. This pattern is consistent with slot-drafter quality limiting further acceptance gains despite the early preparation of more slot candidates.

6 Conclusion

SchemaFill uses tool schemas to draft future argument values concurrently and packs candidates across fields and calls for verification under the actual output prefix. On the Glaive and BFCL test sets, it achieves 3.45× and 4.37× end-to-end throughput over AR, respectively, outperforming EAGLE-3 and ToolSpec. Its central insight is that tool-call structure enables parallel speculation without assuming independent arguments, since the target verifies every token before commitment.

AI USE STATEMENT

Generative AI tools were used to assist with language editing, developing and running experimental code, and generating figures. The authors reviewed the resulting text and verified the experimental code and reported measurements. The authors take responsibility for the final content, claims, and artifacts, including AI-assisted figures.

ETHICS STATEMENT

This work studies inference acceleration for tool-calling models using public datasets and pretrained models. Faster generation can benefit useful automation, but can also increase the scale of harmful or erroneous tool use. Target verification checks candidates against the target model under the actual output prefix; it does not establish the safety, factual correctness, or authorization of a tool call. Deployments should therefore retain tool-access controls, argument validation, and appropriate human oversight. Use of the underlying datasets and models should respect their licenses and applicable privacy constraints.

REPRODUCIBILITY STATEMENT

Section 4 and Algorithm 1 describe the forecast–draft–verify procedure, branch isolation, and cache management. Section 5 specifies the evaluation sets, matched-input comparisons, metric definitions, and timing boundaries. Appendix A documents the target and drafter configurations, training settings, software, and hardware; Appendix C provides additional dataset and measurement details. Appendix B states the assumptions and proofs for greedy consistency and the stochastic extension, and Appendix D reports the configurations used for the additional target models. Reference-derived layouts used in the diagnostic are identified separately in Section 3.3.

REFERENCES

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems*, 7, 2025.

Bradley Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7 (1):1–26, 1979. doi: 10.1214/aos/1176344552. URL https://doi.org/10.1214/aos/1176344552.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 14060–14079, 2024. URL https://proceedings.mlr.press/v235/fu24a.html.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured nlp tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 10932–10952, 2023.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Jsonschemabench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868*, 2025.

Glaive AI. Glaive function calling dataset. Hugging Face dataset, 2023. URL https://huggingface.co/datasets/glaiveai/glaive-function-calling.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International conference on machine learning*, pp. 19274–19286. PMLR, 2023.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 3102–3116, 2023.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 136737–136756. Curran Associates, Inc., 2025. doi: 10.52202/085713-4562. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c7b5a35ea98b62512a869c19ea7b03cb-Paper-Conference.pdf.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. ToolACE: Winning the points of LLM function calling. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 41359–41381, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/663865ea167425c6c562cb0b6bcf76c7-Paper-Conference.pdf.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, et al. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems*, 37:54463–54482, 2024.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 932–949, 2024.

Mistral AI. Model card for Mistral-7B-Instruct-v0.3. Hugging Face model card, 2024. URL https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting llms for efficient parallel generation. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/03d7e13f0092405804f3a381ade8f3f0-Abstract-Conference.html.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, 2024.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.

Ryan Teknium, Jeffrey Quesnelle, and Chen Guang. Hermes 3 technical report, 2024.

Brandon T Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.

Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. doi: 10.1080/01621459.1927.10502953. URL https://doi.org/10.1080/01621459.1927.10502953.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/.

Heming Xia, Yongqi Li, Cunxiao Du, Mingbo Song, and Wenjie Li. Toolspec: Accelerating tool calling via schema-aware and retrieval-augmented speculative decoding. *arXiv preprint arXiv:2604.13519*, 2026.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

A Implementation Details

Target and drafter. The main experiments use Qwen2.5-7B-Instruct (Yang et al., 2024), temperature 0 (greedy decoding), repetition penalty 1.0, BF16 target weights and computation, and FP32 target logits. RoPE (Su et al., 2024) frequency and sine/cosine computations use FP32, with BF16 rounding preserved in rotary application. Reduced-precision BF16 reduction is disabled within target forwards. EAGLE-3 and our method share the same locally trained EAGLE-3 head (Li et al., 2025): epoch 6, a 14,136-token draft vocabulary, and target hidden-state indices $[2, 14, 25]$. In the temperature-one experiments in Appendices C.5 and D, AR, EAGLE-3, and SchemaFill all use temperature 1. ToolSpec retains its temperature-0 configuration as a reference. The drafter was trained on fresh target-generated continuations from separate public tool-calling requests drawn from BFCL and additional public corpora including Glaive, API-Bank, ToolACE, and ToolAlpaca (Glaive AI, 2023; Li et al., 2023; Liu et al., 2025; Tang et al., 2023). Training uses 11,517 positive-mask records constructed from 9,216 public requests, six epochs, learning rate 8×10^{-5}, batch size four with two accumulation steps, and two rollout horizons with loss decay 0.8.

Runtime configuration. The production runtime uses a three-call forecast budget. Candidate tools are ranked by BM25 lexical relevance: the query contains the request and dialogue context, and each tool document contains its name, description, and schema field metadata; ties retain catalog order, and the ranked list is cycled to allow repeated calls. The scheduler admits the current forecast group and the next group, and can rebind an unused matching producer after observed call-order drift. It uses the hint `Select the second request.` only inside the corresponding branch prefix. Multiple active branches share a segmented draft tree with branch-isolated attention; a single active branch uses the standard segmented drafting path. The committed stream and speculative branches share one target tree verification. Target KV allocation capacity, exact prompt-tokenization prefixes, and exact public prompt KV prefixes are reused when valid. Initialization combines causal attention for uncached prompt tokens with masked MAIN/SIDE tree attention in one target forward. Cached prompt features are retained, and branch visibility remains isolated. The measured draft budget is $k = 3$, committed-stream depth 2, branch depth 1, and total draft tokens 8.

Hardware and measurement. Experiments run on one NVIDIA RTX PRO 6000 Blackwell Workstation GPU (96 GB), using PyTorch 2.11.0 (Paszke et al., 2019), CUDA 13.0, Transformers 5.5.4 (Wolf et al., 2020), and batch size one. GPU timing runs are serialized. All methods share SDPA, fused/ordered normalization, exact-shape CUDA graphs, and the target numerical settings. AR, EAGLE-3, and ToolSpec use matching prompts and target settings from same-machine evaluations. Model load, graph setup, warmup, and trace serialization are excluded from E2E time.

B Speculative-Sampling Consistency in the Tool-Call Setting

The target next-token distribution $p_\theta(\cdot \mid P, u)$ is defined at every prefix u of the *serialized* tool-call response. Tool names, field keys, values, punctuation, and closing tokens are all members of the same token sequence. Forecasted slots, SIDE values, and structural bridges are proposals; they do not restrict the target vocabulary or determine the realized call layout.

Greedy consistency at temperature zero. Fix a prompt P and a deterministic tie-breaking rule for g_θ. Assume that every MAIN verification evaluates the same conditional target logits as serial AR decoding at the same token prefix, that the verification tree masks SIDE histories from MAIN, and that only the matching MAIN prefix and the target's first correction are committed. Both decoders use the same stopping rule and, if needed, the same finite length cap. Under these assumptions, SchemaFill emits exactly the greedy AR token sequence.

Proof. Induct on the length of the committed prefix m. Initially m is empty, hence agrees with the AR output. Suppose m agrees with AR and let $z = (z_1, \ldots, z_n)$ be any packed MAIN proposal, including possible tokens from future slots or calls. By Equation 2, each accepted z_i equals $g_\theta(P, m \| z_{<i})$. Therefore appending the accepted prefix follows AR token by token. If $r^\star < n$, the next committed token is $g_\theta(P, m \| z_{\le r^\star})$, which is precisely AR's next token; the remaining proposal is discarded. If $r^\star = n$, the entire accepted proposal is an AR prefix and the next round starts from that prefix. SIDE acceptance cannot alter this argument because a SIDE token is never com-

mitted without a fresh MAIN verification under the actual prefix. The induction continues through proposed closing tokens and ends at the common stopping condition. □

Distributional consistency under exact speculative sampling. We apply the standard speculative-sampling rule (Leviathan et al., 2023; Chen et al., 2023) to the serialized tool-call setting. The following result concerns distributional consistency under a stochastic target policy. At any MAIN prefix u, let $p_u(v) = p_\theta(v \mid P, u)$ be the target next-token distribution at the selected temperature. Conditional on the proposal state available before the next acceptance decision, let $q_u(v)$ be the distribution of the next proposed MAIN token. This state may include public metadata and provisional SIDE values, but must not change p_u; the acceptance draw must be fresh and independent of proposal randomness. A fixed proposed structural or slot token corresponds to a point-mass q_u. For a proposed token v with $q_u(v) > 0$, use the speculative-sampling acceptance probability

$$\alpha_u(v) = \min\left(1, \frac{p_u(v)}{q_u(v)}\right). \tag{6}$$

On rejection, draw a correction from the residual distribution

$$\rho_u(w) = \frac{p_u(w) - \min\{p_u(w), q_u(w)\}}{1 - \sum_{v \in \mathcal{V}} \min\{p_u(v), q_u(v)\}}, \tag{7}$$

whenever its denominator is positive. If the denominator is zero, rejection has probability zero and ρ_u is not used. The accepted token or residual correction is committed; after a rejection, all later tokens in that proposal are discarded. After a fully accepted proposal, decoding resumes at the next target position. Under exact target conditionals and a common stopping rule with almost-sure termination or a common finite length cap, this rule produces the same distribution of stopped tool-call responses as serial sampling from p_θ.

Proof. Fix u and the pre-decision proposal state. The probability of proposing and accepting w is $q_u(w)\alpha_u(w) = \min\{p_u(w), q_u(w)\}$. The rejection probability is $R_u = 1 - \sum_v \min\{p_u(v), q_u(v)\}$. For $R_u > 0$, the probability that a rejection emits w is $R_u \rho_u(w) = p_u(w) - \min\{p_u(w), q_u(w)\}$. Hence the total probability of the next committed token is $p_u(w)$. When $R_u = 0$, the accepted-token term alone equals $p_u(w)$ for every w. This calculation applies again at each new MAIN prefix, including prefixes that cross field or call boundaries: the target conditional is always evaluated under the full serialized prefix, while a SIDE value remains a proposal until checked on MAIN. By induction on output position, every finite committed prefix has the same probability as under serial target sampling. Applying the same deterministic stopping rule to these prefixes gives the same distribution of complete responses. □

The greedy result assumes identical target conditionals at each prefix under exact arithmetic. The stochastic extension additionally uses the acceptance and residual-correction rules above.

C Additional Measurement Details

C.1 Evaluation Sets and Reference Workload Counts

The evaluation uses the Glaive and BFCL test sets described in Section 5. For reference workload statistics, we count the top-level keys of each reference call's `arguments` object and sum them over calls at each decision point. The broader automatically eligible Glaive source pool contains 27,278 deduplicated tool-call decisions (27,278 calls and 51,293 fields), extracted from 52,893 source conversations after structural and quality checks and removal of training/development overlaps. These Glaive counts describe the source pool, not the test set. The BFCL counts cover the complete test set (1,333 calls and 3,355 fields): 50 executable-parallel, 200 parallel, and 200 parallel-multiple. For the executable-parallel category, the locally available frozen processed references supply the 50 labels; the other two categories use their official answer files and the same representative-answer conversion as the test set. Reference annotations are used for these offline statistics and are not inputs to SchemaFill.

C.2 Per-Request KV-Cache Overhead

Table 4: SchemaFill's per-request effective KV-cache overhead with an EAGLE-3 drafter, BF16 weights, and batch size one. KV-cache growth is relative to greedy AR with the same target-KV layout. The model-and-cache comparison holds target and drafter weights fixed and measures the memory increase due to extra KV.

Target model	KV-cache increase	Model-and-cache increase from extra KV
Qwen2.5-7B-Instruct	$+23.07\%$	$+0.23\%$

C.3 BFCL Runtime Time Breakdown

Table 5: SchemaFill time breakdown on the BFCL test set. Percentages use the sum of the four profiled phases.

Phase	Total (s)	Mean (ms/request)	Share (%)
BM25 tool forecast	0.011	0.023	0.015
EAGLE-3 drafting	13.71	30.46	19.24
Target verification	52.64	116.97	73.90
Other generation	4.87	10.83	6.84
Total	71.23	158.29	100.00

We profiled all BFCL test requests with the production configuration of Table 1. BM25 time covers lexical ranking of public tool metadata during forecast preparation. Drafting covers EAGLE-3 candidate generation; target verification covers the cached shared-tree verifier, including tree construction and result evaluation. The remaining generation time covers scheduling, candidate packing, acceptance, KV updates, and stopping. Timed draft and verification calls use CUDA synchronization at their boundaries, so the four rows partition the profiled time. Prompt construction, tokenization, scheduler initialization, target prefill, model loading, graph setup, and warmup are excluded from this breakdown.

Per-phase CUDA synchronization increased end-to-end time from 76.81 to 79.27 seconds; Table 1 reports throughput measured without phase-level synchronization. BM25 scoring is bypassed when only one tool is available.

C.4 Run-to-Run Variance

For the repeated launches in Table 1, sample variance is computed with denominator $n - 1 = 2$. Decimal-valued measurements are rounded to two decimal places. TPS variances for Glaive/BFCL are: AR $0.00/0.02$, EAGLE-3 $0.15/0.49$, ToolSpec $0.10/2.40$, and SchemaFill $0.35/0.29$. SchemaFill's output tokens and target-call counts are identical across its three runs on each dataset, so its observed AL variance is 0.00. In Table 2, stages A and D reuse the three-launch means from Table 1, while B and C each use one run over all BFCL test requests.

C.5 Experiments at Temperature One

We conduct additional experiments at temperature 1 on Glaive and BFCL for AR, EAGLE-3, and SchemaFill. Table 6 reports the results.

Table 6: End-to-end results at temperature 1 for AR, EAGLE-3, and SchemaFill. Each of these methods uses one run per dataset, with relative TPS normalized to AR at temperature 1. ToolSpec is included as a temperature-0 reference; its metrics and relative TPS are retained from Table 1.

Dataset	Method	Accept Length	E2E TPS	Relative TPS
Glaive	AR	1.00	90.78	1.00×
	EAGLE-3 (Li et al., 2025)	2.78	176.90	1.95×
	ToolSpec (Xia et al., 2026)	2.79	202.15	2.21×
	SchemaFill	**5.60**	**297.99**	**3.28×**
BFCL v3	AR	1.00	92.11	1.00×
	EAGLE-3	2.86	198.82	2.16×
	ToolSpec	3.42	257.32	2.77×
	SchemaFill	**6.82**	**387.55**	**4.21×**

At temperature 1, SchemaFill achieves speedups of $3.28\times$ on Glaive and $4.21\times$ on BFCL over AR at the same temperature.

C.6 Slot-Order Diagnostic Model

In the BFCL slot-order diagnostic described in Section 3.3, a source- and call-count-adjusted response-level logistic model associates each additional parameter with $1.35\times$ higher odds of at least one AR deviation (95% interval: 1.21–1.56, estimated by bootstrap resampling (Efron, 1979)). This model is descriptive; the diagnostic uses AR-selected layouts and characterizes candidate behavior rather than estimating held-out end-to-end performance.

D Cross-Target Evaluation

We additionally evaluate Qwen3-8B (Yang et al., 2025), Mistral-7B-Instruct-v0.3 (Jiang et al., 2023; Mistral AI, 2024), and Hermes-3-Llama-3.1-8B (Teknium et al., 2024) on the same BFCL test set. Mistral and the Llama-based Hermes provide non-Qwen targets. Each target has its own EAGLE-3 head, shared by EAGLE-3 and SchemaFill. The heads use target hidden-state indices $[2, 18, 33]$ for Qwen3, and $[2, 16, 29]$ for Mistral and Hermes. Training uses fresh target-generated continuations from separate public requests. The Mistral head was continued for four epochs from its epoch-two weights with a reset optimizer; the other new heads ran for six consecutive epochs. Hermes uses 11,609 positive-mask training records and 553 development records, with a 15,611-token draft vocabulary. Qwen3 uses its non-thinking chat template; Mistral and Hermes use their native chat templates; Qwen2.5 uses the plain-text task prompt. All four decoders receive identical prompt tokens within each target.

All targets use the Table 1 base settings: forecast budget three, $k = 3$, MAIN depth two, SIDE depth one, and total draft budget eight. AR, EAGLE-3, and SchemaFill are evaluated at temperatures 0 and 1, using greedy target decoding and stochastic target decoding, respectively. ToolSpec's temperature-0 results are included as a reference. The optimized backends enable CUDA

graphs, ordered fused normalization, and expanded rotary/KV execution; SchemaFill also uses shared target-prefix reuse, hybrid prompt attention, and multi-branch tree drafting. Each drafter retains its training-time rotary configuration. Table 7 reports one full launch per learned-drafter method, target, and temperature, including request initialization and failed attempts. Model loading and one-time graph setup are outside request time. These single-run measurements are separate from the repeated-run estimates in Table 1.

Table 7: Cross-target evaluation on the BFCL test set at temperatures 0 and 1 for AR, EAGLE-3, and SchemaFill. TPS is end-to-end throughput, and AL includes initialization. Relative TPS for these methods is normalized to AR at the corresponding temperature. ToolSpec's temperature-0 metrics and relative TPS are repeated in both blocks as a reference.

Model	Method	Temperature = 0			Temperature = 1		
		TPS	AL	Relative TPS	TPS	AL	Relative TPS
Qwen2.5-7B	AR	93.07	1.00	1.00×	92.16	1.00	1.00×
	EAGLE-3	204.15	2.87	2.19×	198.52	2.86	2.15×
	ToolSpec	257.19	3.44	2.76×	257.19	3.44	2.76×
	SchemaFill	**406.21**	6.97	**4.36×**	**389.98**	6.83	**4.23×**
Qwen3-8B	AR	80.30	1.00	1.00×	79.76	1.00	1.00×
	EAGLE-3	180.59	2.79	2.25×	176.10	2.78	2.21×
	ToolSpec	226.65	3.49	2.82×	226.65	3.49	2.82×
	SchemaFill	**328.98**	6.47	**4.10×**	**315.30**	6.39	**3.95×**
Mistral-7B	AR	87.58	1.00	1.00×	87.04	1.00	1.00×
	EAGLE-3	233.78	3.34	2.67×	229.66	3.35	2.64×
	ToolSpec	249.90	3.57	2.85×	249.90	3.57	2.85×
	SchemaFill	**352.75**	6.43	**4.03×**	**332.37**	6.05	**3.82×**
Hermes-3-8B	AR	84.01	1.00	1.00×	83.57	1.00	1.00×
	EAGLE-3	206.41	3.07	2.46×	199.25	3.04	2.38×
	ToolSpec	249.10	3.59	2.96×	249.10	3.59	2.96×
	SchemaFill	**352.11**	6.62	**4.19×**	**333.25**	6.38	**3.99×**

SchemaFill exceeds AR and EAGLE-3 on all four targets at both temperatures. At temperature 0, its advantage over ToolSpec ranges from 1.41× to 1.58×. At temperature 1, it achieves 1.45×–1.96× EAGLE-3 throughput and 3.82×–4.23× AR throughput under the same temperature setting. Raising the temperature from 0 to 1 reduces SchemaFill's throughput and AL across all four models, while its AL remains above six committed output tokens per target call.