AGSPEC: PUSHING THE LIMITS OF RETRIEVAL-BASED SPECULATIVE DECODING IN CODING AGENT PIPELINES

Anonymous authors
Paper under double-blind review

ABSTRACT

Retrieval-based speculative decoding (SD) drafts tokens by copying continuations from existing text, which suits coding agents that repeatedly reproduce code, logs, and earlier attempts. Yet existing methods fall short in agent pipelines: much of the reusable text is missing from their corpora or stored in a form that differs from what the agent emits, and their draft lengths ignore that accept length varies across agents and drifts over turns. We present AGSPEC, a framework that supplies the corpus and draft-length policies that existing retrieval engines lack in coding-agent pipelines. AGSPEC retrieves from session, workspace, and global corpora, retaining the ongoing session trajectory and indexing opened files in the agent's emission format. It bounds each agent's draft length with an offline-profiled cap and adapts the length online from verification feedback. On two repository-level multi-agent coding benchmarks, AGSPEC outperforms five retrieval-based drafters and EAGLE-3 in most evaluated settings, raising generation throughput over autoregressive decoding up to 4.37× at batch size 1 and 4.76× at batch size 16. AGSPEC also remains effective on benchmarks without a repository or a multi-agent pipeline, showing that its gains generalize to coding agents broadly.

1 INTRODUCTION

Coding agents have become a representative LLM application (Liu et al., 2026a). Unlike a chatbot, a coding agent handles a single user request as a multi-turn session that alternates between model calls and tool execution (Yang et al., 2024). Within a session, the work is often divided among specialized agents, such as a coder that edits files and an executor that runs tests and scripts to check the result (Hong et al., 2024). As user requests have shifted from synthesizing a single function (Chen et al., 2021) to resolving repository-level issues (Jimenez et al., 2024), agents rarely succeed in one pass and instead refine their attempts over multiple turns (Madaan et al., 2023). Each additional turn can improve the output, but it also lengthens the time a user waits for the result (Tiwari et al., 2026).

Speculative decoding (SD) attacks this cost directly, reducing decoding latency without changing the output (Chen et al., 2023). In SD, a drafter proposes several tokens, and the target model verifies them in parallel and commits the accepted tokens. Each accepted token saves a sequential target-model step. This saving comes at the cost of generating draft tokens and verifying tokens that are later rejected. Rejected tokens are nearly free at small batch sizes but become expensive at large batch sizes, where verification demands much more compute (Liu et al., 2024; Sadhukhan et al., 2025). SD thus pays off only when the drafter is both cheap and accurate. Existing designs range from small draft models (Leviathan et al., 2023) and trained draft heads (Li et al., 2025) to retrieval-based SD, which retrieves draft tokens from a text corpus.

Retrieval-based SD is particularly effective for code generation (Liu et al., 2026b). It copies continuations from text the system already has, such as the current prompt, earlier responses, or a prebuilt datastore, without running a separate draft model. Code suits this copying, since identifiers, code fragments, and repository conventions recur, and coding-agent sessions offer still more of it, as repeated subtasks and refinement loops produce long, predictable sequences. However, previous works do not elicit the full performance of retrieval-based SD at coding agents. They overlook what makes a coding agent different, and leave much of that reuse out of reach.

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.

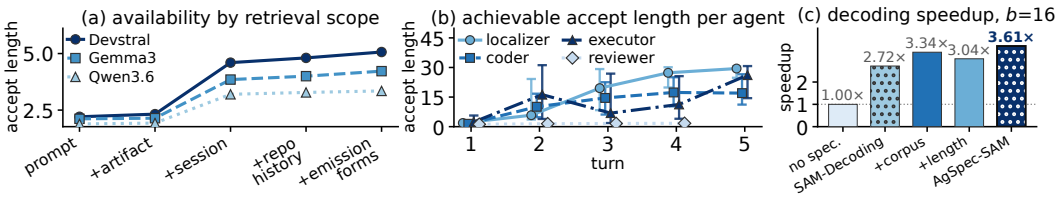


Figure 1: **(a)** Accept length when restoring availability, then matchability. **(b)** Achievable accept length for Gemma3-27B, taking the best available continuation at each step. Markers are medians and bars the interquartile range. **(c)** Decoding throughput of SAM-Decoding and AGSPEC-SAM relative to no speculation for Gemma3-27B. All data is from SWE-bench Verified.

A draft helps only if two conditions hold: the text the agent reuses must be *available* in the corpus, and it must be *matchable* against what the agent generates. Existing corpora fall short on both. First, they limit availability: they hold either a prebuilt datastore or the text of the current call, whereas coding agents reuse text from the ongoing session and the repository. A file drops out of reach once the prompt truncates it, and a retry cannot draw on the previous attempt once its call ends; covering these requires corpora that span the session and the repository. Second, they limit matchability: they store text as the agent read it, yet a coder emits a patch whose copied lines each carry a diff prefix absent from the file. Fig. 1(a) shows that expanding retrieval scope raises accept length on every model, with the largest gain from session history and smaller additional headroom from repository history and emission-form indexing.

Even when a reusable match exists, choosing the draft length remains critical, as it directly determines how many drafted tokens are rejected. Existing methods fix the draft length policy before a session runs, deriving it from a static cap (Saxena, 2023; Hu et al., 2025; He et al., 2024; Zhao et al., 2025) or from properties of the match such as its length (Oliaro et al., 2025) or occurrence count (He et al., 2024; Zhao et al., 2025; Oliaro et al., 2025). However, in coding agents, the achievable accept length follows the agent pipeline. Fig. 1(b) shows that each agent occupies a distinct accept-length range that shifts differently across and within turns. Each agent and context thus has a characteristic accept length, which can guide the draft length better than a fixed or match-derived policy.

These limitations stem not from how existing methods match text but from the policies around the matching. A retrieval-based SD method pairs a retrieval engine that finds a continuation for the current context (*e.g.,* a suffix tree in SuffixDecoding (Oliaro et al., 2025)) with policies that decide what the corpus holds and how many tokens to draft. Existing policies are built around the individual request, the unit an inference engine serves: they index the current call, a prebuilt datastore, or outputs pooled across requests (Saxena, 2023; He et al., 2024; Oliaro et al., 2025), and size drafts by a static cap or the match itself. In a coding agent, however, reuse and acceptance depend on structure above the request, namely the session, the repository, and the role of each agent.

We present AGSPEC, a retrieval-based SD framework that builds its policies on this structure while leaving the engine unchanged, so it applies to existing engines as is. AGSPEC realizes each policy with a simple yet effective design. **(1) Coding-agent-aware retrieval.** For the corpus, AGSPEC organizes text by origin into session, workspace, and global reference corpora. The workspace corpus indexes the artifacts a session opens, in full and in the forms the agent emits. **(2) Agent-aware draft-length control.** For the draft length, AGSPEC profiles agent-specific accept lengths offline to set per-agent draft caps, and learns a draft-length scale that adapts the length per turn and per request from verification feedback.

We compare AGSPEC against AR decoding, five retrieval-based SD methods (Saxena, 2023; He et al., 2024; Oliaro et al., 2025; Hu et al., 2025; Zhao et al., 2025), and EAGLE-3 (Li et al., 2025) on two repository-level benchmarks, SWE-bench and TeamBench (Jimenez et al., 2024; Kim et al., 2026). AGSPEC consistently outperforms most baselines. Especially, with SAM-Decoding as its retrieval engine, AGSPEC raises the speedup of Gemma3-27B on SWE-bench over AR decoding from 2.72× to 3.61× at batch size 16 (Fig. 4). AGSPEC also generalizes beyond this setting, achieving the highest throughput on single-agent and non-repository benchmarks. These results highlight the effectiveness of AGSPEC as a training-free acceleration framework for coding agent applications.

2 BACKGROUND

2.1 CODING AGENTS

Coding agents resolve software engineering tasks by handling each user request as a *session* (Jimenez et al., 2024). A session may span several agents, each of which alternates between *model calls* and tool executions. Throughout the session, agents operate on a persistent *workspace*: the repository checkout together with its execution environment. The workspace holds the *artifacts* agents work on, such as repository files, patches, and tool outputs. Unlike conversation history, artifacts are workspace state; in particular, most repository files predate the session and outlive it. However, what the workspace holds differs from what a model call sees: each prompt exposes only the artifacts the scaffold selects, often truncated to fit the context length (Yang et al., 2024; Wang et al., 2025).

Even when an agent reuses workspace text, its output rarely reproduces it verbatim. An agent reads an artifact as it stands but writes it back in the scaffold's *emission format*. Under unified-diff editing (Free Software Foundation, 2026), a call emits a header naming the file, followed by kept lines prefixed with a space and removed lines prefixed with a minus, so every line copied from the artifact begins with a character the artifact does not contain (Fig. 2). The code is identical, but the token sequence differs, so text indexed as the agent read it may not match what the agent generates. Other scaffolds emit whole files, search-and-replace blocks, or line-addressed edit commands (Xia et al., 2025; Wang et al., 2025; Yang et al., 2024), and the choice measurably affects editing quality (Gauthier, 2023). Because the format is a property of the scaffold, it is known before the session runs.

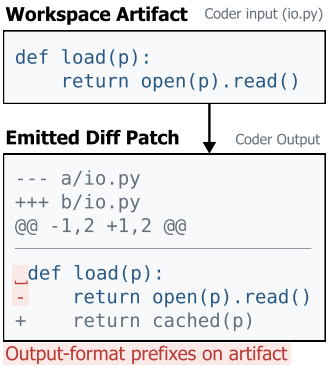


Figure 2: The same code in two forms: as the workspace holds it, and as a unified-diff patch emits it. Red marks the prefix each copied line gains.

The extent to which a call's output repeats earlier text also depends on what the call does and when it is made. Within a session, each model call is issued by an agent, serves a *role*, and belongs to a *turn*. Roles are the functions a session performs, and each generates a different kind of text: a localizer names paths, a coder rewrites files the session has already read, and an executor reproduces test logs. A role describes what a call does rather than which agent issues it: multi-agent systems assign each role to a specialized agent (Hong et al., 2024; Qian et al., 2024; Phan et al., 2024), whereas others fix roles as pipeline stages (Xia et al., 2025) or interleave them within one agent's loop (Yang et al., 2024). A turn comprises a solution attempt and its execution feedback, possibly spanning several roles; failed attempts trigger further turns until the task completes or a retry limit is reached (Yang et al., 2024). Across turns, agents return to the same files and repeat much of the previous attempt. Roles thus determine what kind of text a call generates, and turns determine how much of it repeats.

2.2 RETRIEVAL-BASED SPECULATIVE DECODING

Retrieval-based SD obtains drafts from existing text rather than a draft model: it matches a suffix of the decoding context, such as the prompt and generated prefix, against a corpus and copies the continuation that follows. Its effectiveness hinges on two design choices. The *corpus scope*, which text is indexed and how long it is retained, determines whether a useful match exists. The *draft length*, how many matched tokens are proposed, determines how many of them are rejected.

Existing methods differ in both what their corpora index and how long they retain it. Request-local corpora last for a single call. PLD searches the current prompt and generated prefix (Saxena, 2023), and SAM-Decoding and SuffixDecoding build a per-request suffix automaton or tree over the prompt and output (Hu et al., 2025; Oliaro et al., 2025). Prebuilt corpora are assembled before serving and stay fixed. REST retrieves from a text or code datastore (He et al., 2024), SAM-Decoding pairs its request-local automaton with a static datastore (Hu et al., 2025), and FastCoder adds repository-local files to common code (Zhao et al., 2025). A few corpora persist across requests: SuffixDecoding keeps a global tree of past responses (Oliaro et al., 2025; Snowflake Inc., 2026), and FastCoder caches verified and generated sequences (Zhao et al., 2025).

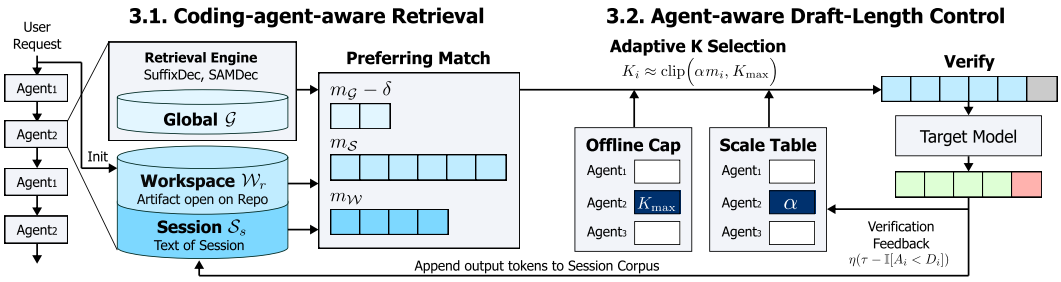


Figure 3: Overview of AGSPEC. Retrieval consults the session, workspace, and global corpora. Verification extends the output, updates the session corpus, and provides feedback to the length controller. The workspace corpus grows as artifacts are opened and is discarded when the session ends.

Draft lengths are either fixed or scaled by the match. PLD and SAM-Decoding cap each continuation at a fixed length, and REST uses a fixed tree length (Saxena, 2023; Hu et al., 2025; He et al., 2024). SuffixDecoding scales its length with match length by a configured factor (Oliaro et al., 2025), so its draft length varies with the input, but the factor itself is set before serving.

These methods accelerate both code generation and agent workloads (He et al., 2024; Zhao et al., 2025; Oliaro et al., 2025), yet they remain limited in the agentic setting. First, agents reuse text from both the ongoing session and the repository workspace, and emit it in the scaffold's emission format. A corpus should therefore span both sources and index text in that format, yet existing methods each cover only some of these sources. Second, their draft lengths ignore the contexts that shape acceptance: they are fixed (Saxena, 2023; He et al., 2024; Hu et al., 2025) or scaled by a preset factor (Oliaro et al., 2025), never corrected by verification outcomes, and blind to roles and turns. AGSPEC addresses these limitations with a source- and emission-aware corpus (§3.1) and an adaptive draft length (§3.2).

3 AGSPEC DESIGN

AGSPEC is a retrieval-based SD framework for coding agents that builds on an existing retrieval engine. A retrieval engine finds the longest suffix match of the decoding context in a corpus and returns its continuation, using an index such as the suffix automaton in SAM-Decoding (Hu et al., 2025) or the suffix tree in SuffixDecoding (Oliaro et al., 2025). AGSPEC operates on top of the engine without modifying it, so it applies to any engine that can search multiple corpora and accept a per-step draft length. AGSPEC combines corpora organized by token source (§3.1) with verification-driven draft-length control (§3.2). Fig. 3 depicts the overview of AGSPEC.

3.1 CODING-AGENT-AWARE RETRIEVAL

AGSPEC addresses the corpus gap by indexing the reusable text in separate corpora keyed by source. Three mechanisms build on this separation: each corpus captures one source of reusable text, workspace artifacts are indexed in the forms the agent emits, and a preference order ranks matches across corpora by how closely their source relates to the active session.

Three corpora by source. AGSPEC maintains one corpus per source of reusable text. The *session corpus* $\mathcal{S}_s$ holds the text of session s, including prompts, tool outputs, and generated tokens, and is shared across its calls and agents, so text from earlier calls and turns remains retrievable. The *workspace corpus* $\mathcal{W}_r$ holds the repository artifacts opened during the current session and grows as additional artifacts are accessed. The *global corpus* $\mathcal{G}$ holds a large body of static reference text that is not specific to the current session; it is assembled before serving and stays fixed. These corpora are kept separate because their provenance and lifetime differ: the session and workspace corpora, which we call *live*, are constructed within the current session and discarded when it ends, whereas the global corpus remains fixed across sessions.

Indexing artifacts in emission form. Covering the workspace is not enough if its text no longer matches what the agent generates. The first time a call opens an artifact, AGSPEC adds it to $\mathcal{W}_r$ in

two forms. It adds the file as is, so the file remains retrievable even after truncation drops it from the prompt. It also adds the file as the agent would write it: because a unified-diff patch prefixes kept lines with a space and removed lines with a minus (Fig. 2), AGSPEC adds one copy with every line prefixed by a space and another with every line prefixed by a minus. Any line the coder copies into its patch thus matches one of these copies. A scaffold that writes whole files through a tool call instead needs one extra copy: the file as that call encodes it, with its newlines and quotes escaped. Later calls in the session reuse these entries instead of reindexing the file.

Preferring matches by source. The retrieval engine queries each corpus with the same suffix of the decoding context and reports its longest match length $m_{\mathcal{S}}$, $m_{\mathcal{W}}$, and $m_{\mathcal{G}}$. Following prior retrieval-based SD (Oliaro et al., 2025; Hu et al., 2025), AGSPEC treats a longer match as a better draft source, since more of the recent context agrees with the corpus text. Between the two live corpora, the longer match wins, with ties going to the session, whose text reflects the ongoing session. The global corpus must beat the live match by a margin δ, because its text is plentiful and unrelated to the repository, so a longer match there is likely coincidental. Formally, AGSPEC drafts from $\mathcal{G}$ only when $m_{\mathcal{G}} > \max(m_{\mathcal{S}}, m_{\mathcal{W}}) + \delta$; otherwise, it drafts from $\mathcal{S}_s$ if $m_{\mathcal{S}} \geq m_{\mathcal{W}}$, and from $\mathcal{W}_r$ if not. Appendix C includes a δ sweep study.

Continuation selection. Within the chosen corpus, a match may occur at multiple positions with different continuations, so AGSPEC must select one occurrence. For both the global corpus $\mathcal{G}$ and the session corpus $\mathcal{S}_s$, AGSPEC groups occurrences by their next three tokens and selects the most frequent group, breaking ties by the most recent occurrence. For the workspace corpus $\mathcal{W}_r$, it selects the first occurrence. Appendix D includes the detailed analysis.

Together, these mechanisms let AGSPEC draft from text that existing corpora miss: the session's own outputs and whole workspace files in the agent's emission format, while preferring the source most relevant to the ongoing session.

3.2 AGENT-AWARE DRAFT-LENGTH CONTROL

A draft token saves a decoding step if accepted but wastes verification if rejected, a waste that grows with batch size as verification becomes compute-bound. The draft length should therefore track how likely each token is to be accepted, which existing methods misjudge in two ways. A static cap ignores that accept length differs across agents and drifts across and within turns (Fig. 1(b)). Match-derived rules (Oliaro et al., 2025) size the length by the match alone, regardless of which agent emits it or when. Both over-draft where acceptance is short. AGSPEC instead estimates the draft length from two sources: an offline-profiled cap for stable differences across agents, and an online scale that adjusts it to runtime drift from verification feedback. This cuts rejected tokens at a small cost in accepted ones, raising throughput most at large batch sizes.

Overview. At decoding step i, AGSPEC sets the draft length for the requesting agent a_i as

$$K_i = \mathrm{clip}\big(\mathrm{round}(\alpha^{(a_i)} m_i),\, 1,\, K_{\max}^{(M,a_i)}\big), \tag{1}$$

where m_i is match length, $\alpha^{(a_i)}$ is the agent's online scale and $K_{\max}^{(M,a_i)}$ is its offline cap under target model M; the lower bound ensures that every match yields at least one draft token. The drafter copies up to K_i tokens that follow the match, so the actual draft length is $D_i = \min\{K_i, \ell_i\}$, where ℓ_i is the number of tokens available after the match. Verification returns the accept length A_i, excluding tokens generated directly by the target model. As some retrieval engines, such as SuffixDecoding (Oliaro et al., 2025), have their own draft length control policy, AGSPEC uses the smaller value for the draft length between our policy and theirs.

Offline profiling of $K_{\max}$. AGSPEC profiles the cap from the trajectories already collected to build $\mathcal{G}$. It replays them and records, at each retrieval step, the reusable length R: the number of retrieved tokens that match the recorded continuation. The distribution of R for each agent gives the probability $\Pr(R \geq k \mid M, a)$ that the k-th draft token would be accepted, which falls as k grows. A draft token pays off only if this probability exceeds λ, the acceptance probability at which one more draft position covers its verification cost (Appendix A). The cap is therefore the last position that still pays off:

$$K_{\max}^{(M,a)} = \max\big\{k : \Pr(R \geq k \mid M, a) \geq \lambda\big\}. \tag{2}$$

For example, suppose $\lambda = 0.1$, so that verifying one more position costs as much time as producing 0.1 tokens. If an agent's 11th draft token is accepted with probability 0.12 and its 12th with 0.09,

the 11th returns 0.12 tokens in expectation for a cost of 0.1, whereas the 12th returns only 0.09; the cap is therefore 11.

Online adaptation of α. Within a session, the useful length for the same agent drifts across and within turns. AGSPEC therefore adapts each agent's scale from verification feedback. Each scale starts at 1.0, which requests exactly as many tokens as were matched, and after step i is updated as

$$\alpha^{(a_i)} \leftarrow \alpha^{(a_i)} + \eta\big(\tau - \mathbb{I}[A_i < D_i]\big), \tag{3}$$

where $\mathbb{I}[\cdot]$ is the indicator function and η is the step size. Under the idealization in Appendix G.1, Equation (3) is a stochastic subgradient step on the quantile loss $\rho_\tau(R_i/m_i - \alpha^{(a_i)})$ (Koenker & Bassett, 1978), where R_i is the number of tokens the target would accept from an unbounded draft. The scale thus tracks the τ-quantile of R_i/m_i, targeting full acceptance ($A_i = D_i$) with probability $1-\tau$; setting $\tau = 1-\lambda$ aligns this target with the cap's break-even point, now for the current session rather than the agent's history. We skip the update when a fully accepted draft reaches $K_{\max}^{(M,a_i)}$, since the cap censors the observation. The scales persist across sessions served by the same server and are initialized once at server start.

Together, the cap and the scale apply one criterion, drafting a token only if it is likely enough to be accepted to cover its verification cost: the cap from each agent's past trajectories, and the scale from recent verification feedback.

4 EVALUATION

4.1 EXPERIMENTAL SETUP

Models. We evaluate Devstral-24B (Mistral AI & All Hands AI, 2025), Gemma3-27B (Gemma Team, 2025), and Qwen3.6-27B (Qwen Team, 2026).

Benchmarks. Our main evaluation uses two repository-level benchmarks, SWE-bench Verified (Jimenez et al., 2024) and TeamBench (Kim et al., 2026). On SWE-bench Verified, five agents—localizer, coder, executor, reviewer, and summarizer—run in containers for up to five turns, with the coder editing repository files through unified diffs. TeamBench provides each task with a repository workspace and spans eighteen task families, including code review, incident response, and infrastructure changes. A planner, executor, and verifier proceed through planning, execution, and verification, with up to 20 turns per phase and two remediation loops, capped at 30 LLM calls per session. We additionally evaluate AGSPEC on LiveCodeBench (Jain et al., 2025) and Terminal-Bench (Merrill et al., 2026) to separately test generality: LiveCodeBench has no repository, while Terminal-Bench uses a single agent for all roles (§4.3). We measure generation throughput excluding prefill and accept length (AL) for all benchmarks.

Baselines. We compare autoregressive (AR) decoding, PLD (Saxena, 2023), REST (He et al., 2024), FastCoder (Zhao et al., 2025), SuffixDecoding (Oliaro et al., 2025), SAM-Decoding (Hu et al., 2025), and EAGLE-3 (Li et al., 2025). We follow the draft-length settings in the original papers, with configuration and implementation details reported in Appendix A. FastCoder extends REST with a project datastore, a cache of verified drafts and model output, and a rule that skips retrieval where it rarely succeeds; we attach its project datastore on both SWE-bench Verified and TeamBench.

AGSPEC Implementation. We implement AGSPEC in vLLM (Kwon et al., 2023) over two retrieval engines: the suffix automaton of SAM-Decoding (AGSPEC-SAM) (Hu et al., 2025) and the suffix tree of SuffixDecoding (AGSPEC-Suffix) (Oliaro et al., 2025). Both engines use the same corpus configuration, draft-length controller, per-agent caps, and workspace scope, differing only in the underlying retrieval structure. For AGSPEC-SAM, we use frequency-majority continuation selection at both batch sizes. The maximum draft length is 64 at batch size 1 and 24 at batch size 16 for both implementations. Detailed configuration is at Appendix A.

Corpus construction. For each benchmark, we construct the global corpus from agent turn logs collected on a disjoint set of coding tasks before evaluation. We use 70 tasks from SWE-bench Verified and 213 from TeamBench, indexing generated text and tool calls rendered in each model's native format. Evaluation tasks are held out from corpus construction.

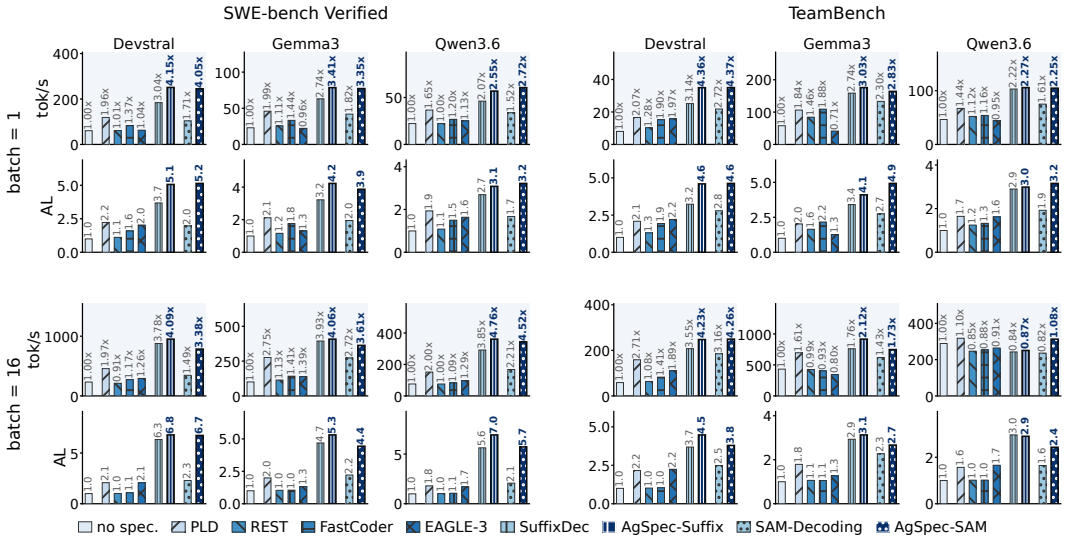


Figure 4: Main comparison across two repository benchmarks and three models. Rows show generation throughput and AL at batch sizes 1 and 16; higher is better. The number above each bar is its ratio to the no-speculation bar in that panel. no spec. denotes AR generation.

4.2 Main Evaluation

Fig. 4 compares AgSpec with autoregressive decoding and six prior speculative methods on SWE-bench Verified and TeamBench, using three models at batch sizes 1 and 16. Within each panel, all methods share the same workload and serving configuration.

AgSpec achieves the highest or the second highest throughput in all evaluated settings, reaching 2.27–4.37× the throughput of autoregressive decoding at batch size 1 and 1.08–4.76× at batch size 16. On average, its throughput is 18.0% higher than that of the fastest prior method. The gains are largest with Devstral at batch size 1 on SWE-bench Verified, achieving 1.36× higher than the fastest prior work. AgSpec also achieves the highest AL in most evaluated settings. This demonstrates the effectiveness of AgSpec for coding agents.

Because AgSpec runs on top of the retrieval engine, we also evaluate each engine with and without it. Both engines benefit in most settings. With SuffixDecoding, throughput rises by 16.5% on average and AL by 18.5%; with SAM-Decoding, throughput rises by 64.6% and AL nearly doubles. The gap reflects what each engine already provides. SuffixDecoding keeps a tree of past responses and scales its draft with the match length, whereas SAM-Decoding does neither. Thus, although SAM-Decoding trails SuffixDecoding in all twelve settings, AgSpec-SAM outperforms it in nine. These results show that existing retrieval engines, on their own, leave much of the speedup available in coding-agent pipelines untapped and AgSpec, as a framework, accelerates existing retrieval engines in coding-agent pipelines without modification.

4.3 Non-Repository and Implicit-Agent Tests

AgSpec primarily targets repository-level coding tasks performed by multi-agent systems. In practice, however, coding workloads also include simpler scenarios, such as standalone code generation or single-agent pipelines like ReAct (Yao et al., 2023). To evaluate whether AgSpec generalizes to these settings, Fig. 5 shows its performance without a repository workspace or explicit agent roles.

On LiveCodeBench, a coder and feedback agent work on each problem for up to five turns without a repository. Across the three models, the faster AgSpec variant reaches 4.87–7.94× the generation throughput of autoregressive decoding and exceeds the fastest prior speculative method by 35–145%. Its accept length also exceeds that of the prior method with the highest accept length. Although the workspace corpus is unavailable, AgSpec can still retrieve from the global corpus and text

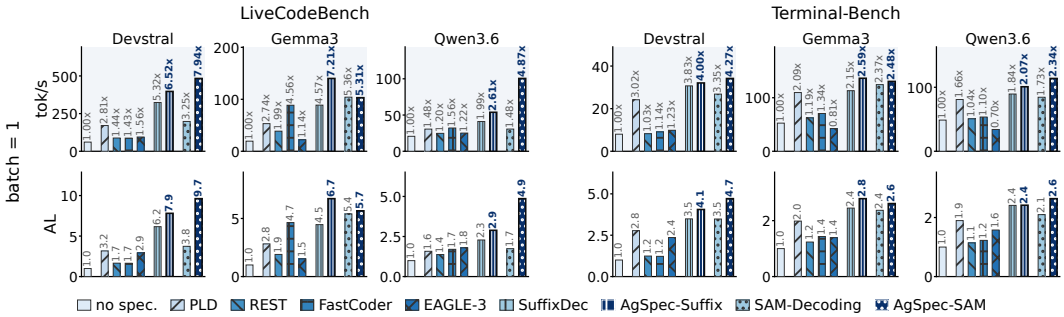


Figure 5: Non-repository and single-agent evaluation at batch size 1 across three models. LiveCodeBench evaluates AGSPEC without repository context, using the global reference and session corpora with an empty workspace corpus. Terminal-Bench evaluates AGSPEC when agent functions are not represented by separate explicit roles.

accumulated within the session, while its draft-length controller continues to adapt from verification feedback, yielding these gains.

On Terminal-Bench, a single Terminus agent inspects files, takes actions, and interprets feedback without separate role assignments. The faster AGSPEC variant improves throughput over the fastest prior speculative method by 9.3–27.2%, while its accept length is 16.7–34.3% higher than that of its corresponding retrieval-engine baseline. For draft-length control, AGSPEC uses a shared cap and a single α table instead of role-specific caps, updating the scale from verification feedback. Thus, AGSPEC can improve decoding even when the workflow has only one explicit agent. Together, these results show that AGSPEC applies to a broader range of coding agents.

4.4 ANALYSIS

4.4.1 IMPACT OF CODING-AGENT-AWARE RETRIEVAL

We measure the contribution of coding-agent-aware retrieval through trace-driven simulation. In particular, we examine two design choices: using three corpora from different sources and indexing artifacts based on emission format. We replay the drafter over traces of AGSPEC-SAM sessions on SWE-bench Verified under each corpus configuration and compare the expected AL (Fig. 6).

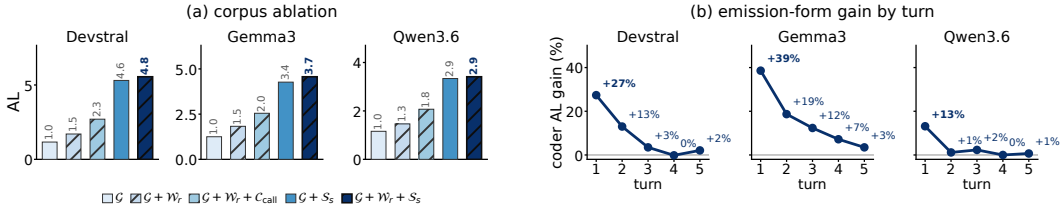


Figure 6: Corpus ablation on SWE-bench Verified at batch size 1. (a) AL as corpora are added; the number above each bar is its ratio to $\mathcal{G}$ alone, and $\mathcal{C}_{\text{call}}$ is a per-call corpus of the current prompt and output. (b) Coder AL gain per turn from indexing workspace artifacts in emission form, relative to raw file text only.

Fig. 6(a) adds corpora to the global corpus one at a time. Retaining text across calls gives the largest gain: replacing a per-call corpus with the session corpus raises AL by 1.6–2.1×, as agents repeatedly reproduce code and outputs from earlier calls. The workspace corpus adds 26–47% on its own and still 2.2–7.0% on top of the session corpus, bringing AGSPEC to 2.9–4.8× the AL of the global corpus alone. This shows that each source contributes significantly to the system.

Fig. 6(b) shows that emission-form indexing helps most in the first turn, raising the coder's AL by 13–39% before its own patches enter the session corpus. From the second turn on, those patches already supply the emitted form, so the gain falls to 3% or less by the fifth turn. Together, these

results show that the session corpus supplies most of the reuse, while the workspace corpus and its emission forms cover what the session has not yet accumulated.

4.4.2 Impact of Agent-Aware Draft-Length Control

We measure the contribution of agent-aware draft-length control on SWE-bench Verified with Devstral (Fig. 7). We use SAM-Decoding as the engine since its draft length is a static cap, which lets the comparison reflect our control alone. Within AgSpec-SAM, we compare our control against fixed caps, $K \in \{8, 32\}$, and separately ablate the online scale.

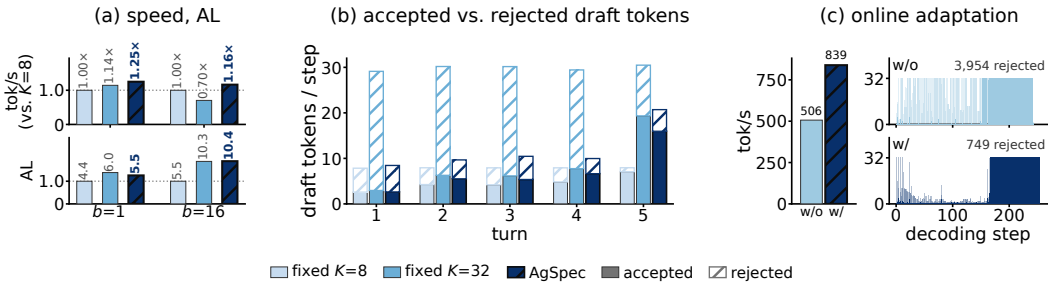


Figure 7: Draft-length control on SWE-bench Verified with Devstral. (a) Generation throughput (top) and AL (bottom) of fixed draft lengths K (every agent capped at K, no online scale) and AgSpec, normalized to $K = 8$ within each group. (b) Mean draft tokens per drafting step by turn at batch size 1, split into accepted (solid) and rejected (hatched) tokens. (c) Fixed $K = 32$ without and with the online scale enabled, at batch size 16: throughput (left) and case study of coder call per decoding step (right), with accepted tokens solid and rejected tokens lightly shaded.

Fig. 7(a) shows that the benefit of long drafts disappears at large batch sizes. At batch size 1, $K = 32$ is 14% faster than $K = 8$, but at batch size 16 it is 30% slower, as rejected tokens consume compute shared by the whole batch. AgSpec is the fastest at both batch sizes, 25% and 16% above $K = 8$. $K = 32$ attains a higher AL than AgSpec at batch size 1 (6.0 vs. 5.5) and a slightly lower AL at batch size 16 (10.3 vs. 10.4), but remains slower at both batch sizes because of the drafts it rejects.

Fig. 7(b) explains this at batch size 1. Rejected tokens add verification cost without producing output, so throughput depends on accepting many tokens while rejecting few. $K = 32$ rejects 11–26 tokens per step, whereas AgSpec rejects at most 6, close to $K = 8$, while still accepting 83–89% as many tokens as $K = 32$. AgSpec lengthens its drafts only as reusable text accumulates, from about 10 tokens in early turns to 21 in the fifth, and thus achieves the highest throughput.

Fig. 7(c) isolates the online scale at batch size 16. To measure its effect alone, we enable only the online scale on top of the fixed $K = 32$ draft length of Fig. 7(a), without per-agent caps. The online scale alone raises throughput by 66%, as the share of rejected draft tokens falls from 78% to 50%. The example coder call shows how: without it, the drafter spends about 26 tokens per step over the first 160 steps but has fewer than 2 accepted; with it, the drafter shortens its drafts to about 4 tokens there and expands to the full draft length once the coder starts copying reusable text. It thus rejects 81.1% fewer tokens while taking only 11 more steps.

5 Conclusion

We proposed AgSpec, a retrieval-based SD framework for coding-agent pipelines. AgSpec has two key designs: it makes the text an agent reuses available for retrieval in the form the agent emits, and it adapts the draft length to each agent using profiled accept lengths and verification feedback. Because AgSpec is orthogonal to the retrieval engine, it can run on top of existing engines without modification. Our evaluation shows that AgSpec achieves the highest throughput in most evaluated settings on SWE-bench Verified and TeamBench, reaching 2.27–4.37× the throughput of autoregressive decoding at batch size 1, and remains effective without a repository or explicit agent roles. These results show that retrieval-based SD for coding agents should follow the agent pipeline, as AgSpec does: it indexes the text the session has produced, in the form agents emit it, and drafts as much as each agent can reuse.

USE OF GENERATIVE AI TOOLS

In this work, we used generative AI tools to assist with translation, drafting and editing manuscript text, and providing feedback on figures and the presentation of our methods and experiments. Also, we exploited AI to get advice for the justification of our online adaptation mentioned at Appendix G.1. The authors reviewed AI-assisted suggestions and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

AGSPEC verifies drafts with the target model, preserving its output distribution for each model call. Its decoding method therefore does not introduce new output-level behavior relative to ordinary target-model decoding. The additional concern is that the workspace and session corpus may contain sensitive prompts, repository content, or tool outputs. Deployments should restrict access to each session's corpus, prevent its disclosure to other sessions, and discard it when the session ends.

REPRODUCIBILITY STATEMENT

§3.1 and §3.2 specify the retrieval and draft-length algorithms. §4.1 and appendix A report the task splits, corpus construction, and measurement protocol, while Table 1 lists the model and serving configurations. For each model and benchmark, we built the global corpus from tasks disjoint from evaluation and held it fixed during measurement (§4.1, appendix A, and table 1); session-specific corpora were reset between sessions (Table 1). We used greedy decoding (Table 1) and the same fixed seed across runs, including model calls that generate tool-call requests. Compared methods used the same evaluation tasks and serving configuration (§4.2).

REFERENCES

Sudhanshu Agrawal, Wonseok Jeon, and Mingu Lee. AdaEDL: Early draft stopping for speculative decoding of large language models via an entropy-based lower bound on token acceptance probability. In *Proceedings of The 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, volume 262 of *Proceedings of Machine Learning Research*, pp. 355–369, 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.

Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Free Software Foundation. Comparing and merging files. GNU Diffutils manual, *Unified Format*, 2026. URL https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html. Accessed 23 September 2026.

Paul Gauthier. Unified diffs make GPT-4 Turbo 3X less lazy. Aider blog, 2023. URL https://aider.chat/2023/12/21/unified-diffs.html.

Gemma Team. Gemma 3 technical report. *CoRR*, abs/2503.19786, 2025. doi: 10.48550/ARXIV.2503.19786. URL https://doi.org/10.48550/arXiv.2503.19786.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D. Lee, and Di He. REST: Retrieval-based speculative decoding. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 1582–1595. Association for Computational Linguistics, 2024.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, pp. 23247–23275, 2024.

Yuxuan Hu, Ke Wang, Xiaokang Zhang, Fanjin Zhang, Cuiping Li, Hong Chen, and Jing Zhang. SAM decoding: Speculative decoding via suffix automaton. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12187–12204. Association for Computational Linguistics, 2025.

Kaixuan Huang, Xudong Guo, and Mengdi Wang. SpecDec++: Boosting speculative decoding via adaptive candidate lengths. In *Conference on Language Modeling (COLM)*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*, pp. 58791–58831, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, pp. 54107–54157, 2024.

Yubin Kim, Chanwoo Park, Taehan Kim, Eugene Park, Samuel Schmidgall, Salman Rahman, Chunjong Park, Cynthia Breazeal, Xin Liu, Hamid Palangi, Hae Won Park, and Daniel McDuff. TeamBench: Evaluating agent coordination under enforced role separation. *arXiv preprint arXiv:2605.07073*, 2026.

Roger Koenker and Gilbert Bassett. Regression quantiles. *Econometrica*, 46(1):33–50, 1978.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626. ACM, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286. PMLR, 2023.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7421–7432, 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 *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Jiacheng Liu, Xiaohan Zhao, Xinyi Shang, and Zhiqiang Shen. Dive into Claude Code: The design space of today's and future AI agent systems. *arXiv preprint arXiv:2604.14228*, 2026a. URL https://arxiv.org/abs/2604.14228v2. Technical report.

Xiaoxuan Liu, Cade Daniel, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Xiangxi Mo, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Optimizing speculative decoding for serving large language models using goodput. *CoRR*, abs/2406.14066, 2024. doi: 10.48550/ARXIV.2406.14066. URL https://doi.org/10.48550/arXiv.2406.14066.

Xiaoxuan Liu, Jiaxiang Yu, Jongseok Park, Ion Stoica, and Alvin Cheung. Speculative decoding: Performance or illusion? In *Proceedings of Machine Learning and Systems (MLSys)*, 2026b.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 46534–46594, 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Jonathan Mamou, Oren Pereg, Daniel Korat, Moshe Berchansky, Nadav Timor, Moshe Wasserblat, and Roy Schwartz. Dynamic speculation lookahead accelerates speculative decoding of large language models. In *Proceedings of The 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, volume 262 of *Proceedings of Machine Learning Research*, pp. 456–467, 2024.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Mistral AI and All Hands AI. Devstral: Fine-tuning language models for coding agent applications. *CoRR*, abs/2509.25193, 2025. doi: 10.48550/ARXIV.2509.25193. URL https://doi.org/10.48550/arXiv.2509.25193.

Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. SuffixDecoding: Extreme speculative decoding for emerging AI applications. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Huy Nhat Phan, Tien N. Nguyen, Phong X. Nguyen, and Nghi D. Q. Bui. HyperAgent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL https://qwen.ai/blog?id=qwen3.6-27b.

Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=CS2JWaziYr.

Apoorv Saxena. Prompt lookup decoding. GitHub repository, November 2023. URL https://github.com/apoorvumang/prompt-lookup-decoding/.

Snowflake Inc. ArcticInference: SuffixDecoding cache implementation. https://github.com/snowflakedb/ArcticInference/blob/aca5d9a8a62474035c15d114d40a01abc8c94b51/arctic_inference/suffix_decoding/cache.py, 2026.

Shubham Tiwari, Tapan Chugh, Nash Rickert, Simon Peter, Ratul Mahajan, and Haiying Shen. CacheWise: Understanding workloads and optimizing KVCache management for efficiently serving LLM coding agents. *arXiv preprint arXiv:2606.16824*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025. doi: 10.1145/3715754.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 50528–50652, 2024.

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

Ziyin Zhang, Jiahao Xu, Tian Liang, Xingyu Chen, Zhiwei He, Rui Wang, and Zhaopeng Tu. Draft model knows when to stop: Self-verification speculative decoding for long-form generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 16696–16708, 2025.

Qianhui Zhao, Li Zhang, Fang Liu, Xiaoli Lian, Qiaoyuanhe Meng, Ziqian Jiao, Zetong Zhou, Jia Li, and Lin Shi. FastCoder: Accelerating repository-level code generation via efficient retrieval and verification. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025.

Table 1: Configuration of the reported AGSPEC experiments.

Hardware	One NVIDIA RTX PRO 6000 Blackwell Server Edition (96 GB) per run.
Serving	vLLM 0.17.1 (V1 engine), TP1, eager execution, prefix caching off, chunked prefill on, and maximum model length 16,000.
Models	Devstral-Small-2507 AWQ 4-bit, Gemma3-27B-it W4A16, and Qwen3.6-27B AWQ INT4, with greedy decoding.
Global corpus $\mathcal{G}$	One fixed corpus per model and benchmark, constructed before serving.
Session corpus $\mathcal{S}_s$	Trailing 2,048 prompt tokens and generated tokens, shared within a session and reset afterward.
Workspace corpus $\mathcal{W}_r$	Current-session repository artifacts in raw and emission forms, reset after the session.
Retrieval	Minimum match length 1 and margin $\delta = 5$ against $\mathcal{G}$. $\mathcal{G}$ and $\mathcal{S}_s$ use frequency; $\mathcal{W}_r$ uses the first occurrence.
Draft control	$\alpha_0 = 1.0$, $\eta = 0.5$, $\alpha \in [0.05, 12.0]$, and $\tau = 1 - \lambda$. Scales initialized once per server.
Draft-length limits	Maximum draft length 64 at batch size 1 and 24 at batch size 16; model- and agent-specific draft caps are profiled offline within these limits.

Table 2: Baseline draft lengths and engine settings.

Method	K	Settings
PLD	10	n-gram length 3–5 over the prompt
REST	10	Match threshold 2; per-model datastore
FastCoder	10	Match threshold 2; same datastore as REST
SAM-Decoding	15	Match threshold 2; $1 + 15$ verified tokens per step
SuffixDecoding	24	Tree depth 24, speculation factor 1.0, minimum token probability 0.1
EAGLE-3	3	Chain depth 3; per-model drafter checkpoint

A IMPLEMENTATION DETAILS

Baseline draft lengths. Table 2 lists the draft length K and the engine settings of each baseline. REST and FastCoder draft over a token tree, so their published policy has no direct chain-length equivalent; we keep the chain approximation of our implementation and report it as such rather than mapping the tree depth to a chain length. SuffixDecoding uses the default configuration of the vLLM implementation. EAGLE-3 uses a chain of depth 3, matching the 3 drafting steps of the published 3-4-8 tree. The autoregressive baseline runs through the same proposer path with the match threshold raised so that no draft is produced, which keeps the per-step timing logs identical across arms. Baseline settings were fixed before measurement and were not revised afterward.

Additional AGSPEC Implementation The session corpus admits the trailing 2,048 tokens of each prompt and is reset between sessions. The workspace corpus is likewise isolated by session, so artifacts from other evaluation sessions are never exposed to retrieval. The margin δ is 5 tokens. For the draft length, $\eta = 0.5$; the scale is initialized to 1.0, clipped to $[0.05, 12.0]$, and not reset between sessions.

Corpus construction. For each model and benchmark, $\mathcal{G}$ is built from instances disjoint from evaluation. SWE-bench Verified uses 70 corpus-building and 30 evaluation instances; TeamBench uses 213 and 30, respectively.

Break-even threshold. We estimate

$$\lambda = \bar{A}\frac{c}{S},$$

where S is target-model step time, $c = dS/dL$ is the marginal cost of one additional draft token, and $\bar{A}$ is the mean accepted tokens per step. We use λ for the offline cap and set $\tau = 1 - \lambda$. The caps and τ of the reported runs were fixed from a cost profile taken before the final runs and were not revised afterward. Table 3 lists λ re-measured on the final batch-size-16 runs themselves, over steps with at least 12 active requests, next to the τ and per-agent caps actually used.

Table 3: Break-even threshold re-measured on the final batch-size-16 runs (steps with $b \geq 12$), with the τ and per-agent caps $K_{\max}$ used in the reported runs. Caps are listed in agent order: planner / executor / verifier for TeamBench, and localizer / coder / executor / feedback / summarizer for SWE-bench Verified; on SWE-bench Verified only the first three are set explicitly and the other two default to the engine budget. On SWE-bench Verified, all three models use the λ measured with Devstral.

Benchmark	Model	λ	$1-\lambda$	τ used	$K_{\max}$ used
TeamBench	Devstral	0.022	0.979	0.944	4 / 14 / 19
	Gemma3	0.133	0.867	0.944	4 / 24 / 11
	Qwen3.6	0.051	0.949	0.896	15 / 3 / 7
SWE-bench Verified	Devstral	0.299	0.701	0.950	24 / 24 / 24 / 24 / 24
	Gemma3	0.299	0.701	0.950	24 / 24 / 24 / 24 / 24
	Qwen3.6	0.299	0.701	0.950	24 / 24 / 24 / 24 / 24

B Additional Corpus Ablations

We further analyze three aspects of the corpus design: the sharing scope of the session corpus, how the workspace corpus grows, and whether it is kept separate from the global corpus (Table 4).

Table 4: Additional corpus controls on SWE-bench Verified at batch size 1. Values are AL changes relative to the reported configuration. Asterisks mark a two-sided Wilcoxon signed-rank test over paired sessions ($^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$; no multiple-comparison correction).

Control	Devstral	Gemma3	Qwen3.6
Per-call session corpus	$-51.4\%^{***}$	$-44.2\%^{***}$	$-39.3\%^{***}$
Per-agent session corpus	$-2.3\%^{***}$	$-1.7\%^{***}$	$-1.4\%^{***}$
Cross-session workspace growth	-0.2%	-0.5%	$+0.3\%$
Workspace pooled into $\mathcal{G}$	$+0.2\%$	0.0%	$+0.5\%$

The session corpus must retain trajectory context across calls: restricting $\mathcal{S}_s$ to the current call sharply reduces AL, because coding agents frequently reuse content from earlier inputs and outputs. Sharing $\mathcal{S}_s$ across agents yields a further, smaller gain, as agents also reuse what other agents have read or produced.

Letting $\mathcal{W}_r$ grow across sessions brings no meaningful benefit, as artifacts from the current session already capture most of the useful workspace reuse. This may change when many sessions run on the same repository, which we leave for future work.

Finally, pooling $\mathcal{W}_r$ into $\mathcal{G}$ performs on par with keeping it separate. We nevertheless keep them separate: in real serving environments, each user works on a different repository, and folding every workspace into a global corpus shared by all users would be difficult to manage.

C Corpus Preference Margin

We vary the preference margin δ to test the sensitivity of corpus selection to the bias against $\mathcal{G}$.

Table 5: Preference-margin sensitivity on SWE-bench Verified ($b = 1$). Values are oracle-normalized continuation agreement (%).

δ	Devstral	Gemma3	Qwen3.6
0	93.8^{**}	93.3^{***}	90.8^{***}
2	95.0^{**}	94.7^{***}	93.6^{**}
$5^{\dagger}$	94.6	94.1	93.3
8	94.3^{***}	93.8^{***}	93.2^{***}
12	94.2^{***}	93.7^{***}	93.2^{***}
∞	94.1^{***}	93.6^{***}	93.2^{***}

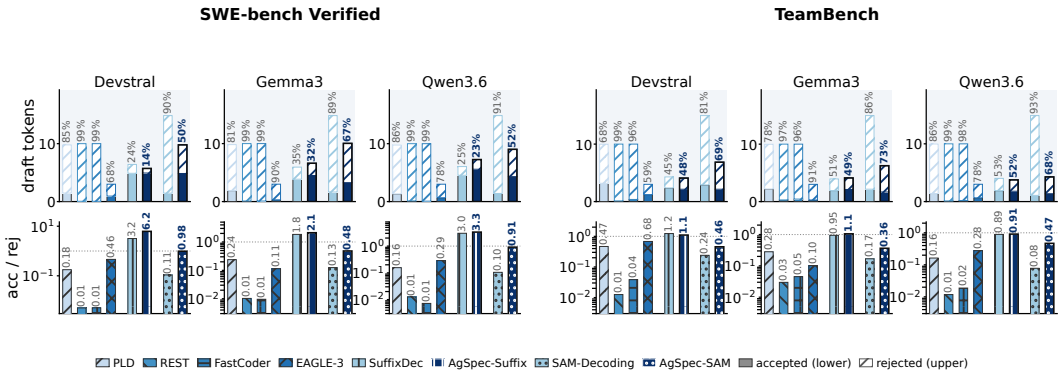


Figure 8: Draft-token composition at batch size 16 over decoding steps with at least 12 active requests. Top: mean accepted (solid) and rejected (hatched) draft tokens per drafting step; labels report rejection rates. Bottom: the ratio of accepted to rejected draft tokens, on a log scale, with the dotted line at equal counts. No-speculation is omitted.

Agreement remains high across a broad range of margins, with $\delta = 2$ giving the highest values across all three models. The reported $\delta = 5$ therefore does not rely on a sharply tuned preference margin.

D Corpus-Specific Continuation Selection

Because the corpora capture different forms of reuse, we replay the same matches and candidates under alternative continuation-selection rules.

Table 6: Continuation-selection replay on SWE-bench Verified. Values are changes in reference-matching continuation length (%) relative to the reported rule, averaged across three models.

Rule	Global $\mathcal{G}$	Session $\mathcal{S}_s$	Workspace $\mathcal{W}_r$
first occurrence	-18.1^{***}	-4.2^{***}	$0^{\dagger}$
Most recent	-7.9^{***}	$+1.1^{***}$	-2.6^{**}
Most frequent	$0^{\dagger}$	$0^{\dagger}$	-1.0
REST-style trie	$+6.5^{***}$	$+0.1$	$+0.3$
Recency–frequency	-0.8^{***}	$+0.6^{***}$	-1.1

†: reported rule. $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$ (two-sided Wilcoxon signed-rank test over paired sessions; uncorrected).

The corpora favor different selection behavior: $\mathcal{G}$ benefits from frequency, while $\mathcal{W}_r$ favors its earliest occurrence. For $\mathcal{S}_s$, alternative rules yield only small differences, so we reuse the frequency rule of $\mathcal{G}$ rather than introducing a separate session-specific rule. REST-style selection shows additional headroom for structured candidate ranking in $\mathcal{G}$.

E Draft Token Composition

Fig. 8 shows the average accepted and rejected token length at batch size 16. On the suffix automaton, AgSpec reduces rejection rates over SAM-Decoding by 12.0–39.8 points while drafting on 98–99% of request steps, compared with 50–56% for SAM-Decoding. This shows that the reduction comes from shorter, better-placed drafts rather than less frequent drafting. On the suffix tree, rejection rates are similar on TeamBench, while AgSpec-Suffix reduces them by 1.9–10.0 points on SWE-bench Verified.

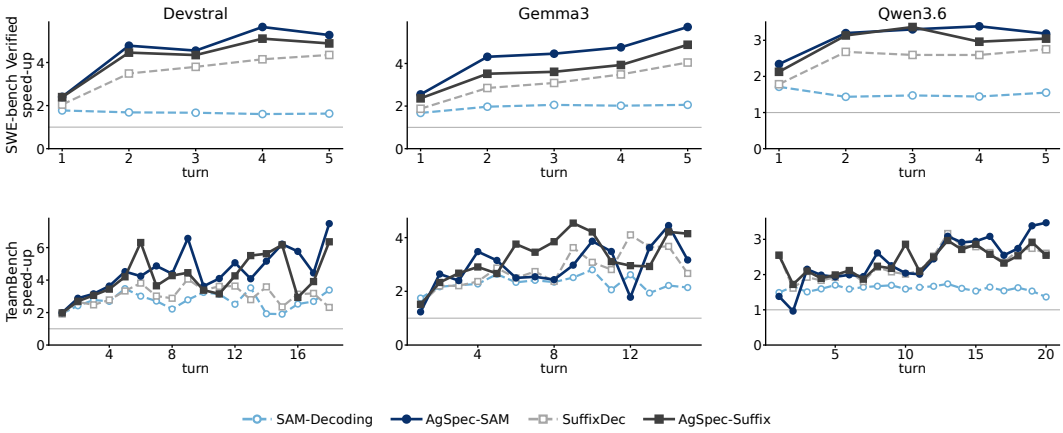


Figure 9: Speed-up over no speculation by turn at batch size 1 on SWE-bench Verified and TeamBench.

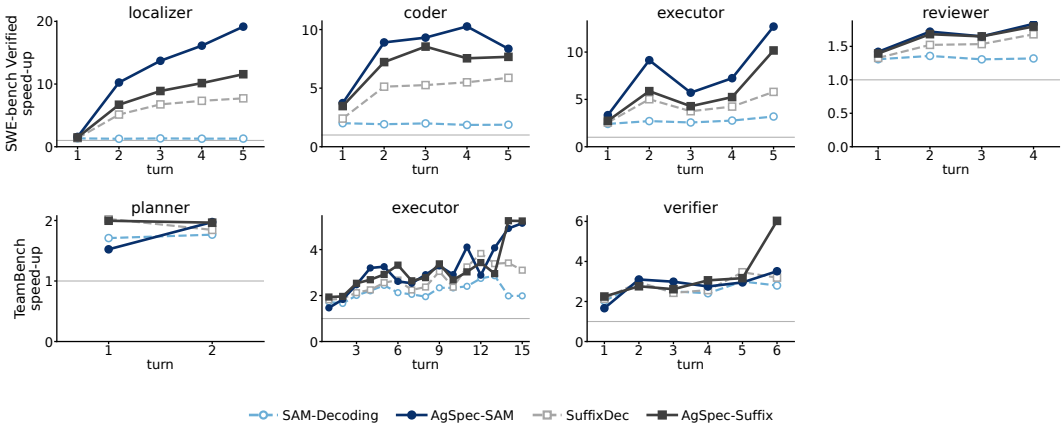


Figure 10: Speed-up over no speculation by agent and turn at batch size 1. TeamBench results are geometric means across available models.

F Speed-up by Turn and Agent

We examine whether speculative-decoding gains remain uniform across agents (Fig. 10) and turns (Fig. 9).

Speed-up varies substantially across both agents and trajectory positions, and its turn-wise trend is not uniformly monotonic. This heterogeneity supports adapting speculation to the observed trajectory rather than using a fixed agent- or turn-level schedule.

G Discussion

G.1 Interpreting the Scale Update

Fix an agent and drop the superscript (a_i). We idealize the draft length as $D_i = \alpha m_i$, ignoring rounding, clipping of α, the cap $K_{\max}$, and corpus continuations shorter than αm_i. Let R_i be the number of tokens the target would accept from an unbounded draft, so that $A_i = \min(R_i, D_i)$ and

$$\mathbb{I}[A_i < D_i] = \mathbb{I}[R_i < \alpha m_i] = \mathbb{I}[U_i < \alpha], \qquad U_i = R_i/m_i.$$

Objective. With the pinball loss $\rho_\tau(u) = u\,(\tau - \mathbb{I}[u < 0])$,

$$\frac{\partial}{\partial \alpha}\,\rho_\tau(U_i - \alpha) = -\big(\tau - \mathbb{I}[U_i < \alpha]\big)$$

for $U_i \neq \alpha$, so Equation (3) is a stochastic subgradient step on $\mathbb{E}[\rho_\tau(U - \alpha)]$, which is minimized at the τ-quantile of U. The loss is defined on the normalized length U_i because α scales the draft relative to the match length. Applying it to the raw length instead, $\rho_\tau(R_i - \alpha m_i)$, yields the subgradient $-m_i(\tau - \mathbb{I}[R_i < \alpha m_i])$, which weights each step by m_i and targets an m-weighted quantile of U; Equation (3) does not apply this weighting.

Equilibrium. If the distribution of U_i were stationary, the expected update $\eta\,(\tau - \Pr[U < \alpha])$ would vanish exactly when $\Pr[U < \alpha] = \tau$, i.e., when drafts are fully accepted ($A_i = D_i$) with probability $1 - \tau$. In practice, η is fixed, sessions are finite, and the distribution of U_i shifts as $\mathcal{S}_s$ grows, so α does not converge; it fluctuates around and tracks this moving target. The ignored effects further perturb this balance. In particular, a fully accepted draft truncated by $K_{\max}$ reveals only $R_i \geq K_{\max}$, not whether $U_i \geq \alpha$; counting it as a full acceptance would bias α upward, which is why AGSPEC skips such updates.

G.2 ADAPTIVE DRAFT LENGTH BEYOND RETRIEVAL

Adaptive draft length has been studied mainly for model-based SD, where a draft model or head generates each token and exposes signals about how likely that token is to be accepted. SpecDec++ trains an acceptance-prediction head and stops drafting once the predicted rejection probability exceeds a threshold (Huang et al., 2025). DISCO feeds the draft model's top-k probabilities and entropy to a small classifier that decides whether to continue (Mamou et al., 2024), and AdaEDL and SVIP stop when the entropy of the draft distribution signals a low acceptance probability (Agrawal et al., 2024; Zhang et al., 2025). EAGLE-2 likewise expands its draft tree along the draft model's most confident branches (Li et al., 2024).

Retrieval-based SD has none of these signals. Its drafter copies a continuation from a corpus instead of sampling one, so there is no token distribution, entropy, or hidden state to read, and training a head to predict acceptance would reintroduce the model that retrieval-based SD avoids. What a retrieval drafter does have is the match itself, such as its length, occurrence count, and source corpus, and the outcome of verification. Existing retrieval methods either fix the draft length or scale it with the match length (§2.2). SmartSpec uses verification outcomes, choosing the length that maximizes estimated goodput under the observed acceptance and system load (Liu et al., 2024), but it targets general request streams rather than agent pipelines, where acceptance differs systematically across agents. AGSPEC uses both available signals and adds the one that coding agents provide, the agent issuing each call: an offline cap per agent bounds the draft, and an online scale fitted to verification feedback sizes it from the match length.

G.3 OVERHEAD OF AGSPEC

Memory. The global corpus $\mathcal{G}$ is loaded once per server and shared across sessions. Its resident host-memory footprint is 2.5–4.3 GB, with 9–15 s load time (Table 7); it does not consume GPU memory or reduce the KV cache. The session and workspace corpora are session-local and released afterward.

Table 7: Global-corpus memory footprint and load time. Token counts are reported for TeamBench.

Benchmark	Model	Resident	On disk	Load	Tokens
SWE-bench Verified	Devstral	3,603 MB	214 MB	12.4 s	–
	Gemma3	4,121 MB	253 MB	14.1 s	–
	Qwen3.6	4,295 MB	258 MB	14.7 s	–
TeamBench	Devstral	2,484 MB	144 MB	9.2 s	1.94 M
	Gemma3	3,180 MB	189 MB	11.5 s	2.51 M
	Qwen3.6	3,640 MB	214 MB	14.0 s	2.94 M

Construction. Building $\mathcal{G}$ from recorded trajectories takes 18–26 s on CPU and requires no GPU. $\mathcal{W}_r$ is built per session, while $\mathcal{S}_s$ grows during decoding; both costs are included in reported throughput.

Retrieval. At batch size 16, querying the three corpora, selecting a draft, and appending the step's tokens to the session corpus takes 2.3–5.8 ms per step for AGSPEC-SAM and 1.6–2.8 ms

for AGSPEC-Suffix. A step takes 47–182 ms, so AGSPEC accounts for 1.1–5.5% of it and the rest is the target model and the engine. We do not split the rest further: the engine launches the forward pass asynchronously, so its CPU-side timer covers only 2–56% of the step. Table 8 gives the split per model.

Table 8: Per-step split at batch size 16, over steps with at least 12 active requests. AGSPEC is the proposer: corpus queries, draft selection, and appending the step's tokens. *Rest* is the target model and the engine.

Benchmark	Model	Engine	AGSPEC	Rest	AGSPEC share
SWE-bench Verified	Devstral	SAM	5.05 ms	98.1 ms	4.9%
		Suffix	2.80 ms	78.7 ms	3.4%
	Gemma3	SAM	5.81 ms	132.8 ms	4.2%
		Suffix	2.57 ms	142.6 ms	1.8%
	Qwen3.6	SAM	2.73 ms	162.6 ms	1.7%
		Suffix	2.54 ms	179.4 ms	1.4%
TeamBench	Devstral	SAM	2.96 ms	147.6 ms	2.0%
		Suffix	1.57 ms	145.6 ms	1.1%
	Gemma3	SAM	2.77 ms	47.9 ms	5.5%
		Suffix	1.60 ms	45.8 ms	3.4%
	Qwen3.6	SAM	2.27 ms	110.5 ms	2.0%
		Suffix	1.94 ms	165.1 ms	1.2%