

Zen4-Coder-Flash: Real-Time Code Intelligence for IDE Environments

Technical Whitepaper v2026.01

Zen LM Research Team
research@zenlm.org
papers.zenlm.org

January 2026

Abstract

Zen4-Coder-Flash is a distilled 8B parameter code intelligence model optimized for real-time IDE deployment, achieving sub-30ms P95 latency for inline code completion while maintaining strong coding accuracy. Distilled from Zen4-Coder (32B) using progressive knowledge distillation and Mixture of Experts (MoE) architecture compression, Zen4-Coder-Flash retains 88.5% of Zen4-Coder’s HumanEval performance (84.3%) and 79.1% MBPP accuracy at 1200 tokens per second throughput. Speculative decoding with a 120M parameter draft model reduces P95 first-token latency to 28ms, enabling responsive autocomplete and inline suggestion experiences that match or exceed prior dedicated completion models while generalizing across 92 programming languages.

Contents

1	Introduction	3
1.1	Model Overview	3
2	Architecture	3
2.1	Compressed MoE	3
2.2	Speculative Decoding	4
2.3	Attention Optimizations	4
2.4	Quantization	4
3	Knowledge Distillation	4
3.1	Progressive Distillation Protocol	4
3.2	Distillation Results	5
4	Evaluation	5
4.1	Accuracy Benchmarks	5
4.2	Latency Benchmarks	5
4.3	IDE-Specific Evaluation	6
4.4	Multi-Language Latency	6
5	Deployment	7
5.1	Hardware Requirements	7
5.2	IDE Plugin Architecture	7
5.3	Privacy Model	7

6	Comparison with Dedicated Completion Models	7
7	Safety Considerations	7
7.1	Autocomplete Safety	7
8	Related Work	8
9	Conclusion	8

1 Introduction

IDE-integrated code completion places qualitatively different constraints on AI models than batch code generation. Where batch workflows tolerate multi-second latencies, IDE users expect responses within the same timeframe as keystroke feedback: tens of milliseconds. Where batch workflows permit large context windows, IDE completion must integrate with a lightweight editor extension that cannot afford the memory footprint of a 32B model inference server.

Prior approaches to this tradeoff either deployed small general-purpose models (low accuracy) or used expensive API calls to large remote models (high latency). Zen4-Coder-Flash resolves this tension through a purpose-built 8B model that combines three techniques:

1. **Progressive Knowledge Distillation:** Token-level distillation from Zen4-Coder (32B) preserves the behavioral distribution of the teacher while drastically reducing parameter count.
2. **Speculative Decoding:** A 120M draft model generates candidate token sequences that the 8B verifier accepts or rejects in parallel, reducing effective per-token latency by $2.8\times$.
3. **Architecture Optimization:** The MoE expert pool is compressed from 64 to 16 experts with larger individual capacity, reducing routing overhead and improving cache efficiency.

1.1 Model Overview

Table 1: Zen4-Coder-Flash Model Specification

Parameter	Value
Architecture	MoE (Compressed)
Total Parameters	8B
Draft Model Parameters	120M
Context Window	32K tokens
Supported Languages	92 programming languages
P95 First-Token Latency	28ms
Throughput	1,200 tok/s
Version	v2026.01
Release Date	January 2026

2 Architecture

2.1 Compressed MoE

The full MoE architecture in Zen4-Coder uses 64 experts with 4 active per token. For real-time deployment, this introduces routing overhead that is unacceptable at sub-30ms latency targets. Zen4-Coder-Flash uses a compressed expert configuration:

Table 2: Expert Configuration Comparison

Model	Total Experts	Active (k)	Expert Dim	Router Overhead
Zen4-Coder (32B)	64	4	2,048	1.8ms
Zen4-Coder-Flash (8B)	16	2	4,096	0.4ms

Individual experts in Zen4-Coder-Flash are wider (4,096 vs. 2,048 dimensional) to compensate for the reduced count, maintaining aggregate representational capacity while reducing

routing computation. The routing mechanism is simplified to a single softmax over 16 logits, compared to the hierarchical two-level routing used in Zen4-Coder-Pro.

2.2 Speculative Decoding

Speculative decoding [1] uses a small draft model to propose γ tokens in parallel, which the main model verifies in a single forward pass. The effective speedup is:

$$S = \frac{\gamma + 1}{1 + \gamma(1 - \beta)} \quad (1)$$

where β is the acceptance rate (fraction of draft tokens accepted by the verifier). With the 120M draft model trained specifically on code distribution, Zen4-Coder-Flash achieves $\beta = 0.81$ and $\gamma = 8$, yielding $S = 2.8\times$ speedup over autoregressive decoding.

The draft model architecture is a lightweight 12-layer decoder-only transformer with:

- Hidden dimension: 768
- Attention heads: 12
- No mixture-of-experts (dense feedforward)
- Vocabulary shared with main model (100K tokens)
- Grouped query attention (4 key-value heads)

2.3 Attention Optimizations

Zen4-Coder-Flash uses several attention optimizations to reduce per-token cost:

1. **Multi-Query Attention (MQA)**: Single key and value head shared across all query heads, reducing KV cache memory by $8\times$ compared to multi-head attention.
2. **Flash Attention 3**: Fused CUDA kernel for attention computation, reducing memory bandwidth requirements and enabling longer effective context at lower latency.
3. **Prefix Caching**: Static code context (imports, class definitions, boilerplate) is cached between completion requests, avoiding redundant computation for the stable prefix that dominates most IDE contexts.

2.4 Quantization

Zen4-Coder-Flash is deployed in FP8 quantization by default with GPTQ-style weight grouping (group size 128). Accuracy degradation from FP8 quantization is less than 0.4% on HumanEval, while reducing model memory footprint from 16GB (BF16) to 8GB (FP8), enabling deployment on a single A100 40GB or consumer H100 NVL.

3 Knowledge Distillation

3.1 Progressive Distillation Protocol

Zen4-Coder-Flash is produced through a four-stage progressive distillation protocol:

Stage 1 – Vocabulary Alignment: The student is initialized with a subset of Zen4-Coder’s embedding matrix (shared tokenizer) and trained to match the teacher’s token log-probabilities on a held-out code corpus:

$$\mathcal{L}_{\text{KD}} = - \sum_t \sum_v p_T(v|x_{<t}) \log p_S(v|x_{<t}) \quad (2)$$

where p_T and p_S are the teacher and student token distributions.

Stage 2 – Layer-Wise Representation Alignment: For each student layer l_s , a corresponding teacher layer l_t is selected. A projection head W_{proj} maps student hidden states to the teacher’s dimension for MSE alignment:

$$\mathcal{L}_{\text{rep}} = \sum_l \left\| h_S^{l_s} W_{\text{proj}} - h_T^{l_t} \right\|_2^2 \quad (3)$$

Stage 3 – Task-Specific Fine-Tuning: The student is fine-tuned on code generation, completion, and inline suggestion tasks using the same RLCE objective as Zen4-Coder, with the teacher used as an additional reward signal.

Stage 4 – Latency-Aware Calibration: Final fine-tuning on a latency-calibrated dataset: short, high-value completions (1–50 tokens) drawn from real IDE telemetry, weighted to optimize the distribution of outputs most commonly needed in autocomplete scenarios.

3.2 Distillation Results

Table 3: Distillation Quality vs. Teacher Model

Stage	HumanEval	MBPP
Scratch (8B, no distillation)	71.3%	64.8%
After Stage 1 (KD only)	76.8%	69.2%
After Stage 2 (+ rep alignment)	80.1%	73.7%
After Stage 3 (+ task fine-tune)	83.4%	78.2%
After Stage 4 (+ latency calib)	84.3%	79.1%
Teacher (Zen4-Coder 32B)	95.2%	89.3%

The distillation process recovers 88.5% of the teacher’s HumanEval performance using 25% of the parameter count.

4 Evaluation

4.1 Accuracy Benchmarks

Table 4: Accuracy Benchmark Results

Model	HumanEval	MBPP	MultiPL-E (avg)	CRUXEval
Zen4-Coder-Flash (8B)	84.3%	79.1%	78.4%	71.3%
Comparable 7B baseline A	79.1%	72.8%	73.2%	65.4%
Comparable 7B baseline B	76.4%	70.3%	70.8%	63.1%
Comparable 13B baseline	81.7%	76.4%	76.1%	69.2%

4.2 Latency Benchmarks

Latency is measured on a single H100 SXM5 80GB GPU with FP8 quantization, serving a single user session with prefix caching enabled.

Table 5: Latency Benchmarks

Metric	Zen4-Coder-Flash	Baseline 7B (Autoregressive)
P50 First-Token Latency	14ms	38ms
P95 First-Token Latency	28ms	67ms
P99 First-Token Latency	41ms	94ms
P50 Completion Latency (32 tok)	67ms	187ms
P95 Completion Latency (32 tok)	124ms	341ms
Throughput (tok/s, batch=1)	1,200	430
Throughput (tok/s, batch=8)	3,800	1,840

4.3 IDE-Specific Evaluation

We evaluate completion quality in a simulated IDE environment using real developer sessions drawn from an opt-in telemetry dataset. 1,200 sessions were evaluated by the original developers who rated each suggestion on a 3-point scale: accepted, modified, or rejected.

Table 6: IDE Completion Quality (developer ratings)

Model	Accept Rate	Modified Rate	Reject Rate	Useful Rate
Zen4-Coder-Flash (8B)	48.3%	31.4%	20.3%	79.7%
Comparable 7B baseline	39.1%	28.7%	32.2%	67.8%
Rule-based completion	21.4%	19.8%	58.8%	41.2%

4.4 Multi-Language Latency

Prefix caching effectiveness varies by language due to differences in common boilerplate patterns. We evaluate P95 latency across major language groups:

Table 7: P95 First-Token Latency by Language Category

Language Category	Cache Hit Rate	P95 Latency
Python (with imports)	74%	22ms
TypeScript / JavaScript	68%	25ms
Go	71%	23ms
Rust	63%	29ms
Java / Kotlin	69%	26ms
C / C++	66%	27ms

5 Deployment

5.1 Hardware Requirements

Table 8: Deployment Hardware Options

Config	Hardware	VRAM	P95 Latency
Production (FP8)	1 × H100 80GB	8GB model	28ms
Development (FP8)	1 × A100 40GB	8GB model	34ms
Edge (INT4)	1 × RTX 4090	5GB model	47ms
CPU fallback (INT4)	32-core CPU + 64GB RAM	5GB model	280ms

5.2 IDE Plugin Architecture

The Zen4-Coder-Flash IDE integration consists of three layers:

1. **Editor Extension** (VS Code, JetBrains, Neovim): Lightweight TypeScript/Lua plugin that captures cursor context, sends completion requests, and renders suggestions. Target memory footprint: < 50MB.
2. **Local Inference Daemon**: A persistent background process that loads the model once and serves completion requests over a local Unix socket, avoiding cold-start latency per request.
3. **Context Aggregator**: Assembles the completion prompt from current file, open files, recent edits, and workspace symbol index, fitting within the 32K token context limit.

5.3 Privacy Model

For enterprise deployments, Zen4-Coder-Flash can run entirely locally (on-device inference) with no code or context transmitted to external servers. Cloud-assisted mode is opt-in and transmits only the minimal context window required for completion, not full file trees.

6 Comparison with Dedicated Completion Models

Prior dedicated autocomplete models achieve lower latency by severely restricting model size and vocabulary, often at the cost of accuracy on complex code patterns. Zen4-Coder-Flash represents a point on the Pareto frontier that improves over both:

Table 9: Accuracy-Latency Pareto Comparison

System	HumanEval	P95 Latency	Languages
Zen4-Coder-Flash (8B)	84.3%	28ms	92
Dedicated 3B completer A	71.4%	19ms	12
Dedicated 1B completer B	62.8%	11ms	8
Remote large model (API)	93.2%	180ms	70+

7 Safety Considerations

7.1 Autocomplete Safety

Even in real-time autocomplete contexts, Zen4-Coder-Flash maintains safety properties:

- **Secret detection:** Inline suggestions never autocomplete credential patterns (API keys, passwords, tokens) even when the surrounding context contains examples.
- **Vulnerable pattern avoidance:** Known vulnerable patterns (SQL injection via string interpolation, unsafe deserialization, command injection) are suppressed in favor of safe alternatives with equivalent functionality.
- **License-aware completion:** The model is trained to avoid direct reproduction of GPL-licensed code in proprietary project contexts.

8 Related Work

Real-time code completion has been addressed through n-gram models [3], small neural language models [4], and speculative decoding applied to general models [1, 2]. Knowledge distillation for code models has been explored in [5]. Zen4-Coder-Flash combines these techniques with the MoE architecture in a unified training protocol optimized for the specific latency and accuracy requirements of IDE deployment.

9 Conclusion

Zen4-Coder-Flash establishes a new accuracy-latency operating point for IDE code intelligence, achieving 84.3% HumanEval accuracy at 28ms P95 first-token latency through a combination of progressive knowledge distillation from Zen4-Coder, compressed MoE architecture, and speculative decoding. The 8B parameter footprint enables on-device deployment on a single consumer GPU while the 1,200 tok/s throughput supports responsive real-time pair programming assistance. With 79.7% useful suggestion rate in developer acceptance studies, Zen4-Coder-Flash provides practical value in production IDE environments.

References

- [1] Leviathan, Y. et al. (2023). Fast Inference from Transformers via Speculative Decoding. ICML 2023.
- [2] Miao, X. et al. (2024). SpecInfer: Accelerating LLM Serving with Tree-based Speculative Inference. ASPLOS 2024.
- [3] Tu, Z. et al. (2014). On the Naturalness of Software. ICSE 2012.
- [4] Amazon. (2023). Amazon CodeWhisperer. AWS Documentation.
- [5] Wei, Y. et al. (2023). MagiCoder: Source Code Is All You Need. arXiv:2312.02120.