

Zen-Coder-Flash: Ultra-Low-Latency Code Completion via Knowledge Distillation

Technical Report v2025.02

Zen LM Research Team
research@zenlm.org

February 2025

Abstract

We present **Zen-Coder-Flash**, an 8 billion parameter code completion model distilled from Zen-Code (14B) using the Zen MoDE (Mixture of Distilled Experts) distillation pipeline. Zen-Coder-Flash targets IDE-grade latency requirements: P95 time-to-first-token below 50ms for typical autocomplete prompts, with throughput of 850 tokens/second on a single A100 GPU. Despite its reduced parameter count, Zen-Coder-Flash achieves HumanEval 81.4%—84.4% of Zen-Code’s performance at 55% of the parameter count and $2.1\times$ lower inference latency. The model supports fill-in-the-middle (FIM), 32K context, and all 80+ languages from Zen-Code, making it the recommended choice for real-time developer tooling where sub-50ms response is a hard requirement.

Contents

1	Introduction	3
2	Architecture	3
2.1	Student Model Specification	3
2.2	Architectural Choices for Latency	3
3	Training Methodology	4
3.1	Overview	4
3.2	Phase 1: Warm-Up Pretraining	4
3.3	Phase 2: Knowledge Distillation	4
3.4	Speculative Decoding Setup	5
4	Evaluation	6
4.1	Code Quality Benchmarks	6
4.2	Latency Benchmarks	6
4.3	Throughput Benchmarks	6
4.4	Speculative Decoding Performance	7
5	Distillation Ablation	7
6	IDE Integration and Deployment	7
6.1	Deployment Architecture	7
6.2	Continue Extension Configuration	8
7	Related Work	8

8	Limitations	8
9	Conclusion	8

1 Introduction

IDE code completion operates under strict latency constraints. Empirical studies of developer experience show that completion latency above 150ms breaks the perceived “inline” feeling and causes developers to wait or dismiss suggestions [1]. At P95, this requires generating a 10–30 token completion in well under 100ms, leaving approximately 30–50ms budget for network round-trip, model inference, and post-processing.

General-purpose code models at 13–14B parameters are too large to meet this budget on widely available server hardware without batching latency. Zen-Coder-Flash solves this through knowledge distillation from Zen-Code, compressing the 14B model into an 8B student that preserves code quality while fitting within the latency budget.

Key contributions:

- An 8B code model distilled from Zen-Code (14B) using sequence-level KD with intermediate-layer feature alignment.
- P95 time-to-first-token of 43ms on A100-80GB at batch size 1, and 850 tokens/s throughput at batch size 16.
- HumanEval 81.4% pass@1, recovering 93% of the teacher model’s score relative to the student’s from-scratch baseline (76.3%).
- Speculative decoding compatibility: Zen-Coder-Flash serves as a draft model for Zen-Code in a speculative decoding setup, achieving $2.8\times$ throughput on Zen-Code at minimal quality loss.

2 Architecture

2.1 Student Model Specification

Zen-Coder-Flash uses the Zen MoDE architecture at 8B parameters. The architecture is a non-trivial reduction of Zen-Code rather than a simple layer-pruning: hidden dimension and FFN width are reduced proportionally to preserve the depth-to-width ratio associated with strong reasoning capability.

Table 1: Zen-Coder-Flash architecture hyperparameters (vs. Zen-Code teacher).

Hyperparameter	Zen-Coder-Flash (8B)	Zen-Code (14B)
Parameters (total)	8.2B	14.4B
Layers	36	40
Attention heads	32	40
KV heads (GQA)	8	8
Hidden dimension	4,096	5,120
FFN intermediate dimension	11,008	13,696
Vocabulary size	151,936	151,936
Context length (training)	32,768	65,536
Position encoding	RoPE ($\theta = 2,000,000$)	RoPE ($\theta = 2,000,000$)

2.2 Architectural Choices for Latency

The hidden dimension of 4,096 (reduced from 5,120) is a critical latency driver: the matrix multiplication cost of each attention projection and FFN layer scales quadratically with hidden

dimension in memory bandwidth and linearly in FLOPs. The 20% hidden dimension reduction yields approximately 36% fewer FLOPs per layer, compounded across 36 layers.

GQA with 8 KV heads (identical to Zen-Code) is retained, yielding minimal KV cache requirements that enable large effective batch sizes:

$$\text{KV cache (per layer)} = 2 \cdot n_{\text{kv}} \cdot d_k \cdot S \cdot 2 = 2 \cdot 8 \cdot 128 \cdot 32,768 \cdot 2 \approx 134 \text{ MB} \quad (1)$$

At 32K context, the full 36-layer KV cache requires 4.8 GB, leaving ample headroom for model weights (16.4 GB in BF16) on a 24 GB consumer GPU.

3 Training Methodology

3.1 Overview

Zen-Coder-Flash training proceeds in three phases:

1. **Warm-up pretraining:** Train on 500B code tokens from scratch with standard next-token and FIM objectives to initialize parameters.
2. **Knowledge distillation:** Distill from Zen-Code using combined token-level KD and intermediate feature alignment losses.
3. **Instruction fine-tuning:** SFT on the same 800K instruction-code pairs as Zen-Code.

3.2 Phase 1: Warm-Up Pretraining

The student is initialized randomly and pretrained on 500B code tokens—a reduced subset of Zen-Code’s 2.5T corpus—using standard cross-entropy loss. This warm-up prevents cold start instability in the subsequent KD phase, where the student would otherwise produce very poor initial approximations of the teacher’s distributions.

- Optimizer: AdamW, LR 3×10^{-4} , cosine to 3×10^{-5}
- Batch: 2M tokens
- FIM rate: 50%
- Duration: 250K steps on 256 H100 GPUs

3.3 Phase 2: Knowledge Distillation

The distillation loss combines three terms:

$$\mathcal{L}_{\text{KD}} = \alpha \mathcal{L}_{\text{CE}} + \beta \mathcal{L}_{\text{KL}} + \gamma \mathcal{L}_{\text{feat}} \quad (2)$$

Cross-entropy loss ($\mathcal{L}_{\text{CE}}$). Standard next-token prediction on ground-truth labels:

$$\mathcal{L}_{\text{CE}} = - \sum_t \log p_s(y_t \mid y_{<t}, x) \quad (3)$$

KL divergence loss ($\mathcal{L}_{\text{KL}}$). Token-level KL divergence between student and teacher output distributions at temperature T :

$$\mathcal{L}_{\text{KL}} = \sum_t D_{\text{KL}}(p_t(\cdot; T) \| p_s(\cdot; T)) \quad (4)$$

with temperature $T = 1.0$ (we find $T > 1$ unnecessarily smooths code-specific distributions where the correct token is often highly peaked). Top- k filtering at $k = 50$ is applied to the teacher logits before KL computation to reduce noise from near-zero probabilities.

Intermediate feature alignment ($\mathcal{L}_{\text{feat}}$). We align intermediate hidden states between the student and teacher at evenly spaced layers using a learned linear projection W_{proj} mapping the student’s 4096-dim hidden states to the teacher’s 5120-dim space:

$$\mathcal{L}_{\text{feat}} = \frac{1}{|M|} \sum_{(i,j) \in M} \|h_i^s W_{\text{proj}} - h_j^t\|_2^2 \quad (5)$$

where M is a mapping from student layers $\{6, 12, 18, 24, 30, 36\}$ to teacher layers $\{5, 10, 15, 20, 28, 35\}$. The projection matrix W_{proj} is trained jointly with the student.

Loss coefficients: $\alpha = 0.3$, $\beta = 0.5$, $\gamma = 0.2$.

Table 2: Distillation phase hyperparameters.

Parameter	Value
Learning rate	$1 \times 10^{-4} \rightarrow 5 \times 10^{-6}$ (cosine)
Batch size	2M tokens
KD temperature	1.0
Top- k filtering	50
CE coefficient α	0.3
KL coefficient β	0.5
Feature coefficient γ	0.2
Duration	500B tokens (250K steps)

3.4 Speculative Decoding Setup

Zen-Coder-Flash serves as a draft model in a speculative decoding pipeline with Zen-Code as the verifier. The draft model generates $\gamma = 6$ speculative tokens per step; the verifier accepts or rejects in parallel.

Expected speedup with acceptance rate α :

$$\text{Speedup} = \frac{1 + \alpha + \alpha^2 + \dots + \alpha^\gamma}{(1 \text{ verifier step cost}) / (\text{draft step cost})} \quad (6)$$

Measured acceptance rate on HumanEval completions: 0.82, yielding a theoretical speedup of $3.1\times$ and observed wall-clock speedup of $2.8\times$ on $1\times$ A100.

4 Evaluation

4.1 Code Quality Benchmarks

Table 3: Zen-Coder-Flash benchmark results versus comparable models.

Benchmark	Flash (8B)	Zen-Code (14B)	Recovery	Comp. A (7B)	Comp. B (8B)
HumanEval (pass@1)	81.4	87.2	93.4%	76.3	78.1
HumanEval+ (pass@1)	76.8	82.4	93.2%	71.2	73.4
MBPP (pass@1)	76.4	82.3	92.8%	70.8	73.1
MultiPL-E (avg)	66.8	72.4	92.3%	61.2	63.7
FIM single-line	79.1	82.3	96.1%	72.4	75.8
FIM multi-line	57.3	61.4	93.3%	51.7	54.2

4.2 Latency Benchmarks

Latency is measured on a single A100-80GB GPU, BF16 precision, with vLLM v0.4 [10] serving, batch size 1.

Table 4: Latency benchmark (single A100-80GB, BF16, batch=1).

Model	TTFT P50 (ms)	TTFT P95 (ms)	TTFT P99 (ms)
Zen-Coder-Flash (8B)	27	43	61
Zen-Code (14B)	52	89	124
Competitor A (7B)	24	40	57
Competitor B (13B)	48	82	118

Time-to-first-token (TTFT) is measured with a 512-token prefix (typical autocomplete context). Zen-Coder-Flash’s P95 of 43ms satisfies the <50ms IDE latency budget.

4.3 Throughput Benchmarks

Table 5: Generation throughput (tokens/second) on single A100-80GB.

Model	Batch=1	Batch=8	Batch=16
Zen-Coder-Flash (8B)	312	640	850
Zen-Code (14B)	148	298	413

At batch size 16 (typical for shared inference serving), Zen-Coder-Flash delivers 850 tok/s, enabling simultaneous low-latency completions for large developer teams from a single GPU.

4.4 Speculative Decoding Performance

Table 6: Speculative decoding (Zen-Coder-Flash draft + Zen-Code verifier).

Configuration	Tok/s	Speedup vs. Zen-Code	HumanEval
Zen-Code alone	148	1.0×	87.2
Spec-Dec ($\gamma = 4$)	312	2.1×	87.1
Spec-Dec ($\gamma = 6$)	414	2.8×	87.0
Spec-Dec ($\gamma = 8$)	431	2.9×	86.8

$\gamma = 6$ provides the best throughput/quality trade-off, recovering 99.8% of Zen-Code’s HumanEval at 2.8× throughput with no quality degradation (speculative decoding is mathematically lossless modulo floating-point differences in acceptance sampling).

5 Distillation Ablation

Table 7: Ablation of distillation loss components on HumanEval and FIM benchmarks.

Loss configuration	HumanEval	FIM single-line	Latency P95
CE only (no KD)	76.3	68.4	43ms
CE + KL	79.8	75.1	43ms
CE + feature align	78.4	73.6	43ms
CE + KL + feature align	81.4	79.1	43ms

Both KL divergence and feature alignment contribute independently: KL improves token-level distribution matching (+3.5pp HumanEval), while feature alignment improves hidden state quality (+2.1pp), with the combined loss exceeding either alone (+5.1pp over CE-only).

6 IDE Integration and Deployment

6.1 Deployment Architecture

Zen-Coder-Flash is designed for always-on inference deployment:

Listing 1: Starting a Zen-Coder-Flash inference server.

```
# Single GPU deployment (24GB VRAM minimum)
docker run --gpus=1 \
  -e MODEL=zen-coder-flash-8b \
  -e MAX_BATCH_SIZE=32 \
  -e MAX_CONTEXT=32768 \
  -p 8080:8080 \
  ghcr.io/zenlm/inference:latest

# Speculative decoding with Zen-Code verifier (2x A100)
docker run --gpus=2 \
  -e MODEL=zen-code-14b \
  -e DRAFT_MODEL=zen-coder-flash-8b \
  -e SPECULATIVE_GAMMA=6 \
  -p 8080:8080 \
  ghcr.io/zenlm/inference:latest
```

6.2 Continue Extension Configuration

The Continue VS Code extension can be configured to use Zen-Coder-Flash for autocomplete with the following configuration:

Listing 2: Continue config.json for Zen-Coder-Flash.

```
{
  "tabAutocompleteModel": {
    "title": "Zen-Coder-Flash",
    "provider": "openai",
    "model": "zen-coder-flash-8b",
    "apiBase": "https://api.zenlm.org/v1",
    "apiKey": "<your-api-key>"
  },
  "tabAutocompleteOptions": {
    "useCopyBuffer": false,
    "maxPromptTokens": 1024,
    "prefixPercentage": 0.85
  }
}
```

7 Related Work

Knowledge distillation for language models was introduced by [2] and applied to transformers in DistilBERT [3]. For generative models, sequence-level KD [4] and token-level KL distillation [5] have emerged as standard approaches. Intermediate layer alignment follows the TinyBERT [6] and PKD [7] patterns. Speculative decoding [8, 9] enables lossless speedup using a small draft model; our use of Zen-Coder-Flash as draft for Zen-Code follows this paradigm.

Fast code completion models have been explored in FauxPilot and similar systems; the key contribution of Zen-Coder-Flash is pairing distillation-preserved code quality with the latency targets required for production IDE deployment.

8 Limitations

The 8B student model recovers 93% of the teacher’s HumanEval score but the 7% gap widens on more complex tasks: SWE-bench recovery is approximately 68% (19.3% vs. 28.4%), as complex multi-file reasoning benefits disproportionately from the teacher’s additional capacity. The 32K context window (vs. Zen-Code’s 64K) limits use in very large repository-level tasks. Speculative decoding latency benefits are reduced under variable-length acceptance patterns typical of code generation (vs. more predictable text completion).

9 Conclusion

Zen-Coder-Flash delivers IDE-grade code completion with P95 latency of 43ms and 850 tok/s throughput at batch size 16 on a single A100 GPU, while maintaining HumanEval 81.4% through structured knowledge distillation from Zen-Code. The combined KL-divergence and intermediate-feature-alignment distillation protocol recovers 93% of teacher performance at 55% of parameter count and $2.1\times$ lower latency. As a speculative decoding draft model for Zen-Code, it additionally provides $2.8\times$ throughput improvement with negligible quality loss. Zen-Coder-Flash is the recommended model for all latency-critical developer tooling integrations in the Zen family ecosystem.

References

- [1] A. Svyatkovskiy et al., “Fast and Low-Cost Code Intelligence with Lightweight Models,” *arXiv:2104.14765*, 2021.
- [2] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” *arXiv:1503.02531*, 2015.
- [3] V. Sanh et al., “DistilBERT, a distilled version of BERT,” *arXiv:1910.01108*, 2019.
- [4] Y. Kim and A. Rush, “Sequence-Level Knowledge Distillation,” *EMNLP*, 2016.
- [5] Y. Gu et al., “MiniLLM: Knowledge Distillation of Large Language Models,” *ICLR*, 2024.
- [6] X. Jiao et al., “TinyBERT: Distilling BERT for Natural Language Understanding,” *EMNLP*, 2020.
- [7] S. Sun et al., “Patient Knowledge Distillation for BERT Compression,” *EMNLP*, 2019.
- [8] Y. Leviathan, M. Kalman, and Y. Matias, “Fast Inference from Transformers via Speculative Decoding,” *ICML*, 2023.
- [9] C. Chen et al., “Accelerating Large Language Model Decoding with Speculative Sampling,” *arXiv:2302.01318*, 2023.
- [10] W. Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” *SOSP*, 2023.
- [11] M. Bavarian et al., “Efficient Training of Language Models to Fill in the Middle,” *arXiv:2207.14255*, 2022.
- [12] M. Chen et al., “Evaluating Large Language Models Trained on Code,” *arXiv:2107.03374*, 2021.