

Zen-Coder-Flash: A Distilled Qwen-Coder Variant for Low-Latency Code Completion

Technical Report v2025.02

Antje Worring, Zach Kelling
Zen LM Research Team
research@zenlm.org

February 2025

Abstract

Zen-Coder-Flash is a smaller code-completion student derived by knowledge distillation from Alibaba’s open-weight **Qwen2.5-Coder** [1] (Apache-2.0). The teacher is the Qwen2.5-Coder-14B checkpoint and the student is initialized from a smaller Qwen2.5-Coder checkpoint (7B-class); we do *not* pretrain a new model from scratch and we do *not* introduce a novel “Zen MoDE” architecture—both the teacher and the student backbone are Qwen’s, the tokenizer and fill-in-the-middle (FIM) format are Qwen’s, and our contribution is the distillation recipe and the deployment packaging. This report describes that recipe: sequence-level KD with intermediate-layer feature alignment, the latency-oriented serving configuration, and the optional use of the student as a draft model for speculative decoding against the larger teacher. Benchmark and latency figures quoted here are illustrative *targets* for the recipe, not validated leaderboard claims; for authoritative quality numbers on the underlying models we refer the reader to the Qwen2.5-Coder technical report [1].

Contents

1	Introduction	3
2	Architecture	3
2.1	Student Model Specification	3
2.2	Architectural Choices for Latency	4
3	Distillation Methodology	4
3.1	Overview	4
3.2	Initialization	4
3.3	Knowledge Distillation	4
3.4	Speculative Decoding Setup	5
4	Expected Performance (Targets)	6
4.1	Code Quality (Target Recovery)	6
4.2	Latency (Target)	6
4.3	Speculative Decoding (Expected)	6
5	Distillation Ablation	6
6	IDE Integration and Deployment	6
6.1	Deployment Architecture	6
6.2	Continue Extension Configuration	7

7	Related Work	7
8	Limitations	7
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 [2]. 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 larger than necessary for this budget on widely available server hardware. Zen-Coder-Flash addresses this by distilling Qwen2.5-Coder-14B [1] into a smaller (~ 7 –8B-class) Qwen2.5-Coder student, aiming to preserve code quality while reducing serving latency.

Scope of contributions (engineering recipe, not new science):

- A distillation recipe applying sequence-level KD with intermediate-layer feature alignment to a Qwen2.5-Coder teacher/student pair [1].
- A latency-oriented serving configuration targeting sub-50ms P95 time-to-first-token for short autocomplete prefixes on a single A100-class GPU.
- Optional speculative decoding: the distilled student serves as a draft model for the larger Qwen2.5-Coder-14B teacher, a standard lossless-speedup setup [9, 10].

The numeric targets in the tables below (e.g. P95 latency, throughput, pass@1) are design goals for this recipe on the stated hardware, *not* independently audited benchmark results. Authoritative quality numbers for the Qwen2.5-Coder family are in the upstream report [1]; we do not restate them as our own.

2 Architecture

2.1 Student Model Specification

Both the teacher and the student are Qwen2.5-Coder [1] checkpoints; the student is a smaller member of the same family (7B-class), not a bespoke “Zen MoDE” architecture. The hyperparameters below are those of the corresponding Qwen2.5-Coder checkpoints and are reproduced from Qwen’s model cards; the depth-to-width ratio and the shared vocabulary/RoPE settings follow from using the same upstream family for teacher and student.

Table 1: Student and teacher hyperparameters (both Qwen2.5-Coder [1]).

Hyperparameter	Student (Qwen2.5-Coder-7B)	Teacher (Qwen2.5-Coder-14B)
Parameters (total)	$\sim 7.6\text{B}$	$\sim 14.7\text{B}$
Layers	28	48
Hidden dimension	3,584	5,120
KV heads (GQA)	4	8
Vocabulary size	151,936	151,936
Context length	32,768	32,768
Position encoding	RoPE ($\theta = 1,000,000$)	RoPE ($\theta = 1,000,000$)

(Figures are the upstream Qwen2.5-Coder specifications; the exact student checkpoint chosen for a given deployment may differ in size, but it is in all cases an existing Qwen2.5-Coder model, not a newly designed architecture.)

2.2 Architectural Choices for Latency

The student’s smaller hidden dimension and lower layer count are the latency drivers: matrix-multiply cost per layer scales with hidden dimension, and total cost scales with layer count. Choosing a smaller existing Qwen2.5-Coder checkpoint as the student is what buys the latency reduction; we do not re-architect the model.

Grouped-query attention (inherited from the Qwen2.5-Coder backbone) keeps KV-cache requirements modest and enables large effective batch sizes. As an order-of-magnitude illustration of KV-cache size for an 8-KV-head, 128-dim-per-head configuration:

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

A modest per-layer KV-cache footprint at 32K context leaves headroom for the BF16 weights of a 7–8B-class student on a single 24 GB consumer GPU.

3 Distillation Methodology

3.1 Overview

Because both teacher and student are existing Qwen2.5-Coder [1] checkpoints, there is **no from-scratch pretraining phase**. The recipe is two steps applied on top of the upstream weights:

1. **Knowledge distillation**: starting from the pretrained Qwen2.5-Coder student checkpoint, distill from the Qwen2.5-Coder-14B teacher using combined token-level KD and intermediate-feature-alignment losses.
2. **Instruction fine-tuning**: a light SFT pass on instruction-code pairs to restore instruction-following after the KD pass.

Earlier drafts described a “Zen MoDE distillation pipeline” and a 500B-token from-scratch warm-up over a proprietary 2.5T corpus. Neither is accurate: the student is not trained from scratch (it is an existing Qwen2.5-Coder checkpoint) and there is no proprietary multi-trillion-token corpus behind this work. Those claims have been removed.

3.2 Initialization

The student is *initialized from a pretrained Qwen2.5-Coder checkpoint*, not randomly initialized. This is precisely why no warm-up pretraining is needed: the student already has strong code priors from Qwen’s upstream pretraining, so distillation begins from a competent starting point rather than from poor initial approximations of the teacher.

Distillation-run settings (illustrative).

- Optimizer: AdamW, LR 1×10^{-4} , cosine to 5×10^{-6}
- Batch: 2M tokens
- FIM rate: 50% (Qwen2.5-Coder FIM format)
- Scale: tens of billions of distillation tokens (orders of magnitude below a pretraining run)

3.3 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 :

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

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 hidden dimension ($d_s = 3584$ for the 7B student) to the teacher’s ($d_t = 5120$):

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

where M maps a set of evenly spaced student layers to evenly spaced teacher layers (the student has fewer layers than the teacher, so the mapping is many-to-fewer). 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
Scale	distillation-scale (tens of billions of tokens)

3.4 Speculative Decoding Setup

The distilled student can serve as a draft model in a speculative decoding pipeline with the Qwen2.5-Coder-14B teacher [1] 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 :

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

A typical draft-model acceptance rate for code completion is in the 0.7–0.85 range; at the upper end this corresponds to a single-host wall-clock speedup of roughly 2–3 $\times$ over running the teacher alone, consistent with the speculative-decoding literature [9, 10]. The exact figure is workload-dependent and should be measured per deployment.

4 Expected Performance (Targets)

The numbers in this section are design targets for the recipe, not audited benchmark results. We have not run an independent leaderboard evaluation of the distilled student, and we do not claim to beat the upstream Qwen2.5-Coder models. For authoritative code-quality numbers on the teacher and on same-size Qwen2.5-Coder checkpoints, see the Qwen2.5-Coder technical report [1]; those upstream numbers bound what a distilled student can be expected to recover.

4.1 Code Quality (Target Recovery)

The aim of distillation is to recover most of the teacher’s code-quality on the smaller student. As a rough target we aim for $\sim 90\%$ + recovery of the teacher’s pass@1 on HumanEval/MBPP; the student cannot exceed the teacher, and a same-size Qwen2.5-Coder checkpoint is the relevant floor. We report no fabricated head-to-head table here; per-deployment numbers should be measured against the upstream Qwen2.5-Coder baselines [1].

4.2 Latency (Target)

The serving goal is sub-50ms P95 time-to-first-token for short (~ 512 -token) autocomplete prefixes on a single A100-class GPU in BF16 with a paged-attention server [11], which a 7–8B-class model can plausibly meet where the 14B teacher cannot. These are engineering targets to be validated on the deployment hardware, not measured results reproduced here.

4.3 Speculative Decoding (Expected)

Using the distilled student as a draft model for the Qwen2.5-Coder-14B teacher is expected to yield a 2–3 $\times$ wall-clock speedup over the teacher alone at the teacher’s exact output distribution, since speculative decoding is lossless modulo floating-point differences in acceptance sampling [9, 10]. Realized speedup depends on the acceptance rate for the target workload and must be measured.

5 Distillation Ablation

We do not report a table of measured pass@1 values for each loss configuration, as we have not run an audited evaluation to support specific numbers. Qualitatively, and consistent with the distillation literature [3, 5, 6, 7], we expect both the KL-divergence term (token-level distribution matching) and the intermediate-feature-alignment term (hidden-state matching) to contribute independently over a cross-entropy-only baseline, with the combined loss outperforming either term alone. Latency is unaffected by the choice of distillation loss (it changes training, not the served architecture). Any quantitative ablation should be reported with the deployment’s own measured numbers.

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 Qwen2.5-Coder-14B verifier (2x A100)
docker run --gpus=2 \
-e MODEL=Qwen/Qwen2.5-Coder-14B-Instruct \
-e DRAFT_MODEL=zen-coder-flash \
-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 [3] and applied to transformers in DistilBERT [4]. For generative models, sequence-level KD [5] and token-level KL distillation [6] have emerged as standard approaches. Intermediate layer alignment follows the TinyBERT [7] and PKD [8] patterns. Speculative decoding [9, 10] enables lossless speedup using a small draft model; our use of the distilled student as a draft for the Qwen2.5-Coder-14B teacher [1] follows this paradigm.

The base models themselves are Qwen2.5-Coder [1]; Zen-Coder-Flash contributes a distillation recipe and a latency-oriented serving configuration on top of those open-weight models, not a new model. Fast code completion has also been explored in systems such as FauxPilot.

8 Limitations

A distilled student is bounded by its teacher: it cannot exceed Qwen2.5-Coder-14B [1], and the gap typically widens on complex multi-file reasoning tasks (e.g. SWE-bench), which benefit disproportionately from the teacher’s additional capacity. The student’s context window is bounded by the chosen Qwen2.5-Coder checkpoint. Speculative-decoding speedups are reduced under the variable-length acceptance patterns typical of code generation. Finally, the quality

and latency figures in this report are *targets*, not audited measurements; deployments should validate against the upstream Qwen2.5-Coder baselines and their own hardware.

9 Conclusion

Zen-Coder-Flash is a smaller, lower-latency code-completion student obtained by knowledge distillation from Alibaba’s open-weight Qwen2.5-Coder-14B [1] into a smaller Qwen2.5-Coder checkpoint. It is not a from-scratch model and does not use a “Zen MoDE” architecture; the teacher, the student backbone, the tokenizer, and the FIM format are all Qwen’s, and the work is released under the upstream Apache-2.0 license with attribution. Our contribution is the distillation recipe (sequence-level KD plus intermediate-feature alignment), a latency-oriented serving configuration, and the option to use the student as a speculative-decoding draft for the teacher. The latency and quality figures here are design targets for that recipe, to be validated per deployment against the upstream Qwen2.5-Coder baselines.

References

- [1] B. Hui, J. Yang, et al. (Qwen Team, Alibaba), “Qwen2.5-Coder Technical Report,” *arXiv:2409.12186*, 2024. Apache-2.0 (0.5B/1.5B/7B/14B/32B). <https://github.com/QwenLM/Qwen2.5-Coder>.
- [2] A. Svyatkovskiy et al., “Fast and Low-Cost Code Intelligence with Lightweight Models,” *arXiv:2104.14765*, 2021.
- [3] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” *arXiv:1503.02531*, 2015.
- [4] V. Sanh et al., “DistilBERT, a distilled version of BERT,” *arXiv:1910.01108*, 2019.
- [5] Y. Kim and A. Rush, “Sequence-Level Knowledge Distillation,” *EMNLP*, 2016.
- [6] Y. Gu et al., “MiniLLM: Knowledge Distillation of Large Language Models,” *ICLR*, 2024.
- [7] X. Jiao et al., “TinyBERT: Distilling BERT for Natural Language Understanding,” *EMNLP*, 2020.
- [8] S. Sun et al., “Patient Knowledge Distillation for BERT Compression,” *EMNLP*, 2019.
- [9] Y. Leviathan, M. Kalman, and Y. Matias, “Fast Inference from Transformers via Speculative Decoding,” *ICML*, 2023.
- [10] C. Chen et al., “Accelerating Large Language Model Decoding with Speculative Sampling,” *arXiv:2302.01318*, 2023.
- [11] W. Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” *SOSP*, 2023.
- [12] M. Bavarian et al., “Efficient Training of Language Models to Fill in the Middle,” *arXiv:2207.14255*, 2022.
- [13] M. Chen et al., “Evaluating Large Language Models Trained on Code,” *arXiv:2107.03374*, 2021.