

Zen-Code: Code Completion and Intelligence at 14B Scale

Technical Report v2024.12

Zen LM Research Team
research@zenlm.org

December 2024

Abstract

We present **Zen-Code**, a 14 billion parameter language model optimized for code generation, completion, and understanding across 80+ programming languages. Built on the Zen MoDE (Mixture of Distilled Experts) architecture, Zen-Code is pretrained on 2T tokens of high-quality code with fill-in-the-middle (FIM) objectives and fine-tuned on repository-scale context tasks. Zen-Code achieves HumanEval 87.2%, MBPP 82.3%, SWE-bench 28.4%, and RepoBench 0.834, establishing it as the capable code-specialist in the Zen family. The model supports long repository-level context up to 64K tokens, multi-file awareness, and standard IDE integration formats including LSP-compatible completions.

Contents

1	Introduction	3
2	Architecture	3
2.1	Model Configuration	3
2.2	Fill-in-the-Middle Token Scheme	3
2.3	Repository-Level Context Structure	4
3	Training Methodology	4
3.1	Pretraining Data	4
3.2	Per-Language Distribution	5
3.3	Pretraining Procedure	5
3.4	Repository-Level Fine-Tuning	5
3.5	Instruction Fine-Tuning	5
4	Evaluation	6
4.1	Code Generation Benchmarks	6
4.2	Multilingual Code Evaluation	6
4.3	Fill-in-the-Middle Evaluation	7
4.4	Repository-Level Evaluation	7
5	IDE Integration	7
6	Ablation Studies	8
6.1	Effect of FIM Training Rate	8
6.2	Effect of Repository Fine-Tuning	8
7	Related Work	8

8	Limitations	8
9	Conclusion	8

1 Introduction

Code generation has emerged as one of the most commercially significant applications of large language models. From IDE autocomplete to automated pull request generation, models that reliably understand and generate code at the repository scale provide substantial productivity gains [1, 2]. General-purpose language models, while capable of code generation, are not optimized for the specific properties of software: multi-file context, precise syntax requirements, test-driven correctness, and sub-token completion granularity.

Zen-Code addresses this gap with a model purpose-built for code. Key contributions:

- A 14B parameter model pretrained on 2T code tokens from 80+ languages, with an additional 500B tokens of technical documentation and Stack Exchange content.
- Fill-in-the-middle (FIM) training enabling prefix-suffix-middle completion, the format required by production IDE integrations.
- Repository-level context training on 64K token windows with structured file-path and symbol-table prefixes.
- Post-training on instruction-following code tasks including debugging, refactoring, test generation, and code review.
- Benchmark results: HumanEval 87.2%, MBPP 82.3%, SWE-bench 28.4%, RepoBench 0.834.

2 Architecture

2.1 Model Configuration

Zen-Code uses the Zen MoDE architecture scaled to 14B parameters, a tier chosen to balance code reasoning capability with inference cost on developer hardware (single A100 or 2× RTX 4090 in BF16).

Table 1: Zen-Code architecture hyperparameters.

Hyperparameter	Value
Parameters (total)	14.4B
Layers	40
Attention heads	40
KV heads (GQA)	8
Hidden dimension	5120
FFN intermediate dimension	13,696
Vocabulary size	151,936
Context length (training)	65,536
Position encoding	RoPE ($\theta = 2,000,000$)
Activation	SiLU
Normalization	RMSNorm
FIM tokens	<code>< fim_prefix ></code> , <code>< fim_middle ></code> , <code>< fim_suffix ></code>

2.2 Fill-in-the-Middle Token Scheme

Zen-Code supports fill-in-the-middle (FIM) completion [3], the standard format for IDE inline completion. The three FIM sentinel tokens partition the input:

Listing 1: FIM token format.

```
<|fim_prefix|>[prefix code]<|fim_suffix|>[suffix code]<|fim_middle|>
```

During training, 50% of code samples are converted to FIM format using the PSM (prefix-suffix-middle) or SPM (suffix-prefix-middle) transformation drawn with equal probability, following the recommendation of [3].

For the PSM transformation of a document d split at position s :

$$d_{\text{FIM}} = [\text{FP}] \cdot d_{[:s]} \cdot [\text{FS}] \cdot d_{[s:]} \cdot [\text{FM}] \cdot d_{\text{middle}} \quad (1)$$

where FP, FS, FM are the prefix, suffix, and middle sentinels respectively, and d_{middle} is the held-out span the model must predict.

2.3 Repository-Level Context Structure

For long-context code tasks, Zen-Code accepts structured repository context using a lightweight file-header format:

Listing 2: Repository context format.

```
# File: src/utils/parser.py
[file content]

# File: src/models/base.py
[file content]

# File: [target file, completion requested here]
[prefix]<|fim_suffix|>[suffix]<|fim_middle|>
```

This format is injected by IDE extensions and agents to provide cross-file context. The 64K context window accommodates typical repository relevant-file sets (10–50 files of 500–1000 lines each).

3 Training Methodology

3.1 Pretraining Data

Zen-Code’s 2.5T-token pretraining corpus is heavily code-dominated.

Table 2: Zen-Code pretraining data composition.

Domain	Tokens (B)	Fraction
Source code (80+ languages)	1,500	60.0%
Technical documentation	375	15.0%
Stack Overflow / Q&A	250	10.0%
Research papers (CS/ML)	125	5.0%
General web (filtered)	125	5.0%
Formal specs (Lean, Coq, TLA+)	75	3.0%
Synthetic code (unit tests)	50	2.0%
Total	2,500	100.0%

3.2 Per-Language Distribution

Table 3: Top programming languages by token count in pretraining corpus.

Language	Tokens (B)	Share of code data
Python	360	24.0%
JavaScript	240	16.0%
TypeScript	150	10.0%
Java	135	9.0%
C/C++	120	8.0%
Rust	90	6.0%
Go	75	5.0%
Ruby	45	3.0%
PHP	45	3.0%
Shell/Bash	45	3.0%
Other (70+)	195	13.0%
Total	1,500	100.0%

Data deduplication uses a two-stage process: exact SHA-256 deduplication at the file level, followed by MinHash near-deduplication with 10-shingling at LSH threshold 0.8 to remove forked repositories and boilerplate.

3.3 Pretraining Procedure

Zen-Code is trained from scratch (not initialized from Zen base) to allow the training distribution to be fully code-optimized without general text forgetting constraints.

- Optimizer: AdamW, LR $3 \times 10^{-4} \rightarrow 3 \times 10^{-5}$ (cosine)
- Warm-up: 1000 steps
- Weight decay: 0.1
- Batch size: 4M tokens
- FIM rate: 50% of code samples
- Precision: BF16
- Training duration: ≈ 625 K steps, ≈ 512 H100 GPUs

3.4 Repository-Level Fine-Tuning

After base pretraining, Zen-Code undergoes a 50B-token repository-level fine-tuning phase using full repository snapshots. Repositories are sampled from GitHub with permissive licenses, filtered for non-trivial size (>10 files, >1 K lines). Files within each repository are packed into 64K-token windows following a topological ordering based on import dependencies.

3.5 Instruction Fine-Tuning

A final SFT phase on 800K instruction-code pairs covering:

- Code generation from natural language descriptions

- Bug finding and fixing (with error message context)
- Code explanation and documentation generation
- Unit test generation from function signatures
- Code refactoring and style improvement
- API usage from documentation

SFT learning rate: 1×10^{-5} , 2 epochs.

4 Evaluation

4.1 Code Generation Benchmarks

Table 4: Code generation benchmark results.

Benchmark	Zen-Code (14B)	Comp. A (15B)	Comp. B (13B)	Comp. C (7B)
HumanEval (pass@1)	87.2	85.1	83.6	76.3
HumanEval+ (pass@1)	82.4	80.3	78.7	71.2
MBPP (pass@1)	82.3	80.1	78.4	70.8
MBPP+ (pass@1)	76.8	74.3	72.1	64.3
LiveCodeBench (3-month)	54.2	51.7	49.3	41.8
SWE-bench Verified	28.4	25.3	22.8	14.6
RepoBench (retrieval)	0.834	0.812	0.798	0.741
MultiPL-E (avg 12 langs)	72.4	70.8	68.3	61.2

4.2 Multilingual Code Evaluation

Table 5: MultiPL-E pass@1 by programming language.

Language	Zen-Code (14B)	Competitor (15B)
Python	87.2	85.1
JavaScript	78.6	76.4
TypeScript	76.4	74.1
Java	74.3	72.8
C++	70.8	68.3
Rust	68.4	65.7
Go	73.1	71.4
Ruby	65.3	62.8
PHP	63.7	61.4
Shell	58.4	55.9

4.3 Fill-in-the-Middle Evaluation

Table 6: FIM completion accuracy on HumanEval-Infilling.

Metric	Zen-Code (14B)	Competitor (15B)
Single-line (exact match)	82.3	79.1
Multi-line (exact match)	61.4	57.8
Single-line (functional)	91.7	89.4
Multi-line (functional)	74.6	71.3

4.4 Repository-Level Evaluation

SWE-bench Verified requires understanding a full repository, identifying which files to modify, and generating a correct patch that passes the repository’s test suite.

Table 7: SWE-bench Verified resolution rates by issue category.

Category	Zen-Code (14B)
Logic errors	34.2%
Type errors	41.8%
Off-by-one	38.7%
Missing handling	31.4%
API usage errors	29.6%
Regression fixes	18.3%
Overall (verified)	28.4%

5 IDE Integration

Zen-Code is designed for IDE integration via a thin inference server exposing an OpenAI-compatible completions API with FIM support:

Listing 3: Example FIM completion request.

```
import openai

client = openai.OpenAI(
    base_url="https://api.zenlm.org/v1",
    api_key="<zen-api-key>"
)

response = client.completions.create(
    model="zen-code-14b",
    prompt="<|fim_prefix|>def merge_sorted_arrays(a, b): \n    \n    "<|fim_suffix|>\n    \n    return result\n<|fim_middle|>",
    max_tokens=128,
    temperature=0.1,
    stop=["<|endoftext|>"]
)

print(response.choices[0].text)
```

Available IDE integrations include VS Code (via Continue and Tabby plugins), JetBrains IDEs, Neovim (via coc.nvim), and Emacs (via copilot.el-compatible backends).

6 Ablation Studies

6.1 Effect of FIM Training Rate

Table 8: HumanEval-Infilling (single-line) vs. FIM training fraction.

FIM rate	HumanEval pass@1	FIM exact match (single)
0% (no FIM)	86.8	41.2
25%	85.9	74.3
50%	87.2	82.3
75%	84.3	83.1

50% FIM rate provides the optimal balance: full left-to-right completion performance is maintained (87.2% vs. 86.8% baseline) while infilling performance improves dramatically. At 75% FIM the model under-trains on left-to-right generation, reducing HumanEval by 2.9pp.

6.2 Effect of Repository Fine-Tuning

Table 9: Impact of repository-level fine-tuning on long-context tasks.

Model	RepoBench	SWE-bench Verified
Base (no repo FT)	0.761	19.3
+ Repo FT (8K context)	0.803	23.7
+ Repo FT (64K context)	0.834	28.4

7 Related Work

Code-specialized language models trace to Codex [1], which demonstrated that training on large code corpora enables competitive function-level generation. CodeGen [2] and its successors explored scaling laws for code. StarCoder [4] trained on The Stack, a permissively licensed code corpus with strong deduplication. Fill-in-the-middle training was introduced in [3] and adopted by most subsequent code models. SWE-bench [5] introduced repository-level task solving as a benchmark; the verified subset reduces noise in evaluation. RepoBench [6] tests long-context retrieval within repository structure.

8 Limitations

Zen-Code’s SWE-bench performance of 28.4% indicates that most real-world repository-level issues remain unsolved in single-pass generation without agentic scaffolding. The model does not have access to runtime execution during generation; code correctness is assessed only at test time. Languages with fewer than 10B training tokens show substantially weaker performance than top-tier languages. Security-sensitive code generation (cryptography, access control) should be reviewed by experts regardless of benchmark scores.

9 Conclusion

Zen-Code provides a purpose-built code intelligence model at 14B parameters, delivering HumanEval 87.2%, MBPP 82.3%, SWE-bench 28.4%, and RepoBench 0.834 through a combination of code-dominated pretraining, fill-in-the-middle objectives, and repository-scale context

fine-tuning. The model’s 64K context window, FIM support, and IDE-compatible API make it a practical choice for developer tooling deployment. Zen-Code is the foundational code model in the Zen family, with Zen-Coder-Flash providing a distilled fast-inference variant for latency-critical IDE autocomplete applications.

References

- [1] M. Chen et al., “Evaluating Large Language Models Trained on Code,” *arXiv:2107.03374*, 2021.
- [2] E. Nijkamp et al., “CodeGen2: Lessons for Training LLMs on Programming and Natural Languages,” *ICLR*, 2023.
- [3] M. Bavarian et al., “Efficient Training of Language Models to Fill in the Middle,” *arXiv:2207.14255*, 2022.
- [4] R. Li et al., “StarCoder: May the Source Be with You!” *arXiv:2305.06161*, 2023.
- [5] C. Jimenez et al., “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” *arXiv:2310.06770*, 2023.
- [6] T. Liu et al., “RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems,” *arXiv:2306.03091*, 2023.
- [7] I. Loshchilov and F. Hutter, “Decoupled Weight Decay Regularization,” *ICLR*, 2019.
- [8] A. Broder, “On the resemblance and containment of documents,” *Sequences*, 1997.