

Zen4-Coder: A Fourth-Generation Code Intelligence Model

with Multi-Repository Understanding and Autonomous Debugging

Technical Whitepaper v2026.01

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

January 2026

Abstract

Zen4-Coder is our fourth-generation code specialist at 32 billion parameters, built on the Mixture of Experts (MoE) architecture and trained on real developer workflows spanning open-source repositories, issue trackers, pull request histories, and code review datasets. Unlike prior generation models that treat code generation as isolated function synthesis, Zen4-Coder understands multi-repository dependency graphs, reasons across codebases spanning millions of lines, and autonomously executes debugging loops by forming and testing hypotheses. On standard code generation benchmarks, Zen4-Coder achieves 95.2% on HumanEval, 58.4% on SWE-bench, 0.891 on RepoBench, and 89.3% on MBPP+, establishing new state-of-the-art results among models in its parameter class. This paper describes the architecture, training methodology, capability evaluations, and safety considerations for Zen4-Coder.

Contents

1	Introduction	3
1.1	Model Overview	3
2	Architecture	3
2.1	Mixture of Experts (MoE)	3
2.2	Multi-Repository Attention	4
2.3	Debugging Loop Architecture	4
3	Training Methodology	5
3.1	Pre-Training Data	5
3.2	Instruction Fine-Tuning	5
3.3	Reinforcement Learning from Code Execution	5
4	Evaluation	6
4.1	Code Generation Benchmarks	6
4.2	Software Engineering Benchmarks	6
4.3	Debugging Capability	6
4.4	Test Generation Quality	6
4.5	Multi-Language Performance	7

5	Multi-Repository Understanding	7
5.1	Dependency Graph Reasoning	7
5.2	API Contract Verification	7
6	Agentic Capabilities	7
6.1	Tool Use	7
6.2	Long-Horizon Planning	8
7	Safety and Reliability	8
7.1	Code Safety	8
7.2	Hallucination Rates	8
8	Deployment	8
8.1	Inference Requirements	8
8.2	API Compatibility	9
9	Related Work	9
10	Conclusion	9

1 Introduction

Software development is fundamentally a multi-step, multi-context activity. Developers navigate large codebases, reason about existing APIs and invariants, debug failures by forming causal hypotheses, and synthesize changes that integrate cleanly with surrounding infrastructure. Early code generation models treated code as isolated text sequences and optimized for function-level completion accuracy. This framing captures only a small fraction of real engineering work.

Zen4-Coder represents a qualitative shift in capability: a model trained not merely to complete function signatures but to act as an engineering collaborator across the full development lifecycle. The key advances introduced in this generation are:

1. **Multi-Repository Context Understanding:** Zen4-Coder encodes cross-repository dependency graphs and can reason about how changes in one module propagate to consumers across separate repositories.
2. **Autonomous Debugging:** Given a failing test or runtime error, Zen4-Coder forms a ranked list of hypotheses, generates candidate patches, evaluates them symbolically, and iterates until the failure is resolved.
3. **Test Generation:** Zen4-Coder synthesizes comprehensive test suites including property-based tests, regression tests for known failure modes, and edge-case coverage that complements existing test infrastructure.
4. **MoE Architecture:** The 32B parameter model uses sparse expert routing to activate specialized subnetworks for distinct programming languages, paradigms, and reasoning tasks, achieving high accuracy without proportional inference cost.

1.1 Model Overview

Table 1: Zen4-Coder Model Specification

Parameter	Value
Architecture	Mixture of Experts (MoE)
Total Parameters	32B
Active Parameters per Token	8B
Context Window	128K tokens
Supported Languages	92 programming languages
Version	v2026.01
Release Date	January 2026

2 Architecture

2.1 Mixture of Experts (MoE)

Zen4-Coder is built on the MoE architecture, which combines sparse mixture-of-experts routing with knowledge distillation from an ensemble of specialized teacher models. The core insight is that different programming tasks require qualitatively different reasoning patterns: syntactic completion, semantic reasoning about types and invariants, debugging (causal inference), and architecture design each activate different cognitive processes in human experts.

The MoE architecture formalizes this intuition. Given a token sequence $x_{1:t}$, the router R_θ computes a probability distribution over N expert networks $\{E_1, \dots, E_N\}$:

$$R_\theta(x_{1:t}) = \text{softmax}(W_r \cdot h_t + b_r) \in \mathbb{R}^N \quad (1)$$

where h_t is the hidden state at position t . The top- k experts are activated and their outputs combined:

$$y_t = \sum_{i \in \text{top}_k(R_\theta)} \alpha_i \cdot E_i(x_{1:t}), \quad \alpha_i = \frac{R_\theta(x_{1:t})_i}{\sum_{j \in \text{top}_k} R_\theta(x_{1:t})_j} \quad (2)$$

For Zen4-Coder, $N = 64$ experts and $k = 4$, yielding 8B active parameters from a 32B total parameter budget. Expert specialization emerges naturally through training but is reinforced by an auxiliary load-balancing loss:

$$\mathcal{L}_{\text{aux}} = \alpha \cdot \sum_{i=1}^N f_i \cdot P_i \quad (3)$$

where f_i is the fraction of tokens routed to expert i and P_i is the mean routing probability for expert i .

2.2 Multi-Repository Attention

Standard attention operates over a single contiguous context window. Multi-repository understanding requires relating code fragments that may be separated by repository boundaries, not contiguous text. Zen4-Coder extends the attention mechanism with a *cross-repo attention* layer that operates over a structured index of repository chunks:

$$\text{CrossRepoAttn}(Q, K_{\text{index}}, V_{\text{index}}) = \text{softmax} \left(\frac{QK_{\text{index}}^T}{\sqrt{d_k}} \right) V_{\text{index}} \quad (4)$$

The index $\{K_{\text{index}}, V_{\text{index}}\}$ is constructed from a retrieval corpus of relevant code chunks selected by a sparse retrieval step using BM25 + embedding similarity. This allows the 128K token context to incorporate information from multi-million-line codebases without exceeding context limits.

2.3 Debugging Loop Architecture

Autonomous debugging is implemented as a structured agentic loop with four phases:

1. **Observation:** Parse the failing test output, error message, and stack trace into a structured failure report.
2. **Hypothesis Generation:** Generate a ranked list of root cause hypotheses using the model’s causal reasoning capability.
3. **Patch Synthesis:** For each hypothesis, generate a minimal code patch that would resolve the hypothesized root cause.
4. **Validation:** Execute the patch in a sandboxed environment, check test outcomes, and update the hypothesis ranking based on results.

The loop terminates when a patch passes all relevant tests or when a budget of K iterations is exhausted. Default $K = 8$ for interactive use, $K = 32$ for batch agentic pipelines.

3 Training Methodology

3.1 Pre-Training Data

Zen4-Coder was pre-trained on a curated corpus of 6.5 trillion code tokens drawn from:

Table 2: Pre-Training Data Composition

Source	Tokens (B)	Fraction
Public code repositories	3,200	49.2%
Pull request & code review history	800	12.3%
Issue tracker & bug report corpora	600	9.2%
Documentation and API references	700	10.8%
Stack Overflow and technical forums	500	7.7%
Academic CS papers and textbooks	300	4.6%
Test suites and CI/CD configs	250	3.8%
General natural language (filtered)	150	2.3%
Total	6,500	100%

3.2 Instruction Fine-Tuning

After pre-training, Zen4-Coder was fine-tuned on curated instruction-response pairs covering:

- **Code generation:** Natural language to code, with multi-step specification clarification.
- **Bug fixing:** Given a bug report and codebase context, produce a correct patch.
- **Code review:** Identify defects, suggest improvements, and explain rationale.
- **Test generation:** Given an implementation, synthesize a comprehensive test suite.
- **Refactoring:** Restructure code for improved readability, performance, or maintainability.
- **Multi-repo tasks:** Trace cross-repository dependencies and suggest coordinated changes.

3.3 Reinforcement Learning from Code Execution

A critical component of Zen4-Coder’s training is Reinforcement Learning from Code Execution (RLCE). Unlike RLHF which relies on human preference signals, RLCE uses executable ground truth: a candidate solution earns reward proportional to the fraction of test cases it passes.

$$r(y) = \frac{1}{|T|} \sum_{t \in T} \mathbf{1}[\text{execute}(y, t) = \text{pass}] - \lambda \cdot |y|_{\text{norm}} \quad (5)$$

where T is the set of test cases, $|y|_{\text{norm}}$ is the normalized code length (penalizing unnecessary verbosity), and $\lambda = 0.01$ is a length regularizer. The RLCE reward is dense and objective, enabling stable policy gradient optimization without reward model approximation errors.

4 Evaluation

4.1 Code Generation Benchmarks

Table 3: Code Generation Benchmark Results

Model	HumanEval	MBPP+	HumanEval+	LiveCodeBench
Zen4-Coder (32B)	95.2%	89.3%	93.1%	88.7%
Prior generation (22B)	89.4%	82.1%	87.6%	81.3%
Comparable 34B baseline	91.8%	84.7%	89.3%	84.2%
Comparable 70B baseline	93.6%	87.4%	91.7%	86.9%

4.2 Software Engineering Benchmarks

Table 4: Software Engineering Benchmark Results

Model	SWE-bench	SWE-bench Verified	RepoBench
Zen4-Coder (32B)	58.4%	61.2%	0.891
Prior generation (22B)	48.3%	51.7%	0.847
Comparable 34B baseline	52.1%	55.4%	0.862
Comparable 70B baseline	55.8%	59.3%	0.874

4.3 Debugging Capability

We evaluate autonomous debugging on a proprietary benchmark of 500 real-world bug reports sampled from public issue trackers, each paired with a ground-truth patch and full repository context.

Table 5: Autonomous Debugging Benchmark

Model	Pass@1	Pass@4	Pass@8	Avg Iterations
Zen4-Coder (32B)	54.2%	71.6%	78.4%	3.2
Prior generation (22B)	41.8%	59.3%	67.1%	4.7
Non-agentic baseline	31.4%	48.7%	56.2%	N/A

4.4 Test Generation Quality

Table 6: Test Generation Benchmark Results

Model	Branch Coverage	Bug Detection Rate	Test Validity
Zen4-Coder (32B)	87.3%	72.1%	96.4%
Prior generation (22B)	79.1%	63.4%	93.2%
Baseline 30B model	82.6%	67.8%	94.7%

4.5 Multi-Language Performance

Table 7: HumanEval Performance by Programming Language

Language	Zen4-Coder	Prior Gen
Python	95.2%	89.4%
TypeScript / JavaScript	94.1%	87.8%
Go	92.7%	85.3%
Rust	91.4%	83.1%
C / C++	90.8%	82.7%
Java	93.6%	86.9%
Kotlin	91.2%	84.4%
Swift	89.7%	81.6%

5 Multi-Repository Understanding

5.1 Dependency Graph Reasoning

A distinctive capability of Zen4-Coder is the ability to reason across repository boundaries. We evaluate this on the Multi-Repo Dependency Benchmark (MRDB), which contains 200 tasks requiring changes that must be coordinated across 2–5 interdependent repositories.

Table 8: Multi-Repository Benchmark Results (MRDB)

Model	Coordination Accuracy	Breaking Change Detection
Zen4-Coder (32B)	71.4%	84.3%
Prior generation (22B)	52.8%	69.7%
Single-repo baseline	34.1%	51.2%

5.2 API Contract Verification

Zen4-Coder can verify that a proposed change does not violate the API contracts of downstream consumers. This requires understanding call sites across repositories, type signatures, and the implicit behavioral contracts encoded in test suites.

In a held-out evaluation of 150 API-breaking changes, Zen4-Coder correctly identified 127 (84.7%) as breaking and proposed backward-compatible alternatives for 109 (85.8% of detected breaks), compared to 68.3% detection and 71.4% remediation for the prior generation.

6 Agentic Capabilities

6.1 Tool Use

Zen4-Coder is designed for agentic deployment with native support for the following tool categories:

- **Shell:** Execute commands, run test suites, invoke build systems.
- **File System:** Read, write, and navigate large code repositories.
- **Search:** Semantic and syntactic code search across indexed repositories.

- **Language Server:** Query LSP-compatible language servers for type information, references, and diagnostics.
- **Git:** Read commit history, blame annotations, and branch diffs.
- **CI/CD:** Query build and test results from CI pipeline APIs.

6.2 Long-Horizon Planning

On the Agentic Coding Evaluation (ACE) benchmark, which measures performance on tasks requiring 10+ sequential steps, Zen4-Coder achieves 63.7% task completion, compared to 48.2% for the prior generation and 41.4% for non-agentic 32B baselines evaluated with ReAct-style prompting.

7 Safety and Reliability

7.1 Code Safety

Zen4-Coder includes training-time and inference-time mitigations for code safety risks:

- **Malicious code refusal:** Zen4-Coder refuses requests to generate code with obvious malicious intent (e.g., credential theft, network exploitation), with a false positive rate of $< 0.3\%$ on benign security research tasks.
- **Secret detection:** When generating code that handles credentials or API keys, Zen4-Coder defaults to placeholder patterns and warns against hardcoding secrets.
- **Dependency safety:** Zen4-Coder prefers pinned, widely-used dependencies and flags packages with known CVEs.

7.2 Hallucination Rates

A persistent failure mode in code generation is hallucinating API signatures, function names, or library behaviors. We evaluate hallucination rates on a benchmark of 1,000 API usage tasks covering 50 popular libraries.

Table 9: API Hallucination Rates

Model	Hallucination Rate	Syntactically Valid
Zen4-Coder (32B)	3.2%	98.7%
Prior generation (22B)	6.8%	97.1%
32B baseline	5.1%	97.8%

8 Deployment

8.1 Inference Requirements

Table 10: Inference Resource Requirements

Configuration	Hardware	Throughput
FP8 (recommended)	$2 \times$ H100 80GB	480 tok/s
BF16	$4 \times$ H100 80GB	380 tok/s
INT4 (edge deployment)	$1 \times$ H100 80GB	650 tok/s

8.2 API Compatibility

Zen4-Coder exposes an OpenAI-compatible API endpoint, supporting both standard completion and function/tool-calling interfaces. Integration with major IDE plugins (VS Code, JetBrains, Neovim) is available through the Zen LM SDK.

9 Related Work

Code intelligence has advanced rapidly through several generations of transformer-based models [1, 2, 3, 4]. The transition from function-level to repository-level understanding has been driven by longer context windows and retrieval-augmented generation [5, 6]. Agentic approaches that combine code generation with execution feedback [7, 8] represent the current frontier, which Zen4-Coder advances through the unified MoE architecture and RLCE training objective.

10 Conclusion

Zen4-Coder establishes a new capability tier for code intelligence at the 32B parameter scale, with multi-repository understanding, autonomous debugging, and comprehensive test generation distinguishing it from prior generation models. The MoE architecture enables efficient deployment with 8B active parameters at inference time while retaining the representational capacity of a 32B model. Benchmark results across HumanEval, SWE-bench, RepoBench, and MBPP+ confirm state-of-the-art performance among models in its parameter class.

Future work will focus on deeper integration with distributed version control workflows, formal verification of generated code properties, and tighter coupling with continuous integration systems for real-time feedback during development.

References

- [1] Chen, M. et al. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374.
- [2] Li, Y. et al. (2022). Competition-level code generation with AlphaCode. Science, 378(6624).
- [3] Li, R. et al. (2023). StarCoder: May the source be with you. arXiv:2305.06161.
- [4] Roziere, B. et al. (2023). Code Llama: Open Foundation Models for Code. arXiv:2308.12950.
- [5] Zhang, F. et al. (2023). RepoCoder: Repository-Level Code Completion Through Iterative Retrieval. arXiv:2303.12570.
- [6] Jimenez, C. et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR 2024.
- [7] Xia, C. et al. (2024). Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489.
- [8] Yang, J. et al. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793.