



From Code Foundation Models to Agents and Applications: A Comprehensive Survey and Practical Guide to Code Intelligence

BUAA-SKLCCSE, Alibaba, ByteDance, M-A-P, BJTU, OPPO, HKUST (GZ), BUPT, TeleAI, Shanghai AI Lab, Manchester, StepFun, UoS, SCU, CASIA, NJU, Kuaishou, HIT, Huawei Cloud, Tencent, Monash/CSIRO, NTU, ZJU, BIT, Ubiquant, NUS, HNU, PKU, CSU

Abstract

Large language models (LLMs) have fundamentally transformed automated software development by enabling direct translation of natural language descriptions into functional code, driving commercial adoption through tools like Github Copilot (Microsoft), Cursor (Anysphere), Trae (ByteDance), and Claude Code (Anthropic). While the field has evolved dramatically from rule-based systems to Transformer-based architectures, achieving performance improvements from single-digit to over 95% success rates on benchmarks like HumanEval. In this work, we provide a comprehensive synthesis and practical guide (a series of analytic and probing experiments) about code LLMs, systematically examining the complete model life cycle from data curation to post-training through advanced prompting paradigms, code pre-training, supervised fine-tuning, reinforcement learning, and autonomous coding agents. We analyze the code capability of the general LLMs (GPT-4, Claude, LLaMA) and code-specialized LLMs (StarCoder, Code LLaMA, DeepSeek-Coder, and QwenCoder), critically examining the techniques, design decisions, and trade-offs. Further, we articulate the research-practice gap between academic research (e.g., benchmarks and tasks) and real-world deployment (e.g., software-related code tasks), including code correctness, security, contextual awareness of large codebases, and integration with development workflows, and map promising research directions to practical needs. Last, we conduct a series of experiments to provide a comprehensive analysis of code pre-training, supervised fine-tuning, and reinforcement learning, covering scaling law, framework selection, hyperparameter sensitivity, model architectures, and dataset comparisons.

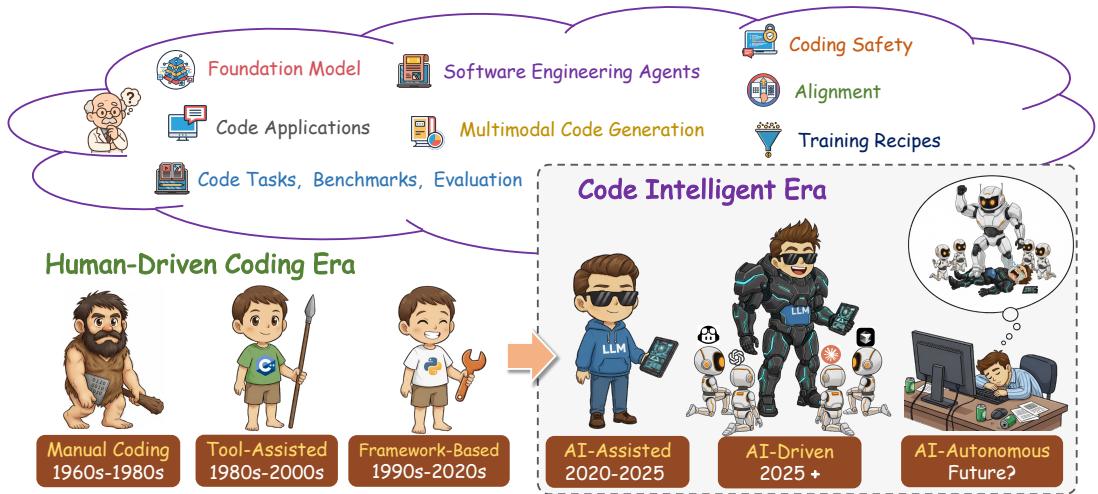


Figure 1. Evolution of programming development and research landscapes in AI-powered code generation. The upper section highlights the key research areas covered in this work. The timeline below illustrates the six-stage evolution from the human-driven coding era to the emerging code intelligence era.

Contents

1	Introduction	8
2	Code Foundation Models	10
2.1	General Large Language Models	10
2.1.1	The Rise of General LLMs	10
2.1.2	Model Architectures	12
2.1.3	Multimodality	15
2.1.4	Limitations of General LLMs	15
2.2	Code Large Language Models	16
2.2.1	Closed-source Code Large Language Models	16
2.2.2	Open-source Code Large Language Models	20
2.2.3	Evolution of Open-Source Code Large Language Models	21
2.2.4	Model Pre-Training Tasks	30
2.2.5	Model Training Stages	33
2.3	Open-source Code Pre-training Data	36
2.3.1	The Github Datasets	36
2.3.2	StarCoderData	37
2.3.3	Others	37
2.4	Future Trends	37
3	Code Tasks, Benchmarks, and Evaluation	38
3.1	Evaluation Metrics	40
3.1.1	Extensions Based on Traditional Metrics	40
3.1.2	LLM-as-a-Judge Paradigm	40
3.1.3	Execution-Based Metrics	42
3.1.4	Multi-Agent & Advanced Reasoning Framework	43
3.1.5	Statistical & Consistency Analysis Metrics	43
3.1.6	Other Unique Paradigms	43
3.2	Statement, Function, and Class-Level Tasks and Benchmarks	44
3.2.1	Code Completion and Code FIM	44
3.2.2	Code Generation	45
3.2.3	Code Edit and Bug Fix	48

3.2.4	Code Efficiency	49
3.2.5	Code Preference	50
3.2.6	Code Reasoning and Question Answering	50
3.2.7	Code Translation	51
3.2.8	Test-Case Generation	53
3.3	Repository-Level Tasks	54
3.3.1	Code Generation and Completion	54
3.3.2	Domain-Specific and Complex Code Generation	55
3.3.3	Code Editing, Refactoring, and Agent Collaboration	56
3.3.4	Commit Message Generation	57
3.3.5	Software Engineering Tasks	58
3.3.6	Comprehensive Software Development	59
3.3.7	Repository-Level and Long Context Understanding	60
3.4	Agentic Systems	61
3.4.1	Agent Tool Use	61
3.4.2	Deep Research Benchmarks	61
3.4.3	Web Search Benchmarks	61
3.4.4	Benchmarking Agents for Graphical User Interfaces	62
3.4.5	Terminal Use	63
4	Alignment	63
4.1	Supervised Fine-tuning (SFT)	63
4.1.1	Single-Turn Supervised Fine-tuning	64
4.1.2	Multi-Turn Supervised Fine-tuning	65
4.1.3	SFT for Repository Tasks	65
4.1.4	Reasoning-based Methods	67
4.1.5	Training Strategies	68
4.1.6	Challenges	68
4.2	Cold-start / Distill Reasoning SFT data for Code LLMs	69
4.2.1	Data Sourcing	69
4.2.2	Data Cleaning and Decontamination	70
4.2.3	Question Filtering and Quality/Difficulty Assessment	71
4.2.4	Reasoning Chain Generation	71

4.2.5	Solution Filtering and Refinement	72
4.2.6	Final Dataset Construction	73
4.3	Multilingual Code Understanding and Generation	74
4.3.1	Multilingual Code LLMs	74
4.3.2	Multilingual Code Evaluation	76
4.4	Multimodal Code Understanding and Generation	77
4.4.1	Vision-Language Foundation Models for Code	77
4.4.2	Core Challenges and Technical Positioning	79
4.4.3	Frontend Interface Generation	79
4.4.4	Web-Embodied Intelligence	81
4.4.5	Software Engineering Artifact Generation	82
4.4.6	Technical Trends and Future Outlook	84
4.5	Task-based Overview of Reinforcement Learning in Code Intelligence	85
4.5.1	Reinforcement Learning (RL) Algorithms	85
4.5.2	RL for Code Generation	88
4.5.3	RL for Code Understanding	90
4.5.4	RL for Software Engineering	91
4.5.5	RL for Code Security	92
4.5.6	Code Testing	93
4.6	Applying Reinforcement Learning with Verifiable Rewards	93
4.6.1	RLVR-Suitable Datasets for Code Tasks	94
4.6.2	Representative RLVR-Trained Open-Source Code LLMs	97
4.6.3	Reward Shaping in Code Post-training	100
4.6.4	Quality-Oriented Rewards	101
5	Software Engineering Agents	103
5.1	SWE Agents Operate Across Lifecycles in Software Engineering	103
5.1.1	Requirements Engineering	103
5.1.2	Software Development	106
5.1.3	Software Testing	123
5.1.4	Software Maintenance	125
5.1.5	End-to-End Software Agents	131
5.2	General Code Agents in Software Engineering	131

5.3	Training Techniques for SWE Agents	133
5.3.1	Fine-tuning SWE Agents	133
5.3.2	Reinforcement Learning for SWE Agents	135
5.4	Future Trends: Towards Integrated and Autonomous Software Engineering Ecosystems	139
6	Code for Generalist Agents	142
6.1	Code as Interaction Protocols	142
6.1.1	Tool Use	142
6.1.2	Model Context Protocol	143
6.1.3	Multi-Agent Coordination	144
6.2	Code as Agentic Capabilities	144
6.2.1	Thinking in Code	144
6.2.2	Acting in Code	145
6.2.3	Memory With Code	146
6.3	Code as Environment Interfaces	147
6.3.1	Code as Simulation Gym	147
6.3.2	Computer-Use Agents	148
7	Safety of Code LLMs	150
7.1	Safety Pre-training for Code LLMs	151
7.1.1	Data Provenance, Security, and License Compliance	152
7.1.2	Training-data Auditing and Cleaning	153
7.1.3	The Regulatory and Standards in Data Security	154
7.1.4	Robustness Against Adversarial Code Transformations	154
7.1.5	Privacy Risk Assessment and Mitigation in Pre-training Data	155
7.1.6	Bias Assessment and Mitigation	156
7.2	Safety Post-training for Code LLMs	157
7.2.1	Pre-training Limitations and the Necessity of Post-training Alignment	157
7.2.2	Data as the Cornerstone: Constructing Safety-related Training Datasets	158
7.2.3	Safety Supervised Fine-Tuning for Code LLMs	159
7.2.4	Advanced Preference Optimization for Localized Flaws	160
7.2.5	Coding Safety Alignment via Reinforcement Learning	160
7.3	Red-teaming Techniques for Code LLMs	162

7.3.1	Prompt-Level Manipulation: Subverting Input-Output Behavior	162
7.3.2	Semantic and Contextual Manipulation: Exploiting the Interpretation Layer	163
7.3.3	Agentic Workflow: Subversion of Agent Systems and Tool Use	163
7.4	Mitigation Strategies for Coding and Behavioral Risks in AI Agent Systems	165
7.4.1	Foundations in Secure Execution Environments	165
7.4.2	Proactive Defense and Pre-Execution Validation	166
7.4.3	Runtime Oversight and Intent Grounding	167
8	Training Recipes for Code Large Language Model	167
8.1	Distributed Training Framework Introduction	168
8.2	Pre-Training Guidelines	169
8.3	Supervised Finetune Training Guidelines	172
8.4	Reinforcement Learning Training Guidelines	177
9	Code Large Language Model for Applications	183
9.1	IDE-integrated Development Assistants	184
9.2	Cloud-native Coding Platforms	187
9.3	Terminal-based Autonomous Agents	188
9.4	Code Repair and Verification Applications	190
9.5	Pull Request Review and Quality Assurance	191
10	Contributions and Acknowledgements	193

1. Introduction

The emergence of large language models (LLMs) [66, 67, 192, 423, 434, 749, 752, 754, 755] has catalyzed a paradigm shift in automated software development, fundamentally reconceptualizing the relationship between human intent and executable code [1307]. Modern LLMs have achieved remarkable capabilities across a wide range of code-related tasks, including code completion [98], translation [1158], repair [618, 970], and generation [139, 161]. These LLMs effectively distill years of accumulated programming expertise into accessible, instruction-following tools that can be deployed by developers at any skill level using code from sources such as GitHub, Stack Overflow and other code-related websites. Among LLM-related tasks, code generation stands as one of the most transformative, enabling the direct translation of natural language descriptions into functional source code, thereby dissolving traditional barriers between domain knowledge and technical implementation. This capability has transcended academic curiosity to become a commercial reality through a series of commercial and open-source tools, including (1) GitHub Copilot (Microsoft) [321], which provides intelligent code completion within development environments; (2) Cursor (AnySphere) [68], an AI-first code editor that enables conversational programming; (3) CodeGeeX (Zhipu AI) [24], which offers multilingual code generation; (4) CodeWhisperer (Amazon) [50], which integrates seamlessly with AWS services; (5) Claude Code (Anthropic) [194]/Gemini CLI (Google) [335], which are both command-line tools that allow developers to delegate coding tasks directly to Claude or Gemini [67, 955] from their terminal for agentic coding workflows. These applications reshape software development workflows, challenge conventional assumptions about programming productivity, and redefine the boundary between human creativity and machine assistance.

In [Figure 1](#), the evolutionary trajectory of code generation reveals a compelling narrative of technological maturation and paradigm shifts. Early approaches, constrained by heuristic rules and probabilistic grammar-based frameworks [42, 203, 450], were inherently brittle—optimized for narrow domains and resistant to generalization across the vast diversity of programming contexts. The advent of transformer-based architectures [291, 360] represented not merely an incremental improvement but a fundamental reconceptualization of the problem space, leveraging attention mechanisms [997] and scale to capture the intricate relationships between natural language intent and code structure. More remarkably, these models exhibit emergent instruction-following capabilities that were neither explicitly programmed nor directly optimized for, suggesting that the capacity to translate high-level goals into executable implementations may be a natural consequence of learning rich representations at scale. This democratization [138, 863] of coding, enabling non-experts to generate sophisticated programs through natural language, carries profound implications for workforce development, innovation pace, and the very essence of computational literacy in the 21st century [223, 903].

The contemporary landscape of code LLMs reveals a strategic bifurcation between generalist and specialist approaches, each with distinct advantages and trade-offs. General-purpose models like the GPT [746, 749, 752], Claude [66, 67, 192], and LLaMA [689, 690, 979, 980] series offer remarkable breadth, leveraging vast corpora of natural language alongside code to develop a nuanced understanding of context, intent, and domain knowledge. Conversely, specialized code LLMs such as StarCoder [562], Code LLaMA [858], DeepSeek-Coder [232], CodeGemma [1295], and QwenCoder [434, 824] achieve superior performance on code-specific benchmarks through focused pre-training on programming-centric data and task-specific architectural optimizations. Dramatic performance improvements from single digits to 95%+ success rates on standardized benchmarks like HumanEval [161] reflect both algorithmic innovations and deeper insights. While code is highly formalized, it shares core characteristics with natural language, particularly

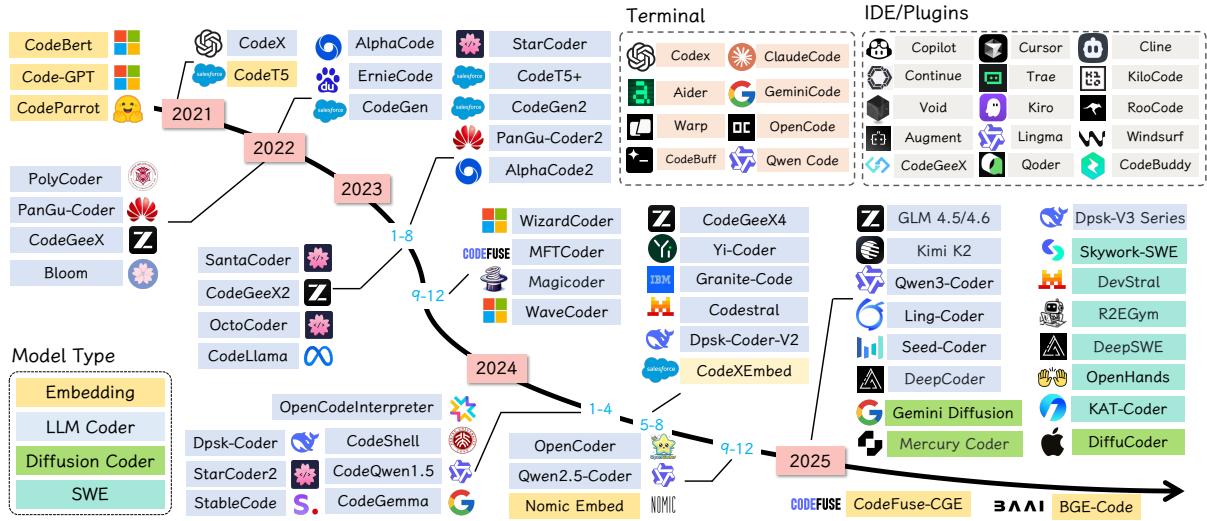


Figure 2. Overview of the evolution of code large language models (Code-LLMs) and related ecosystems from 2021 to 2025. The landscape begins with early models and quickly expands into a diverse set of LLM coders across 2022–2024. From 2025 onward, research focus shifts toward reinforcement learning (RL)-based training, software engineering (SWE) agents, and novel architectures such as diffusion-based code models. In parallel, a rich ecosystem of terminal tools, IDE integrations, and plugins emerges, highlighting the transition from pure modeling to practical developer-oriented applications.

in compositional semantics and contextual dependencies.

Despite vigorous research activity and rapid commercial adoption, a critical gap persists between the breadth of innovation and the depth of systematic analysis in the literature. Existing surveys have largely adopted panoramic approaches, surveying broad categories of code-related tasks, or focusing on earlier generations of models, leaving contemporary advances inadequately synthesized. Crucially underexplored are the sophisticated data curation strategies of state-of-the-art systems, which balance quantity with quality instruction tuning methods to align model behavior with developer intent. Such alignment techniques involve incorporating human feedback to refine outputs, advanced prompting paradigms including chain-of-thought reasoning and few-shot learning, the emergence of autonomous coding agents capable of multi-step problem decomposition, retrieval-augmented generation (RAG) approaches that ground outputs in authoritative references, and novel evaluation frameworks that move beyond simple binary correctness to assess code quality, efficiency, and maintainability.

In Figure 2, recent LLMs like Kimi-K2 [957], GLM-4.5/4.6 [25, 1248], Qwen3Coder [824], Kimi-Dev [1204], Claude [67], Deepseek-V3.2-Exp [234], and GPT-5 [752] embody these innovations, yet their contributions remain scattered across disparate publications without cohesive integration. Table 1 compares various surveys related to code intelligence or LLM, evaluating them across eight dimensions: domain, whether focus on Code, LLM usage, pretraining, supervised fine-tuning (SFT), reinforcement Learning (RL), Training Recipes for code LLM, and applications. These surveys cover diverse areas, including general code generation, software engineering using GenAI, code summarization, and LLM-based agents. Most surveys focus on code and applications, but vary significantly in their coverage of technical aspects. While some address LLMs and pretraining, very few cover reinforcement learning methods. This survey offers a comprehensive and contemporary synthesis of research literature on large language models (LLMs) for code intelligence, providing a systematic examination of the entire model life

cycle. It explores critical phases—from initial data curation and instruction tuning to advanced code applications and the development of autonomous coding agents.

To provide a comprehensive and practical study from code foundation models to agents and applications, we present a detail guide that bridges theoretical foundations with implementations in modern code generation systems, as shown in [Table 1](#). Our work makes several key contributions: (1) We provide a unified taxonomy of contemporary code LLMs, tracing their evolution from early transformer-based models to the latest generation of instruction-tuned systems with emergent reasoning capabilities; (2) We systematically analyze the complete technical pipeline from data curation and preprocessing strategies, through pretraining objectives and architectural innovations, to advanced fine-tuning methodologies including supervised instruction tuning and reinforcement learning; (3) We examine cutting-edge paradigms that define state-of-the-art performance, including prompting techniques (e.g., chain-of-thought [1174]), retrieval-augmented generation approaches, and autonomous coding agents capable of complex multi-step problem solving; (4) We critically evaluate the landscape of benchmarks and evaluation methodologies, discussing their strengths, limitations, and the ongoing challenge of assessing not merely functional correctness but code quality, maintainability, and efficiency; (5) We synthesize insights from recent breakthrough models (e.g., GPT-5, Claude 4.5 among others) to identify emerging trends and open challenges that will shape the next generation of code generation systems. This survey aims to serve as both a comprehensive reference for researchers entering the field and a strategic roadmap for practitioners seeking to leverage these technologies in production environments. (6) We perform extensive experiments to comprehensively examine code pre-training, supervised fine-tuning, and reinforcement learning across multiple dimensions including scaling laws, frameworks, hyperparameters, architectures, and datasets.

2. Code Foundation Models

2.1. General Large Language Models

2.1.1. *The Rise of General LLMs*

The advent of LLMs built on the transformer architecture [996] marked a decisive shift in AI. Before transformers, progress was fragmented across specialized systems, including sequence-to-sequence models for translation [84, 925, 1093], handcrafted pipelines for dialogue [1074, 1144, 1220], and domain-specific engines for program synthesis [48, 358, 797]. Transformer-based pretraining and knowledge transfer unified these strands into a single, scalable framework that could be adapted across tasks and modalities [122, 247, 828]. Scaling laws show predictable gains with more model parameters, data, and compute [483], while reports of *emergent* abilities, defined as capabilities that appear only at larger scales, suggest LLMs generalize beyond their training distribution [1062]. Yet recent work argues some emergence may stem from metric choice rather than true leaps in capability, offering a more nuanced view of the benefits of scale [864]. Two classes of abilities are especially salient: coding and agentic behavior. First, general-purpose LLMs revealed surprising coding competence, catalyzing the development of models explicitly trained on code. OpenAI’s Codex demonstrated functional code generation from natural-language prompts and introduced standardized evaluation like HumanEval [161]. LLMs have achieved outstanding performance on HumanEval, as illustrated in [Figure 3](#). In parallel, DeepMind’s AlphaCode [577] showed that large-scale sampling and filtering could reach competitive-programming proficiency at roughly the median human level under simulated Codeforces settings. These results established that linguistic modeling and code synthesis share exploitable structure, making LLMs immediately useful for tasks from boilerplate generation to

Table 1. Comparison between our study and existing works.

Survey	Scope	Focus on Code	LLM	Pretrain	SFT	RL	Application	Training Recipes
A Survey on Language Models for Code [1292]	All	✓	✓	✓	✓	✗	✓	✗
Deep Learning for Code Generation: A Survey [1284]	Deep Learning, Code Generation, Automated SE	✓	✗	✗	✗	✗	✓	✓
Code to Think, Think to Code [1172]	Code reasoning, planning, debugging	✓	✗	✗	✗	✗	✓	✗
A Survey on LLMs for Code Generation [457]	Code Generation, Data Process	✓	✓	✓	✗	✗	✓	✗
A Survey of ML for Big Code and Naturalness [44]	Code patterns, model design	✗	✗	✗	✗	✗	✓	✗
A Survey on Code Generation with LLM-based Agents [1032]	Code Gen, LLM Agents, Multi-agent Systems	✓	✓	✓	✓	✓	✓	✗
A Survey of Automatic Source Code Summarization [622]	Code Summarization, Program Analysis, NMT	✓	✗	✗	✗	✗	✓	✓
A Review of Automatic Source Code Summarization [288]	Code Summarization, Program Analysis, NMT	✓	✗	✗	✗	✗	✓	✗
Survey on NN-based Automatic Source Code Summarization [307]	Intelligent SE, Code Summarization, Deep Learning	✓	✗	✗	✗	✗	✓	✗
A Survey of Large Language Models [1301]	General LLM	✗	✓	✓	✓	✗	✓	✗
Source code data augmentation for deep learning: A survey [1338]	Code Data Augmentation, Program Analysis, Deep Learning	✓	✓	✗	✓	✗	✓	✗
A Survey of Vibe Coding with LLMs [317]	Vibe Coding	✗	✓	✓	✓	✗	✓	✗
Ours	All	✓	✓	✓	✓	✓	✓	✓

algorithmic problem solving [82, 397, 469, 562, 858].

Second, when paired with external tools, memory, and closed-loop reasoning, LLMs begin to look like decision-making agents rather than static predictors. Methods such as ReAct [1209] interleave reasoning traces with environment actions to plan, gather information, and correct course [1209]. Complementary approaches such as Toolformer [867] show that models can learn *when* and *how* to call APIs in a self-supervised way, improving reliability on tasks that benefit from calculators, search, or retrieval [242, 625, 719, 866, 894, 1208]. Among them, the most representative software engineering (SWE) agents have made remarkable progress, as shown in Figure 4.

Taken together, these developments mark a clean break from narrow, task-specific systems to general coding system, which provides a unified substrate for language, programming, and tool-mediated reasoning. At the same time, their breadth exposes limits in accuracy, security, and system-level reliability in professional software settings [926, 927, 929], which in turn motivate the specialized coding models and agents represented in the rest of this work.

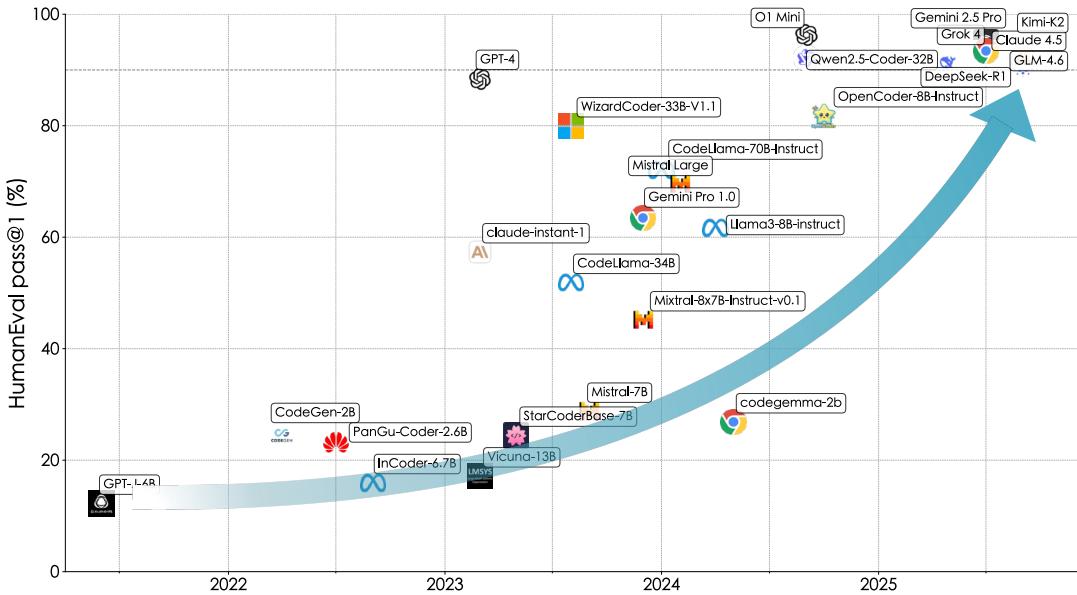


Figure 3. The timeline of code language models’ progress on HumanEval. The dashed line represents a score of 90. The vertical axis does not indicate actual scores but signifies that model scores exceed 90 points.

2.1.2. Model Architectures

Alongside tremendous growth in scale and data [398, 657], innovations in model architecture have been a central pillar of the rapid progress of LLMs. This architectural evolution is primarily defined by a shift away from dense models, where every parameter is engaged in every computation, and toward sparser, more specialized designs that optimize the trade-offs between efficiency, scalability, and performance.

Dense Models The transformer model [996] remains the foundation of modern LLMs, leveraging dense architectures where every parameter is involved in processing each token. This design, built on stacks of attention and feed-forward layers, has enabled remarkable progress in capturing long-range dependencies and driving breakthroughs across NLP tasks. Building on this, models like LLaMA [344, 979, 980] and its successors have shown that high-quality open models can rival proprietary systems, scaling from 7B to 70B parameters. The GLM series [272, 328] extended dense architectures into bilingual and multilingual domains, while the Qwen family [85, 825, 962, 1162] emphasized strong performance in both understanding and generation with scalable dense models. Meanwhile, Mistral [452] highlighted how careful engineering, such as grouped query attention (GQA), can deliver competitive results with fewer parameters. Collectively, these dense models illustrate a consistent trend: while computationally demanding, they continue to evolve toward greater efficiency and versatility, cementing their central role in modern NLP research and applications.

Mixture-of-Experts (MoE) MoE expands model capacity through conditional computation without proportionally increasing activated compute: each token is routed to only a small number of experts, typically the top- k experts, for forward computation, thereby trading sparse activation for higher effective capacity [267, 286, 530]. In the open-source community, the Mixtral series made two-expert routing a de facto engineering standard: 8x7B demonstrated that

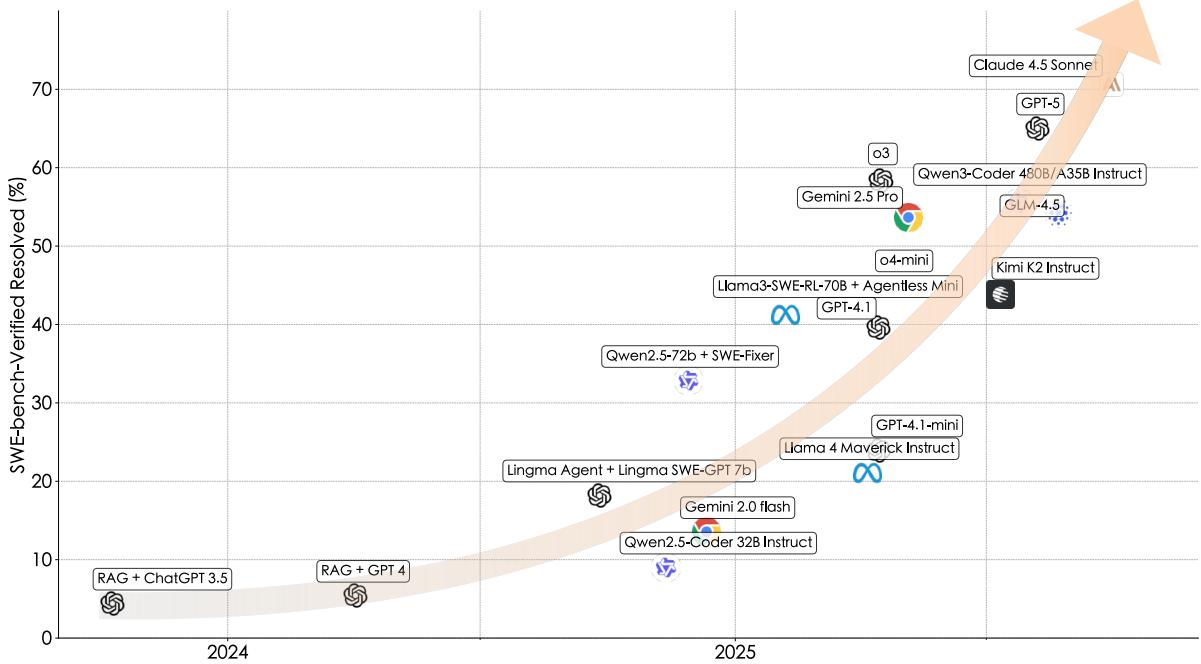


Figure 4. The timeline of code language models’ progress on SWE-bench-Verified. All models without scaffold annotations uniformly use mini-SWE-agent.

activating fewer parameters can outperform larger dense baselines, and the subsequent 8 \times 22B further pushed the limits of capability and throughput in open-source models [453]. The Qwen series introduced MoE variants across its 1.5/2.5/3 versions [825, 962, 1163]. DeepSeek [231] systematized efficient co-design of sparse experts and Multi-head Latent Attention (MLA) in its V2/V3 series. V2 has 236B total parameters with about 21B activated, while V3 has 671B total parameters with about 37B activated. These models offered replicable open paradigms balancing cost and stability [235, 238]. DeepSeek R1 further built on V3-Base with reinforcement learning to significantly enhance chain-of-thought reasoning [237]. GLM-4.5 employed large-scale MoE, integrating hybrid reasoning modes into a unified model for coding, reasoning, and agent applications [1248]. In addition, the entire LLaMA-4 series also adopts the MoE architecture [690]. Overall, MoE has become one of the mainstream architectures for optimizing the effective capacity ratio, and in practice it works synergistically with long-context handling, KV cache compression, and multi-token prediction, forming an efficient paradigm for large-scale production environments.

Recurrent Models Recurrent-style architectures revisit sequence modeling to cut memory and latency while preserving parallel training. RWKV [153, 785, 786] blends transformer-like parallelizable training with recurrent inference, activating a constant-size state at each step so that decoding scales linearly and can approach transformer quality at similar sizes. Retentive Networks (RetNet) [922] replace attention with a retention operator that supports fully parallel training and either recurrent or chunkwise-recurrent inference, yielding linear-time long-sequence processing with strong language-modeling results. Mamba [345] introduces selective state-space models whose parameters are input-dependent, enabling linear-time decoding and competitive performance on language while maintaining high throughput; a follow-up theoretical line frames transformers and SSMs under a shared state-space duality with efficient algorithms [222]. Closely related long-range operators such as Hyena [796] use implicitly

parameterized long convolutions with gating to match attention quality at subquadratic cost, pushing feasible context lengths far beyond standard attention regimes and complementing recurrent approaches in practice. Additionally, DeltaNet [1197] introduces a hardware-efficient way to parallelize linear transformers with the delta rule (a state update mechanism), which improves associative retrieval and enables scaling to standard language-modeling settings. Gated DeltaNet [1196] combines gating with the delta update to better control memory and consistently surpasses Mamba-2 and DeltaNet on long-context and retrieval benchmarks.

Diffusion-based Models Diffusion-based language models replace left-to-right decoding with iterative denoising steps that refine a noisy sequence into fluent text, enabling strong global control over attributes and structure. Foundational work on discrete diffusion formalized corruption/denoising processes directly in token space (D3PM [81]), establishing principled transition kernels for categorical data such as text. Building on this, Diffusion-LM [568] operates in a continuous embedding space and leverages gradient-based guidance for fine-grained controllability while remaining non-autoregressive. For conditional generation, DiffuSeq [333] adapts diffusion to sequence-to-sequence tasks and reports performance that is competitive with strong autoregressive baselines. To better align diffusion with token vocabularies and practical decoding, SSD-LM [377] performs simplex-based diffusion over the discrete vocabulary and generates text in blocks, enabling modular classifier guidance that matches or surpasses GPT-style models. AR-Diffusion [1090] introduces an explicit autoregressive ordering within diffusion to reconcile sequential dependencies with iterative refinement. Lately, several larger efforts have pushed diffusion LMs beyond small-scale prototypes: LLaDA [729] trains diffusion models for language from scratch via a masking schedule and reverse denoising with a vanilla transformer, reporting competitiveness with similarly sized autoregressive baselines. On the commercial side, Mercury Coder [505] frames coding as parallel multi-token denoising and markets substantial speed/throughput gains relative to autoregressive (AR) models. Gemini Diffusion [230] is another research model exploring diffusion for text generation, signaling continued interest in non-autoregressive decoding at production scale. While diffusion LMs offer controllability and parallelizable training objectives, they typically require many sampling steps, motivating research on faster samplers and hybrid AR-diffusion decoders.

Hybrid Architectures Hybrid architectures interleave complementary sequence operators, typically combining transformer attention with state-space or recurrent blocks, often in addition to MoE feed-forwards to trade off quality, context length, and throughput in one stack. Jamba [589] is a canonical example: it interleaves transformer and Mamba layers with MoE, achieving high throughput at long contexts while retaining strong performance. In the Qwen line, Qwen3-Next [963] adopts a hybrid attention design that mixes gated DeltaNet-style linear operators with gated attention and sparse-activation MoE, targeting 256K+ (more than 256K tokens) contexts with low active parameters per token. The DeepSeek family also fuses multiple ideas: V3 introduced MLA with DeepSeek-MoE for efficient training/inference [238], and the recent V3.2-Exp [234] adds an experimental DeepSeek Sparse Attention (DSA) mechanism as an intermediate step toward its next-generation hybrid architecture, emphasizing longer-context efficiency across diverse hardware.

In summary, model architecture has diversified from a one-size-fits-all dense transformer to a toolkit of sparsity, recurrence/state-space, diffusion, hybrids, and efficient attention. These choices let practitioners trade off capacity, latency, and context length, providing the capabilities that underpin both general LLMs and the specialized coding systems discussed later.

2.1.3. Multimodality

Code LLMs need to process visual information like diagrams, screenshots, and UI elements to understand and generate code in real-world scenarios [285, 495, 558, 1079, 1168]. These capabilities form the foundation for code-oriented workflows. Modalities such as audio or speech are outside the present scope.

2.1.4. Limitations of General LLMs

The progress highlights the breadth and versatility of general-purpose LLMs, spanning dense and sparse architectures, recurrent and hybrid designs, as well as emerging multimodal capabilities. These developments underscore how far the field has advanced from narrow task-specific systems toward unified substrates for language, coding, and perception–action reasoning. Yet, this very breadth also exposes their limitations: general LLMs, while impressive in scope, often lack the depth, robustness, and domain alignment required for professional software engineering. We therefore turn next to a closer examination of their key shortcomings.

Specialization and Accuracy Despite their breadth, general-purpose LLMs often lack the depth required for professional software engineering. They may produce functionally-looking code that superficially appears correct but fails to satisfy domain constraints such as subtle API contracts, security policies, and they struggle to maintain invariants across large systems. Evidence from repository-scale evaluations further indicates that real-world issue resolution remains challenging even for strong models and agentic toolchains [469].

Security and Reliability A growing body of empirical studies shows that *functionally correct* code from general LLMs can still be *insecure*. Large-scale evaluations involving more than one hundred models across eighty tasks report that about 45% of generations contain known vulnerabilities, with little improvement from newer or larger models. Smaller focused studies likewise find that ChatGPT and similar LLMs often emit code that is not robust to attacks [489], and recent outcome-driven benchmarks that evaluate both functionality and security confirm substantial rates of works-but-insecure solutions [787, 971].

Repository-Level Understanding Even with expanded context windows, general LLMs do not robustly exploit very long inputs: performance degrades when pertinent information lies in the *middle* of the context rather than near its ends [615], and repository-level benchmarks covering tasks such as multi-file completion, retrieval, and editing reveal persistent difficulties in cross-file dependency tracking and global reasoning.

Multimodal Friction General multimodal models provide useful perception for screenshots, documents, and diagrams, but fine-grained UI hierarchy and interaction semantics remain weak points. Recent analyses in GUI understanding note that existing systems often specialize in narrow sub-tasks rather than achieving holistic and consistent screen comprehension, which in turn limits stable perception-to-action transitions in real applications.

Agentic Constraints For tool-augmented settings, benchmarked agents still fail due to brittle long-horizon reasoning, decision-making, and instruction following. Systematic evaluations highlight sizeable gaps across interactive environments and domains [625], and new diagnostics document *tool hallucinations* such as wrong tool choice, incorrect timing, or fabricated tool outcomes. These studies further propose reliability alignment to mitigate such issues, underscoring that robust planning and faithful tool use remain open challenges for general LLMs [1136, 1290].

Overall, breadth without domain alignment leads to gaps in depth, reliability, and system-level coherence. Addressing these limitations motivates *coding-specialized* pretraining, data

curation, safety alignment, and evaluation, with models optimized to act as expert programmers rather than generalists.

2.2. Code Large Language Models

2.2.1. Closed-source Code Large Language Models

In Figure 5, closed-source code LLMs have evolved from basic generation to agentic systems with repository-level capabilities. The GPT series [752, 755, 758] from OpenAI and Claude [66, 67, 192] from Anthropic achieve state-of-the-art results on SWE-Bench through reasoning and RL on engineering tasks.

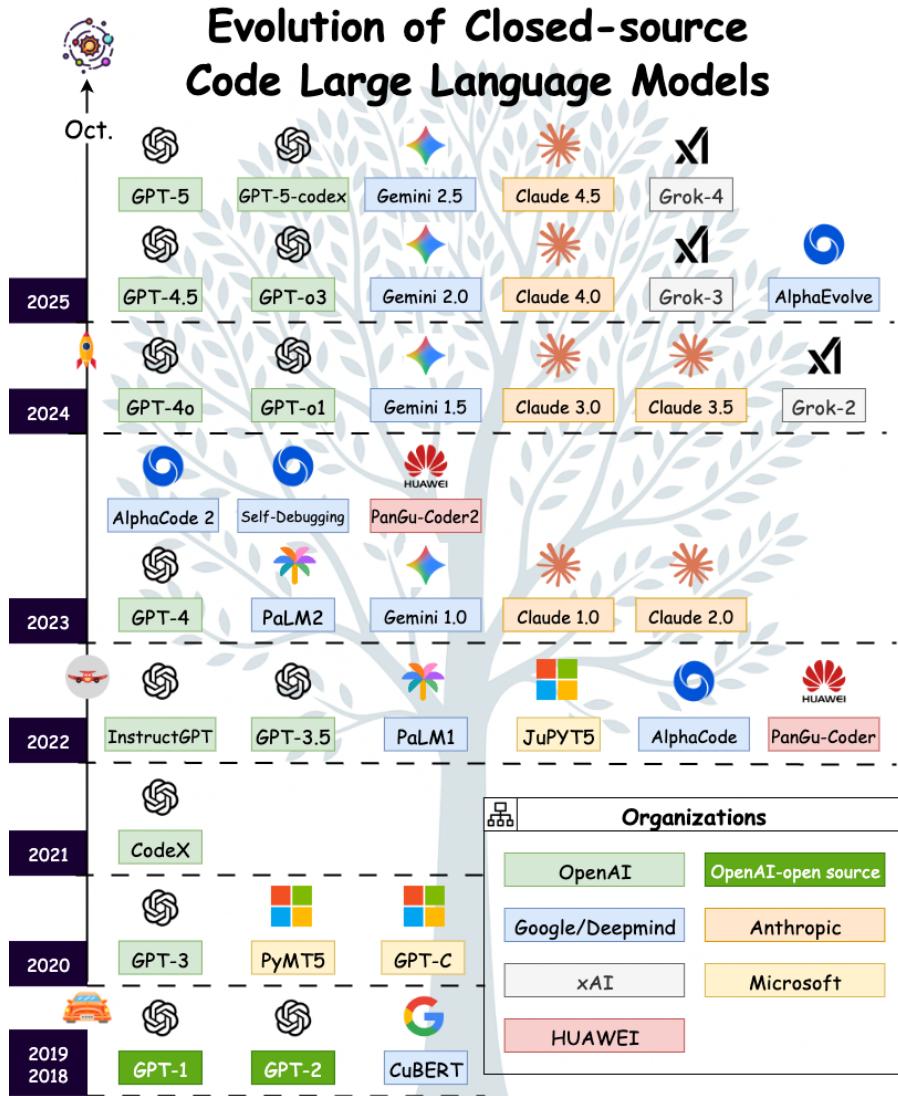


Figure 5. Evolution of closed-source large language models from 2018 to 2025. This figure depicts the chronological development of major proprietary LLMs released by leading research organizations, illustrating key milestones in the progression of model capabilities and architectures across systems such as GPT, Gemini, Claude, and Grok.

GPT Series The GPT series from OpenAI has strongly shaped code intelligence. Early open-weight GPT-1/2 validated generative pre-training. Proprietary successors—GPT-3, Codex, GPT-4, and the reasoning-focused *o*-series—expanded from in-context learning and code synthesis to multimodal use and repository-level repair. GPT-OSS[759] reintroduced open weights via mixture-of-experts. Most recently, GPT-5 and GPT-5-Codex set leading results on SWE-Bench and Aider Polyglot, pushing from passive generation toward agentic, feedback-driven software engineering. Overall, the family charts a path from general language modeling to systems optimized for end-to-end coding.

- **GPT-3** [124] scaled autoregressive pre-training on diverse web and curated text, and *in-context learning* showed models can adapt from a few examples without gradient updates. It delivered strong zero-/few-shot results across language, reasoning, and code tasks, cementing large-scale pre-training as a foundation for code synthesis and program understanding.
- **Codex** [161] continued GPT-3 training on large GitHub corpora across many languages under an autoregressive decoder. It performed well on code generation and completion benchmarks (e.g., HumanEval, APPS) and powered GitHub Copilot. Conditioned on natural language, Codex synthesized code, translated between languages, and generated docstrings—an early large-scale alignment of LLMs to programming.
- **InstructGPT** [769] aligned models with reinforcement learning from human feedback via supervised demonstrations, preference-based reward modeling, and PPO optimization. The resulting models were preferred by human raters, with fewer hallucinations and safer behavior; notably, a smaller aligned model surpassed a much larger base GPT-3 in preference evaluations and showed preliminary transfer to non-English and code instructions.
- **ChatGPT** [745] (GPT-3.5) built on InstructGPT with additional instruction tuning and RLHF, stabilizing multi-turn dialogue and adding safety and refusal behaviors. Despite undisclosed details, it is broadly viewed as an extension of the GPT-3 line. As the first widely deployed conversational LLM with robust coding ability, it generated, explained, and debugged code in IDE workflows, paving the way for GPT-4.
- **GPT-4** [749, 750, 753, 758] advanced reasoning and code synthesis over GPT-3. GPT-4 Turbo improved efficiency for production use; GPT-4o integrated text, vision, and audio while keeping strong code performance; GPT-4o mini emphasized cost efficiency. GPT-4.1 expanded context and code-editing capabilities, enabling repository-level software engineering within the series.
- ***o*-series** targets *reasoning-centered* modeling for complex problem solving with coding as a core focus. Early o1 and o1-mini introduced step-by-step internal deliberation, with o1-mini noted for software tasks [751]. Successors o3 and o3-mini [757] scaled context and optimized for repository-level editing and automated repair. On SWE-Bench Verified, the series outperformed prior GPT-4 models, establishing state-of-the-art proprietary performance in program repair and maintenance.
- **GPT-5** was introduced as OpenAI’s most capable coding model to date, with leading results on *SWE-Bench Verified* and *Aider Polyglot* [756]. **GPT-5-Codex** [755] specializes in agentic coding via RL on real engineering tasks, sandboxed execution, and controlled tool use, deployed across CLI, IDEs, and cloud. External commentary suggests strong gains over baseline GPT-5 on synthesis tasks, though estimates remain provisional. Together they combine stronger benchmark results with interactive, feedback-driven development workflows.

PaLM–Gemini Series Google’s **PaLM–Gemini** lineage evolves from dense, decoder-only Pathways scaling with SwiGLU and parallelized attention/FFN [188] through an efficiency-oriented redesign with multilingual pre-training and UL2-style denoising [52], to native multimodality with sparse expert routing and memory-efficient long-context attention [952, 954]. Across generations, the series consolidates code intelligence for program synthesis, multilingual editing, and repository-level reasoning via scaled sequence modeling and integrated tool use.

- **PaLM** [188] is a large decoder-only transformer using SwiGLU and parallelized attention/FFN to improve scaling. Trained on mixed natural language and substantial code, it transfers effectively to programming tasks; the finetuned *PaLM-Coder* further strengthens generation, repair, and translation, showing general models adapt well to coding workloads.
- **PaLM 2** [52] refines the scaling/data balance with multilingual pre-training and UL2-style denoising, delivering stronger results at more compute-efficient sizes. Its code-specialized variant **PaLM 2-S***—trained on multilingual code—shows competitive performance on HumanEval, MBPP, ARCADE, and BabelCode, highlighting robust cross-lingual synthesis and understanding.
- **Gemini 1 & 1.5** [952, 954] introduce native multimodality (text/code–vision–audio) under Pathways. Gemini 1.5 adds sparse MoE, efficiency improvements, and million-scale context, enabling repository-level comprehension and more reliable long-range code reasoning, with consistent gains over Gemini 1 on coding benchmarks (e.g., HumanEval, Natural2Code).
- **Gemini 2 & 2.5** [205, 338] emphasizes efficiency, reasoning, and code intelligence. 2.0 Flash optimizes attention and memory for long contexts while retaining multimodality; 2.5 extends context length, parallelism, and agentic capabilities (tool use, iterative reasoning). Trained on mixed natural language and code and finetuned for repair, translation, and synthesis, the series reports strong results on Natural2Code, Bird-SQL, LiveCodeBench, Aider Polyglot, and SWE-Bench Verified.

Anthropic Claude Series Anthropic’s **Claude** line evolves from RLHF/Constitutional-AI-aligned, decoder-only LLMs to long-context, tool-augmented agentic coders. **Claude 1→2** adds long-context and safer instruction following, boosting standardized code synthesis and editing [54–56]. **Claude 3/3.5** introduces native multimodality and function calling with documented gains on HumanEval and multi-file repository edits under sandboxed evaluation [57–59, 61]. **Claude 4/4.5** integrates deliberative reasoning and a computer-use stack (terminal, editor, package manager, browser) with policy-controlled tool use and parallel test-time compute, showing strong results on repository-level program repair and terminal-coding suites [63–65].

- The **Claude** family comprises proprietary decoder-only LLMs aligned via RLHF and Constitutional AI, with successive generations emphasizing longer context, safer instruction following, and robustness for structured outputs (JSON/XML and code) [54, 56]. **Claude 2** expands context and introduces training/service refinements for multistep reasoning and tool-friendly formatting, aiding repository comprehension, refactoring, and test-driven edits. Under standardized evaluation (e.g., HumanEval), Claude 2 shows clear gains in program synthesis [55], translating to stronger generation, explanation, debugging, and cross-language editing in closed-source models.
- The **Claude 3** family (Opus/Sonnet/Haiku) are proprietary, multimodal decoder-only LLMs with native tool use and vision inputs, with reported improvements in coding

reliability over prior generations [59, 61]. On HumanEval, Claude 3 demonstrates strong unit-style synthesis [59]. **Claude 3.5 Sonnet** further improves code performance and shows gains on repository-style multi-file editing in offline, sandboxed evaluations [57, 58]. Long-context retrieval is also strengthened, supporting large-codebase comprehension [57]. Overall, the 3 → 3.5 transition centers on multimodal, tool-augmented modeling with improved synthesis and repository-level editing under controlled tests.

- The **Claude 4** family integrates hybrid (deliberative) reasoning with first-class agentic coding and a computer-use toolchain (sandboxed shell, editor, package manager, browser), trained and aligned via RLHF and Constitutional AI [64, 65]. The system card details coding-specific safeguards and safety instrumentation for tool use, alongside dedicated evaluations for agentic coding and terminal workflows [64]. On SWE-bench Verified, Claude 4 reports strong program-repair accuracy, further improved by parallel test-time compute. **Claude 4.5 (Sonnet)** advances repository-level repair and shows gains on terminal-coding and tool-use suites [63, 64]. Collectively, Claude 4/4.5 shift toward long-horizon, tool-augmented coding agents that deliberate, invoke tools under policy controls, and iteratively validate patches, yielding measurable improvements in repair and structured editing.

Others

- **Grok Series** xAI’s **Grok** evolves from a proprietary, instruction-following decoder-only LLM into an agentic, code-oriented family with longer context and specialized coding variants. **Grok-1** shipped with Chat and later released as open weights, enabling public inspection and downstream use [1101, 1102]. **Grok-1.5** introduced a 128k-token window with stronger math/coding and long-context reasoning for repository-scale comprehension/editing [1102]. **Grok-2** reported gains on standardized coding evaluations such as HumanEval [1103]. The **Grok-4** generation emphasizes native tool use and “think” modes with real-time search, plus a code-specialized endpoint (grok-code-fast-1) for synthesis, refactoring, and repair loops [1104–1106]. Overall, Grok integrates longer context, tool-grounded reasoning, and a code-optimized serving path aligned with developer workflows.
- **PanGu-Coder** [191] uses a decoder-only transformer (PanGu- α) for function-level synthesis, translating between docstrings, signatures, and method bodies. Training follows large-scale causal pre-training on mixed language/code, then task adaptation on docstring-function pairs with code-focused losses (e.g., CODE-CLM). Emphasizing code tokens during fine-tuning outperforms docstring-side denoising, and the released 317M model is competitive on HumanEval under multi-sample evaluation. **PanGu-Coder2** [884] scales to 15B with longer context and introduces ranking-feedback alignment (RRTF): offline solutions are ranked by unit-test signals and teacher preferences, then optimized with a ranking loss. With expanded, leakage-screened instructions, it reports consistent gains on HumanEval and broader suites, showing that execution-aware ranking improves code generation without heavy online RL.
- **PyMT5** [193] casts method-level NL↔Python generation as a T5-style seq2seq multi-mode translation problem, where a single encoder-decoder model reconstructs any missing method feature (signature, docstring, body) through span-masking and feature-filling denoising. This unified formulation enforces cross-feature consistency and preserves syntactic structure, enabling controlled, feature-conditioned generation of Python methods. **JuPyT5** [148] extends this paradigm to Jupyter notebooks via a cell-infilling objective that predicts each cell from its surrounding context, modulated by cell-type control codes.

This notebook-aware seq2seq scheme models cross-cell dependencies and executable semantics, framing notebook code generation as structured infilling under test-driven supervision.

- **AlphaCode** [576] treats competitive programming as sequence-to-sequence translation from long natural-language statements to full programs, coupling large encoder-decoder transformers (pretrained on multilingual GitHub code and fine-tuned on the CodeContests dataset [576]) with massive stochastic sampling, execution-based filtering on public tests, and behavioural clustering over model-generated test inputs to select a small, diverse set of candidate solutions. **AlphaCode 2** [950] refines this pipeline with Gemini-based policies and a learned scoring model, applying two-stage fine-tuning on an updated *CodeContests* v2 dataset and a curated higher-quality problem set [950], while aggressive execution filtering, behavioural clustering, and reranking concentrate the submission budget on high-likelihood, semantically diverse candidates. **AlphaEvolve** [736] instead casts program synthesis as evolutionary search in code-edit space, maintaining a population of programs and iteratively applying LLM-generated diff-style patches, compiling and executing under task-specific tests, and selecting high-scoring descendants, thereby exploiting structured edits and test-time compute for scientific and algorithmic discovery.

2.2.2. Open-source Code Large Language Models

As shown in [Table 2](#), This subsection concisely reviews classical encoder-centric NL (natural language) - PL (programming language) embedding methods, highlighting core architectures, denoising/contrastive pre-training over code-text corpora, and their primary uses in retrieval and code understanding that underpin modern open-source Code LLMs. [Figure 6](#) illustrates representative model structures including encoder-only, encoder-decoder, and decoder-only designs.

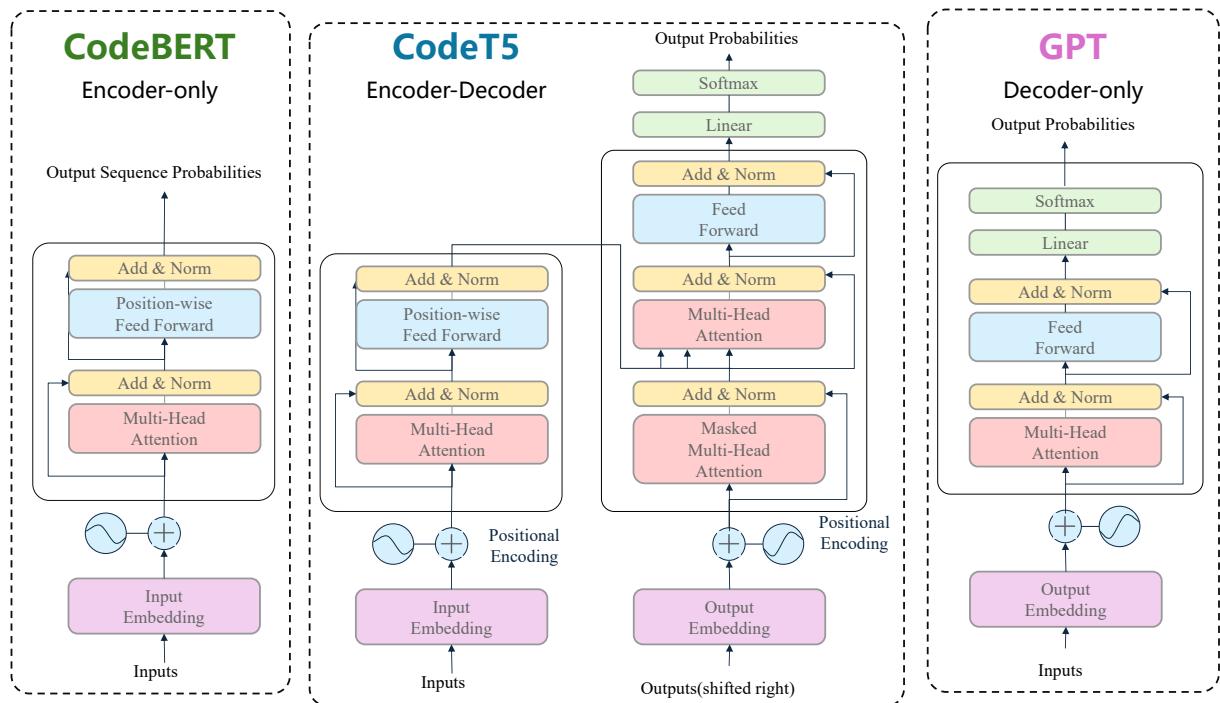


Figure 6. Comparison of model architectures for CodeBERT, CodeT5, and GPT.

Table 2. Open-source code-specialized LLMs.

Model	Layers	Hidden Size	Intermediate Size	Attention Method	Max Context	Extra
StarCoder 15B	40	6144	24576	MQA	8192	Multi Query Attention
StarCoder2-3B	30	3072	12288	GQA	16384 (sliding 4096)	BigCode consortium
StarCoder2-7B	32	4608	18432	GQA	16384 (sliding 4096)	Multiple data sources
StarCoder2-15B	40	6144	24576	GQA	16384 (sliding 4096)	Largest variant
Code Llama-7B	32	4096	11008	GQA	16k training (supports 100k)	Based on Llama2 architecture
Code Llama-13B	40	5120	13824	GQA	16k	Python specialization
Code Llama-34B	48	8192	22016	GQA	16k	Larger version
Qwen2.5-Coder-7B	28	3584	18944	GQA	131072 (with YaRN)	Base context 32768
Qwen2.5-Coder-32B	64	5120	27648	GQA	131072	State-of-the-art
Qwen3-Coder-30B-A3B	48	5120	25600	GQA+MoE	262144 (1M w/ YaRN)	MoE: 30B total 3.3B active (128 experts, 8 activated)
Qwen3-Coder-480B-A35B	62	6144	8192	GQA+MoE	262144 (1M w/ YaRN)	MoE: 480B total 35B active (160 experts, 8 activated)
IBM Granite Code-3B	28	2560	10240	GQA	8192	116 languages
IBM Granite Code-8B	36	4096	16384	GQA	8192	Enterprise focused
IBM Granite Code-20B	52	6144	24576	GQA	8192	High performance
IBM Granite Code-34B	88	6144	24576	GQA	8192	Depth upscaling
DeepSeek-Coder V2-Lite	27	2048	—	MLA + MoE	128k	MoE: 16B total 2.4B active (2 shared + 64 routed)
DeepSeek-Coder V2-236B	60	5120	12288	MLA + MoE	128k	MoE: 236B total 21B active 338 languages
Codestral-22B	56	6144	16384	GQA	32k	80+ languages FIM support Latest version
Codestral 25.01	-	-	-	-	256k	2x faster 80+ languages API only State Space
Codestral Mamba-7B	64	4096	—	Mamba2 SSM	256k	Model 7.3B params
Microsoft Phi-4	40	5120	17920	Full Attention	16k (ext. from 4k)	Strong math reasoning
Replit Code v1-3B	32	2560	10240	MQA	4096	Trained on Stack Dedup
StableCode-3B	32	2560	10240	MQA	16384	Fill-in-the Middle
WizardCoder-15B	40	6144	24576	MQA	8192	Fine-tuned StarCoder
Magicoder-6.7B	32	4096	11008	GQA	16384	Based on Code Llama
CodeGeeX2-6B	28	4096	13696	MHA	8192	Based on ChatGLM2
CodeGeeX4-ALL-9B	40	4096	14336	GQA	131072	Multi-language
OctoCoder-15.5B	40	6144	24576	MQA	8192	Fine-tuned StarCoder
Yi-Coder-1.5B	28	2048	8192	GQA	131072	52 languages
OpenCoder-1.5B	24	2240	6144	GQA	4096	Fully open-source
OpenCoder-8B	32	4096	14336	GQA	8192	2.5T tokens training

2.2.3. Evolution of Open-Source Code Large Language Models

The development of open-source code large language models can be systematically categorized into four distinct evolutionary stages based on their architectural innovations and functional capabilities. This taxonomy provides a comprehensive framework for understanding the technological progression in the open-source code intelligence community.

Icon legend. Throughout this subsection, we annotate models with small icons indicating their architectures and primary capabilities.

Architecture icons

-  — encoder-only

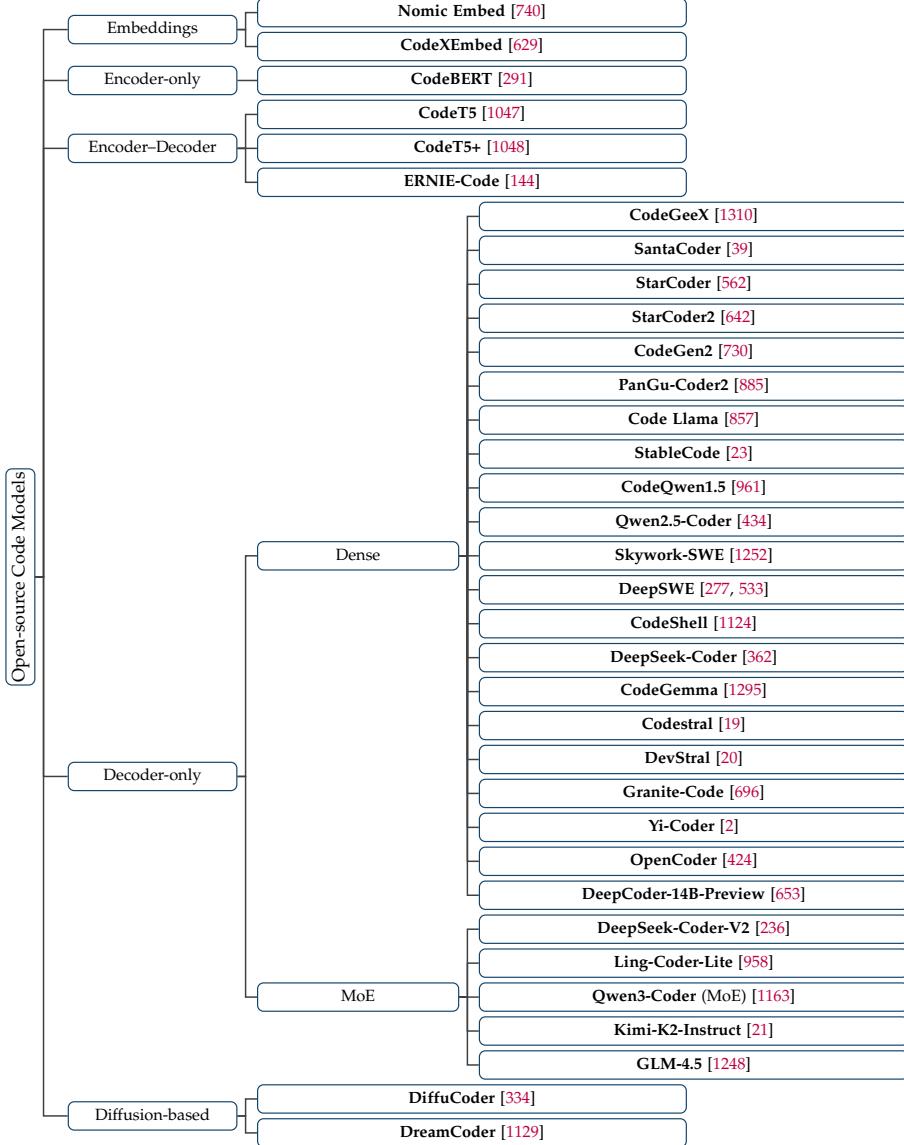


Figure 7. Taxonomy of selected open-source code models grouped by architecture.

- — encoder-decoder
- — decoder-only
- — diffusion-based
- — mixture-of-experts (MoE)

Functional icons

- — retrieval / embedding
- — code understanding
- — code generation
- — fill-in-the-middle / infilling
- — software-engineering agents

Stage 1: Pre-trained Encoder Models. The initial stage was dominated by encoder-based pre-trained models such as CodeBERT [360], GraphCodeBERT [360], and CodeT5 [168]. These

open-source models primarily focused on code understanding tasks, establishing fundamental code-text alignment through bidirectional attention mechanisms. Their core strengths lay in code classification, vulnerability detection, and semantic code search.

-  **CodeBERT** () [291] is an encoder-only RoBERTa-style model pre-trained on paired natural language and source code using a hybrid objective (masked prediction on NL–PL pairs plus replaced-token detection) with data from CodeSearchNet. It is primarily used for representation tasks (retrieval, reranking, classification) and is typically combined with a decoder when applied to generation.
-  **ERNIE-Code** ( ) [144] is a multilingual text–code encoder–decoder built on the T5 line with a single vocabulary covering many natural and programming languages and added tokens to capture code layout. Its pretraining mixes span-corruption on text and code with a pivot-based translation objective to promote cross-lingual and cross-modal alignment; fine-tuned ERNIE-Code shows strong transfer on summarization, text-to-code, translation and program repair.

Stage 2: Generative Models. The second stage witnessed the emergence of open-source generative models including CodeT5 [168] and CodeGPT [1244], which introduced encoder–decoder architectures capable of both code understanding and generation. These models demonstrated proficiency in code generation, cross-language translation, and automated code completion tasks.

-  **CodeParrot** () [279] is a family of decoder-only GPT-2 models (110M, 1.5B parameters) specifically trained for Python code generation tasks. Built upon the cleaned CodeParrot dataset derived from GitHub dumps with aggressive deduplication and filtering heuristics, the model employs a GPT-2 tokenizer trained on code-specific vocabulary. The training methodology uses standard autoregressive language modeling with left→right generatio. CodeParrot excels at Python code completion, docstring→code generation, and unittest generation tasks, demonstrating strong performance on code synthesis despite being trained on significantly fewer tokens than larger models. The model architecture and training data are fully open-sourced, enabling reproducible research in neural code generation.
-  **CodeGPT** ( ) [1244] is a family of GPT-style transformer models (110M, 1.5B parameters) developed by Microsoft Research for Python code understanding and generation. Built upon large-scale filtered GitHub repositories with aggressive data cleaning and deduplication strategies, the model employs a combination of masked language modeling and next token prediction during pre-training. The training methodology incorporates multi-task learning across diverse code-related objectives including code completion, NL→code generation, code→NL summarization, and bug detection tasks. PyCodeGPT excels at syntactic and semantic code understanding, enabling applications in automated code review, documentation generation, and program repair. The model architecture and training approach contribute to Microsoft’s broader research initiative in AI-assisted software development, demonstrating strong capabilities in code completion, comment generation, and educational programming assistance.
-  **T5 series** ( ) [1047, 1048] are T5-derived models for code understanding and generation that use a code-aware tokenizer and identifier-aware pretraining, alternating unimodal and bimodal data and employing a dual-generation stage to align NL and PL. The family spans encoder/decoder and seq2seq variants (from compact to large), applies

a two-stage pretraining plus instruction tuning recipe, and is competitive across many code tasks.

Stage 3: Large Language Models. The third stage marked a paradigm shift with the advent of large-scale open-source language models such as StarCoder [562], CodeLlama [858], DeepSeek-Coder [232], and CodeQwen [961]. These models exhibited remarkable capabilities in complex code generation, multi-turn conversational programming, and instruction following, demonstrating that open-source models could achieve competitive performance with proprietary counterparts.

- **SantaCoder** () [39] is an open-source decoder transformer from the BigCode project. It adopts Multi-Query Attention (MQA) for efficient serving and a Fill-in-the-Middle (FIM) objective to support both left-to-right generation and code infilling. Pretraining used permissively-licensed code (Python/Java/JavaScript) with strict data governance: opt-out honoring, PII redaction, aggressive near-deduplication, and documentation-aware filtering. A two-phase training recipe validated design choices before a final large-scale run. SantaCoder supports text-to-code, infilling, and multilingual synthesis, performs well on multi-language code benchmarks (e.g., MultiPL-E) compared to some larger models.
- **OctoCoder** () [711] is an instruction-tuned Code LLM (StarCoder-16B base) trained on permissively licensed, commit-derived instructions mixed with natural-language targets. The work also releases **HumanEvalPack**, extending HumanEval to three tasks—code repair, explanation, and synthesis—across six languages (Python, JS, Java, Go, C++, Rust). OctoCoder attains the best average pass@1 among commercially usable (permissive) models on this suite—e.g., strong gains in bug-fixing from commit-style data and solid synthesis—while closed models like GPT-4 remain higher overall. The paper underscores practical deployability (permissive licensing, OpenAI-output-free data), multilingual generalization from pretraining, and the importance of mixing NL targets to avoid code-only bias.
- **CodeGeeX** () [1310] is a multilingual GPT-style decoder LLM aimed at generation and translation. It was pretrained on a large multilingual code corpus spanning many languages and introduces **HumanEval-X**, a multi-language suite of canonical problems for generation and cross-lingual translation evaluation. The work emphasizes deployability (INT8 quantized inference, integration with FasterTransformer) and developer tooling (IDE plugins), and shows that CodeGeeX is a top-performing open multilingual baseline, competitive with comparable models depending on language; a fine-tuned variant further improves translation. The release also reports substantial real-world usage and user-reported productivity gains.
- **StarCoder** () [562] and **StarCoderBase** are open-access decoder-only code models with long context and FIM training. StarCoderBase was trained on a large permissive-code collection (The Stack) and StarCoder is obtained after targeted fine-tuning on additional Python data. The project prioritizes data governance (near-deduplication, benchmark decontamination, PII redaction) and practical engineering (tokenizer sentinels for FIM, efficient attention/backends for long context). Evaluated across diverse code benchmarks, StarCoderBase/StarCoder lead among open multilingual code LLMs and compare favorably to some closed models. The release includes IDE demos and an OpenRAIL-M license that pairs permissive access with documented usage restrictions.
- **CodeGen2** () [730] presents an open-source family of decoder-only code LLMs and a single, practical recipe that unifies architecture choices, sampling modes (left-to-right & infill), and mixed NL/PL data. The study tests a Prefix-LM unification but finds *no*

consistent gains over a causal decoder; the final recipe therefore uses a decoder-only transformer with a mixed objective: with probability $p = 0.5$ train by next-token prediction (CLM), otherwise apply within-file span corruption (dynamic mask ratios/lengths) to enable infilling. Infill training is shown to trade off slightly with pure left-to-right performance, NL+PL mixing offers a robust compromise when one model must cover both modalities, and continued multi-epoch pretraining (the CodeGen2.5 variant) yields clear scaling benefits. Overall lessons: Prefix-LM is not superior for these tasks, infill is not free, a CLM+span-corruption mixture is effective, NL+PL mixing is promising under constrained compute, and multi-epoch training is important.

-  **Code LLaMA** ( ) [857] is a LLaMA 2-based code family that emphasizes strong infilling, long-context support, and instruction-following for programming. It ships in foundation, Python-specialized, and instruction-tuned variants across multiple scales and is trained/finetuned for long sequences and repository-scale completion (RoPE base period increased from 10^4 to 10^6 during long-context fine-tuning). Training continues from LLaMA 2 on a code-heavy corpus; the Python and Instruct variants add focused token streams for language specialization and alignment. Ablations report modest trade-offs from infill training, clear gains from long-context fine-tuning for repository tasks, and consistent benefits from language specialization; safety-tuned instruct models improve truthfulness and reduce toxicity while preserving coding ability.
-  **MFTCoder** ( ) [598] proposes a multi-task fine-tuning framework that trains a single decoder-only backbone to handle completion, text-to-code, commenting, translation, and unit-test generation concurrently. It addresses multi-task issues via a data-balanced, token-weighted loss, focal-style emphasis at sample and task levels, and a validation-driven dynamic reweighting that prioritizes slowest-converging tasks. Efficiency techniques—dynamic padding, packed SFT (concatenating samples with eos), and PEFT support (LoRA/QLoRA)—reduce padding and enable practical fine-tuning of large bases on modest hardware. Applied across multiple models, MFTCoder shows consistent gains over single-task SFT and mixed-data SFT and better generalization on unseen tasks.
-  **DeepSeek-Coder** (  ) [362] is an open-source code LLM family (1.3B–33B) trained from scratch on multi-language corpora. A key idea is repository-level pretraining that models cross-file dependencies, improving repo understanding and cross-file completion. It integrates a fill-in-the-middle objective with long context (up to 16,384 tokens), enhancing FIM infilling and long-range code reasoning. It reports strong results on HumanEval and MBPP, exceeding GPT-3.5 without proprietary data. Instruction-tuned variants show robust multi-turn problem solving, and the permissive license supports reproducibility and practical adoption.
-  **StableCode** (  ) [23] is a 3B lightweight open model for code generation and understanding, trained on large GitHub corpora. It supports completion and natural language → code, with long-context handling (up to 16,384 tokens) for multi-file reasoning. Performance on HumanEval/MBPP is competitive among compact open models, trading peak accuracy for efficiency and easy deployment under limited compute.
-  **StarCoder2** (  ) [642] advances the BigCode line with **The Stack v2** (larger and more diverse, partnered with Software Heritage). Models at 3B/7B/15B are trained across hundreds of languages plus issues/PRs, docs, and math/logic data. Training uses two stages (4k → 16k) with repository-context formatting and FIM. On benchmarks, **StarCoder2-3B** surpasses similar-size peers, and **StarCoder2-15B** matches or exceeds larger models, with strong math and low-resource language performance.
-  **CodeShell** ( ) [1124] is a 7B foundation model (8k context) extending GPT-2 with grouped-query attention and RoPE for efficient inference. Its hallmark is rigorous data

governance: multi-stage filtering (deduplication, perplexity, structure rules, model-based scoring) to build a high-quality corpus. Despite modest scale, CodeShell outperforms comparable 7B models and shows competitive results on MultiPL-E and code completion, supporting the view that careful data curation can rival sheer scaling.

- **CodeGemma** ([1295]) adapts Gemma for coding via large-scale code-centric pretraining and instruction tuning. The suite includes a 2B model for low-latency completion/infilling and 7B variants (pretrained and instruction-tuned). Curated corpora employ deduplication, contamination removal, and multi-file packing guided by dependency graphs and tests; an improved FIM objective (high-rate) supports both prefix-suffix-middle and suffix-prefix-middle modes. The 7B-IT model performs strongly on HumanEval/MBPP and multilingual coding (e.g., BabelCode), with solid mathematical reasoning, while the 2B model offers competitive accuracy with fast inference for IDE use.
- **Granite-Code** ([696]) is a decoder-only open-source family from IBM (3B–34B) trained in two stages (large-scale code pretraining → mixed code+NL enhancement). It integrates Fill-in-the-Middle (PSM/SPM) with causal LM, improving infilling and completion. The series shows solid results on coding and explanation/fix tasks while emphasizing enterprise-grade data transparency and Apache 2.0 licensing for practical deployment.
- **Codestral** ([19]) is a 22B open-weight code model focused on instruction following and FIM across many languages, with a ~32K context for repository-level reasoning. Public materials report strong long-range and FIM performance, including on RepoBench and Python-oriented evaluations. It is released for research under the Mistral AI Non-Production License with separate commercial options.
- **Yi-Coder** ([2]) targets high coding quality under compact sizes (1.5B/9B; base and chat) with up to 128K context over 52 languages. The models prioritize inference efficiency and interactive debugging, showing robust outcomes on HumanEval, MBPP, and LiveCodeBench relative to larger peers. Weights, code, and deployment guides are provided under Apache 2.0 for straightforward IDE and production integration.
- **CodeQwen1.5 & Qwen2.5-Coder** ([961]) is a 7B decoder-only model trained on large-scale code corpora, covering many languages and long-context (up to 64K). It adopts GQA for efficient inference and extended context, demonstrating strong text→SQL, bug fixing, and debugging. Qwen2.5-Coder [434] expands to a family (0.5B–32B) trained on a balanced mix of code, natural language, and math. It combines file→repository pretraining with FIM, and scales context to 128K via YARN. Instruction tuning blends multilingual synthesis and DPO with execution feedback, yielding solid results on MultiPL-E, RepoEval, and CrossCodeEval without relying on narrow prompt formats.
- **OpenCoder** ([424]) emphasizes full reproducibility: weights, inference code, curated RefineCode data, processing pipelines, and checkpoints are all released. Models (1.5B/8B) use a LLaMA-style transformer (RoPE, SwiGLU) and a two-stage instruction plan (general SFT → code-specific SFT). The 8B variants report strong HumanEval/MBPP, multilingual (MultiPL-E), and debugging performance, surpassing StarCoder2-15B and CodeLlama-7B. The project serves both as a capable model and an open recipe for scientific reuse.

Stage 4: Advanced Scaling and Agentic Models. The current stage represents two major developments: massive parameter scaling through mixture-of-experts (MoE) architectures that maintain high inference efficiency while dramatically increasing model capacity, and the evolution toward agentic systems that integrate tool usage, multi-step reasoning, and environment interaction capabilities. Representative models like DeepSeek-Coder-V2 [1333] demonstrate

how MoE architectures enable unprecedented scaling while preserving computational efficiency. These models excel at complex software engineering tasks, including repository-level programming and systematic code maintenance as demonstrated in benchmarks like SWE-bench [928].

- **DeepSeek-Coder-V2** () [236] adopts a Mixture-of-Experts backbone (16B and 236B; small active experts per token) continued from DeepSeek-V2 with mixed code/math/NL data and extended context (up to 128K via YARN). It delivers strong results across synthesis, competitive programming, bug fixing, and math reasoning, with a lightweight variant offering compelling efficiency. Released under a permissive license, it narrows the gap with top closed models in both coding and reasoning.
- **Ling-Coder-Lite** () [958] is an MoE code LLM (few active parameters per token) with shared+routed experts, top-6 routing, and a refined NormHead design. Training proceeds via continued pretraining and instruction optimization (SFT → DPO) over high-quality, execution-aware, repository-structured data. It shows competitive results on HumanEval, MBPP, LiveCodeBench, and BigCodeBench against similarly sized peers, achieving a favorable performance-efficiency trade-off for low-latency deployment.
- **Skywork-SWE** () [1252] presents an execution-aware curation pipeline plus an open 32B agent, revealing clear SWE data scaling laws with LLMs. It collects PR-issue pairs, builds per-instance Docker runtimes validated by tests, and filters multi-turn agent trajectories to retain only passing solutions. Fine-tuning within **OpenHands** on validated trajectories yields **Skywork-SWE-32B**, which improves over its base on SWE-bench-Verified and further benefits from test-time scaling. Ablations indicate log-linear gains with more trajectories and that execution-grounded data and framework quality matter more than parameter count. The work releases the checkpoint and practical guidance for leakage control and scalable evaluation.
- **DeepCoder** () [653] is a fully open, RL-trained 14B code-reasoning model fine-tuned from DeepSeek-R1-Distilled-Qwen-14B. It targets repository-level coding with long-context editing (trained at 32K, inference-time scaled to 64K) and reaches competitive LiveCodeBench performance versus strong proprietary baselines. Training uses verifiable tasks and rewards (e.g., TACO-Verified, a verified subset of PrimeIntellect SYNTHETIC-1, and LiveCodeBench from 2023-05-01 → 2024-07-31) enforced by unit tests. The release includes the RL pipeline, datasets, evaluation logs, and traces for reproducible study.
- **DeepSWE** () [277] is an open-source *RL-only* coding agent on **Qwen3-32B** with a thinking mode. A compact RL recipe rapidly lifts SWE-bench-Verified, and test-time scaling with a *DeepSWE-Verifier* selects high-quality patches; combining execution-free and execution-based verifiers yields additional gains. The public write-up details the rLLM-based¹ setup on real repository-level tasks with stability tweaks for long-horizon, multi-file editing, showing that RL-only post-training + lightweight TTS narrows the gap to larger proprietary systems.
- **Devstral** () [20] is an Apache 2.0 agentic code LLM co-developed by Mistral AI and All Hands AI. *Devstral Small* (24B, 128K context) targets repository-scale SWE on accessible hardware, while *Devstral Medium* provides stronger cost-performance via API. On SWE-bench-Verified, both achieve top-tier open-weight results and are designed as agent backbones emphasizing multi-file reasoning, long-context editing, and verifier-friendly test-time scaling.
- **Qwen3-Coder** () [824] advances agentic capabilities (e.g., Qwen3-Coder-480B-A35B-Instruct), showing strong open-model performance on agentic coding, browser use, and foundational coding tasks, competitive with leading assistants. It offers native

¹<https://github.com/agentica-project/rllm>

256K context (extendable to 1M via Yarn), a structured function-call schema, and integration with Qwen Code/Cline. With permissive licensing, the series stands as a leading open-source code-LLM family.

- **GLM-4.5/4.6** () [1248] is an open MoE foundation model for agentic, reasoning, and coding tasks, featuring hybrid “think” ↔ direct modes with ~32B active parameters per token within a larger MoE design. Both GLM-4.5 and its GLM-4.6 successor adopt GQA, QK-Norm, and an MoE multi-token prediction head for speculative decoding; training spans diverse corpora with mid-training that upsamples repo-level code, synthetic reasoning, and long-context/agent trajectories, with context extended from 4K → 32K → 128K in GLM-4.5 and further to 200K tokens in GLM-4.6. Post-training blends expert models via SFT and unified self-distillation, with RL innovations (difficulty curricula, long-output RL, dynamic temperature, code-weighted losses) yielding consistent gains across TAU-Bench [1210], AIME [3, 4], SWE-bench-Verified, and BrowseComp, while GLM-4.6 additionally improves coding, tool-augmented reasoning, and agentic performance across a broader suite of public benchmarks and enhances writing quality.
- **Kimi-K2-Instruct** () [21] is the instruction-tuned variant of the Kimi-K2 Mixture-of-Experts (MoE) series developed by Moonshot AI. It employs a large-scale MoE design with $\sim 10^{12}$ total parameters and 3.2×10^{10} active per forward pass, as shown in [Figure 8](#). The model is pretrained with the MuonClip optimizer on trillions of tokens, followed by agentic data synthesis and reinforcement learning for improved instruction following and tool usage. On code-related benchmarks, Kimi-K2-Instruct shows strong performance across multiple evaluation sets. It also maintains robust reasoning on mathematical and logic tasks, reflecting its cross-domain capability. With native tool invocation and extremely long context support ($\geq 128K$ tokens), it serves as a versatile open-weight foundation for agentic code assistants and general-purpose reasoning systems.
- **KAT-Dev** () [1257] is an open-weight code-centric model series from Kwaipilot, built on the Qwen3 architecture and released under Apache 2.0. The 32B variant reaches 62.4% on SWE-Bench Verified, ranking among the strongest open models. Its training pipeline combines mid-training for tool-use and instruction following, supervised and reinforcement fine-tuning across diverse programming tasks, and large-scale agentic RL. With long-context support and native tool invocation, KAT-Dev serves as a versatile foundation for autonomous coding agents and general-purpose software reasoning.
- **DeepSeek-V3/V3.1/V3.2** () [233] is an open Mixture-of-Experts LLM series for agentic reasoning and code generation, featuring hybrid “thinking” vs. direct modes and ~37B active parameters per token (671B total) with a 128K context window. DeepSeek-V3 adopts Multi-Head Latent Attention and a multi-token prediction head for efficient long-context inference, and is pre-trained on 14.8T tokens with auxiliary-loss-free MoE load balancing, followed by SFT and RL fine-tuning. It achieves state-of-the-art open-source performance on coding benchmarks, rivaling closed models on code tasks. Its successor DeepSeek-V3.1 underwent extended training (an additional 840B tokens to reach 32K then 128K context) and integrated the “DeepThink” chain-of-thought mode, which boosted multi-step tool use and coding-agent capabilities. Post-training optimizations made V3.1 significantly stronger on software engineering challenges (e.g. SWE-bench, Terminal-Bench), outperforming earlier V3 and R1 models in code generation and search-agent benchmarks. The experimental DeepSeek-V3.2 builds on V3.1-Terminus with a novel DeepSeek Sparse Attention mechanism that yields near-linear attention scaling, cutting inference cost by 50% for long inputs while maintaining output quality on par with V3.1. This improves efficiency in handling large code bases and retrieval-augmented coding tasks without degrading coding accuracy.

- **MiniMax-M1/M2** () is an open MoE model pair for long-context reasoning, coding, and agentic tasks. M1 introduced a hybrid Mixture-of-Experts architecture with a custom “lightning” attention mechanism, enabling a 1-million-token context window (8× DeepSeek-R1’s length) while maintaining high FLOP efficiency. Trained via large-scale reinforcement learning, M1 excels at complex multi-step reasoning, software engineering, and tool use, outperforming earlier open models on long-horizon tasks. Its successor M2 emphasizes deployment efficiency – using 230B total (10B active) parameters to deliver near-frontier performance in code generation and autonomous tool use with only 200K context. Post-training alignment further boosts M2’s capabilities across end-to-end coding benchmarks and agent planning tasks (e.g. SWE-Bench[928], BrowseComp[1064]), making it one of the most capable and practical open LLMs for complex workflows.

Alternative Architecture Explorations. Beyond the mainstream autoregressive transformer paradigm, diffusion-based language models for code have recently begun to attract attention. On the proprietary side, models such as Gemini Diffusion [230] and Mercury Coder [505] illustrate that discrete text diffusion can achieve competitive code quality while substantially reducing generation latency compared to standard autoregressive decoders. In parallel, the open-source community is also exploring this design space: for example, DiffuCoder [334] investigates masked diffusion models for code generation and reports encouraging results on standard coding benchmarks, suggesting that diffusion LLMs are a viable alternative architecture for code synthesis tasks.

- **DiffuCoder** () is an open 7B masked diffusion coder that serves as a canonical testbed for diffusion-native code generation and reinforcement learning. Trained on 130B effective tokens of code, DiffuCoder delivers performance competitive with strong autoregressive coders while enabling non-autoregressive, globally planned decoding over the entire sequence. Using this model, the authors introduce local and global “AR-ness” metrics to quantify how closely diffusion LMs follow left-to-right generation, and show that raising the sampling temperature diversifies not only token choices but also generation order, creating a rich rollout space for RL. Building on this insight, they propose *coupled-GRPO*, a diffusion-native variant of GRPO that applies complementary mask noise to paired completions, reducing variance in likelihood estimates and better exploiting the non-AR search space. Coupled-GRPO training yields a +4.4% improvement on EvalPlus with only ~21K RL samples, further strengthening DiffuCoder-Instruct and establishing DiffuCoder as a strong open baseline for future diffusion-based coding assistants and RL research.

Code Retrieval Embeddings. Parallel to the main evolutionary trajectory, open-source code retrieval embedding models have undergone their own transformation. Early approaches relied on BERT-based encoder models such as CodeBERT and UniXcoder [361] for generating code representations. Recent developments have shifted toward open-source LLM-based embedding models, leveraging the rich semantic understanding of large language models to produce more sophisticated code embeddings for retrieval and similarity tasks.

- **Nomic Embed Code** () [740] is a 7B parameter code embedding model that achieves state-of-the-art performance on CodeSearchNet through high-quality contrastive training. Built upon the CoRNStack dataset—a large-scale curated corpus derived from deduplicated Stackv2 with dual-consistency filtering—the model converts code retrieval tasks into semantic similarity matching using cosine distance between pooled representations. The

training methodology employs a novel curriculum-based hard negative mining strategy with softmax-based sampling to progressively introduce challenging examples during contrastive learning. Nomic Embed Code excels at NL→code, code→NL, and code→code retrieval tasks across multiple programming languages while maintaining full open-source availability of training data, model weights.

-  **CodeXEmbed**(🔍) [629] is an open family of multilingual and multi-task retrieval models spanning both encoder and decoder architectures. The 400M variant is a BERT-style bi-encoder trained from scratch for efficiency-oriented deployment, while the 2B and 7B variants are decoder-only LLMs adapted into dual-tower encoders for generalist retrieval. All variants map diverse text–code tasks into a unified query–document matching framework, where pooled embeddings are compared via cosine similarity. A two-stage LoRA contrastive training pipeline—first on large-scale text retrieval and then jointly on text–code pairs with hard negatives—produces models specialized for Text→Code, Code→Text, and Code→Code retrieval, as well as repository-level RAG. The 7B model achieves state-of-the-art results on CoIR, while smaller models maintain strong BEIR performance with lower latency and cost.
-  **CodeSage**(🔍) [1262] is a family of bidirectional encoder models (130M, 356M, 1.3B) trained for large-scale code representation learning across nine programming languages. It employs a two-stage training scheme: first, a mix of identifier deobfuscation and masked language modeling (without the 80-10-10 corruption) for token-level denoising, and second, bimodal contrastive learning using text–code pairs with hard positives and hard negatives. This design promotes semantic alignment between natural and programming languages. Evaluated on NL→Code, Code→Code, and classification benchmarks, CodeSage consistently outperforms prior models such as UnixCoder, GraphCodeBERT, and OpenAI-Ada embeddings. Larger variants yield stronger cross-lingual and retrieval performance, while smaller ones balance speed and efficiency.
-  **BGE-Code**(🔍) [533] is a generalist code-embedding bi-encoder (Qwen2.5-Coder 1.5B) trained with an *Annealing* curriculum to transfer from text-only to code-centric retrieval. It relies on the synthetic **CodeR-Pile** built under DRU (Diversity, Reliability, Usability), spanning Text2Code, Code2Text, Code2Code, and Hybrid tasks across many languages. Data are synthesized via LLM brainstorming, instruction refinement, pair generation/annotation, and *hard-negative* mining on real code. LoRA-based training with staged schedules and difficulty filtering produces strong results on CoIR/CoIR-filter/CodeRAG. Ablations support that broader task coverage, mined negatives, and the curriculum ≫ single-shot mixed training.
-  **CodeFuse-CGE**(🔍) [197] is an open decoder-only family for code retrieval that adapts causal LLMs into dual-tower encoders via a lightweight cross-attention embedding head. The Large model builds on CodeQwen1.5-7B-Chat and the Small model on Phi-3.5-mini-instruct; both are LoRA-tuned to project text and code into a shared vector space and scored by cosine similarity. CGE reframes NL→Code search as query–document matching and targets repository-level workflows; it reports strong results on CodeSearchNet and AdvTest, and has been used as the semantic retriever in repo-level systems. Compared with encoder baselines, the decoder-based design captures richer cross-modal semantics while remaining practical to deploy.

2.2.4. Model Pre-Training Tasks

Next Token Prediction Next Token Prediction (NTP) is the most fundamental and widely used self-supervised training task, whose goal is to predict the next likely word or subword

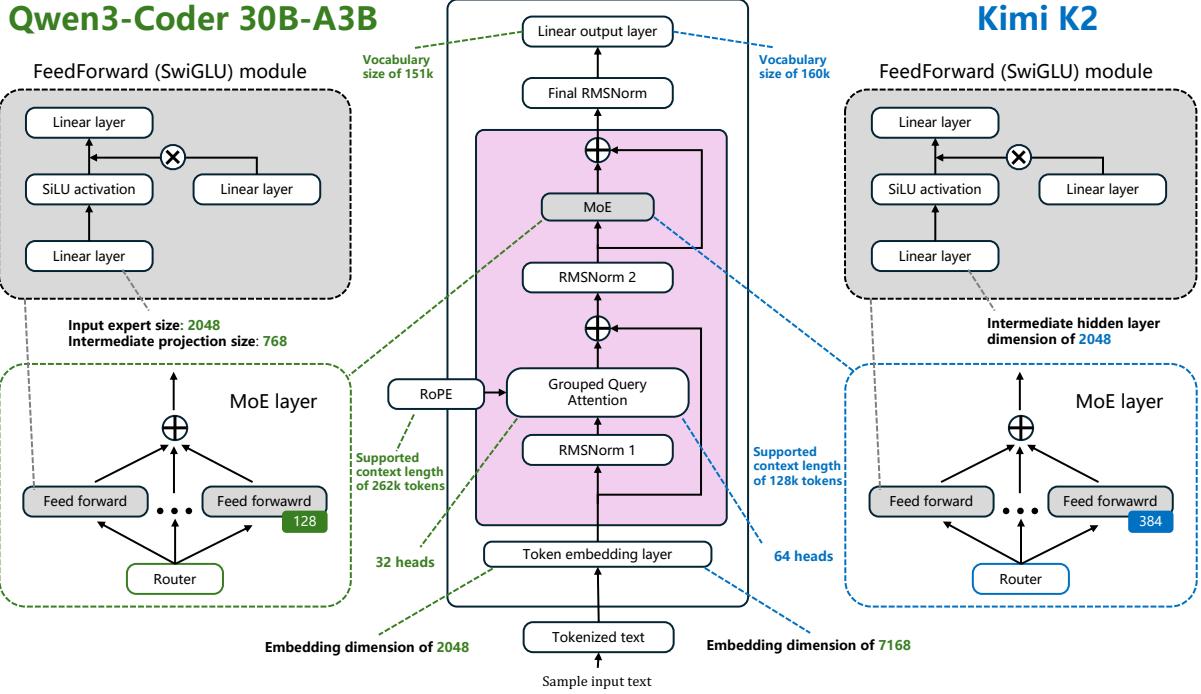


Figure 8. Architectural comparison between Kimi-K2-Instruct and Qwen3-Coder.

unit based on a given preceding context sequence. Essentially, this task is a form of Causal Language Modeling (CLM), where the model can only access information up to the current moment and cannot “peek into” future content. During the training process, the input sequence is slid token by token, and the label for each position is the token that immediately follows it. The model learns the conditional probability distribution $P(x_{t+1} | x_1, x_2, \dots, x_t)$ by minimizing the cross-entropy loss. When training, given a text sequence of length T , the model sequentially predicts the $t + 1$ -th token for each position $t \in [1, T - 1]$. This process enables the models to capture the statistical laws of language, grammatical structures, semantic correlations, and world knowledge, thereby establishing robust capabilities in language understanding and generation.

Multi-Token Prediction In Figure 9, we provide a comparison between next token prediction (NTP) and multi-token prediction (MTP) training objectives in large language models. Multi-Token Prediction (MTP) is an extended task built on the foundation of Next Token Prediction. Its objective is to enable the model to predict multiple consecutive tokens at once based on the preceding context sequence, thereby improving the model’s text generation efficiency and coherence.

Fill-in-the-Middle In Figure 10, fill-in-the-Middle (FIM) is a non-autoregressive language modeling task. Its core objective is to enable the model to predict the missing token segment in the middle of a text sequence based on the given prefix and suffix of the sequence, thereby enhancing the model’s ability to understand the global semantics of text and its sequence completion capability. The execution logic of this task differs from autoregressive sequential prediction: first, the model inputs both the prefix and suffix sequences into the network simultaneously, and uses the bidirectional attention mechanism of the transformer to jointly encode the semantics of the prefix and suffix, capturing the semantic association between them; then, the model

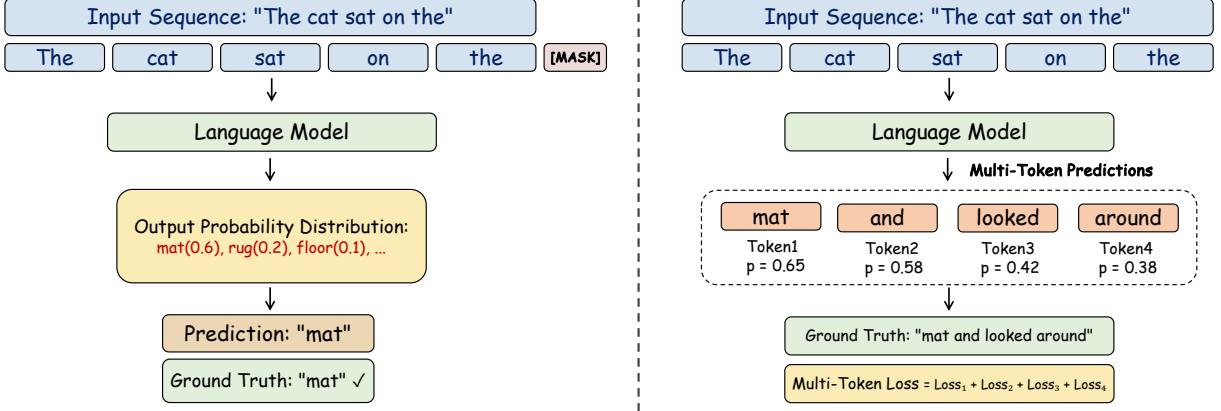


Figure 9. Comparison between next token prediction (NTP) and multi-token prediction (MTP) training objectives in large language models. In NTP (left), the model predicts the next single token (“mat”) given the preceding context. In contrast, MTP (right) extends this to predict multiple future tokens (“mat and looked around”) simultaneously, optimizing a combined loss over all predicted tokens. This parallel formulation can improve training efficiency and better capture longer-range dependencies in sequence modeling.

generates possible token sequences for the missing middle region, and optimizes its parameters by calculating the loss between the generated sequence and the real middle sequence. Formally, the input format is often expressed as: <prefix>...<suffix> → <mask>, and the model’s output is the original text with the masked part restored. The Code LLaMA [857] models all adopt the FIM strategy for training. To enhance diversity and robustness, half of the segmentations use the prefix-suffix-middle (PSM) format, while the other half use the compatible suffix-prefix-middle (SPM) format. To support this new task, Code LLaMA extends the original tokenizer of Llama 2 with four special tokens: <fim_prefix>, <fim_suffix>, <fim_middle>, and <fim_pad>, which are used to clearly identify the boundaries of each part in the input sequence. By jointly optimizing FIM as a multi-task objective parallel to standard autoregressive prediction, the Code LLaMA model can directly implement intelligent completion for cursor positions or arbitrary code blocks in environments such as IDEs during inference without additional fine-tuning, significantly improving its practicality in real-world programming scenarios.

Diffusion Coder Training Task Diffusion Coder is a language model designed based on the principles of diffusion models. Its core objective is to enable the model to generate text token sequences that comply with semantic logic from random noise sequences through a “gradual denoising” process, primarily for enhancing the diversity and quality of text generation, as illustrated in Figure 11. The execution logic of this task is divided into two phases: the first is the “forward diffusion phase”, in which random noise is gradually added to the real text token sequence in accordance with a preset noise scheduling strategy, causing the sequence to gradually approximate pure noise; the second is the “reverse denoising phase”, where the model needs to learn to start from the noisy sequence, gradually remove noise based on the noise level, and restore the real text sequence. During the training process, the model optimizes its parameters by calculating the loss between the “predicted noise” and the “actually added noise”, and ultimately gains the ability to generate high-quality text from noise. Compared with autoregressive models, Diffusion Coder has the advantages of higher diversity in generated text and the ability to improve generation efficiency through parallel computing.

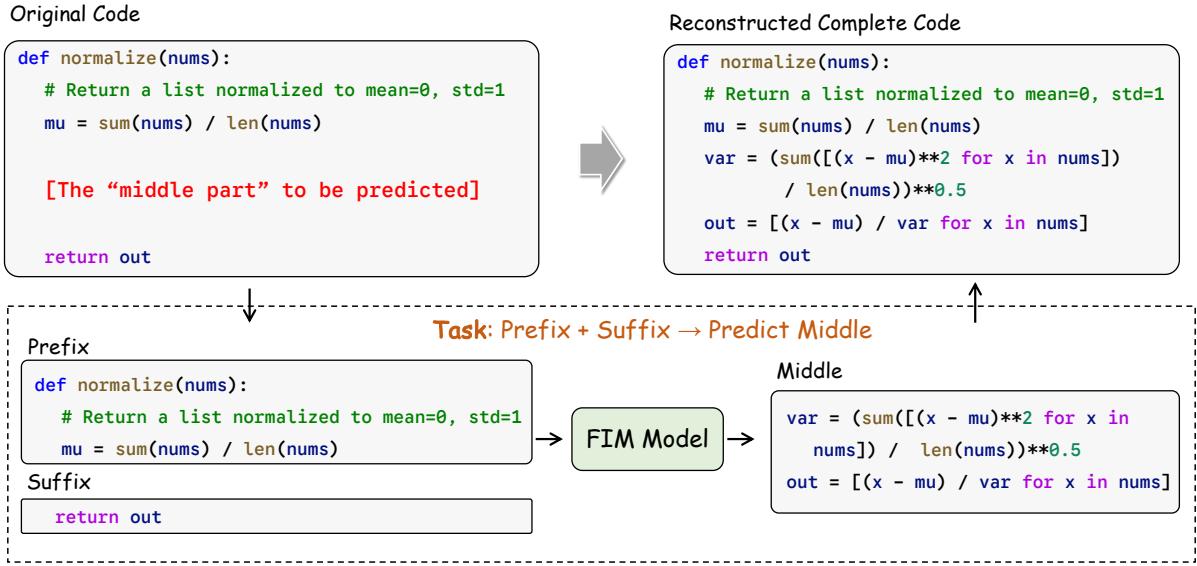


Figure 10. Illustration of the Fill-in-the-Middle (FIM) training and inference process for code completion. The model receives both a prefix and a suffix of a function and learns to generate the missing middle segment. In this example, given the beginning and end of the `normalize()` function, the FIM model predicts the intermediate computation steps for variance and normalization, reconstructing the complete code.

2.2.5. Model Training Stages

The training stages of large language models are a critical process through which models progress from learning general linguistic knowledge to adapting to specific tasks and generating content that meets human needs. This process determines the model’s capability boundaries and practical value. It primarily consists of two main parts: **Pre-training** and **Post-training**, as shown in [Figure 12](#). Pre-training establishes the model’s foundation in general language capabilities, while Post-training enables the model to operate more accurately and efficiently in specific tasks and interactive scenarios.

Pre-Training Stages Pre-training is the core phase where large language models learn general language patterns and accumulate world knowledge. By learning from massive amounts of unlabeled data, the model acquires basic grammar, semantic understanding, and generation capabilities.

- **Data Collection and Processing** Data collection and processing form the foundational step of pre-training. Data collection requires broadly sourcing texts spanning multiple domains and types, such as general web text, books, academic literature, and specialized domain materials. Data processing involves cleaning and deduplicating the collected raw text, filtering out low-quality and sensitive content, tokenization (splitting text into model-recognizable tokens), format standardization, unified encoding, and adding special markers, thereby transforming it into an input form that the model can directly learn from. Data collection and processing typically utilize methods like web crawlers and integration of public datasets, employing NLP tools and custom scripts to complete the cleaning, tokenization, and other processing workflows.
- **PT (Pre-training Stage)** Pretraining refers to the process of training a model starting

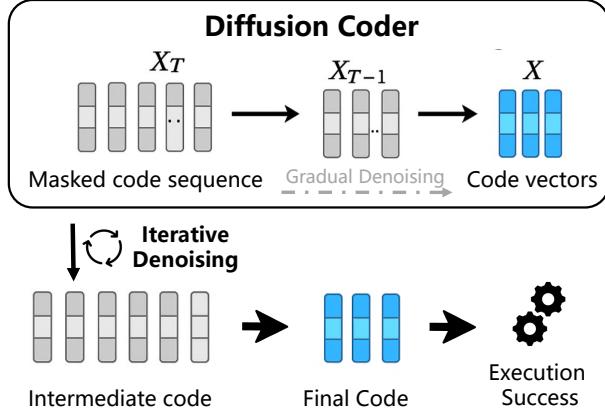


Figure 11. Overall architecture of Diffusion Coder.

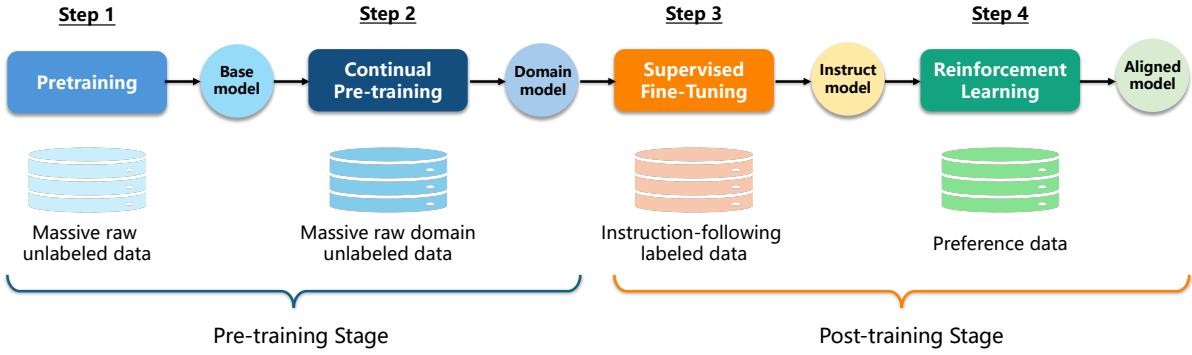


Figure 12. Overview of the model training stages.

from randomly initialized parameters, utilizing processed large-scale unlabeled corpora through self-supervised learning objectives. These objectives include autoregressive language modeling, which involves predicting the next token based on preceding context, or masked language modeling, which entails predicting randomly masked tokens within the input sequence. During the training process, the model progressively learns grammatical structures, lexical relationships, and extensive general world knowledge by continuously optimizing its parameters.

- **CPT (Continual Pre-training Stage)** CPT refers to the pre-training optimization task where, based on an existing pre-trained model, additional pre-training is performed using domain-specific or newly added corpora to continuously enrich the model's knowledge and enhance its performance. It is not training from scratch but rather a continuation of the pre-training process, often used for vertical domain adaptation or knowledge updates. The primary execution methods include freezing part of the lower-layer parameters or performing full fine-tuning, and continuing to perform autoregressive or masked language modeling tasks on the new corpus to learn new contextual patterns.
- **Annealing Strategy** The annealing strategy primarily involves the dynamic adjustment of training hyperparameters such as the learning rate. Typically, a larger learning rate is used in the early stages of training to accelerate convergence, followed by a gradual reduction in the learning rate for fine-tuning the model parameters. This helps prevent the model from struggling to converge to an optimal solution in the later stages due to an excessively large learning rate.

Table 3. Training phases of different Code LLMs.

Model	Pre-training				Post-training		
	PT	CPT	ANN	Repo PT	SFT	RL	Repo SFT
SantaCoder [39]	✓				✓		
OctoCoder [714]		✓			✓		
CodeLlama [858]		✓			✓		
StarCoder [562]	✓				✓		
CodeT5+ [1049]		✓			✓		
PanGu-Coder2 [885]		✓			✓	✓	
AlphaCode2 [950]		✓			✓		
WizardCoder [661]		✓			✓		
MFTCoder [598]		✓			✓		
WaveCoder [1229]		✓			✓		
DeepSeek-Coder [232]	✓			✓	✓		
StarCoder2 [226]	✓				✓		
CodeGemma [1295]		✓			✓		
Yi-Coder [2]		✓			✓		
Granite-Code [697]	✓				✓		
Qwen2.5-Coder [825]		✓		✓	✓		
DeepSeek-Coder-V2 [235]		✓			✓	✓	
Opencoder [424]	✓		✓		✓		
DeepSeek-V3 [238]		✓			✓	✓	
DeepSeek-R1 [238]		✓			✓	✓	
Ling-Coder [958]		✓	✓		✓		
Seed-coder [872]	✓		✓		✓	✓	
Qwen3-Coder [824]	-	✓	✓	✓	✓	✓	✓

PT: Pre-training. **CPT:** Continued pre-training. **ANN:** Annealing training. **Repo PT:** Repository-level pre-training. **SFT:** Supervised fine-tuning. **RL:** Reinforcement learning. **Repo SFT:** Repository-level supervised fine-tuning.

Post-training Stage Post-training is the crucial step following pre-training that enables the model to adapt to specific tasks and align with human interaction needs. It transforms the model from “understanding general language” [291] to “accurately completing specific tasks and generating content satisfactory to humans” [1177].

- **Supervised Fine-Tuning (SFT)** Supervised Fine-Tuning is a post-training method that uses manually annotated “input-output” data pairs to perform supervised training on the pre-trained model, enabling it to quickly adapt to specific tasks. This involves collecting annotated data for specific tasks (e.g., dialogue generation, text summarization, code generation), inputting this data into the pre-trained model, and adjusting the model parameters by optimizing the loss function to minimize the prediction error on the task data. This allows the pre-trained model to learn task-specific mapping relationships from this data, making its output more aligned with task requirements.
- **Reinforcement Learning (RL)** Reinforcement Learning is a post-training method that further optimizes the model’s output by incorporating human feedback or reward mechanisms, guiding the model to generate content that better aligns with human preferences. By introducing a reward signal, the model adjusts its behavior through interaction with specific task scenarios to maximize this signal, thereby producing higher-quality content. A common implementation is Reinforcement Learning from Human Feedback (RLHF).

The process typically involves first training a Reward Model (RM) using annotated data to quantify human preferences, then using reinforcement learning algorithms to optimize the generative model’s output under the guidance of the Reward Model.

2.3. Open-source Code Pre-training Data

The capabilities of open-source code LLMs are fundamentally shaped by their pre-training data. The evolution of these datasets shows a clear trend: a shift from prioritizing sheer volume to a rigorous focus on data quality, licensing, and governance. This maturation has been critical for the open-source community to build models that are not only powerful but also safer and more reliable. [Table 4](#) shows a few key datasets have defined this landscape.

Table 4. Open-source pre-training code datasets.

Dataset	Size	Languages	Source	Key Features
The Github Dataset				
The Stack v1 [492]	2.9 TB	358	GitHub repositories	Permissively licensed; two-stage deduplication (exact-match + MinHash LSH)
The Stack v2 [641]	32.1 TB	600+	Software Heritage, GitHub PRs/issues, Jupyter notebooks	4x larger than v1; formal opt-out process; industry standard
OpenCoder [424]	3.3 TB	13	GitHub,skypike cc,FineWeb AutoMathText	Complete Open Source Reproducible
Derived Datasets				
StarCoderData [562]	783 GB	86	The Stack v1.2	Additional decontamination; includes GitHub issues and commits; contamination-free
Other Foundational Datasets				
The Pile [308]	825 GB	30+	GitHub, Stack Exchange	EleutherAI; pioneering open reproducible dataset; foundation for GPT-Neo and Pythia
RedPajama [206]	120 GB	50+	GitHub	Permissive licenses only (MIT, BSD, Apache 2.0); LLaMA data replication
CodeParrot [279]	50 GB	1 (Python)	GitHub	Python-only; ~70% data removed via deduplication

2.3.1. The Github Datasets

The Stack dataset family, from the BigCode collaboration, is the most influential resource² for pre-training code LLMs.

The Stack v1 [492]. In the first version, The Stack v1 was a landmark, providing 3.1 TB of permissively licensed source code across 358 programming languages after extensive deduplication. Its creation set a vital precedent by sourcing exclusively from GitHub repositories with licenses permitting redistribution, addressing key legal concerns in open-source development. The data pipeline was notable for its two-stage deduplication strategy, combining exact-match hashing

²<https://huggingface.co/datasets/bigcode/the-stack>

with a MinHash LSH algorithm for near-deduplication, a technique proven to significantly boost model performance.

The Stack v2 [641]. Building on this, The Stack v2 represents the current industry standard. It expands the data scale fourfold to approximately 900 billion tokens. Its sources are more diverse, drawing primarily from the Software Heritage archive and supplemented with GitHub pull requests, issues, and Jupyter notebooks. The Stack v2 also increased coverage to over 600 languages and introduced a crucial governance mechanism: a formal opt-out process, allowing developers to have their code removed and ensuring the dataset is periodically updated to respect these requests.

2.3.2. *StarCoderData*

StarCoderData [562]. Derived from The Stack, the StarCoderData dataset was curated specifically for training the StarCoder model series. While originating from The Stack v1.2, it underwent an additional, cleaning and decontamination phase to filter against common evaluation benchmarks, preventing data leakage and ensuring fair model assessment. Totaling 783 GB, it provides a rich mix of source code across 86 languages, augmented with contextual data from GitHub issues and commits, making it a benchmark for high-quality, contamination-free pre-training.

2.3.3. *Others*

Beyond these large-scale multilingual datasets, several others have been instrumental.

The Pile [308]. The Pile, an 825 GB collection from EleutherAI, is a dedicated effort that included significant code from GitHub and Stack Exchange. It is one of the first large-scale, open, and reproducible datasets, serving as the foundation for influential early models like GPT-Neo and Pythia.

RedPajama [206]. Similarly, the code portion of RedPajama-Data-1T is a vital contribution, created as an open-source replication of the data used to train the original LLaMA models. Its 59-billion-token code slice was carefully filtered to include only projects with highly permissive licenses (MIT, BSD, Apache 2.0), making it a valuable resource for training models in the LLaMA architectural family without legal ambiguity.

CodeParrot [279]. Language-specific datasets have also proven highly effective. CodeParrot, for instance, is a high-quality dataset focused exclusively on Python. Its processing revealed the high degree of duplication in public code repositories, with its pipeline removing nearly 70% of the raw data, underscoring the critical need for this step in creating efficient training corpora.

Collectively, the trajectory of these open-source datasets highlights a maturation in data engineering. The community has moved from raw data collection to implementing sophisticated, transparent, and reproducible pipelines for filtering, deduplication, PII redaction, and license verification. These high-quality, well-governed datasets have become the bedrock upon which the modern ecosystem of open-source Code LLMs is built.

2.4. Future Trends

Based on the comprehensive evolution of code foundation models above, we identify three key trends that will likely shape the future landscape of code intelligence:

From General to Specialized Code Intelligence. The trajectory from general-purpose LLMs to dedicated code specialists represents a fundamental shift in model development philosophy. While early models like GPT-3 [244] demonstrated surprising code competence as an emergent capability, the success of specialized systems like GPT-5-Codex [755], and Claude-4 [192]’s coding variants illustrates the substantial gains achievable through domain-specific optimization. We anticipate continued differentiation between general conversational AI and purpose-built coding assistants, with the latter achieving superior performance on repository-level tasks, complex debugging scenarios, and multi-step software engineering processes.

Agentic Training and Complex Scenario Mastery. The emergence of agentic code models represents a paradigm shift from passive code generation to active software engineering. Future developments will likely emphasize training methodologies that enable models to operate autonomously across complex, multi-step programming scenarios. This includes reinforcement learning from execution feedback, curriculum learning on progressively complex repository-level tasks, and integration with external tools and environments. Models will be trained not just to write code, but to understand project contexts, navigate codebases, execute iterative debugging cycles, and collaborate with human developers through extended interactions. The success of systems like SWE-Bench[928] agents suggests that future code LLMs will increasingly function as autonomous software engineers capable of end-to-end problem solving rather than mere code completion tools.

Scaling Laws and Scientific Model Development. The application of scaling laws to code model development will drive more systematic and efficient training strategies. Unlike the early era of empirical model scaling, future code LLMs will leverage principled understanding of how model performance scales with parameters, training data, and compute resources specifically for coding tasks [657]. This scientific approach will inform optimal resource allocation between model scale, data curation quality, and training duration. Additionally, scaling laws will guide the development of mixture-of-experts architectures optimized for code tasks, enabling models to achieve superior performance while maintaining computational efficiency. We expect future code models to be developed through data-driven optimization of the scaling trade-offs unique to programming domains, leading to more capable and cost-effective systems.

These trends collectively point toward a future where code foundation models evolve into sophisticated, specialized systems that combine deep programming expertise with autonomous problem-solving capabilities, developed through scientifically principled scaling strategies that maximize both capability and efficiency.

3. Code Tasks, Benchmarks, and Evaluation

Figure 13 presents a hierarchical taxonomy of coding tasks and benchmarks. The tree organizes the landscape along three major granularities: (i) statement, function, and class-level tasks; (ii) repository-level tasks; and (iii) agentic systems. At the statement/function/class level, the taxonomy covers completion and fill-in-the-middle, code generation, editing and bug fixing, code efficiency, code preference, reasoning and question answering, code translation, and test-case generation, with representative benchmarks listed at the leaves. The repository-level branch groups benchmarks for generation and completion in larger codebases, domain-specific and complex code, code editing/refactoring and agent collaboration, commit message generation, software engineering (SWE) task resolution, and comprehensive software development work-

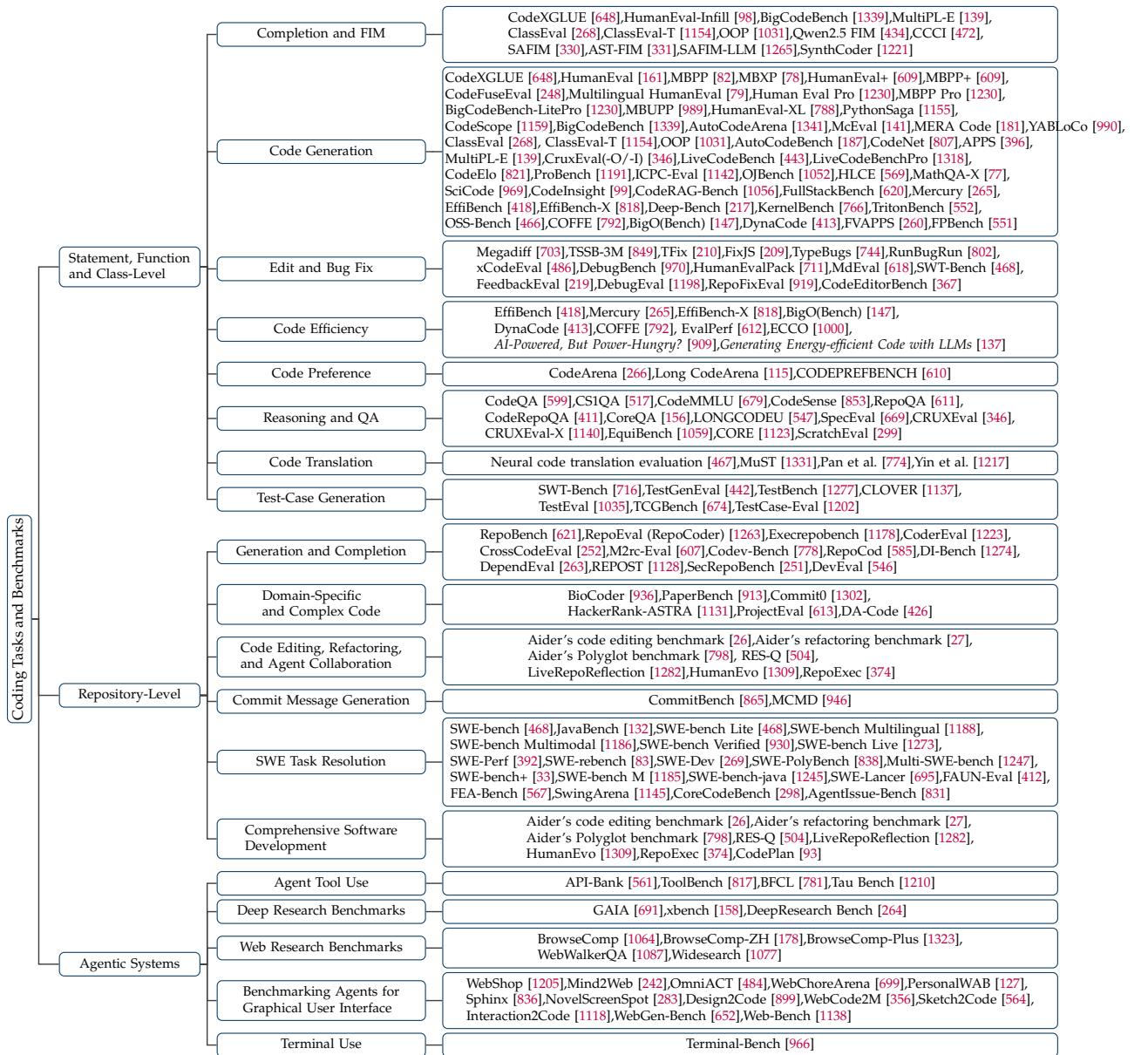


Figure 13. A Taxonomy of coding tasks and benchmarks.

flows. The agentic systems branch highlights benchmarks for tool use, deep and web research, graphical user interface interaction, and terminal use. Together, the leaves enumerate both widely used datasets and recent additions, providing a consolidated map of current evaluation resources across capabilities and scales. [subsection 3.1](#) first introduces the evaluation metrics for code LLMs and then [subsection 3.2](#) describes code tasks across three major granularities across diverse code llm tasks like code generation, code completion, code edit, code efficiency and code preference, code translation, and test case generation, while [subsection 3.3](#) introduces repo-level code benchmarks. Finally, [subsection 3.4](#) presents advanced code benchmarks, such as agent tool use, deep research, web search and terminal use.

3.1. Evaluation Metrics

[Table 5](#) lists the evolution from the string-matching methods to the execution-based methods. CodeBLEU [845] measures syntax-level similarity using n -gram matching and abstract tree parsing but fails to assess functional correctness. Pass@ k then emerges as an execution-based metric, testing whether code passes predefined test cases across k attempts. Most recently, LLM-based code judgment [1177] leverages LLMs for comprehensive quality assessment, including correctness, efficiency, and readability beyond simple pass/fail outcomes.

3.1.1. Extensions Based on Traditional Metrics

- CodeBLEU [845] borrows BLEU [779, 1176] from machine translation, rewarding n -gram overlap between the model output and a single reference. That helps track superficial similarity but struggles with code with small lexical changes. And there are many correct implementations that share few tokens. It extends BLEU [779] by incorporating n -gram matching, abstract syntax tree (AST) matching, and data flow semantic matching to provide a more comprehensive code quality assessment.
- CodeBERTScore [1324] is a pre-training model-based code generation evaluation metric that assesses generation quality by encoding natural language context and code snippets to calculate semantic similarity. Round-trip correctness (RTC) [45] is an unsupervised evaluation method for LLMs that verifies semantic consistency through the model’s own bidirectional tasks (e.g., code translation), avoiding reliance on manually annotated data. $RTC_{sim}(x)$ in [Table 5](#) measures the semantic similarity between original code x and round-trip generated code \hat{x} .
- Pass@ k [161, 500] in [Table 5](#) sample k completions for each task and score the chance that at least one passes the unit tests. pass@ k directly measures functional correctness, captures the benefit of sampling, handles multiple valid solutions, and tends to correlate better with developer utility, making it the de facto standard for modern LLM code generation benchmarks.

3.1.2. LLM-as-a-Judge Paradigm

- ICE-Score [1335] is the first evaluation framework that instructs LLMs to rate code quality using structured, criteria-driven scoring rubrics. In contrast to test-based evaluation, ICE-Score leverages rubric-guided LLM judgments to assess dimensions such as *usefulness* and *functional correctness*. In the formula presented in [Table 5](#), the evaluation function f takes as input the task description p and the generated code r , and outputs a discrete score $S \in \{0, 1, 2, 3, 4\}$ based on predefined evaluation criteria. Higher scores indicate better semantic alignment with the intended functionality. In a continuous adaptation, this score may be normalized as $S/4$, where values closer to 1 denote stronger correctness.
- CodeJudge [978] is a code evaluation framework built upon ICE-Score, enhancing evaluation reliability through a slow-thinking mechanism. In the formula presented in [Table 5](#), f denotes the evaluation function, which outputs either a binary judgment (correct/incorrect) or a deviation score ranging from 0 to 1 points. This function in the continuous setting, lower deviation scores indicate smaller semantic errors (i.e., better alignment with the reference).
- LLM-as-a-Judge [389] is a method that uses LLMs to directly evaluate code quality. In the formula presented in [Table 5](#), src represents the task source instruction, $target$ denotes the code/text to be evaluated, and the LLM directly outputs a score. The primary advantage of this approach is its ability to leverage LLMs for analyzing deep semantic aspects of code

functionality. However, its limitation lies in unstable performance, attributed to LLMs' propensity for generating verbose explanations.

- CodeJudgeBench [456] adopts the LLM-as-a-Judge paradigm, where an LLM directly evaluates the functional correctness of code responses without executing the code. The core formula of the framework, presented in [Table 5](#), involves concatenating p (the programming task description), r (the response to be evaluated), and q (the model's judging instruction), after which the LLM outputs J to represent the judgment result.
- BigCodeReward [1341] is the first benchmark designed for evaluating reward models in practical coding scenarios. Unlike prior reward-model benchmarks that focus on general domains, BigCodeReward targets code-specific evaluation by incorporating real-world execution feedback, including textual logs, webpage screenshots, interactive application states, and plots. In the formulation shown in [Table 5](#), the evaluation function f takes as input a coding prompt p , two candidate responses (r_A, r_B), and optional execution feedback e , and the model outputs a preference label $J \in \{A, B, \text{Tie}\}$. BigCodeReward operates entirely within the LLM-as-a-Judge paradigm, requiring models to directly judge which response better satisfies user intent, both with and without execution results. This design enables assessment of whether reward models can reliably leverage multimodal execution signals that extend beyond pure text.

Table 5. Summarized metrics for code evaluation.

Metrics	How to calculate	Explanation
Extensions Based on Traditional Metrics		
Biased pass@k [161]	$1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$	$n = \text{total generation times}$, $c = \text{correct generation times}$, $k = \text{Number of samples from all } n \text{ generation without replacement}$.
Unbiased pass@k [500]	$1 - \left(\frac{n-c}{n}\right)^k$	$n = \text{total generation times}$, $c = \text{correct generation times}$, $k = \text{Number of samples from all } n \text{ generation with replacement}$.
CodeBERTScore [1324]	$P = \frac{1}{ \hat{y} } \sum_j \max_i \text{sim}(y_i^*, \hat{y}_j)$ $R = \frac{1}{ y^* } \sum_i \max_j \text{sim}(y_i^*, \hat{y}_j)$ $F_1 = \frac{2PR}{P+R}, F_3 = \frac{10PR}{9P+R}$	$\hat{y} = \text{generated tokens}$, $y^* = \text{reference tokens}$, $\text{sim}(\cdot, \cdot) = \text{embedding similarity}$, $P, R = \text{precision, recall}$; $F_1, F_3 = \text{harmonic f-scores}$
RTC [45]	$RTC_{sim}(x) \approx \frac{1}{N_f N_b} \sum_{y \sim M(x)} \sum_{\hat{x} \sim M^{-1}(y)} \text{sim}(\hat{x}, x)$	$N_f, N_b = \# \text{forward/backward samples}$, $M, M^{-1} = \text{round-trip models}$, sim denotes code similarity
LLM-as-a-Judge Paradigm		
ICE-Score [1335]	$f(\text{Score, Task})$	$f = \text{scoring function}$, $\text{Score} = \text{numeric rating}$, $\text{Task} = \text{problem description}$
CodeJudgeBench [456]	$J \leftarrow \text{LLM}(p \oplus r \oplus q)$	$J = \text{judgment score}$, $p = \text{prompt template}$, $r = \text{reference solution}$, $q = \text{model output}$, $\oplus = \text{text concatenation}$
BigCodeReward [1341]	$J \leftarrow f(p, r_A, r_B, e)$	$J \in \{\text{A, B, Tie}\} = \text{preference judgment}$, $p = \text{task description}$, $r_A, r_B = \text{candidate responses}$, $e = (\text{optional}) \text{execution feedback}$, $f = \text{LLM-as-a-Judge evaluator}$
Execution-Based Metrics		
ProbeGen [43]	$\exists p_k : p_k(f_i) \neq p_k(f_j) \Rightarrow f_i \neq f_j$	$p_k = \text{test probe input}$, $f_i, f_j = \text{two candidate functions}$, $p_k(f) = \text{function output on probe}$
REFUTE [906]	$H(x^*) \cap \neg P(x^*)$	$x^* = \text{denotes input}$, H represents the first presupposed hypothesis, and P represents the second hypothesis.
EvaLooop [284]	$ASL = \frac{\sum_{i=1}^M n_i i^2}{M \cdot N}$	$n_i = \# \text{tests with } i \text{ iterations}$, $i = \text{loop-iteration count}$, $M = \# \text{distinct } i \text{ values}$, $N = \text{total test cases}$
Multi-Agent & Advanced Reasoning Framework		
MCTS-Judge [1050]	$Reward = \varepsilon \text{ if } f(t, g) = h(x)$	$\varepsilon = \text{small reward constant}$, $h(x) = \text{expected output for input } x$ $f(t, g) = \text{model's predicted output given trace } t \text{ and goal } g$
Statistical & Consistency Analysis Metrics		
Incoherence [991]	$I_{\text{Gen}}(d) = P([\Pi_1^d]^{1(X)} \neq [\Pi_2^d]^{1(X)})$	$d = \text{decoding depth}$, $P = \text{probability}$, $\Pi_1^d, \Pi_2^d = \text{two runs sampling depth } d$, $[\cdot]^{1(X)} = \text{first token on input } X$,
MAD [704]	$\frac{1}{K} \sum_{k=1}^K \text{Acc}_{\text{org}}^{(k)} - \text{Acc}_{\text{bias}}^{(k)} $	$K = \# \text{of trials}$, $\text{Acc}_{\text{bias}}^{(k)} = \text{accuracy with biased prompt}$ $\text{Acc}_{\text{org}}^{(k)} = \text{accuracy on trial } k \text{ with original prompt}$
Other Unique Paradigms		
SBC [799]	$(0.7 \times \text{semantic}) + (0.1 \times \text{BLEU}) + (0.2 \times \text{completeness})$	$\text{semantic} = \text{semantic-similarity score}$, $\text{BLEU} = \text{n-gram precision}$, $\text{completeness} = \text{test-coverage measure}$
Copilot Arena [183]	$\text{Rank} = \text{BradleyTerry}(\text{User Votes})$	$\text{Rank} = \text{model's overall rank}$, $\text{User Votes} = \text{pairwise preferences aggregated via Bradley-Terry}$
BigCodeArena [1341]	$\text{Rank} = \text{BradleyTerry}(\text{User Votes})$	$\text{Rank} = \text{model's overall rank}$, $\text{User Votes} = \text{pairwise preferences aggregated via Bradley-Terry}$
CodeCriticBench [1258]	$MSE = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2$	$N = \text{number of samples}$, $\hat{y}_i = \text{predicted rating}$, $y_i = \text{true rating}$

3.1.3. Execution-Based Metrics

- ProbeGen [43] leverages LLMs to generate targeted test cases (probes) that verify code semantic equivalence through execution feedback. Its key insight is captured by the formula in Table 5. when a probe p_k causes different implementations, f_i and f_j , to produce divergent outputs, their functional inequivalence is immediately established. REFUTE [906] is an evaluation framework specifically designed to assess the ability

of large language models (LLMs) to find counterexamples. Its core concept involves prompting models to generate test cases that cause buggy code to fail. Its core formula is presented in [Table 5](#), where x^* denotes the counterexample input, H represents the constraint, and P stands for the expected correctness of the buggy code.

- EvaLoop [284] introduces a self-contained feedback-loop evaluation framework to assess the robustness of LLMs in programming. The framework leverages two dualities to repeatedly transform the model’s outputs back into new prompts until functional correctness fails, including (a) code generation↔code summarization and (b) cross-language code translation. Its core metric, average sustainable loops (ASL), is defined in [Table 5](#), where M denotes the maximum loop count, n_i the number of tasks still passing on loop i , and N the total number of tasks. A higher ASL indicates stronger semantic consistency and robustness across iterations.

3.1.4. Multi-Agent & Advanced Reasoning Framework

- MCTS-Judge [1050] is a test-time computation framework for code correctness evaluation that enables LLMs to conduct multi-perspective reasoning through monte carlo tree search (MCTS). Each node in the search process represents a distinct reasoning perspective, such as boundary condition analysis, exception handling, or specification compliance. In [Table 5](#), $f(t, g)$ denotes the aggregated prediction along the reasoning trajectory, $h(x)$ indicates the verification result from simulated test execution, and ϵ represents the reward assigned when the prediction aligns with the verification outcome.
- CodeVisionary [1038] constructs a two-stage evaluation framework. The first stage collects comprehensive information through multi-source knowledge analysis, while the second stage employs a multi-LLM collaborative scoring mechanism. In [Table 5](#), S_i represents the consensus score from multi-LLM collaboration and n denotes the number of LLMs involved.

3.1.5. Statistical & Consistency Analysis Metrics

- Incoherence [991] quantifies the uncertainty of LLMs in code generation tasks and measures the probability that two independently generated programs for the same problem produce different outputs on identical inputs. Higher incoherence values reflect increased stochasticity and reduced consistency in model reasoning, whereas lower values imply more deterministic and consistent behavior. In [Table 5](#), computes the divergence rate across program pairs $[\Pi_1^d]$ and $[\Pi_2^d]$ sampled for task d over input distribution $\text{Gen}(d)$.
- Mean absolute deviation (MAD) [704] evaluates the robustness of LLM-based evaluators under superficial perturbations and measures the deviation between the original evaluation accuracy (Acc_{org}) and the accuracy after introducing K systematic biases (Acc_{bias}), such as variable renaming, comment injection, or code formatting changes. A smaller MAD indicates higher evaluator consistency and less sensitivity to presentation-level noise.

3.1.6. Other Unique Paradigms

- The core innovation of the SBC [799] (Semantic similarity score: measuring meaning-level alignment between requirements, BLEU score: capturing lexical overlap, and Completeness score: identifying missing and extra elements) metric lies in its reverse generation technique: instead of directly evaluating code, it leverages LLMs to reverse-engineer requirements from the generated code and then compares these reverse-engineered requirements with the original ones. Its core calculation formula is presented in [Table 5](#), where *Semantic*

refers to cosine similarity, and *Completeness* is based on penalizing missing keywords and redundancy extracted from nouns and verbs.

- Copilot Arena [183] and BigCodeArena [1341] are platforms that collect user preference votes for LLM-generated code in real developer environments and computes the relative win rates of models using the Bradley-Terry model (a probability model that estimates the likelihood of one item being preferred over another in a series of pairwise comparisons).
- CodeCriticBench [1258] is a comprehensive code critique benchmark covering two main tasks (code generation and code QA) and structured into two dimensions: basic evaluation and advanced evaluation. The basic evaluation focuses on binary classification of correctness (ACC metric), while the advanced evaluation uses multi-dimensional fine-grained checklists for scoring with MSE metric (mean squared error).

3.2. Statement, Function, and Class-Level Tasks and Benchmarks

This section presents statements, function, and class-level code benchmarks across different tasks, such as code completion, FIM, generation, editing, and bug-fixing.

3.2.1. Code Completion and Code FIM

This section reviews benchmarks for code completion and Fill-in-the-Middle (FIM) tasks, which assess a model’s ability to predict correct code fragments in partially observed contexts. Unlike function-level generation from prompts or docstrings, code completion focuses on continuing or repairing existing code, often using local context. FIM further generalizes this by requiring the prediction of missing segments within a sequence, reflecting real-world scenarios such as IDE autocompletion, refactoring, and bug fixing. [Table 6](#) list some code completion benchmarks.

Table 6. Code Completion and Fill-in-the-Middle Benchmarks Overview.

Benchmark	Year	Granularity	Languages	Size (K)	Evaluation
Code Completion Benchmarks					
CodeXGLUE [648]	2021	Statement	3+	104K	Exact Match
HumanEval-Infill [98]	2022	Function	Python	164	Unit Tests
ExecRepoBench [1179]	2024	Statement	Python	1.2K	Tests+Accuracy
BigCodeBench [1339]	2024	Function	3+	1.1k	Functional Tests
MultiPL-E [139]	2022	Function	3+	12.7K	Diversity
ClassEval [268]	2024	Class	Python	100/412	Unit Tests
ClassEval-T [1154]	2024	Class	Java, C++	94	Unit Tests
OOP [1031]	2024	Class	Python	431	OOP Patterns
Fill-in-the-Middle (FIM) Benchmarks					
CCCI [472]	2025	Statement	Java	289	Contextual Quality
SAFIM [330]	2024	Syntax-aware	3+	17,720	Syntax Correctness
AST-FIM [331]	2025	AST-based	12	30k+	Structural Accuracy

Code Completion

- **Statement-level** completion evaluates the prediction of individual lines or expressions based on preceding context. CodeXGLUE [648] includes a next-line completion task across

six programming languages, using exact match accuracy as the primary metric. However, it lacks functional validation. More rigorous benchmarks like HumanEval-Infill [98] adapt function-level datasets by treating docstring-conditional body generation as a completion task, with correctness determined via unit test execution. Qwencoder2.5 [434] further refines this with a dedicated base completion benchmark that tests Python statement prediction within functions, measuring both lexical accuracy and test-based correctness under varying context scopes.

- **Function-level** completion involves generating full implementations from signatures or partial bodies. BigCodeBench [1339] includes function recovery tasks where models complete stubs extracted from real repositories, evaluated on functional correctness and complexity preservation. MultiPL-E [139] extends this by requiring multiple valid solutions per prompt, probing a model’s generative diversity—a key aspect of realistic completion behavior. Class-level completion introduces challenges in modeling stateful interactions and method dependencies. ClassEval [268] requires models to implement individual methods within a partially defined class structure, assessing coherence with existing logic through unit tests. Results show significant performance drops compared to function-level tasks, especially for smaller models. Its multilingual extension, ClassEval-T [1154], adds Java and C++ variants and identifies common errors such as incorrect field initialization and broken inter-method dependencies. OOP [1031] complements these by evaluating object-oriented design patterns during method completion, including proper use of inheritance and encapsulation.

Fill-in-the-Middle (FIM) FIM specifically targets infilling missing code segments given both left and right context, simulating real editing and refactoring scenarios. While originally used as a pretraining objective (e.g., in StarCoder [642]), systematic FIM evaluation has only recently emerged. CCCI [472] improves code completion by incorporating contextual information like database relationships and object models into LLMs, achieving higher functional correctness and code quality. SAFIM [330] introduces a syntax-aware FIM benchmark across multiple programming languages, demonstrating that pretraining strategies and data quality significantly affect FIM performance. AST-FIM [331] leverages abstract syntax trees to mask syntactic structures during pretraining, improving model performance on real-world FIM tasks.

3.2.2. Code Generation

In Table 7, this section surveys code generation benchmarks, organized along different granularities (function-level, class-level, and domain-specific tasks) and extended dimensions such as efficiency and verification.

Table 7. Code generation Benchmarks Overview.

Benchmark	Year	Size	Languages	Domain	Difficulty
Function-Level Benchmarks					
CodeXGLUE [648]	2021	104k	Java	General	
HumanEval [161]	2021	164	Python	Algorithmic	
MBPP [82]	2021	974	Python	Intro Coding	
MBXP [78]	2022	12.4k	Multi-PLs	Intro Coding	
HumanEval+ [609]	2023	164	Python	Algorithmic	
MBPP+ [609]	2023	378	Python	Algorithmic	
CodeFuseEval [248]	2023	820	Multi-PLs	General	

Continued on next page

Table 7 continued from previous page

Benchmark	Year	Size	Languages	Domain	Difficulty
Multilingual HumanEval [79]	2023	1.9K	Python, Java, JS	Algorithmic	
HumanEval Pro [1230]	2024	164	Python	Algorithmic	
MBPP Pro [1230]	2024	378	Python	Algorithmic	
BigCodeBench-LitePro [1230]	2024	57	Multi-PLs	General	
MBUPP [989]	2024	466	Python	Algorithmic	
HumanEval-XL [788]	2024	22k	Multi-PLs&NLs	Cross-lingual	
PythonSaga [1155]	2024	185	Python	Library/API	
CodeScope [1159]	2024	400	Multi-PLs	Control Flow	
BigCodeBench [1339]	2024	1.1k	Python	General	
McEval [141]	2025	12k	Python	Mathematical	
MERA Code [181]	2025	1.3k	Python, Java	Systems	
YABLoCo [990]	2025	208	C, C++	Systems	
IFEvalCode [1180]	2025	1.6K	Multi-PLs	Systems	
Class-Level Benchmarks					
ClassEval [268]	2024	100	Python	OOP	
ClassEval-T [1154]	2024	94	Python, Java, C++	OOP	
OOP [1031]	2024	431	Python	OOP	
utoCodeBench [187]	2025	3.9k	Multi-PLs	General	
Competition Benchmarks					
CodeNet [807]	2021	14M+	Multi-PLs	Cross-lingual	
APPS [396]	2021	10k+	Python	Algorithmic	
Multip-E [139]	2022	12.7k	Multi-PLs	Cross-lingual	
CruxEval(-O / -I) [346]	2024	800	Multi-PLs	Algorithmic	
LiveCodeBench [443]	2024	121	Multi-PLs	Dynamic	
LiveCodeBenchPro [1318]	2024	584	C++	Dynamic	
CodeElo [821]	2025	387	C++, Python	Algorithmic	
ProBench [1191]	2025	790	C++, Java, Python	Cross-lingual	
ICPC-Eval [1142]	2025	118	C++	Algorithmic	
OJBench [1052]	2025	232	C++, Python	Algorithmic	
HLCE [569]	2025	235	C++, Python	Algorithmic	
Other Domain-Specific Benchmarks					
MathQA-X [77]	2022	5.6k	Python, Java, JS	Mathematical	
SciCode [969]	2024	338	Python	Scientific	
CodeInsight [99]	2024	3.4k	Python	Hybrid	
CodeRAG-Bench [1056]	2024	2.1k	Python	RAG	
FullStackBench [620]	2024	3K	Multi-PLs	Web	
Mercury [265]	2024	1.9k	Multi-PLs	Efficiency	
EffiBench [418]	2024	1k	Multi-PLs	Efficiency	
EffiBench-X [818]	2025	623	Multi-PLs	Efficiency	
Deep-Bench [217]	2025	520	Python	Deep Learning	
KernelBench [766]	2025	250	Python	GPU kernel	
TritonBench [552]	2025	184	Triton	GPU kernel	
AutoCodeArena [1341]	2025	600	Multi-PLs	Hybrid	
OSS-Bench [466]	2025	17.9k	PHP, SQL	Software	
COFFE [792]	2025	756	Python	Efficiency	
BigO(Bench) [147]	2025	3.1k	Python	Complexity	
DynaCode [413]	2025	405	Python	Complexity	
FVAPPS [260]	2025	4.7k	Python	Verification	
FPBench [551]	2025	1.8k	Python	Verification	

Function-Level Benchmarks Early benchmark suites such as CodeXGLUE [648] includes code generation tasks (e.g., NL2Code, code-to-code translation), which are primarily formulated at the function or statement level. Function-level evaluation was further developed by Hu-

manEval [161] and MBPP [82], which target Python code synthesis from docstrings using pass/-fail outcomes on unit tests as correctness criteria. To address limited test coverage, EvalPlus [609] introduces strengthened variants—HumanEval+ and MBPP+—that incorporate adversarially generated and hidden test cases, and automatically synthesize more comprehensive test suites with robust metrics. Building on this line, HumanEval Pro, MBPP Pro, BigCodeBench-Lite Pro [1230], and MBUPP [989] extend classical settings with self-invoking code execution, larger problem pools, and one-to-many evaluation. Multi-lingual evaluation expands the scope beyond Python. MBXP [78] and Multilingual HumanEval [79] provide function-level coding problems across multiple programming languages, HumanEval-XL [788] scales this to 22,080 problems in 23 natural languages and 12 programming languages, and PythonSaga [1155] offers a curated Python set to assess generation quality under diverse problem types. CodeScope [1159] adds execution-based feedback for multilingual code generation, while MERA Code [181] introduces a systematic, cross-granularity benchmarking framework spanning function-level, algorithmic, and system-level tasks. McEval [141] focuses on method-level code completion in large codebases, emphasizing contextual coherence. BigCodeBench [1339] samples from real-world repositories to assess functional correctness and code complexity. CodeFuseEval [248] further evaluates models on multi-task scenarios including code completion, cross-language translation, and code generation from Chinese commands.

Class-Level Benchmarks Beyond single-function tasks, ClassEval [268] introduces a manually curated benchmark of 100 Python classes (412 methods) that require modeling inter-method dependencies and stateful behavior. Results across 11 LLMs show significant performance drops compared to function-level tasks. Its multilingual extension, ClassEval-T [1154], adds Java and C++ variants and provides a fine-grained failure taxonomy, including dependency and initialization errors. OOP [1031] further targets object-oriented programming in Python, complementing ClassEval benchmarks by evaluating class-level design, method interactions, and stateful behavior.

Domain-Specific Benchmarks To better reflect domain-specific requirements, specialized benchmarks have emerged. For competitive programming, CodeNet [807] provides a large-scale, multilingual dataset with performance metadata, supporting diverse evaluation scenarios. APPS [396] is a widely used benchmark featuring programming competition problems and human-written solutions. MultiPL-E [139] assesses solution diversity by requiring models to generate multiple distinct correct implementations per problem, thereby probing deeper algorithmic reasoning rather than mere pattern matching. CruxEval [346], comprising CRUXEval-I and CRUXEval-O, curates problems from online judges that emphasize reasoning and execution, enabling evaluation of both input and output prediction to assess code understanding and runtime behavior. LiveCodeBench [443] evaluates performance on LeetCode problems with temporal alignment, capturing the evolution of model capabilities over time and in relation to human solution trends. Several follow-up benchmarks extend this paradigm: CodeElo [821] introduces Elo-style ranking for competition-level problems, while ProBench [1191], ICPC-Eval [1142], OJBench [1052], LiveCodeBench Pro [1318], and HLCE [569] probe increasingly difficult Olympiad- and ICPC-level challenges with human baselines, highlighting the gap between LLMs and expert programmers. In **other specialized domains**, benchmarks target diverse tasks: Deep-Bench [217] focuses on code generation for deep learning frameworks; MathQA-X [77] provides a domain-specific evaluation for mathematical coding problems; FullStackBench [620] is a multilingual full-stack programming benchmark covering multiple domains and difficulty levels with reference solutions and automated correctness tests; Ker-

nelBench [766] and TritonBench [552] probe low-level GPU kernel and operator generation; YABLoCo [990] stresses long-context code generation beyond typical function-level inputs; SciCode [969] evaluates scientific coding tasks curated by researchers; CodeInsight [99] collects practical coding solutions from Stack Overflow; AutoCodeBench [187] and OSS-Bench [466] propose meta-benchmarking frameworks that automatically construct new evaluation datasets; and retrieval-augmented generation benchmarks such as CodeRAG-Bench [1056] highlight the importance of contextual grounding for realistic evaluation.

3.2.3. Code Edit and Bug Fix

Code edit and bug fix are the process of modifying the source code of a software program to correct errors, malfunctions, or unexpected behaviors, known as “bugs”. This is a fundamental and continuous activity in the software development lifecycle, aimed at improving the stability, reliability, and correctness of an application. [Table 8](#) lists most popular bug fixing benchmarks.

Table 8. Code Editing and Bug Fixing Benchmarks Overview.

Benchmark	Year	Language	Size	Source	Key Feature
Statement-Level Bug Fixing					
NL2SQL-BUGs [626]	2025	SQL	2K	Curated (expert-annotated)	Semantic error detection for Text-to-SQL (9 main + 31 subcategories)
Megadiff [703]	2021	Java	663K	VCS Commits	Real-world changes
TSSB-3M [849]	2022	Python	3M	Synthetic	Mutation-based
FixJS [209]	2022	JavaScript	324K	GitHub	Large-scale patches
PyTer [744]	2022	Python	93	Curated	Type errors
RunBugRun [802]	2023	Multi-PLs	450K	GitHub	Executable pairs
xCodeEval [486]	2023	Multi-PLs	4.7M	Multilingual	Cross-lingual repair
DebugBench [970]	2024	C++/Java/Python	4.2K	Curated	Controlled debugging
HumanEvalPack [711]	2023	Multi-PLs	984	HumanEval	Multilingual debug
MdEval [618]	2024	Multi-PLs	3.5K	Instruction	18 languages
Interactive and Feedback-Based Repair					
SWT-Bench [468]	2024	Python	1.9K	GitHub	Test-driven repair
FeedbackEval [219]	2025	Python	394	Synthetic	Iterative refinement
DebugEval [1198]	2024	Python	4253	Curated	Self-debugging
CodeEditorBench [367]	2024	Multi-PLs	1216	IDE simulation	Incremental editing

Statement-Level Benchmarks These benchmarks evaluate LLMs on localized bug fixes, typically at the statement or function level. Early datasets are primarily derived from version control commit histories: Megadiff [703] aggregates 663K Java changes mined from real-world repositories, while TSSB-3M [849] provides 3M synthetic single-statement bugs in Python, systematically generated by mutation operators. Beyond commit logs, several resources emphasize specific bug types or languages: TFix [210] applies sequence-to-sequence repair to 105K JavaScript fixes paired with unit tests; FixJS [209] expands this direction with 324K JavaScript patches from GitHub, and PyTer [744] targets Python type errors, offering controlled benchmarks for static typing issues. Later efforts improve “scale, executability, and multilinguality”: RunBugRun [802] contributes 450K executable bug-fix pairs across eight languages (e.g., Java, Python,

C++), enabling end-to-end validation. xCodeEval [486] scales to 4.7M multilingual samples (10+ languages) spanning understanding, generation, and repair. DebugBench [970] provides 4,253 curated debugging tasks in C++, Java, and Python with accompanying tests, emphasizing controlled evaluation. More recent datasets integrate debugging into instruction tuning: HumanEvalPack [711] extends HumanEval to six languages with debugging variants, and MdEval [618] introduces 3,513 debugging tasks across 18 languages, highlighting multilingual generalization.

Interactive and Feedback-Based Benchmarks Recent benchmarks move beyond one-shot fixes toward multi-step, context-sensitive repair with test feedback, self-refinement, or IDE-like interactions. SWT-Bench [468] collects 1,900 real GitHub bugs with executable test suites, showing how model-generated tests can validate candidate patches. FeedbackEval [219] systematically studies iterative repair with structured external hints, finding that feedback improves performance but with diminishing returns over multiple refinement cycles. DebugEval [1198] explores self-debugging, where models generate internal diagnostic signals to guide multi-step patching. Finally, CodeEditorBench [367] simulates IDE-like incremental editing, assessing behaviors such as local modification, error propagation, and interactive change application. Together, these benchmarks emphasize repair as an iterative process requiring state tracking, execution feedback, and realistic editing workflows, moving beyond isolated bug-fix pairs. Debug-gym [1235] develops a gym environment that equips agents with debugger tools (like pdb) to teach LLMs to use stateful tools through SFT/RL, addressing their scarcity in pre-training data.

3.2.4. *Code Efficiency*

Code efficiency task is dedicated to evaluating and optimizing the performance of large language model generated code across multi dimensional efficiency metrics, including runtime, memory usage, algorithmic complexity, and energy consumption, aiming to generate code that is not only functionally correct but also resource-efficient.

Performance and Complexity Benchmarks Recent work has moved beyond functional correctness to systematically examine the runtime, memory, and energy efficiency of LLM-generated code. EffiBench [418] establishes a benchmark over 1,000 algorithmic tasks, showing that GPT-4 solutions can be up to 3 \times slower and use 14~44 \times more memory than optimized human baselines. Mercury [265] extends this evaluation with a multi-dimensional framework that measures execution time, memory footprint, and best/worst-case complexity across diverse programming problems. EffiBench-X [818] further generalizes efficiency benchmarking to multiple programming languages, enabling cross-lingual comparisons of computational resource usage. Several benchmarks focus on algorithmic complexity. BigO (Bench) [147] evaluates whether generated code achieves the correct asymptotic complexity, while DynaCode [413] introduces input scaling to expose inefficiencies that are not visible under small test cases. COFFE [792] emphasizes practical runtime evaluation under realistic execution loads, providing a complementary system-level perspective. EvalPerf [612] introduces a Differential Performance Evaluation framework to systematically assess the efficiency of LLM-generated code on performance-challenging tasks.

Energy Consumption and Efficiency Optimization Energy consumption has also become an important evaluation axis. ECCO [1000] examines whether efficiency improvements can be obtained through code-level transformations without affecting correctness. Solovyeva et al. [909] quantifies the energy overhead of LLM-generated solutions compared to human-written

code. Cappendijk et al. [137] explores fine-tuning strategies to reduce power consumption, and a study on *StarCoder2* [226] analyzes the impact of quantization on inference cost and downstream code efficiency. Several approaches aim to directly incorporate efficiency into modeling. *EffiCoder* [420] introduces fine-tuning methods that integrate runtime and memory signals into the training objective. *ACECode* [1170] applies reinforcement learning to jointly optimize correctness and efficiency. *Rethinking Code Refinement* [875] trains models to detect and improve inefficient code through iterative refinement, offering a post-generation strategy for efficiency enhancement.

3.2.5. Code Preference

Code preference is a task that evaluates whether code language models can make preference judgments between different code solutions that align with human developer preferences across dimensions like correctness, efficiency, security, and readability.

Holistic and Composite Scoring Benchmarks CodeArena [266] calculates a dynamic score from two components. The correctness component is weighted by a problem’s overall pass rate, thus giving more credit for solving difficult problems. The efficiency component is determined by a solution’s runtime ranking among all correct submissions, with faster solutions scoring higher. The final score is a composite of these two. Moving beyond algorithm-level problems, Long CodeArena [115] assesses the understanding of LLMs at the project level, requiring a holistic comprehension of an entire codebase rather than a small part of code or one single file, such as library-based code generation, CI build repair, and commit message generation.

Aligning with Human Preferences CodePrefBench [610] focuses on evaluating whether a code LLM’s preferences align with human developers. It tests the judgment of LLMs by presenting them with pairs of code solutions that differ across dimensions like correctness, efficiency, security, and human preference and requiring LLMs to select the superior option. Similarly, other arena-style evaluations employ an LLM-as-a-judge to systematically compare two models. In this setup, the judge selects the better of two model-generated responses, providing a scalable method for assessing which model’s output is more aligned with human preferences. CodeArena (Yang) [1177] employs an LLM-as-a-judge to systematically compare two LLMs. The judge selects the better of two LLM-generated responses, providing a scalable method for assessing which LLM’s output is more aligned with human preferences (if LLM judgment can reflect the human preference). AutoCodeArena [1341] extends this arena-style evaluation paradigm by replacing costly human preference collection with automatic judgments from a strong LLM. Motivated by the high resource demands of BigCodeArena [1341], AutoCodeArena leverages an LLM-as-a-judge to compare each model’s output against a fixed baseline system, using the Bradley-Terry [918] model to aggregate pairwise comparisons into final preference scores. To approximate real-world usage, the benchmark selects 600 representative prompts sampled across six programming topics, executes model-generated code using a local Docker-based sandbox, and provides execution outputs to the judge model. This enables a scalable and efficient automatic arena for tracking LLM coding capability while maintaining alignment with human preferences.

3.2.6. Code Reasoning and Question Answering

Code reasoning and question answering is a task category that evaluates language models’ ability to understand, analyze, and reason about code semantics through question-answer formats, deep

semantic reasoning challenges, and specialized tasks like code execution prediction, program equivalence checking, and static analysis.

Evaluation Based on Question-Answer Pairs Some benchmarks evaluate models through question-and-answer formats. For example, CodeQA [599] converts code comments into question-answer pairs through syntactic and semantic analysis, creating a dataset with over 100,000 entries. On the other hand, CS1QA [517] collects question-answer pairs from chat logs of an introductory Python programming course. To more comprehensively evaluate a model’s code reasoning ability, CodeMMLU [679] designed a variety of tasks, including code repair, execution inference, and fill-in-the-blank challenges. Similarly, CodeSense [853] proposes a benchmark covering fine-grained code semantic reasoning tasks in real-world software projects, aiming to address the inadequacy of existing benchmarks that largely rely on synthetic data or educational programming problems.

Deep Code Semantic Reasoning There are also benchmarks that measure models’ deep code reasoning capabilities through more specific tasks. SpecEval [669] innovatively uses formal program specifications to evaluate models’ code understanding ability, designing four tasks including specification judgment, selection, completion, and generation. CRUXEval [346] and CRUXEval-X [1140] design two sub-tasks, output prediction and input prediction, to measure models’ code reasoning and understanding capabilities. CRUXEval-X extends the benchmark to 19 different programming languages, providing a more comprehensive evaluation. EquiBench [1059] evaluates models’ understanding of program semantics through program equivalence checking, i.e., determining whether two functionally identical but syntactically different pieces of code are equivalent. This task directly tests the model’s understanding of code execution semantics. CORE [1123] evaluates LLMs’ code reasoning capabilities through fundamental static analysis tasks such as data dependency, control dependency, and information flow. With the development of multimodal models, new evaluation dimensions have been introduced. ScratchEval [299] utilizes Scratch, a block-based visual programming language for children, to evaluate the reasoning abilities of large multimodal models in integrating visual information and programming logic.

3.2.7. *Code Translation*

The automated translation of code from one programming language to another is a long-standing challenge, aimed at modernizing legacy systems, improving performance, and unifying disparate codebases. The field has evolved from early structure-aware models to the current paradigm dominated by Large Language Models (LLMs) enhanced with sophisticated verification and reasoning techniques.

Foundational Approaches: From Syntax to Sequence Early research focused on capturing the rigid structure of programming languages. Methods like Tree-to-tree Neural Networks [167] and Grammar-Driven models [262] treated translation as a transformation between Abstract Syntax Trees (ASTs), ensuring syntactic correctness by design. This structural approach was later advanced by sequence-to-sequence models like TransCoder [506], which demonstrated the power of unsupervised pre-training on vast monolingual code corpora. These models set the stage for modern LLMs by showing that deep semantic patterns could be learned without requiring parallel, line-by-line translation examples.

Ensuring Correctness through Execution and Feedback A key challenge in code translation is that syntactic validity does not guarantee functional equivalence. A translated program might compile but produce incorrect results. To address this, a significant trend has emerged: leveraging program execution as a feedback signal. This test-and-repair paradigm involves using automated unit tests to filter incorrect translations [856], employing compiler feedback within a reinforcement learning loop to guide the model toward valid programs [446], and using dynamic analysis to compare runtime states and pinpoint semantic errors [1130]. This approach, which integrates generation with verification, has become critical for producing reliable translations.

Prompt Engineering and Reasoning for Large Language Models While powerful, general-purpose LLMs require specialized strategies to excel at the nuanced task of code translation. A primary method is the prompt engineering and reasoning. Techniques like the “Explain-then-Translate” method [942] improve reasoning by forcing the model to first create a natural language summary. Others use intermediate languages and planning algorithms to decompose complex translations into manageable steps [675]. Another major direction is the development of agentic and automated workflows, which move beyond single-shot generation, employing multiple LLM agents that collaborate to repair syntax and semantics [370] or creating closed-loop, self-correcting frameworks that automatically use compiler errors to fix their own output until the code is executable [228].

Specialization: Safety-Critical Translation A particularly high-stakes application of code translation is the migration from memory-unsafe languages (e.g., C/C++) to memory-safe languages like Rust. The goal here is not just correctness but also the elimination of entire classes of security vulnerabilities. This has spurred the development of specialized tools that combine LLMs with formal methods and rigorous testing. These frameworks use techniques like differential fuzzing to verify equivalence [275], generate reference oracle programs for validation [1161], and leverage static analysis to guide the LLM in producing safer, more idiomatic Rust code [734, 1327].

Overcoming Data Limitations and Improving Generalization The performance of any model is dependent on its training data, and high-quality parallel code corpora are rare. Researchers have developed several techniques to overcome this bottleneck. These include training on aligned code snippets from multiple languages to improve low-resource performance (e.g., MuST [1331]), using back-translation to generate synthetic parallel data (e.g., BabelTower [1071]), and employing federated learning to train models across organizations without sharing private code (e.g., FedCoder [502]). These methods are crucial for building models that can generalize across a wide range of languages and domains.

Error Analysis As translation models become more capable, evaluating them becomes more complex. Simple metrics like BLEU are often insufficient. New evaluation frameworks propose a multi-level taxonomy of translation complexity, from simple token replacement to complex algorithmic rewriting, providing a more nuanced assessment of model capabilities [467]. Furthermore, significant research has focused on taxonomizing common LLM translation errors [774] and developing lightweight, post-hoc models designed specifically to recognize and rectify these errors, thereby improving the reliability of any underlying translation model [1217].

3.2.8. Test-Case Generation

In Table 9, test case generation is the task of automatically creating input-output test cases that can effectively evaluate and distinguish between correct and incorrect code implementations, serving as a critical component for assessing program correctness in both software engineering and competitive programming domains.

Table 9. Test case benchmarks overview. **Solutions per problem (SPP)** indicates the average number of solutions that are used to evaluate generated test cases for each problem in the benchmark.

Benchmark	Year	Source	Language	#Problems	SPP
Software Engineer					
SWT-Bench [468]	2024	SWE-Bench [928]	Python	1900+	1
TestGenEval [442]	2024	SWE-Bench [928]	Python	1210	1
TestBench [1277]	2024	Github	Java	108	1
CLOVER [1137]	2025	Github	Python	845	1
Algorithm Competition					
TestEval [1035]	2025	Leetcode	Python	210	1
CodeForce-SAGA [674]	2025	Atcoder Codeforces Nowcoder	Python	1840	36.66
TestCase-Eval [1202]	2025	Codeforces	C++ Python Java	500	200
TCGBench [134]	2025	NOIP Luogu	C++	208	5

Test Case Generation for Software Engineering The correctness of code is often evaluated through test cases, making the generation of these test cases a new, core, and critical problem. Influenced by traditional software testing, early evaluations of test cases primarily focused on engineering practicality, such as code coverage and the ability to distinguish between historical (buggy) and current (correct) code. SWT-Bench [716] and TestGenEval [442] transform tasks from SWE-Bench, providing buggy code and its corresponding fix patch, requiring test cases to fail on the buggy code while passing on the fixed version. As these benchmarks are specific to Python, TestBench [1277] extends this approach to Java. CLOVER [1137] utilizes the data from GitHub and supplements the benchmark to address the lack of long-form evaluation.

Test Case Generation for Competitive Programming Beyond the field of software engineering, there is also a significant demand for test case generation in competitive programming. This is due to the inaccessibility of private test cases, necessitating the generation of a sufficient number of test cases to determine the correctness of a generated solution. TestEval [1035] collects 210 problems from LeetCode; however, its evaluation is still limited to coverage metrics. In contrast, CodeForce-SAGA, TCGBench, TestCase-Eval and TCGBench [134, 674, 1202] collects

a large number of wrong and correct code submissions to conduct an end-to-end evaluation of the proportion of test cases that could reject the wrong code while passing the correct code. This progression highlights the critical shift from coverage-based metrics to more robust, functionality-driven benchmarks that assess the ability of test cases to precisely discriminate between correct and incorrect program behaviors.

3.3. Repository-Level Tasks

This section presents repository-level code benchmarks, using various sources like repository-based commit messages, issues, and PRs to evaluate the performance of multiple large language models (LLMs). We will introduce their contributions and methods.

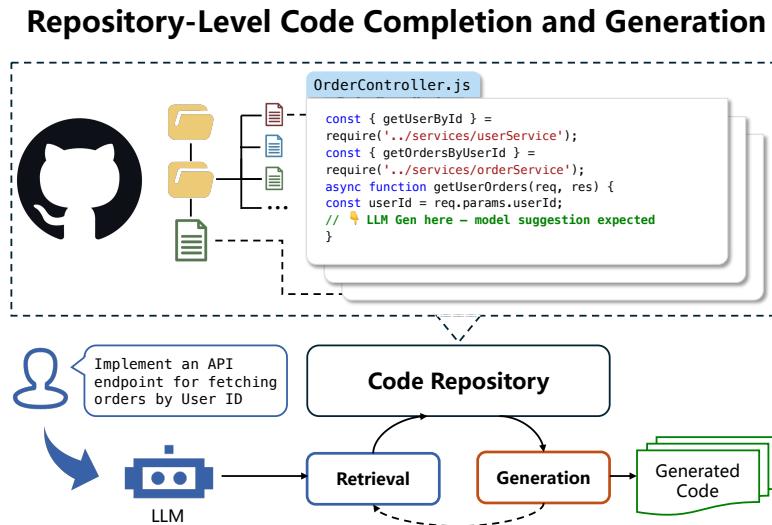


Figure 14. Illustration of the repository-based code generation and completion task.

3.3.1. Code Generation and Completion

In Figure 14, repository-level code completion and generation refers to AI-powered techniques that leverage the entire codebase context (including multiple files, project structure, dependencies, and cross-file relationships) to predict, complete, or generate code segments that are contextually aware of the broader software repository rather than just the immediate file or function. **RepoBench** [621] is a repository-level code completion benchmark using pre-2022 GitHub repos for training and post-2023 for testing to ensure temporal validity, covering Python and Java. It evaluates models (Codex, StarCoder variants) on metrics and tasks (Retrieval, Completion), showing better Python performance. **RepoEval** [1263] evaluates code completion from 14 real repositories using unit tests; its RepoCoder agent, leveraging full repository context, outperforms others in 90% of tests. **Excrepobench** [1178] assesses repository-based completion of LLM by regenerating multi-granularity masked spans (expressions, statements, functions) and validating via repository test files and fine-tune base Qwen2.5-Coder on it to enhance completion performance. **CoderEval** [1223] is a practical benchmark with 460 Java/Python problems from open-source repositories, focusing on non-standalone functions; models perform significantly worse on these than standalone ones. **CrossCodeEval** [252] is a multilingual benchmark (10k examples) testing cross-file context needs via static analysis. **M2rc-Eval** [607] is a large-scale multilingual repository-level benchmark with AST-based annotations. **Codev-Bench** [778] uses industrial data to evaluate repository-level completion; specialized code LLMs outperform

general ones, but all struggle with incomplete suffix scenarios. **RepoCod** [585] is a Python benchmark (980 tasks, 50%+ needing repo context) from 11 projects. **DI-Bench** [1274] evaluates dependency reasoning across 4 languages (581 testable repos); even top models achieve low pass rates. **DependEval** [263] hierarchically assesses repository-level dependency understanding across 8 languages (15,576 repos); significant performance gaps exist, with advanced models struggling. **REPOST** [1128] builds repository-level environments via sandbox testing; models trained on its REPOST-TRAIN dataset show modest gains on HumanEval/RepoEval, but perform poorly on REPOST-EVAL. **SecRepoBench** [251] evaluates secure code generation at the repository level (318 tasks, 15 CWE types in C/C++). **DevEval** [546] is a manually annotated benchmark (1,874 samples, 117 repos); current LLMs perform poorly, confirming its difficulty.

Domain-Specific and Complex Code Generation

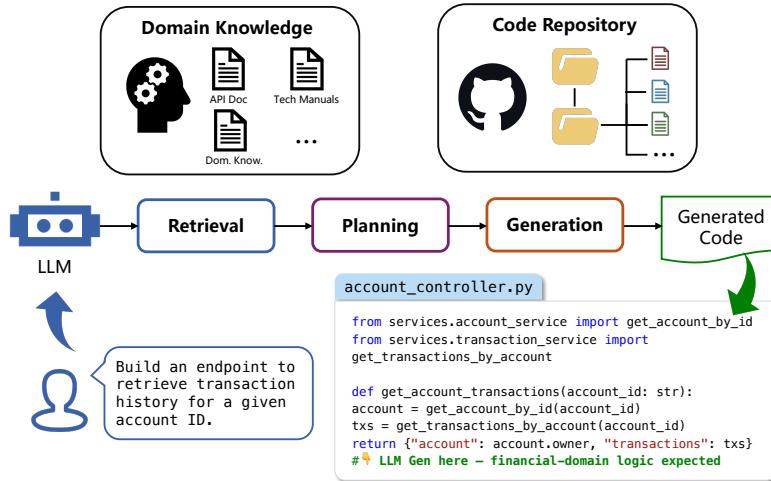


Figure 15. Illustration of the domain-specific and complex code generation task.

3.3.2. Domain-Specific and Complex Code Generation

In Figure 15, the domain-specific and complex code generation task refers to the automated creation of source code that requires specialized knowledge in a particular field and involves intricate logic, multiple dependencies, or sophisticated algorithmic implementations. **BioCoder** [936] is a code generation benchmark for bioinformatics. It contains 2,269 high-quality coding problems from 1,700 bioinformatics repositories. **PaperBench** [913] uses the reproduction of 20 ICML 2024 papers as its benchmark standard, encompassing understanding core contributions, implementing code, and conducting experiments. A rubric tree decomposes the overall task into subtasks; every leaf node is an independently scored unit that cannot be split further. PaperBench contains 8,316 individually gradable tasks. Rubrics are co-developed with the authors of each ICML paper for accuracy and realism. To enable scalable evaluation, PaperBench also develops an LLM-based judge to automatically grade replication attempts against rubrics and assess our judge's performance by creating a separate benchmark for judges. PaperBench evaluates several frontier models in experiments. **Commit0** [1302] is a benchmark that tests AI agents' ability to write software libraries from scratch. It includes 54 Python libraries where LLMs are given specification documents and empty function bodies to complete. The task is to implement the functions and pass all unit tests. The results show that current agents can pass some unit tests but cannot reproduce entire libraries. Interactive feedback is very helpful for models to generate code that passes more tests. **HackerRank-ASTRA** [1131] is a benchmark that evaluates

the correctness of LLMs on cross-domain multi-file project problems. This benchmark assesses LLMs by creating multi-file project problems, with each problem run 32 times for evaluation. **ProjectEval** [613] is a new benchmark that uses simulated user interaction to test how well LLM agents can generate code at the repository level. It is built with LLMs and human review, has 284 test cases, and offers three levels of input, including **Level 1** natural language prompt (NL Prompt), **Level 2** Natural Language Checklist (NL Checklist), and **Level 3** Skeleton. The results show that current agents perform poorly, revealing that creating systematic project code and understanding the whole project are key challenges for LLM agents. **DA-Code** [426] is a code generation benchmark designed to evaluate LLMs on agent-based data science tasks. It includes 500 complex data science tasks covering various aspects and a developed DA-Agent baseline, which shows that even advanced LLMs perform poorly on DA-Code, indicating that current models still face significant challenges with complex data science tasks.

Code Editing, Refactoring, and Agent Collaboration

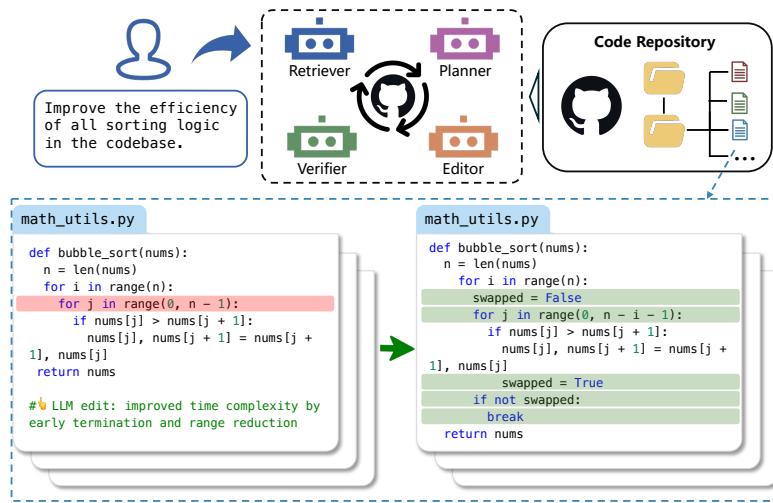


Figure 16. Illustration of the code editing, refactoring, and agent collaboration task.

3.3.3. Code Editing, Refactoring, and Agent Collaboration

In Figure 16, code editing, refactoring, and agent collaboration tasks are a type of tasks where code is modified, restructured for improvement, and multiple AI agents work together to complete programming objectives. **Aider’s code editing benchmark** [26] asks the LLM to edit Python source files to complete 133 small coding exercises from Exercism³. This measures the LLM’s coding ability and whether it can write new code that integrates into existing code. The model also has to successfully apply all its changes to the source file without human intervention. **Aider’s refactoring benchmark** [27] asks the LLM to refactor 89 large methods from large Python classes. This is a more challenging benchmark, which tests the model’s ability to output long chunks of code without skipping sections or making mistakes. It was developed to provoke and measure GPT-4 Turbo’s “lazy coding” habit (refers to an LLM tendency to avoid fully writing long, exact code—skipping sections, hand-waving with placeholders or summaries, and introducing mistakes instead of producing a complete, faithful implementation). The refactoring benchmark requires a large context window to work with large source files. Therefore, results are available for fewer models. **Aider’s Polyglot benchmark** [798] is based on Exercism coding exercises like Aider’s original code editing benchmark. The new polyglot

³<https://github.com/exercism>

benchmark contains coding problems in C++, Go, Java, JavaScript, Python, and Rust. The old benchmark was solely based on Python exercises. It focuses on the most difficult 225 exercises out of the 697 that Exercism provides for those languages. The old benchmark simply included all 133 Python exercises, regardless of difficulty. **RES-Q** [504] is a benchmark based on GitHub commits, containing 100 handcrafted repository-level editing tasks. It is used to evaluate LLMs in software development tasks, focusing on their ability to follow human instructions and then make code changes. The results show that RES-Q can effectively distinguish the performance of different LLMs. **LiveRepoReflection** [1282] establishes a rigorous benchmark for evaluating code comprehension and generation capabilities in multi-file repository contexts. LiveRepoReflection contains 6 programming languages to ensure diversity and increase complexity, and demonstrates significantly greater difficulty than Aider Polyglot Benchmark in experimental evaluations. Additionally, it provides the RepoReflection-Instruct dataset and fine-tunes the RepoReflectionCoder based on Qwen2.5-Coder-32B. **HumanEvo** [1309] is an evolution-aware repository-level code generation benchmark designed to address the issue of overestimated performance in LLMs caused by ignoring project dynamics. It constructs a benchmark of 400 task instances with evolution-aware settings, combining dependency level categorization and automated evaluation tools to compare code generation capabilities across multiple mainstream LLMs. The results show that neglecting project evolution leads to performance overestimation ranging from 10.0% to 61.1%, validating the critical importance of dynamic context for realistic evaluation. **RepoExec** [374] is a repository-level code generation benchmark. It constructs executable environments to analyse how context affects the quality of generated code. The benchmark designs three context modes and compares 18 LLMs using the pass@k metric and the Dependency Invocation Rate(DIR). The results show that complete dependencies significantly improve model performance, and instruction-tuned models are better at using dependencies than pre-trained models. **CodePlan** [93] is a framework for repository-level coding tasks. It automates complex tasks that require extensive edits across the entire repository. CodePlan uses incremental dependency analysis and other algorithms to create a multi-step chain of edits. The results show that CodePlan performs better than baseline methods on 2 repository-level tasks. Most repositories pass the validity checks when using CodePlan.

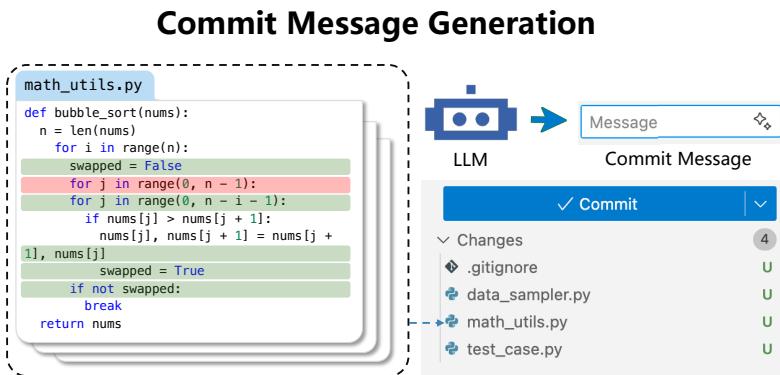


Figure 17. Illustration of the commit message generation task.

3.3.4. Commit Message Generation

In Figure 17, the commit message generation task is the process of automatically creating concise, informative textual descriptions that summarize the changes made in a code commit based on the modified code differences. Commit message generation aims to automatically summarize source code changes into concise natural-language descriptions. Early works treat this as a

supervised translation problem from code diffs to text. Jiang and McMillan [461] apply naive bayes classification to predict verbs and objects from diffs, but struggles with full-sentence generation due to limited semantic modeling. Loyola et al. [640] introduce an attention-based encoder–decoder for diff-to-text generation, while Jiang et al. [462] add a verb–direct-object (VDO) structural filter to improve syntactic coherence and BLEU scores.

Later studies focus on representation and retrieval. Liu et al. [634] propose *NMTGen*, a nearest-neighbor retrieval model that reuses messages from similar diffs, achieving higher performance and faster inference efficiency than neural baselines. van Hal et al. [993] revisited NMT methods with stricter preprocessing, showing that performance gains often stemmed from memorization rather than semantic understanding. Xu et al. [1141] developed *CoDiSum*, which integrates code structure and a copy mechanism for out-of-vocabulary (OOV) words, improving BLEU, METEOR [96], and Recall. Liu et al. [617] propose *ATOM* combines abstract syntax trees (ASTs) with retrieval and CNN-based ranking to boost accuracy.

With the rise of pre-trained models, transformer-based and context-aware approaches became dominant. Jung [476] introduces *CommitBERT*, jointly encoding added and deleted code with a code-pretrained BERT model, outperforming earlier NMT systems. Wang et al. [1017] presented *CoRec*, a retrieval–generation hybrid mitigating exposure bias and handling rare vocabulary via decay sampling. Wang et al. [1024]’s *ExGroFi* incorporate linked issue descriptions, enhancing the rationale and conciseness of messages, while Eliseeva et al. [274]’s *CommitChronicle* leveraged commit history to improve temporal and stylistic coherence.

For standardized evaluation, *CommitBench* [865] compiles 1.6M commit–diff pairs from 72K GitHub repositories, enabling large-scale comparison of transformer models such as CodeTrans [202] and T5 [829]. Using ROUGE-L [852] and BLEU [779], code-pretrained transformers consistently outperforms general-purpose LMs. The MCMD framework [946] proposes the BLEU variant B-Norm, which applies smoothing and case-insensitivity and was shown to correlate more strongly with human judgments than traditional BLEU measurements..

Overall, research has evolved from rule-based and RNN systems to pre-trained, retrieval-augmented transformers that incorporate structure and history. Remaining challenges include aligning messages with developer intent, handling multilingual repositories, and balancing informativeness with brevity.

3.3.5. Software Engineering Tasks

Software engineering task is the process of analyzing, implementing, and completing specific software development tasks, including bug fixes, feature implementations, and technical problem-solving, from identification through to verified completion.

The SWE-bench family of benchmarks [468] has become the foundation for evaluating large language models on real-world software engineering tasks. The original SWE-bench contains 2,294 Python issue–PR pairs, where models must resolve GitHub issues verified via “fail-to-pass” unit tests. **JavaBench** [132] extends this evaluation to Java, focusing on object-oriented reasoning across four open-source projects, where models consistently lag behind trained human developers. To reduce computational cost, **SWE-bench Lite** offers a 300-task subset that preserves the overall distribution of the full benchmark. **SWE-bench Multilingual** [1188] broadens coverage to nine programming languages (42 repositories, 300 manually verified tasks), addressing the Python-centric limitation of earlier versions. Meanwhile, **SWE-bench Multimodal** [1186] introduces 517 image-based issues, testing whether models can interpret visual cues such as screenshots or GUI errors during debugging. **SWE-bench Verified** [930]

contributes 500 human-curated examples with Docker-based evaluation, ensuring consistent reproducibility across models and environments.

Recent extensions aim to enhance temporal and practical realism. **SWE-bench Live** [1273] continuously incorporates new GitHub issues (1,565 tasks across 164 repositories), making the benchmark more dynamic and substantially harder than static datasets. Beyond bug fixing, **SWE-Perf** [392] evaluates performance optimization through 140 repository-level pull requests, revealing that models still fall short of expert programmers in efficiency-related tasks. **SWE-rebench** [83] automates large-scale issue collection (>21,000 tasks), enabling longitudinal evaluation of model degradation and temporal generalization. **SWE-Dev** [269] shifts the focus from bug fixing to feature implementation, introducing 14K training and 500 evaluation examples—fine-tuned 7B models approach GPT-4-class performance. BugPilot [911] generates more realistic synthetic bugs by having LLMs add new features rather than directly inserting bugs, achieving SOTA results on SWE-bench-Verified with Qwen3 models. Gistify [524] requires agents to distill a repository into minimal single-file code that replicates its runtime behavior, testing deeper codebase understanding beyond SWE-bench’s localization-focused tasks. Several newer datasets extend this paradigm in specific directions. **SWE-PolyBench** [838] evaluates multi-language repositories across 21 projects (2,110 tasks), revealing substantial variance in cross-language reasoning ability. **Multi-SWE-bench** [1247] similarly covers seven languages (1,632 verified issues), showing strongest results in Python and weakest in C/C++/Rust. **SWE-bench+** [33] mitigates data leakage by including post-cutoff repositories, demonstrating significant performance drops compared to pre-2023 datasets. Finally, **SWE-bench M** [1185] evaluates visual debugging across 17 JavaScript projects (619 image-grounded issues), where even top-tier AI systems struggle with multimodal reasoning.

Beyond the SWE-bench family, **SWE-Lancer** [695] reframes evaluation as freelance-style project execution using 1,488 real UpWork tasks (valued at roughly \$1M), probing economic value creation rather than accuracy alone. **FAUN-Eval** [412] benchmarks fine-grained GitHub issue resolution (300 manually curated entries) and finds that open- and closed-source models excel on different categories. **FEA-Bench** [567] measures repository-level feature implementation across 83 repositories, with the best models solving only around 10% of cases. **SwingArena** [1145] and **CoreCodeBench** [298] extend evaluation toward CI-integrated and composite tasks, respectively. Finally, **AgentIssue-Bench** [831] examines the self-maintenance abilities of software agents (50 tasks derived from 201 issues), revealing persistent challenges in long-term memory and LLM compatibility.

3.3.6. Comprehensive Software Development

Recent benchmarks are expanding the scope of evaluation from isolated code snippets to the broader and more complex ecosystem of software development. This trend reflects a growing understanding that a model’s value lies not just in writing code, but in its ability to comprehend, document, and interact with the entire development lifecycle. This shift is evident in benchmarks targeting different facets of this lifecycle. README Eval [842] moves beyond code to assess high-level project understanding, tasking models with generating repository documentation from contextual metadata like issues and commits. Pushing into the collaborative process, OmniGIRL [369] evaluates a model’s ability to resolve GitHub issues, introducing multilingual and, critically, multimodal challenges by including images within bug reports—a task where current LLMs significantly struggle. Furthermore, new benchmarks are beginning to evaluate the use of essential developer tools. GitGoodBench [595] is the first to test AI agents on their mastery of version control systems, revealing that even sophisticated agents fail at complex

but common tasks like resolving merge conflicts. To ensure these evaluations remain relevant, EvoCodeBench [545] introduces a dynamic paradigm, with tasks derived from evolving, real-world repositories to better reflect the moving target of ongoing software development. Underpinning all these advanced capabilities is the foundational importance of data quality, with projects like Stack-Repo [493] demonstrating that curating massive, deduplicated source code datasets is critical for improving model performance across all these real-world tasks.

3.3.7. Repository-Level and Long Context Understanding

As shown in [Figure 18](#) of IDE, the repository-level and long context understanding refers to the task of comprehending and reasoning across entire codebases or extensive documents that span multiple files and require maintaining context over thousands or millions of tokens

As models' ability to process longer contexts improves, evaluating repository-level code understanding becomes particularly important. Benchmarks such as RepoQA [611] focus on the repository-level question-answering task, where a system must answer natural-language queries by grounding its response in the contents of a code repository. The task requires retrieval and reasoning over heterogeneous project artifacts (e.g., source files, README documents, test files, issues/PRs) distributed across multiple files and directories, and producing an answer that is both correct and supported by one or more evidence items from the repository (for example, file paths, code snippets, or documentation excerpts). Formally: given a repository R composed of artifacts $A = \{a_1, a_2, \dots, a_n\}$ and a natural-language question q , the goal is to produce an answer a based on the repository's contents.

RepoQA [611] and CodeRepoQA [411] both focus on evaluating the long-context code understanding capabilities of large language models. Specifically, RepoQA requires models to find functions based on natural language descriptions, while CodeRepoQA assesses their repository-level question-answering ability in the field of software engineering. Similarly, CoreQA [156] also targets code repository-level question-answering tasks, building its dataset by collecting questions and comments from real GitHub repositories to reflect the complexity of real-world software development. Furthermore, LongCodeU [547] provides a more comprehensive and challenging set of tasks, evaluating models' long code understanding ability across four key aspects: code unit awareness, intra-code unit understanding, inter-code unit relationship understanding, and long document understanding.

To reduce the context length, LongCodeZip [892] proposes a novel, training-free framework designed to efficiently compress long code contexts for large language models (LLMs). Traditional context pruning and retrieval-based methods struggle with the structural and semantic complexity of code. To address this, LongCodeZip introduces a two-stage hierarchical compression strategy: (1) coarse-grained compression that ranks and selects function-level chunks based on conditional perplexity (approximated mutual information with respect to the instruction), and (2) fine-grained compression that segments retained functions into blocks via perplexity-based boundary detection, followed by adaptive token-budget optimization using a knapsack algorithm. Experiments across code completion, summarization, and question answering tasks demonstrate that LongCodeZip achieves up to 5.6 \times compression without degrading performance, outperforming baselines. It also generalizes well across models (from 0.5B to 8B parameters), significantly reduces API costs and latency, and represents the first dedicated framework for long-context code compression in LLMs.

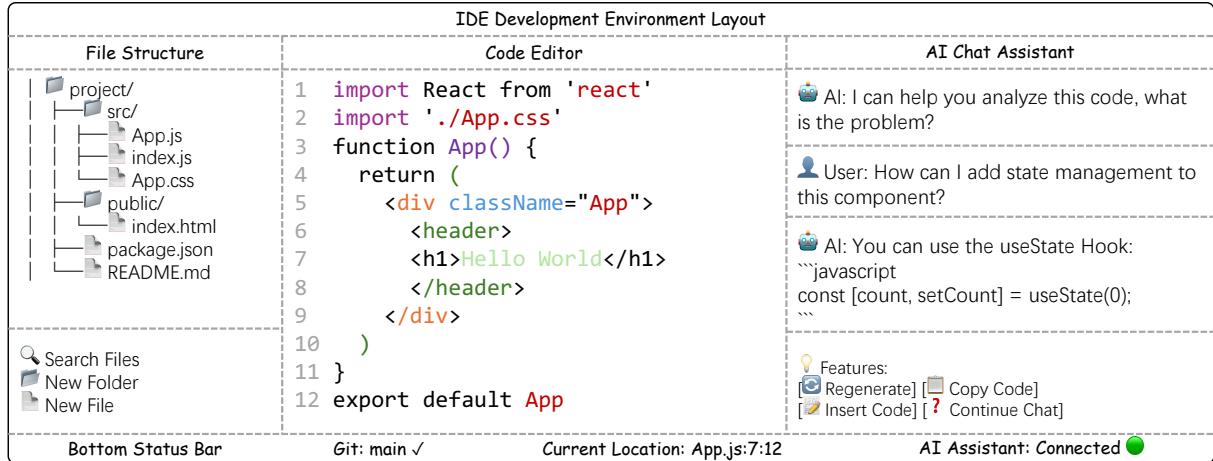


Figure 18. The core task of RepoQA [611] is searching needle function (SNF), asking a model to find and reproduce a target function from an entire repository using only a natural language description of what the function does.

3.4. Agentic Systems

3.4.1. Agent Tool Use

A fundamental capability of any agent is its ability to interact with the digital world through tools like APIs and functions. Early benchmarks like API-Bank [561] and ToolBench [817] established the foundation for this evaluation, testing an agent’s ability to correctly select and invoke the right tool for a given task. However, the complexity of these evaluations is quickly advancing. BFCL [781] exemplifies this trend with its iterative versions, progressing from single function calls to complex, multi-turn, and multi-step scenarios that better mimic real-world interactions. Furthermore, benchmarks are moving from generic tool-use tasks to domain-specific applications, with frameworks like Tau Bench [1210] assessing agent performance in realistic business workflows like customer service.

3.4.2. Deep Research Benchmarks

The true measure of an advanced agent lies in its ability to go beyond simple information retrieval and perform deep, human-like reasoning. Benchmarks in this category are designed to be trivially easy for humans but expose profound limitations in current AI systems. GAIA [691] pioneered this approach with real-world questions that require a combination of reasoning, web browsing, and tool use, revealing a stark performance gap where leading models fail on tasks humans solve with over 90% accuracy. This challenge is being pushed further into specialized, high-stakes domains. xbench [158] evaluates agents on professional workflows like talent sourcing and marketing, while DeepResearch Bench [264] raises the bar to an academic standard, requiring agents to synthesize analyst-grade reports on PhD-level topics, testing the limits of autonomous research and synthesis.

3.4.3. Web Search Benchmarks

The open web presents perhaps the most unconstrained and unforgiving environment for autonomous agents. Unlike closed benchmarks with predefined states, real-world web search

requires persistence, adaptability, and the ability to filter noisy or contradictory information. Recent benchmarks have begun to capture this challenge more faithfully. For example, BrowseComp [1064] frames information retrieval as a compositional reasoning task, requiring agents to aggregate fragmented evidence distributed across multiple websites. Its extensions, including BrowseComp-ZH [178] for multilingual search and BrowseComp-Plus [1323], which decouples reasoning from retrieval, push evaluation toward more realistic and interpretable agent behavior. Complementary efforts such as WebWalkerQA [1087] emphasize systematic exploration and question answering across hyperlinked environments, while Widesearch [1077] focuses on large-scale, open-domain information synthesis. The results reported so far remain sobering: even top-performing models achieve near-zero success on end-to-end tasks, underscoring that robust web-scale reasoning remains a major unsolved problem.

3.4.4. Benchmarking Agents for Graphical User Interfaces

The development of autonomous agents capable of understanding and interacting with graphical user interfaces (GUIs)—across web, desktop, and mobile platforms—represents a significant milestone toward general-purpose artificial intelligence. Early work in this area typically focused on narrow tasks, but the integration of visual and linguistic reasoning in Large Language Models (LLMs) has enabled agents with far greater adaptability. Nonetheless, grounding natural language commands into concrete UI actions, maintaining coherent multi-step plans, and generalizing across heterogeneous interfaces remain open challenges. These difficulties have motivated the creation of increasingly sophisticated benchmarks designed to systematically evaluate and advance GUI-agent capabilities. Broadly, these benchmarks target two complementary dimensions of capability: interface navigation (acting as a user) and interface development (acting as a creator).

Benchmarks for Frontend Navigation A core challenge for GUI agents is navigating complex, interactive environments. Foundational benchmarks such as WebShop [1205] and Mind2Web [242] established this paradigm by providing thousands of goal-oriented tasks that require agents to operate within real or simulated websites. These evaluations revealed that even powerful models struggle with long-horizon reasoning and precise UI grounding. Building on this foundation, recent benchmarks have expanded along several directions. **Cross-Platform Generalization:** OmniACT [484] introduces unified benchmarks spanning web, desktop, and mobile settings, exposing limitations in cross-domain generalization. **Higher-Level Cognitive Reasoning:** Web-ChoreArena [699] challenges agents with compound “chore” tasks requiring long-term memory and computation, while PersonalWAB [127] introduces personalization through user histories and preferences. **Fine-Grained Capability Diagnosis:** Benchmarks such as Sphinx [836] and NovelScreenSpot [283] decompose GUI interaction into subskills, such as goal understanding, UI grounding, and planning.

Benchmarks for Frontend Development The second major direction focuses on automating GUI creation, translating human intent into executable frontend code. This research area has evolved from single-page generation to complex multi-file workflows that simulate real development pipelines. Early efforts such as Design2Code [899] and WebCode2M [356] provide large-scale paired datasets linking design inputs with corresponding code implementations. Increasingly sophisticated tasks have since emerged: Sketch2Code [564] examines generation from informal sketches, while Interaction2Code [1118] evaluates dynamic, interactive webpage generation. Recent benchmarks reflect a broader ambition to develop autonomous agents capa-

ble of full-stack web creation. WebGen-Bench [652] requires the generation of entire multi-file websites from scratch, while Web-Bench [1138] models realistic software engineering workflows with sequentially dependent tasks, shifting evaluation from single-turn code generation toward continuous, project-level reasoning.

3.4.5. Terminal Use

Terminal-Bench [966] is an emerging benchmark that evaluates a code agent's ability to autonomously operate in a terminal environment on real tasks. It transcends the traditional paradigm of code generation or fixing within a defined environment, requiring the code agent to have system-level development capabilities. This includes being able to explore the system and execute shell commands and various command-line tools to complete complex, system-level tasks, such as compiling and booting a complete Linux kernel from source, or deploying a functional server from scratch. Unlike SWE-bench [928], which has clear problem boundaries (a repo/folder), in Terminal-Bench, the entire environment is the problem space to be solved. The agent's objective is to deliver a successfully running or configured system by performing system configuration, dependency management, and executing a complete workflow.

4. Alignment

Alignment in code LLMs refers to the process of adapting pre-trained LLMs to follow human instructions and perform coding tasks effectively. [subsection 4.1](#) introduce the progress about supervised fine-tuning (SFT) for code LLMs, which learn from labeled instruction-following datasets covering tasks like code generation, repair, and translation. [subsubsection 4.5.1](#) and Reinforcement Learning (RL), which uses reward signals to further refine model behavior. A particularly powerful variant is Reinforcement Learning with Verifiable Rewards (RLVR), where models receive deterministic pass/fail feedback from test cases or compilers, enabling them to develop structured reasoning, self-verification, and error-correction capabilities. Together, these alignment methods transform general pre-trained models into specialized coding assistants capable of understanding requirements, generating correct solutions, and handling complex real-world software development scenarios.

4.1. Supervised Fine-tuning (SFT)

Supervised fine-tuning (SFT) is the process of training a pre-trained language model on a labeled dataset. A common and powerful way to format this data is through instruction tuning, where the model learns to follow natural language commands. This method not only enables the model with the ability to understand and execute instructions but also significantly enhances its performance on targeted tasks [143, 1279]. In the training of code LLMs, SFT is a key strategy for improving model performance.

The scope of code instruction tuning data is broad, covering a diverse range of programming-related tasks, including but not limited to code generation, code repair, and code translation. A multi-task dataset design aims to cultivate comprehensive capabilities in LLMs, allowing them to respond flexibly to human instructions and perform various code-related tasks [598]. The emergence of high-quality code instruction datasets has significantly enhanced the generality and adaptability of code LLMs, bringing them closer to meeting the demands of real-world scenarios. Early code instruction tuning data were extracted from various developer communities and platforms [712], such as code-comment pairs from GitHub repositories [435] and user question-answer data from StackExchange [575]. Data of this type are written by humans and

conform to real-world data distributions; therefore, fine-tuning techniques based on such data are also known as Natural-Instruct [698]. However, because each data point is provided by different users from diverse sources, the quality is often inconsistent and difficult to filter. Moreover, the original data were not specifically created for instruction tuning and may not conform to the required format. For instance, code in a repository may not have a corresponding natural language comment. These factors limit the quality and extraction efficiency of Natural-Instruct data.

In contrast, self-instruct [661], an approach to enhance the instruction-following abilities of pretrained language models by iteratively learning from their own generated outputs, for leveraging more powerful LLMs to generate higher-quality and more standardized data through high-quality demonstrations and in-context learning. The Alpaca dataset [948] in the field of LLMs is a typical example based on self-instruct technology. Following this work, Code-Alpaca [152] replaced the general-purpose seed examples with code-related ones, thereby constructing the first Self-Instruct-based dataset in the code domain. Inspired by this, subsequent works have emerged, optimizing the self-Instruct method for code-related tasks. [Figure 19](#) summarizes the three typical methods for synthesizing code alignment data.

4.1.1. Single-Turn Supervised Fine-tuning

Complexity The code in the CodeAlpaca dataset mostly consists of basic operations such as object creation and arithmetic, lacking advanced algorithms and complex logic. To address this, Luo et al. [661] proposed the Evol-Instruct method, which guides the model to generate more complex augmented data based on the original data using a series of manually crafted heuristic rules. The augmentation process can be iterated for multiple rounds to continuously increase data complexity.

Diversity As the volume of generated data increases, the likelihood of the Self-Instruct method producing repetitive or similar data also increases, limiting the quality of the dataset. An effective solution is to introduce human-written code data from Natural-Instruct during the generation process, leveraging its rich diversity to avoid repetition. To this end, Luo et al. [656] proposed the Semi-Instruct method, which uses diverse data from Natural-Instruct as a foundation and employs the Self-Instruct method to rewrite it, thereby enhancing data standardization. Similarly, the OSS-Instruct method [1067] adopts a similar construction process, with the difference that the raw data consists of code snippets rather than complete code, providing greater flexibility in the subsequent rewriting phase. Furthermore, Yu et al. [1229] constructed the Code Ocean dataset by first selecting diverse representative data through heuristic rules and embedding-based semantic similarity, and then using a CoT-like approach to verify the correctness of the selected data with a large model, thus obtaining correct and diverse results. Wu et al. [1094] explored using the inverse generation capability of the model to be fine-tuned, without relying on a larger model. It first cleans the outputs from Evol-Instruct, then uses the model itself to generate multiple instructions, and finally, the model itself determines whether the generated instructions are correct.

Sensitivity Luo et al. [658] propose a new sensitive dimension, which means the ability to capture detailed changes in instructions. It uses the CTF-Instruct framework, which augments counterfactuals to improve sensitivity. Such as changing use bullet points to use numbered lists or tweaking a length constraint. To measure this, they generate minimal-edit instruction pairs and check whether the model's outputs differ in exactly the prescribed way, using both

automated metrics (e.g., edit-distance, style classifiers) and targeted human judgments.

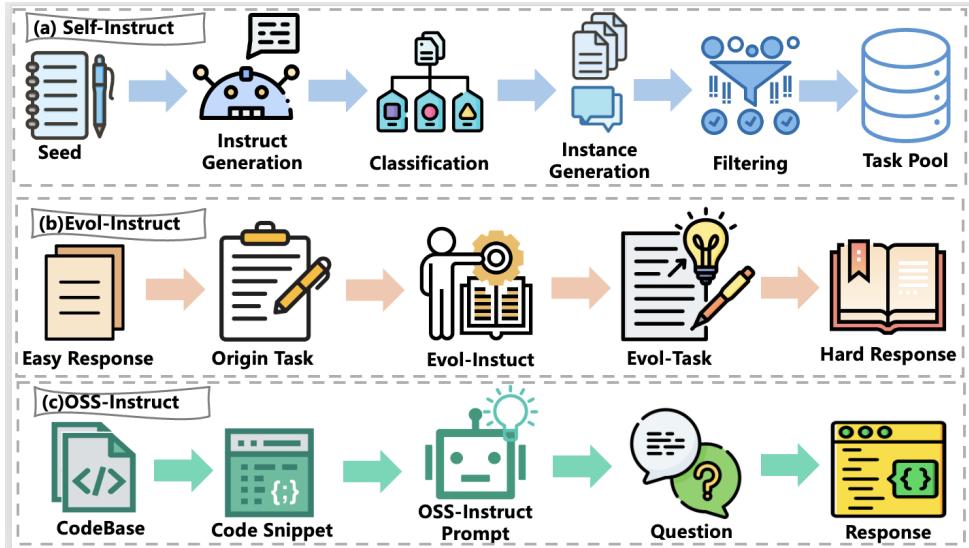


Figure 19. Three typical methods for synthesizing code alignment data.

4.1.2. Multi-Turn Supervised Fine-tuning

Execution Feedback A key feature that distinguishes code from natural language is its executability. Therefore, generated code can be verified using compilers and interpreters, providing low-cost and high-efficiency feedback signals without human intervention to guide improvements after errors occur.

Multi Agent AIEV-Instruct (Instruction Tuning with Agent-Interaction and Execution-Verified) [527] sets up a questioner agent and a programmer agent based on the same large model. When the code generated by the programmer agent fails verification, the questioner agent intervenes, asking questions based on the error to facilitate improvement. Data constructed with feedback are inherently multi-turn, whereas users in practical applications generally prefer single-turn interactions whenever possible. To bridge this gap between training and inference, Ren et al. [843] proposed a self-distillation method. It adds a summary-like turn at the end of the multi-turn data, which directly generates the multi-turn optimized code based on the user’s original question. The training process adopts a method similar to scheduled sampling [103], masking all but the last turn with a certain probability, and gradually increasing the masking probability as training progresses. This allows the model to fully learn from the knowledge in multi-turn interactions during training while ultimately adapting to the single-turn generation requirement at the inference stage. Yuan et al. [1231] introduce multi-turn interaction from the verification level to the problem-solving level using the feedback at each step.

4.1.3. SFT for Repository Tasks

While traditional code instruction tuning has focused on isolated functions or file-level tasks, the complexity of real-world software development demands models that can understand and navigate entire repository structures. Repository-level SFT datasets have emerged as a critical resource for training code LLMs to handle cross-file dependencies, multi-file edits, and long-context reasoning that characterize practical software engineering scenarios.

Software Engineering Task Datasets The SWE family of datasets has become the cornerstone for training autonomous coding agents. SWE-smith [1187] introduces a scalable pipeline that generates a large collection of task instances from GitHub repositories, representing an order of magnitude increase over previous works. SWE-Synth [793] complements this by synthesizing verifiable bug-fix data through LLM-driven debugging workflows, producing not only bug-fix pairs but also test cases and structured repair trajectories. SWE-Gym [773] provides Python task instances with executable runtime environments, enabling reinforcement learning approaches. SWE-Dev [270] specifically targets feature-driven development, addressing a task type that comprises a significant portion of real-world development efforts. Skywork-SWE [1252] tries to demonstrate clear data scaling laws across thousands of task instances from multiple repositories.

Offensive Cybersecurity Task Datasets On the offensive cybersecurity side, recent work has introduced CTF-style benchmarks that target vulnerability discovery and exploitation. Cyber-Zero [1342] proposes a runtime-free framework that synthesizes high-quality agent trajectories from public CTF writeups, using persona-driven LLM agents to reverse-engineer plausible environment behaviors and generate long-horizon interaction sequences capturing both successful exploits and realistic failed attempts. The resulting corpus spans thousands of challenges across diverse CTF categories and enables open-weight models to approach the performance of frontier proprietary systems on standard CTF benchmarks. Complementing this trajectory-focused perspective, CTF-Dojo [1343] provides a large-scale, execution-ready environment containing hundreds of fully functional CTF challenges packaged in secure Docker containers. Built on top of pwn.college artifacts via the CTF-FORGE pipeline, which automatically constructs and validates runtime environments with over 98% success rate, CTF-Dojo supports systematic trajectory collection from multiple LLMs and reveals key factors for building effective cybersecurity agents, including the importance of writeup-guided interaction, runtime environment augmentation, and teacher-model diversity.

Code Completion and Repository Navigation Several datasets focus on repository-level code completion and editing capabilities. RepoBench [621] evaluates auto-completion through three interconnected tasks across 10,345 Python and 14,956 Java repositories, specifically measuring cross-file context understanding. CoEdPilot [601] addresses incremental code edits, collecting over 180,000 commits from 471 projects across 5 programming languages to evaluate edit location prediction. RepoST [1127] introduces sandbox testing to isolate target functions for execution, containing 7,415 functions from 832 repositories. For hardware design, RTL-Repo [40] extends repository-level training to Verilog, a hardware description language (HDL) [1175], with over 4,000 samples including full repository context ranging from 2K to 128K tokens.

Repository-level SFT data presents unique challenges compared to traditional code datasets. The computational cost of maintaining execution environments is substantial, with some datasets requiring terabytes of storage for Docker images. Ensuring data quality while scaling remains difficult, as automated generation methods may introduce subtle errors that are hard to detect without comprehensive testing. Additionally, the diversity of repository structures, dependency management systems, and coding conventions across projects complicates unified training frameworks.

Despite these challenges, the consistent improvements across benchmarks and the clear data scaling laws indicate that investing in larger, more diverse repository-level datasets is a promising path for advancing autonomous programming systems.

4.1.4. Reasoning-based Methods

Paradigm Shift Towards Reasoning Reasoning-based LLMs [237] represent a significant advancement over traditional instruction-tuned models by incorporating explicit chain-of-thought (CoT) processes during inference, enabling them to decompose complex problems into intermediate steps and perform more systematic analysis. While instruction-tuned models are trained to directly map inputs to outputs through supervised fine-tuning on task-specific demonstrations, reasoning models employ techniques such as reinforcement learning from human feedback and process supervision to develop metacognitive capabilities that allow them to verify their own logic, explore multiple solution pathways, and self-correct errors before producing final answers. Specifically, CoT reasoning prompts models to generate a step-by-step thinking process, transforming complex requests into sequences of simpler, manageable steps that allow for more computational focus on intricate problems [1063]. This approach has proven highly effective in code generation, enabling models to build upon their own intermediate conclusions and verify logical consistency at each step [454]. By making the reasoning process explicit, CoT provides a more robust framework for tackling complex coding challenges in real-world software development (e.g., repository-based tasks) [1283] that require deep semantic understanding rather than just syntactic fluency.

Mechanism Interpretation The mechanism behind CoT’s success is tied to how it structures the problem-solving process for the model. Generating an explicit reasoning chain serves as a computational “scratchpad”, allowing the model to offload intermediate steps and conclusions into its context window [1063]. This process helps the model organize relevant information and focus its attention, which is crucial for complex tasks. Interestingly, the effectiveness of this process seems to depend more on the structure of the reasoning than the factual correctness of its content. Studies have shown that models can learn effective reasoning behaviors even when fine-tuned on CoT demonstrations with flawed logic or incorrect final answers [536, 1009]. This suggests that the primary role of the CoT during fine-tuning is to teach the model a structured, deliberative approach to problem-solving. The model learns the pattern of breaking down a problem, exploring alternatives, and verifying steps, a process that is more critical for success than learning the specific logical rules from any single example [536].

Supervised Fine-Tuning for Reasoning SFT on datasets of reasoning traces has become the primary method for instilling this capability in code LLMs [1283]. Research has shown that fine-tuning on a relatively small number of high-quality reasoning demonstrations can lead to significant performance gains on challenging benchmarks [536]. The quality of these demonstrations is crucial and selecting for difficult problems that require diverse reasoning strategies, such as exploration, backtracking, and self-verification, is more sample-efficient than simply scaling up the quantity of data [454, 570]. Furthermore, techniques that prune redundant or irrelevant tokens from the reasoning chain during generation can improve both performance and efficiency, encouraging the model to maintain a clear and focused thought process [186].

Rejection Sampling Fine-Tuning and Reinforcement Learning This SFT-based approach to teaching reasoning has a symbiotic relationship with Reinforcement Learning with Verifiable Rewards (RLVR) in [subsubsection 4.5.1](#), the other major paradigm for training advanced reasoning models. SFT provides the foundational reasoning structures and patterns, essentially teaching the model how to think. An important enhancement to standard SFT is rejection sampling fine-Tuning (RFT), which bridges supervised learning and reward-based optimization by

generating multiple candidate solutions, filtering them using verifiable rewards, and fine-tuning exclusively on correct outputs [1237]. This selective approach substantially improves data quality and reasoning diversity compared to basic SFT, while remaining more computationally efficient than full RL. RLVR then builds upon this foundation, using rewards to further refine the model’s policy and amplify the generation of correct solutions [240]. However, RLVR is fundamentally constrained by the reasoning patterns established during its initial training phases, including both SFT and RFT. It primarily sharpens the model’s existing capabilities, concentrating probability mass on known, high-reward reasoning paths rather than discovering entirely new ones [1082]. This limitation exists because RLVR gains are driven almost entirely by optimizing the model’s policy at a small fraction of high-entropy “forking” tokens, which represent critical decision points in the reasoning process [1030, 1157]. Therefore, high-quality initial training, encompassing both diverse SFT and selective RFT, is a crucial prerequisite for developing powerful and robust code LLMs.

4.1.5. Training Strategies

After data expansion, some work has focused on how to train more effectively. One aspect is quality filtering of large datasets. Tsai et al. [985] first clusters all data and retains a different number of data points from each cluster based on the density of the cluster center, thereby achieving data de-duplication. Wang et al. [1044] first focuses on data difficulty and correctness before considering diversity, training a complexity scorer to select the most difficult data and a test generator to compute the test case pass rate for each data point to select the most accurate ones. Besides data quality, multi-task balance is also important. Wang et al. [1043] treats code generation and code evaluation as a two-stage training task, significantly improving model performance. Liu et al. [598] integrates multiple loss functions and proposes a multi-task fine-tuning framework that can fine-tune tasks in parallel. Wang et al. [1046] introduces a denoising strategy that transforms some output tokens into random noise while keeping the LLMs predicting the following correct tokens based on them. This not only keeps the instruction following ability, but also adds denoising ability to accelerate inference.

4.1.6. Challenges

Although various methods and datasets have emerged in code instruction tuning, several challenges still remain.

Data Leakage Current instruction tuning datasets suffer from potential data leakage, where training data may contain information from test or benchmark sets. This leakage not only leads to inflated performance but may also conceal the model’s deficiencies in real-world scenarios [685]. Therefore, there is an urgent need to develop more rigorous and effective data filtering methods to ensure the purity of instruction datasets [562, 850, 1195].

Data Bias Existing instruction tuning datasets often exhibit a significant task complexity bias, with an excessive focus on simple programming tasks, such as programs that can be completed in just a few lines of code. This bias leads to poor model performance on complex tasks (e.g., long code generation, complex algorithm implementation, and system design), resulting in more pronounced performance disparities across tasks of varying difficulty [444]. Consequently, it is crucial to construct more balanced datasets that cover a wider range of task complexities.

Insufficient Multilingual Support Current code instruction tuning datasets are predominantly based on a narrow set of popular programming languages, such as Python or JavaScript. This focus limits the applicability and effectiveness of models for developers working in other critical, high-demand languages (e.g., C++, Rust, Go, or Swift). The lack of broad programming language support not only restricts the model’s utility across the diverse software development ecosystem but may also lead to significant performance degradation for tasks in less-represented languages. To address this issue, it is necessary to construct instruction datasets that cover a wider spectrum of programming languages. This requires not just a simple “translation” of problems into different syntaxes. Languages vary fundamentally in both surface syntax (variable declaration syntax, type annotations, comment formats, file extensions) and deeper semantics (idioms, standard libraries, memory management, and design patterns).

Functions	Control Flow	Resource Handling
Python — Implicit Typing <pre>def add(a, b): return a + b</pre> Java — Explicit Typing <pre>int add(int a, int b) { return a + b; }</pre> Swift — Explicit Typing <pre>func add(_ a: Int) -> Int { return a }</pre> JavaScript — Dynamic Typing <pre>function add(a, b) { return a + b; }</pre>	Python — Indentation-based Syntax <pre>if x > 0 and y > 0: print(" both positive ")</pre> Java — Parentheses + Braces Syntax <pre>if (x > 0) { System.out.println("positive"); }</pre> Swift — Mixed Syntax <pre>if x > 0 && y > 0 { print(" both positive ") }</pre> Ruby — Keyword-delimited Syntax <pre>if x > 0 and y > 0 then puts "both positive" end</pre>	Python — Context Manager <pre>with open("db.txt") as f: data = f.read()</pre> CPP — RAII <pre>{std::ifstream f("data.txt"); f >> data; } // f auto-closed here</pre> C — Manual Release <pre>FILE *f=fopen("d.txt","r"); fread(buf, 1, 100, f); fclose(f);</pre> Go — Deferred Cleanup <pre>f, _ :=os.Open("data.txt") defer f.Close() data, _ := io.ReadAll(f)</pre>

Figure 20. A comparison of programming language syntax across “Functions”, “Control Flow”, and “Resource Handling”, highlighting key differences in their design philosophies for typing, block structure, and memory management.

4.2. Cold-start / Distill Reasoning SFT data for Code LLMs

4.2.1. Data Sourcing

The objective of data sourcing is to acquire the raw problems and materials that form the foundational layer of a dataset. Across the research, a consistent practice is to begin with large-scale existing resources, which include online forums like Art of Problem Solving and Math StackExchange, academic systematicallycompetitions such as AIME and AMC, as well as extensive synthetic or distilled datasets. A variety of data sourcing methods are currently employed. OpenThoughts3 [354] systematically evaluated numerous sources and found that mixing a small number of high-quality sources, such as StackExchange, CodeGolf, and Open-CodeReasoning [12], produced the best results. The rationale is that quality of the source data is more beneficial than sheer diversity from a large number of sources. LIMO [1216] and s1 [713] focused on extracting a minimal yet highly effective set of samples from their extensive corpora. This approach is motivated by the hypothesis that a small number of carefully chosen examples are sufficient to elicit complex reasoning in models that already have a rich knowledge

base from pre-training. DeepMath-103K [394] and OpenMathReasoning [706] concentrated on extracting raw content from diverse but less structured sources, such as Math StackExchange and Art of Problem Solving community forums, and converting this content into a structured format. This method is designed to find novel and unique problems that are not present in more common, well-formatted datasets. AceReason-Nemotron [170] and Skywork-OR1 [388] sourced data from a combination of distilled datasets (e.g., DeepScaler [687], NuminaMath [544]) and competitive programming platforms to gather problems with high-quality, verifiable answers and test cases. This strategy is essential for their reinforcement learning pipelines, which rely on precise, rule-based reward signals.

In conclusion, while all projects begin with sourcing from large datasets, their strategies diverge based on their specific goals. Some prioritize finding a few high-quality samples to prove a hypothesis about data efficiency, while others aim for a massive scale to push the boundaries of model performance. The choice of source materials is closely tied to the subsequent data curation and filtering pipelines, which are tailored to the desired training paradigm, whether it's standard SFT, or more advanced methods like RL with verifiable rewards.

4.2.2. *Data Cleaning and Decontamination*

The primary purpose of data cleaning and decontamination is to enhance the integrity and reliability of the training data. This process is crucial to prevent models from overfitting to specific examples and to ensure that evaluation results are not skewed by contamination from benchmark datasets. By removing low-quality, incomplete, or duplicate samples, researchers can create a more robust foundation for training powerful reasoning models. Several distinct strategies for data cleaning and decontamination have been developed. To address the issue of subtle overlap with evaluation benchmarks, DeepMath-103K [394] implemented a highly rigorous process using an LLM-Judge (specifically, Llama-3.3-70B-Instruct) to perform semantic comparisons. This method identifies not only exact duplicates but also paraphrased problems, ensuring a truly clean dataset. OpenMathReasoning [706] and OpenThoughts3 [354] employed similar LLM-based comparisons or n-gram matching to remove similar questions from their datasets, providing a strong defense against data leakage. This level of scrutiny is an advantage as it goes beyond simple string matching to catch more nuanced forms of contamination. For models trained with reinforcement learning or tool-integrated reasoning, data quality is defined by more than just uniqueness. Skywork-OR1 [388] and AceReason-Nemotron [170] focused on filtering out problems that were unsuitable for rule-based verification, such as proofs, multiple-choice questions, or those with insufficient test cases. DeepMath-103K [394] took this a step further by filtering out problems that were not only verifiable but also produced consistent answers across multiple solutions generated by a teacher model. This ensures that the reward signals used in RL are unambiguous and reliable, a crucial aspect for stable training. The s1 [713] project's methodology included filtering out questions that were either too easy or too difficult for the target model (Qwen2.5-32B-Instruct) to answer. This approach ensures that the final dataset contains only a concise set of high-leverage examples, which is a key advantage for projects aiming for sample efficiency.

In summary, current practices for data cleaning and decontamination go far beyond basic deduplication. They now incorporate sophisticated techniques like semantic analysis with LLMs to identify hidden contamination, verifiability checks to ensure data is suitable for advanced training paradigms like RL, and model-aware filtering to create highly-efficient, targeted datasets. These rigorous steps are fundamental to building robust and generalizable reasoning models.

4.2.3. Question Filtering and Quality/Difficulty Assessment

The objective of question filtering and quality assessment is to select a high-leverage subset of problems from a larger pool, ensuring that these problems are challenging enough to stimulate complex reasoning without being unsolvable. This process is critical because training on every available data point is often computationally prohibitive, and a carefully curated subset can be significantly more effective for developing robust model capabilities. The research employs several distinct methodologies for question filtering and quality assessment. A common approach is to use model performance as a proxy for problem difficulty. The LIMO [1216] and s1 [713] projects adopt a “Goldilocks” strategy by retaining problems of intermediate difficulty. LIMO [1216], for example, filters out problems that a weak model could easily solve and keeps only those a strong model (DeepSeek-R1-Distill-Qwen-32B) could solve in a narrow range of attempts (1-3 out of 32). This method is highly advantageous for creating a minimal, high-signal dataset that is exceptionally sample-efficient. In contrast, DeepMath-103K [394] and AceReason-Nemotron [170] deliberately seek out highly difficult problems. DeepMath-103K [394] uses GPT-4o to rate problems on a 1-10 scale and retains only those at level 5 or higher. AceReason-Nemotron [170] uses the pass rate of a powerful model (DeepSeek-R1-671B) over multiple rollouts to assign a difficulty score, ultimately filtering problems that the model consistently fails to solve. The benefit of this strategy is to push the model’s reasoning boundaries and enable it to solve problems that were previously unsolvable. Researchers have increasingly leveraged LLMs for qualitative filtering tasks. OpenThoughts3 [354] found that for math and science problems, filtering for questions that elicit longer LLM-generated responses was an effective strategy. This is based on the intuition that a longer reasoning trace indicates a more complex problem. Similarly, the Skywork-OR1 [388] project used LLMs to classify problems (e.g., as proof-based, multiple-choice, or valid) and to assess data quality based on criteria like clarity, completeness, and formatting. This approach ensures that problems are not only difficult but also well-formed and suitable for the intended training task. For datasets designed for reinforcement learning with verifiable rewards, filtering for verifiability is paramount. AceReason-Nemotron [170] and Skywork-OR1 [388] explicitly filter out problems such as proofs, interactive problems, or questions lacking comprehensive test cases, as these cannot provide reliable reward signals. This step is a key advantage for ensuring training stability by eliminating sources of noisy or ambiguous reward signals.

In conclusion, question filtering is a multi-faceted process that has evolved from simple data statistics to sophisticated, model-aware strategies. The optimal approach is highly dependent on the research goal: for maximizing sample efficiency, a “Goldilocks” strategy that targets intermediate-difficulty problems is effective; whereas for pushing the performance ceiling with RL, actively curating a dataset of high-difficulty and verifiable problems is crucial. The pervasive use of LLMs in this phase has enabled more nuanced filtering that assesses not just difficulty, but also problem quality, diversity, and suitability for specific training paradigms.

4.2.4. Reasoning Chain Generation

The primary purpose of reasoning chain generation is to create detailed, step-by-step solutions, such as CoT or tool-integrated reasoning, for each problem. These explicit reasoning traces are crucial for teaching a model the logical process required to solve complex problems, as opposed to simply memorizing the final answer. By providing a “cognitive template”, these chains enable a model to apply and structure its pre-trained knowledge effectively at inference time. Current methods for reasoning chain generation vary significantly in their approach and complexity. The most common method involves using one or more large, powerful “teacher” models to

generate solutions. These teachers, such as DeepSeek-R1, QwQ-32B, and Gemini Thinking Experimental, are often state-of-the-art models that can produce high-quality reasoning traces. The advantage of this approach is its relative simplicity and scalability, as demonstrated by projects like OpenThoughts3 [354] and s1 [713] which use this method to generate millions or thousands of high-quality samples, respectively. DeepMath-103K [394] takes this a step further by generating three distinct solutions per problem to support diverse training paradigms. For more specialized tasks, a multi-stage approach is often required. OpenMathReasoning [706] developed a complex pipeline specifically for building TIR data, which integrates natural language reasoning with Python code execution. This process involves an iterative cycle of generating, filtering, and re-training models to produce increasingly high-quality TIR solutions. This method’s advantage is its ability to create datasets that teach models to use external tools effectively, a capability that standard reasoning models often lack. The generated solutions are also tailored to the downstream training method. For example, AceReason-Nemotron [170] and Skywork-OR1 [388] create solutions that are formatted for rule-based verification, a necessary component for their reinforcement learning pipelines. This ensures that the generated data provides reliable and unambiguous reward signals, which is critical for stable and effective RL training.

In summary, the generation of reasoning chains has become a sophisticated process that goes beyond simple prompting. While distillation from powerful models remains a cornerstone, projects are increasingly employing more specialized techniques, such as iterative tool-integrated generation or targeted formatting for RL, to create datasets that are not only large but are also precisely engineered for their specific training goals. The ultimate goal is to craft reasoning traces that act as effective “cognitive template”, allowing models to generalize and solve new, unseen problems.

4.2.5. Solution Filtering and Refinement

The main purpose of solution filtering and refinement is to ensure that generated solutions are not just correct, but also of high quality and in an appropriate format. This process aims to create ideal “cognitive templates” for the model to learn from, which are characterized by logical coherence, clarity, and efficiency. By refining these solutions, researchers can improve the model’s ability to structure its own reasoning process, leading to better generalization and performance on unseen problems. Several distinct methodologies for filtering and refining solutions have been developed, each with specific advantages. To identify the most effective reasoning chains, LIMO [1216] employed a unique rule-based scoring system. This system evaluated solutions based on “meta-reasonin” features such as elaborated reasoning, self-verification, an exploratory approach, and adaptive granularity. The advantage of this approach is that it allows for the precise selection of a minimal set of high-quality examples, which is crucial for achieving high performance with few training samples. For specialized tasks like tool-integrated reasoning, filtering ensures that generated solutions provide meaningful value. OpenMathReasoning [706] applied a strict filtering process to its TIR data to retain only solutions where code execution was “novel and significant”. This involved using an LLM to classify whether a code block performed a new calculation or merely verified a previous step. The advantage is that it prevents the model from learning to use tools for trivial or redundant tasks, thereby reinforcing effective and purposeful tool usage. DeepMath-103K [394] implement a rigorous answer verification process to ensure the robustness of its data. After generating three distinct solutions for each problem with a powerful teacher model, it retained only problems where all three solutions produced an identical, verifiable final answer. This strategy’s main advantage is that it creates a highly reliable dataset with unambiguous ground

truths, which is essential for stable reinforcement learning with verifiable rewards. Interestingly, OpenThoughts3 [354] conducted extensive experiments on various answer filtering techniques and found that none of them provided a significant performance improvement over simply training on all generated answers. As such, their final pipeline omits this step. This suggests that for some training paradigms, the noise introduced by lower-quality or redundant answers may be negligible and not worth the computational cost of filtering.

In conclusion, solution filtering and refinement has become a sophisticated, goal-oriented process. While some projects, like LIMO [1216] and OpenMathReasoning [706], use complex filtering to curate highly specific datasets for sample-efficient or tool-integrated learning, others, like OpenThoughts3 [354], find that simpler pipelines can be just as effective. The optimal strategy depends on the trade-off between data quality, data volume, and the specific requirements of the downstream training paradigm.

4.2.6. Final Dataset Construction

The final step of assembling the training dataset is to compile all processed problems and their refined solutions into a cohesive format ready for model fine-tuning. The primary goal is to create a dataset that aligns with the specific research objectives, whether that's to demonstrate data efficiency or to push the boundaries of large-scale model performance. This stage represents the culmination of all prior steps, from sourcing and cleaning to filtering and refining, ensuring the final product is optimized for its intended use. Current approaches to constructing the final dataset vary primarily in their scale and the strategic rationale behind that scale. Some projects, like LIMO [1216] and s1 [713], prioritize creating extremely small yet high-quality datasets to prove that extensive data isn't always necessary for achieving strong reasoning capabilities. The final LIMO [1216] dataset, for instance, contains just 800 examples, while s1 [713] is built on only 1,000 samples. This strategy's main advantage is its ability to demonstrate the power of data curation over data quantity, showing that carefully selected "cognitive templates" can be highly effective for alignment and generalization with minimal computational resources. Other projects, aiming for state-of-the-art performance and broad generalization, construct much larger datasets. OpenThoughts3 [354] scaled its final dataset to 1.2 million samples to achieve its performance gains. Similarly, DeepMath-103K [394] and OpenMathReasoning [706] built datasets with over 100,000 and 500,000 unique problems, respectively, to create a robust foundation for training powerful reasoning models across multiple domains. The advantage of this approach is that a large volume of diverse, high-quality data can push models to higher performance ceilings and enhance their ability to handle a wider variety of complex problems. Certain datasets are designed to serve multiple purposes. OpenMathReasoning [706] compiled a final dataset of 5.5 million samples from three different tasks: CoT solution generation, tool-integrated reasoning, and generative solution selection. This multi-task approach allows a single model to learn different inference modes, such as providing a CoT solution, using a code interpreter, or selecting the best answer from multiple candidates. This creates a versatile and powerful model capable of adapting to various reasoning challenges.

In summary, final dataset construction is a strategic decision that reflects the core hypothesis of a research project. The choice between a minimalist or large-scale approach dictates the kind of model that can be trained, with some researchers prioritizing data efficiency and others focusing on maximizing performance and versatility. Regardless of scale, the quality and intentional design of the final dataset remain the most critical factors for success.

Table 10. Summary of datasets and filtering strategies.

Dataset	Scale	Source	Model	Filtering Strategy
OpenThoughts3 [354]	1.2M (850k math, 250k code, 100k science)	OpenMath-2-Math StackExchange-CodeGolf OpenCodeReasoning StackExchange-Physics	QwQ-32B	≤2 top sources per domain Code: difficulty-based filter Math/science: answer-length filter Exact deduplication (math/science only) 16x answer sampling for all No answer filtering
LIMO [1216]	800	NuminaMath [544]-CoT DeepScaler [687] AIME, MATH	DeepSeek R1 DeepSeek-R1-Distill-Qwen-32B QwQ-32B	Multi-stage difficulty filtering: - Remove problems solvable by weak model - Keep those strong model solved 1-3/32 attempts N-gram deduplication Rule-based scoring on elaborateness
Skywork-OR1 [388]	105k math 13.7k code	Math: NuminaMath [544]-1.5 Code: LeetCode, TACO	DeepSeek-R1-Distill -Qwen-7B/32B	Keep verifiable, correct, challenging problems Model pass-rate difficulty estimation Human + LLM-as-Judge for quality check
DeepMath-103K [394]	103k	Math StackExchange MMIQC WebInstructSub	DeepSeek-R1	LLM-Judge semantic decontamination GPT-40 difficulty rating (≥ level 5) Retain problems with 3 consistent solutions
AceReason-Nemotron [170]	49k math 8.5k code	Math: DeepScaler [687], NuminaMath [544] Code: competitive platforms	DeepSeek-R1-671B	Remove benchmark contamination Model pass-rate scoring (8 attempts) Rule-based verification: - Math: sympy - Code: sandbox + test cases
OpenMathReasoning [706]	540k problems 3.2M CoT solutions	AoPS forums	DeepSeek-R1 QwQ-32B	LLM extracts and classifies problems LLM-based decontamination Filter CoTs not reaching expected answer Iterative TIR pipeline
s1 [713]	1k	NuminaMath [544] OlympicArena AGIEval s1 [713]-prob	Gemini Thinking Experimental DeepSeek-r1	Remove API errors and poor formatting Exclude problems easily solved Classify by domain for diversity Favor longer reasoning traces

4.3. Multilingual Code Understanding and Generation

4.3.1. Multilingual Code LLMs

Table 11 comprehensively illustrates the development trajectory of code large language models from 2020 to 2025, revealing significant evolutionary progress in model scale, language coverage, task capabilities, and training data.

Development Phases and Scale Evolution Early models (2020-2021) such as JavaBERT and C-BERT primarily focused on single programming languages with relatively small parameter scales (125M-350M) and training data from limited sources (e.g., specific GitHub repositories or Linux kernel code). The intermediate period (2022-2023) witnessed breakthrough progress, with multilingual models like Codex and AlphaCode expanding to 12B-41B parameters and supporting 6-12 mainstream programming languages. The latest generation of models (2024-2025) achieved a leap: DeepSeek-Coder-V2 reached 236B parameters supporting 338 languages, while StarCoder2 supports 600+ languages trained on 4TB+ data, marking the entry of code intelligence into the era of “ultra-large-scale multilingualism [641]”

Data and Architecture Innovation Training corpora have evolved from single GitHub repositories (hundreds of MB) to ultra-large-scale multi-source fusion (10TB+), with data sources including GitHub, Stack Overflow, CodeSearchNet, The Stack series, The Pile, synthetic textbook data, execution traces, and parallel translation corpora. Architecturally, the field exhibits diversity: BERT-based encoder models (CodeBERT, UniXcoder), T5-based encoder-decoder models

Table 11. Representative multilingual code models.

Model	Year	Task Type	Scale	Language	Corpus Type
Single Programming Languages					
JavaBERT [227]	2020	Java code understanding	125M	Java	Java GitHub repositories
C-BERT [1005]	2021	C code understanding	125M	C	Linux kernel code
JSCoder [5]	2021	JavaScript generation	350M	JavaScript	npm packages
GPT-Neo-Python [113]	2021	Python generation	2.7B	Python	Python code corpus
PyMT5 [193]	2021	Python code generation	220M	Python	GitHub Python corpus
InCoder-Python [295]	2022	Python code infilling	6.7B	Python	GitHub Python
RustCoder [902]	2023	Rust code generation	2.7B	Rust	Rust GitHub repositories
WizardCoder-Python [661]	2023	Python code generation	7B-34B	Python	Evo-Instruct + Code Llama
VeriGen [967]	2023	Verilog code generation	16B	Verilog	GitHub, Verilog textbooks
VerilogEval [614]	2023	Verilog evaluation benchmark	-	Verilog	HDLBits (156 problems)
ReasoningV [814]	2025	Verilog code generation	7B	Verilog	ReasoningV-5K, PyraNet
Multiple Programming Languages					
Codex [161]	2021	Code generation	12B	Python, JavaScript, Go, Java, PHP, Ruby	GitHub
CodeBERT [291]	2020	Code search, generation	125M	Python, Java, JavaScript, PHP, Ruby, Go	GitHub, CodeSearchNet
GraphCodeBERT [360]	2021	Code understanding, generation	125M	Python, Java, JavaScript, PHP, Ruby, Go	GitHub repositories
CodeT5 [168]	2021	Code generation, summarization, translation	220M	Java, Python, JavaScript, PHP, Ruby, Go, C, C#	GitHub, CodeSearchNet
CodeT5+ [1049]	2023	Code understanding, generation	220M-16B	Python, Java, JavaScript, C, C++, Go, etc.	GitHub, The Stack
PolyCoder [682]	2022	Code generation	2.7B	12 languages (C, C++, Python, Java, etc.)	GitHub (245GB)
AlphaCode [577]	2022	Competitive programming	41B	C++, Python, Java	Codeforces, GitHub
CodeGen [731]	2022	Code generation	350M-16B	Python, Java, JavaScript, C, C++, Go	The Pile, BigQuery, BigPython
PanGu-Coder [884]	2022	Code generation	317M-2.6B	Python, Java, JavaScript, C, C++	GitHub
PanGu-Coder2 [885]	2023	Code generation	15B	Python, Java, JavaScript, C++, Go	PanGu, internal data
SantaCoder [39]	2023	Code completion	1.1B	Python, Java, JavaScript	The Stack
StarCoder [630]	2023	Code completion, generation	15.5B	80+ languages	The Stack (GitHub)
CodeGen2 [730]	2023	Code generation, infilling	1B-16B	Python, Java, JavaScript, C, C++	The Stack, StarCoder data
Code Llama [858]	2023	Code generation, infilling	7B-34B	Python, C++, Java, PHP, C#, TypeScript, Bash	GitHub, Stack Overflow
WizardCoder [661]	2023	Code generation	15B-34B	Python, Java, JavaScript, C++, C#, PHP	Evo-Instruct + StarCoder
phi-1 [359]	2023	Code generation	1.3B	Python, JavaScript, Java	Synthetic data, StackOverflow
phi-1.5 [573]	2023	Code generation	1.3B	Python, Java, JavaScript, C++	Synthetic data, web corpus
DeepSeek-Coder [232]	2023	Code generation	1.3B-33B	87 languages	GitHub (2TB)
DeepSeek-Coder-V2 [1333]	2024	Code generation	16B-236B	338 languages	GitHub (10TB+)
Phind-CodeLlama	2023	Code generation	34B	Python, JavaScript, C++, Java, TypeScript	Code Llama + instruction tuning
CodeFuse [248]	2023	Code generation	13B-15B	Python, Java, JavaScript, C++, SQL	GitHub, internal data
Magiccoder [174]	2023	Code generation	7B	Python, Java, JavaScript, C++, TypeScript	OSS-Instruct synthetic data
OctoCoder [714]	2023	Code generation	15.5B	Python, Java, JavaScript, C++, TypeScript	StarCoder + Git commits
Yi-Coder [2]	2024	Code generation	1.5B-9B	52 languages	GitHub, StackOverflow
OpenCodeInterpreter [1312]	2024	Code execution, generation	6.7B-33B	Python, JavaScript, Java, C++, SQL	Code execution data
Qwen2.5-Code [343]	2024	Code generation	0.5B-32B	92 languages	GitHub, synthetic data (5.5TB)
CodeStral [19]	2024	Code generation	22B	80+ languages	Multilingual code corpus
TransCoder [921]	2020	Unsupervised code translation	-	C++, Java, Python	GitHub (monolingual)
DOBF [507]	2021	Code translation	-	Java, C#	Parallel corpus
TreeBERT [727]	2021	Code understanding	125M	Java, Python, JavaScript, Ruby, Go, PHP	Abstract syntax trees
SPT-Code [735]	2022	Code translation	220M	C++, Java, Python, C#, JavaScript	Syntax-preserved translation
UniCoder [361]	2022	Cross-lingual code understanding	125M	Python, Java, JavaScript, Ruby, Go, PHP, C, C#	GitHub, CodeSearchNet
CodeRL [512]	2022	Code generation with RL	770M	Python, Java, JavaScript, C++, Go, Rust	CodeNet, GitHub
CodeTransOcean [1158]	2023	Low-resource language translation	-	Java, C#, Python, JavaScript, C, C++	Parallel corpus with back-translation
CodeExecutor [600]	2023	Execution-based translation	-	Python, Java, C++, JavaScript, Ruby, PHP	Execution traces
REEF [1267]	2024	Low-resource code generation	-	Fortran, COBOL, Ada, Lisp	Legacy code corpus
Qwen2.5-xCoder [1181]	2025	code generation	-	Fortran, COBOL, Ada, Lisp	Legacy code corpus
Massively Multiple Programming Languages					
CodeGeeX [24]	2023	Multilingual code generation	13B	23 languages (Python, Java, C++, etc.)	The Pile, CodeParrot, GitHub
CodeGeeX4 [1311]	2023	Multilingual code generation	6B	100+ languages	The Stack, GitHub
CodeShell [1124]	2023	Code generation	7B	Python, Java, C++, JavaScript, Go, Rust	GitHub, internal Chinese data
StarCoder2 [641]	2024	Code generation	3B-15B	600+ languages	The Stack v2 (4TB+)
CodeQwen1.5 [961]	2024	Code generation	7B	92 languages	GitHub, internal data
Granite-Code [697]	2024	Code generation	3B-34B	116 languages	The Stack, GitHub
CodeGemma [1295]	2024	Code generation	2B-7B	20+ languages	GitHub, web corpus
PolyglotCode [861]	2024	Cross-family code translation	-	25+ languages across paradigms	Parallel multilingual corpus

(CodeT5 series, PyMT5), GPT-based decoder models (Codex, Code Llama), and graph neural network approaches integrating code structure (GraphCodeBERT, TreeBERT).

Ecosystem and Application Trends Table 11 reflects a complete industrial ecosystem: open-source foundation models (StarCoder, DeepSeek-Coder), commercial closed-source models (Codex, AlphaCode), instruction-tuned variants (WizardCoder, Phind-CodeLlama), domain-specific models (CodeFuse, CodeShell, incorporating Chinese data), and evaluation benchmarks (VerilogEval). Latest trends include: (1) small efficient models (phi series 1.3B, Yi-Coder 1.5B) pursuing the balance between performance and cost; (2) cross-lingual translation emerging as an independent research direction (TransCoder, CodeTransOcean, PolyglotCode); (3) multimodal fusion (code-natural language-execution results); (4) specialized optimization for low-resource and legacy languages, demonstrating that code intelligence is transitioning from generalization to refinement, and from high-resource languages to long-tail languages along a mature development path.

4.3.2. Multilingual Code Evaluation

Table 12 chronicles the evolution of code evaluation benchmarks from 2021 to 2025, showing a clear progression from single-language Python-focused datasets to comprehensive multilingual frameworks. Early benchmarks like HumanEval (164 problems) and MBPP (974 tasks) establish foundational evaluation paradigms for code generation, which were subsequently enhanced with expanded test coverage and specialized domains including data science (DS-1000), algorithmic challenges (APPS), and real-world software engineering tasks (SWE-bench series). The field evolved dramatically toward multilingual evaluation starting in 2022, with benchmarks like MultiPL-E (18 languages), HumanEval-X (5 languages), and eventually McEval (40 languages) addressing the need for cross-lingual code generation and understanding. Recent benchmarks (2023-2025) demonstrate increasing sophistication, incorporating diverse tasks such as code reasoning (CRUXEval-X across 19 languages), multilingual debugging (MdEval with 18 languages), full-stack programming (FullStackBench with 16 languages), and cross-lingual natural language generalization (HumanEval-XL with 23 natural languages \times 12 programming languages). The most recent entries reflect emerging trends toward multimodal evaluation (SWE-bench Multimodal) and practical repository-level tasks (Multi-SWE-bench), indicating a maturation from simple code generation benchmarks to comprehensive, real-world software engineering evaluation frameworks that span multiple programming languages, natural languages, domains, and modalities.

Specialized Single-Language Tasks The evolution of single-language benchmarks diversified into specialized domains. DS-1000 (2022) focused specifically on data science code generation across 7 Python libraries, while APPS (2021) tackled algorithmic challenges with 10,000 problems. More sophisticated benchmarks emerged to evaluate code reasoning and execution, such as CRUXEval (2023), which introduced input/output prediction tasks. The field further matured with real-world engineering challenges through the SWE-bench series (2024), which brought authentic GitHub issues into the evaluation paradigm, offering variants like Verified and Lite versions for different use cases.

The Multilingual Expansion Starting in 2021-2022, the field witnessed a paradigm shift toward multilingual benchmarks. Early efforts like mCoNaLa (2021) and MBXP (2022) began exploring code generation across multiple programming languages. MultiPL-E (2022) made significant strides by translating HumanEval to 18 different languages, while HumanEval-X (2023) provided hand-written multilingual tasks across 5 major languages. These benchmarks recognized that modern developers work across diverse programming ecosystems and that models need cross-lingual capabilities.

Emerging Trends and Future Directions The most recent benchmarks (2024-2025) reflect increasing sophistication and practical orientation. HumanEval-XL (2024) introduced an ambitious cross-product of 23 natural languages and 12 programming languages, creating over 22,000 prompts to evaluate true cross-lingual generalization. The field is also expanding beyond pure code generation: SWE-bench Multimodal (2025) incorporates visual elements into programming tasks, while Multi-SWE-bench (2025) tackles multilingual issue resolution, reflecting the complex, multimodal nature of modern software development. This progression demonstrates a clear trajectory from simple, single-language code generation toward comprehensive, real-world software engineering evaluation across languages, modalities, and domains.

Table 12. Multilingual code benchmarks and evaluation datasets.

Benchmark	Year	Task Type	Languages	Description
Single Language				
HumanEval [161]	2021	Code generation	Python	164 hand-written problems
MBPP [82]	2021	Basic programming problems	Python	974 entry-level tasks
HumanEval+ [609]	2023	Enhanced test coverage	Python	HumanEval with 80x more tests
MBPP+ [609]	2023	Enhanced test coverage	Python	MBPP with 35x more tests
APPS [396]	2021	Algorithmic challenges	Python	10,000 coding problems
DS-1000 [510]	2022	Data science code generation	Python (7 libraries)	1,000 StackOverflow problems
CRUXEval [346]	2023	Code reasoning & execution	Python	800 functions, input/output prediction
BigCodeBench [1339]	2024	Practical programming	Python	1,140 challenging real-world tasks
SWE-bench [928]	2024	Software engineering	Python	2,294 real GitHub issues
SWE-bench Verified [928]	2024	Human-filtered issues	Python	500 curated problems
SWE-bench Lite [928]	2024	Cost-effective subset	Python	300 selected issues
EvalPlus [609]	2024	Enhanced test suite framework	Multiple languages	Augmented HumanEval/MBPP
Multiple Languages				
mCoNaLa [1053]	2021	Multilingual code generation from NL	Python, Java, JavaScript	StackOverflow, CoNaLa
MBXP [78]	2022	Multi-lingual execution-based	13 languages (Python, Java, Go, etc.)	MBPP transpiled to multiple PLs
ODEX [1054]	2022	Open-domain execution	Python (79 libraries), 4 NLS	945 StackOverflow NL-code pairs
XLCoST [1330]	2022	Cross-lingual code translation	C++, Java, Python, C#, JavaScript, PHP, C	Parallel code snippets
MultiPL-E [139]	2022	Multilingual evaluation	18 languages (Lua, Racket, Scala, etc.)	HumanEval translations
CodeContests [575]	2022	Competitive programming	Multiple languages	AlphaCode training data
HumanEval-X [24]	2023	Cross-lingual generation	5 languages (Python, C++, Java, JS, Go)	820 hand-written multilingual tasks
xCodeEval [486]	2023	Cross-lingual multitask	Python, Java, JavaScript, C++, Go, Rust, etc.	GitHub + synthetic data
XCodeSearchNet [873]	2023	Multilingual code search	Python, Java, JavaScript, Go, PHP, Ruby, C, C#	Extended CodeSearchNet
CodeScore [258]	2023	Cross-lingual evaluation	20+ languages including Dart, Kotlin, Swift	Execution-based metrics
CrossCodeEval [252]	2023	Cross-lingual understanding	11 languages (Scala, Swift, Kotlin, Rust, etc.)	Parallel benchmark dataset
ClassEval [268]	2023	Class-level generation	Python	100 classes with 410 methods
RepoBench	2023	Repository-level completion	Python, Java	Cross-file code completion
CRUXEval-X [1140]	2024	Multilingual code reasoning	19 languages (C++, Java, Rust, etc.)	12,660 subjects, 19K tests, I/O prediction
HumanEval-XL [788]	2024	Cross-lingual NL generalization	23 NLS × 12 PLs	22,080 prompts, parallel data
McEval [141]	2024	Massively multilingual	40 languages (16K samples)	Code generation, completion, understanding
McEval [618]	2024	Multilingual debugging	18 languages (3.6K samples)	APR, code review, bug identification
FullStackBench [620]	2024	Full-stack programming	16 languages (3,374 problems)	11 domains, real-world scenarios
SWE-bench M [1185]	2025	Issues with visual elements	Python	517 multimodal tasks
Multi-SWE-bench [1247]	2025	Multilingual issue resolving	Multiple languages	Multilingual benchmark for issue resolving

4.4. Multimodal Code Understanding and Generation

4.4.1. Vision-Language Foundation Models for Code

Modern multimodal LLMs trace back to contrastive vision–language pretraining like CLIP [826] that established robust zero-shot perception by aligning image and text embeddings at scale. Subsequent “connector” designs couple frozen encoders with LLMs. BLIP [556] and BLIP-2 [557] demonstrate that lightweight adapters can efficiently bridge modalities while retaining strong generation and understanding. Architectures that ingest interleaved image–text sequences push few-shot generalization: Flamingo [32] conditions a language backbone on visual tokens for strong in-context transfer, and instruction-tuned stacks like LLaVA [531, 605, 606] demonstrate that synthetic visual dialogue corpora can elicit broad perception–reasoning skills with modest compute.

Large open families extend coverage and resolution handling: the Qwen-VL line [86, 88, 1029] brings grounding, OCR and dynamic-resolution processing. The InternVL Series [175, 176, 1034, 1329] evolve toward competitive open performance via model, data and test-time scaling. DeepSeek-VL [645, 1099] target real-world screenshots, documents, and charts, adding Mixture-of-Experts variants for efficiency.

Foundation releases from major labs have made vision first-class: Meta’s Llama-3.2-Vision [689] adds image understanding to an open LLM family, Google’s Gemma 3 brings a lightweight, open, multimodal stack derived from Gemini [956], while the Gemini series itself advances long-context multimodality [953, 955] and newer models focus on agentic use [204]. OpenAI’s GPT-4V [748] established broad image-understanding with a detailed safety analysis, GPT-4o [749] unified end-to-end text-vision-audio for low-latency interaction, and GPT-5 [754] continues the trend with updated capabilities.

Across these lines, we observe converging design motifs such as frozen or lightly trained

visual front-ends, compact cross-modal adapters, and instruction or preference tuning at scale, all of which yield steady gains in perception-reasoning breadth. These foundation models provide the underlying capabilities for specialized code-related multimodal tasks discussed in the following sections. Meanwhile, there are many popular multimodal benchmarks in Table 13 to evaluate the performance of the multimodal code understanding and generation.

Table 13. Representative Multimodal Code Generation Benchmarks.

Benchmark	Year	Task Type	Scale	Key Metrics	Innovation
Frontend Interface Generation					
Design2Code [899]	2024	Screenshot→HTML	484 pages	TreeBLEU, DOM-ED	Real-world benchmark
UICoder [1086]	2024	UI→Code	3M pairs	Compile-Render-CLIP	Auto feedback loop
Interaction2Code [1118]	2024	Interaction→Code	374 interactions	Event accuracy	Dynamic interaction
Sketch2Code [564]	2024	Sketch→Code	731 sketches	User preference	Interactive eval
Data Visualization					
nvBench [659]	2021	NL→Chart	Cross Domain	Cross Domain Database	First Large Scale Benchmark
Plot2Code [1080]	2024	Chart→Python	Scientific plots	Execution rate	Scientific chart focus
ChartMimic [1168]	2024	Chart→Code	Research papers	Cross-modal reasoning	Reasoning eval
nvAgent [767]	2025	NL→Chart	nvBench	SOTA performance	Multi-agent workflow
DeepVis [898]	2025	NL→Chart	nvBench	SOTA performance	Transparent Decision Refine
nvBench [655] 2.0	2025	NL→Chart	Cross Domain	Ambiguity Identification	Disambiguation Reasoning
ChartCoder [1303]	2025	Chart→Code	Large dataset	Code quality	Open-source enhance
Web-Embodied Intelligence					
WebArena [1326]	2024	Web tasks	800+ tasks	Task completion	Multi-domain platform
VisualWebArena [495]	2024	Visual web tasks	910 tasks	Success vs human	Visual + navigation
WebVoyager [383]	2024	End-to-end agent	15 websites	GPT-4V eval	Screenshot-based
Agent-E [6]	2024	Hierarchical agent	WebVoyager	Error recovery	Planner + Navigator
Flow2Code [390]	2025	Flowchart→Code	15 languages	Logic accuracy	Flowchart understanding
ArtifactsBench [1259]	2025	Code→Artifacts	Multi-type	Auto eval	Programmatic rendering
MMCCode [558]	2024	Visual programming	Rich visuals	Programming ability	Multi-visual elements
HumanEval-V [1264]	2024	Visual reasoning	Complex diagrams	High-level reasoning	Complex chart focus
MM-Coder [439]	2025	Design→Code	Multi-language	Code quality	UML + Flowchart

Table 14. Evolution of Evaluation Metrics in Multimodal Code Generation

Generation	Metrics	Focus
Traditional	BLEU [779], Code similarity [160]	Syntactic correctness
Structural	TreeBLEU [356], DOM-ED [923]	Hierarchical structure
Multimodal	Visual fidelity [465], Rendering similarity [169]	Cross-modal alignment
Automated	Code-in-the-Loop, MLLM judging [1259]	End-to-end evaluation

Table 15. Key Technical Trends in Multimodal Code Generation

Trend	Evolution	Representative Works
Agent Workflows	Plan→Execute→Observe→Reflect	nvAgent, Agent-E, Frontend Diffusion
Self-Correction	One-shot→Iterative optimization	UICoder, DesignCoder, ChartIR, ReLook
Hierarchical Generation	Flat→Multi-level decomposition	UICopilot, DesignCoder, WebDreamer
Code-in-the-Loop	Static evaluation→Dynamic rendering	ArtifactsBench, UICoder

With the rapid development of vision-language multimodal large language models (MLLMs), code generation research is expanding from pure text input to a new paradigm that incorporates multiple modalities such as images, sketches, and interactive signals. Multimodal Code Generation (MCG) aims to enable models to understand visual design intentions and generate executable, renderable high-quality code, thereby bridging the gap between high-level abstract thinking and low-level code implementation.

4.4.2. Core Challenges and Technical Positioning

Multimodal code generation faces two unified challenges: **Fidelity** - how to achieve high-fidelity restoration of visual details, structural hierarchies, and functional semantics in cross-modal transformations; and **Executability** - how to ensure that generated code is syntactically correct, renders without errors, and is functionally complete. These challenges drive researchers to explore new model architectures, training strategies, and evaluation paradigms.

Based on application scenarios and technical focus, this section divides multimodal code generation into three core subfields: frontend interface generation, web-embodied intelligence, and software engineering artifact generation.

4.4.3. Frontend Interface Generation

Task Evolution Trajectory Frontend interface generation, as a pioneering field in multimodal code generation, has undergone an evolution from single modality to multimodality, and from static to dynamic.

Early screen-to-code systems such as pix2code [101] demonstrated the feasibility of mapping GUI screenshots to platform-specific code, establishing the foundational paradigm for subsequent research. This pioneering work validated that deep learning approaches could bridge the gap between visual UI representations and executable code.

- **Image-to-Code** represents the starting point of this field. pix2code [102] pioneer the use of CNNs to extract GUI screenshot features combined with LSTMs to generate corresponding platform code.
- **Design-to-Code** represents standardization efforts in this field. Design2Code [899] built a large-scale benchmark containing 43,000 webpage screenshot-HTML pairs, systematically evaluate 9 MLLMs (multimodal large language models), and proposed hierarchical evaluation metrics such as TreeBLEU [356], DOM-ED [923]. The study finds that even GPT-4V still has many label omissions in maintaining hierarchical structures, highlighting the severity of fidelity challenges. Prototype2Code [1119] further utilize Figma APIs to build a 9k real prototype→React code dataset, proposing a pipeline consisting of hierarchical layout trees, component library retrieval, and incremental repair.
- **Sketch-to-Code** explores more natural interaction methods. Sketch2Code [564] released the first benchmark containing 731 high-quality hand-drawn sketches and designed two interactive evaluation modes: “passive feedback acceptance” and “active questioning”. The researchers find that although current models perform poorly in active questioning, this mode is more favored by UI/UX experts. WireGen [289] utilizes generative LLMs to automatically generate medium-fidelity wireframes from simple design intent descriptions. [329] further explored the feasibility of integrating visual code assistants in IDEs, validating the practical potential of sketch-to-code generation. Sketches are quick, usually hand-drawn UI drawings used early in design to explore ideas and communicate layout concepts without worrying about visual polish. Wireframes are more structured, low-/medium-fidelity blueprints that define the information hierarchy, layout, and interaction flows of a screen while intentionally omitting final styling and detailed content.
- **Interaction-to-Code** extends tasks to the dynamic level. Interaction2Code [1118] defined a new task paradigm, releasing a large benchmark containing 127 webpages, 374 interactions, and 31 event types, systematically revealing four major failure modes of SOTA MLLMs in interaction generation: event omission, logical errors, detail confusion, and visual detail loss.

Key Methodological Innovations Hierarchical Generation and Layout Modeling has become the mainstream approach for handling complex UI structures. UICopilot [357] splits HTML generation into a two-stage process of coarse-grained hierarchical skeleton followed by fine-grained tags and CSS, significantly reducing MLLM context length. LayoutCoder [1081] introduces element relationship graphs and layout tree prompts to better capture complex grid and float layouts. Further advancing this paradigm, DesignCoder [173] proposes a UI Grouping Chain to automatically group mock-ups into nested hierarchies, which is combined with a divide-and-conquer decoding strategy to enhance performance on industrial datasets.

- **Automated feedback and self-correction loops** significantly improve generation quality. UICoder [1086] pioneered compile-render-CLIP triple automatic feedback, filtering 3M LLM synthetic data for fine-tuning, enabling 7B-13B open-source models to achieve 12-18 point BLEU improvements on the Design2Code benchmark, approaching GPT-4V performance. DesignCoder’s self-inspection module uses browser screenshots + A Vision Encoder to detect and fix missing styles and invalid logic. ChartIR [118] transfers this idea to chart generation through a two-stage process of initial generation and optimization to gradually improve code quality. ReLook [574] introduces a reinforcement learning framework that closes the generate-diagnose-refine loop, leveraging a multimodal LLM as a vision-grounded critic and coach. By internalizing this self-correction capability during training, the model can perform low-latency self-editing at inference, even without relying on an external critic.
- **Agentic workflows** represent a shift toward intelligent workflows. Frontend Diffusion [255] proposes a Sketch-to-PRD (product requirements document)-to-Code three-stage agent chain, combining Pexels⁴ image retrieval with Claude-3.5, validating the feasibility of LLM-Agents in “Code→Rende→Self-Assessmen→ Revision” closed loops. User studies reveal AI-human bidirectional alignment requirements and the authors propose prompt layering and visual iterative interfaces.

Evaluation System Development The rapid development of this field benefits from continuous improvement of high-quality benchmarks. Design2Code not only provides large-scale data but also breaks through limitations of traditional sequence metrics like BLEU, proposing hierarchical metrics such as TreeBLEU [356], DOM-ED [923]. Web2Code [1242] builds large-scale webpage-to-code datasets, proposing comprehensive evaluation frameworks including webpage understanding and code generation, with experiments proving that this dataset improves not only webpage task performance but also general vision tasks. To avoid data leakage, the Snap2Code dataset specifically distinguishes between seen and unseen sites to ensure evaluation fairness.

Recent evaluation efforts have broadened the scope of UI understanding tasks. WebUIBench [594] offers a comprehensive benchmark covering element classification, visual grounding, OCR, layout understanding, and code generation, providing a holistic assessment framework for WebUI-to-Code capabilities. Complementary directions explore grammar-guided layout generation, where LLMs act as high-level planners for UI composition [650], and HTML structure understanding for web automation [371].

These developments suggest that general LLMs can increasingly bridge visual comprehension with front-end engineering. However, achieving robustness in fine-grained DOM (document object model) grounding and generating faithful, production-quality code remains a

⁴<https://www.pexels.com/>

key challenge for the field.

4.4.4. Web-Embodied Intelligence

Problem Definition and Challenges Web-embodied intelligence transcends the scope of static code generation, requiring AI agents to complete complex tasks in real web environments through “observe-reason-act” loops. These tasks not only test code generation capabilities but also require comprehensive reasoning, planning, and environmental interaction abilities.

Methodological Development Timeline

- **Stage 1: Foundational Frameworks** provide theoretical foundations for this field. ReAct [1209] proposes the “reasoning-acting” interleaved paradigm, becoming the cornerstone of modern agent architectures. Models first perform Chain-of-Thought reasoning, then execute actions, observe results, and proceed to the next step of thinking, effectively reducing hallucination phenomena. Workflow-Guided Exploration [603] guides reinforcement learning exploration through human workflows, laying foundations for learning in complex web environments. Toolformer [866] proposes methods for language models to self-learn tool usage, providing theoretical support for agent tool-use capabilities.
- **Stage 2: Task-Specific Platforms** drive practical progress. WebShop [1205] creates simulated online shopping website environments specifically for evaluating multimodal understanding and multi-step reasoning capabilities. WebArena [1326] provides more realistic and complex open platforms, including functionally complete environments replicated from real websites and over 800 long-term planning tasks, greatly promoting general web agent research.
- **End-to-End Multimodal Navigation** achieves significant breakthroughs. WebGUM [304] innovatively combines transformer language models (T5) and vision models (ViT), capable of simultaneously processing webpage HTML text and screenshot information. Web-Voyager [382] is a milestone work in this field, first achieving end-to-end multimodal web agents that directly use screenshots as input to simulate human browsing behavior, and innovatively uses GPT-4V for automated evaluation. WebLNX [304] releases large-scale benchmarks supporting multi-turn dialogue, containing over 100,000 expert demonstrations recorded on 150+ real websites.
- **Stage 3: Game Environments as Open-Ended Testbeds** provide rich, interactive settings for developing and evaluating multimodal agent capabilities. In Minecraft, video pre-training (VPT) [94] learns to act from large-scale human videos using a small labeled set for inverse dynamics, demonstrating long-horizon control from native mouse–keyboard interfaces. MineDojo [282] contributes an internet-scale knowledge base plus a simulation suite, enabling language-conditioned agents and reward shaping from video–language models. Building on this ecosystem, Voyager [1014] uses an LLM to drive open-ended exploration and lifelong skill acquisition via an evolving library of executable programs. These results indicate that general LLMs, when augmented with perception and code execution, can acquire reusable competencies in complex environments. Complementing these interactive game-playing agents, V-GameGym [1281] addresses the inverse problem of visual game generation, providing a comprehensive benchmark of 2,219 Pygame samples with multimodal evaluation across code correctness, visual quality, and gameplay dynamics.
- **Stage 4: Advanced Agent Architectures** improve complex task handling capabilities. Agent-E [6] proposes hierarchical agent architectures, decomposing complex web tasks

into “planner” and “browser navigator”, enabling agents to more effectively handle long-term tasks, error recovery, and backtracking. WebDreamer [351] innovatively uses LLMs as internet world models, first simulating consequences of each possible action, then evaluating and selecting optimal paths, effectively reducing trial-and-error costs on real websites.

- **Stage 5: Multi-Agent Collaboration** explores swarm intelligence. AgentVerse [164] provides general multi-agent collaboration frameworks, improving problem-solving efficiency through dynamic task decomposition, agent assignment, and collaborative execution. Voyager [1014] achieves lifelong learning in Minecraft, demonstrating possibilities for open-ended continuous learning. Generative Agents [780] creates virtual towns with 25 generative agents, simulating credible human behavior and pioneering large-scale social behavior simulation.
- **Stage 6: Tool-Augmented Multimodal Reasoning** enables LLMs to leverage specialized visual modules for complex perception-action tasks. Systems such as Visual ChatGPT [1079] and MM-ReAct [1201] orchestrate calls to specialist vision models under LLM control for multi-step tasks, demonstrating the effectiveness of modular architectures. Code-driven assemblers like ViperGPT [924] compose VLM modules via generated Python, yielding explicit, verifiable pipelines for complex queries. For end-to-end interactive evaluation, benchmarks like Mind2Web [242] target general web instruction following, while AndroidWorld [840] focuses on smartphone tasks. These studies highlight that long-horizon perception-to-action competence remains challenging even for top models, underscoring open problems in UI grounding, memory, safety, and reliability.

Evaluation Environment Evolution The progression of this field relies on a foundation of robust evaluation environments, which require constant iteration and improvement. From WebShop’s [1205] closed shopping scenarios to WebArena’s [1325, 1326] open multi-domain websites, to WebVoyager’s [285] innovative evaluation paradigm using GPT-4V image scoring, evaluation dimensions have expanded from text correctness to comprehensive assessment of visual understanding, long-term planning, and interaction robustness.

4.4.5. Software Engineering Artifact Generation

Task Value and Positioning Software engineering artifact generation aims to create supporting artifacts for code, including UML diagrams, data charts, flowcharts, architecture diagrams, and other visual software engineering documents. These tasks have important value throughout the software engineering lifecycle: business logic understanding in the requirements phase, architecture visualization in the design phase, supporting documentation generation in the development phase, and system understanding support in the maintenance phase.

Sub-task Classification and Progress

- **Data Visualization Generation** is the most active branch in this field. nvAgent [767] proposes a multi-agent collaborative natural language to visualization system, utilizing four roles - an insight miner, visualization recommender, code generator, and narrative generator - for collaboration, achieving SOTA performance on the nvBench [659] benchmark. Similarly, DeepVis [898] propose an interactive visual interface that tightly integrates with the CoT reasoning process, allowing users to inspect reasoning steps, identify errors, and make targeted adjustments to improve visualization outcomes. Plot2Code [1080]

establishes a comprehensive benchmark for evaluating multimodal large language models in code generation from scientific plots. ChartCoder [1303] advances multimodal large language models for chart-to-code generation through instruction fine-tuning datasets. VisCoder [726] focuses on fine-tuning LLMs for executable Python visualization code generation. MatPlotAgent [767] explores single-agent data science visualization methods, while METAL [767] investigates multi-agent frameworks for chart generation with test-time scaling. ChartMimic [1168] emphasizes the cross-modal reasoning difficulty of Chart-to-Code, providing important evaluation benchmarks for this subfield. Beyond recognition and captioning, chart understanding increasingly targets *executable* reconstruction by producing plotting code that faithfully reproduces the input figure. ChartMimic [1169] frames this as a chart-to-code benchmark with multi-level automatic metrics, revealing substantial headroom even for strong proprietary models. Dedicated models like ChartCoder [1303] pair code-centric backbones with large-scale chart-to-code corpora to boost executability and visual fidelity. Broader chart reasoning datasets such as Chart-Bench [1152] probe complex visual-logical skills that underlie chart regeneration and editing. For code-centric multimodal pipelines, these tasks tie perception to verifiable outputs, offering precise failure signals that are valuable for iterative improvement. Furthermore, nvBench 2.0 [655] extend the task to ambiguous scenarios, defining patterns of ambiguity in natural language queries for visualizations and resolving said ambiguity through step-wise reasoning.

- **Software Diagram and Model Generation** covers broader software engineering scenarios. DiagrammerGPT [1243] generates open-domain diagrams from natural language through plan and review dual loops. Draw with Thought [213] uses chain-of-thought to reconstruct scientific diagrams into editable XML code. Flow2Code [390] evaluates flowchart-to-code mapping capabilities, covering 15 programming languages. Code-Vision [1015] focuses on flowchart-to-program logic reasoning. Unified UML Generation [97] explores automatic UML code generation from UML images. From Text to Visuals [525] converts mathematical problem text to SVG graphics, completing mathematical and scientific diagram sub-tasks. MM-Coder [142] proposes a multilingual multimodal software developer for code generation that can jointly understand software design diagrams (UML, flowcharts) and textual descriptions.
- **Comprehensive Multimodal Software Engineering Tasks** reflect systematic development in this field. SWE-bench Multimodal [1184] targets visual bug fixing in JavaScript software development, while CodeV [1275] addresses issue resolving with visual data, particularly chart-related bug fixing methods. MMCode [558] benchmarks multimodal large language models for code generation with visually rich programming problems, and HumanEval-V [1264] benchmarks high-level visual reasoning with complex diagrams in coding tasks. Both incorporate visual elements such as trees, graphs, charts, tables, and pseudocode into programming evaluation, building more comprehensive multimodal programming benchmarks.

Evaluation Paradigm Innovation ArtifactsBench [1259] proposes a breakthrough automated multimodal evaluation paradigm, solving the pain point of traditional evaluation ignoring visual fidelity and interaction completeness. This framework programmatically renders visual artifacts, captures dynamic behavior, then uses MLLM judges to comprehensively score code, visuals, and interactions according to task checklists. The work focus on: (1) Automated closed loop - completing the full process from code generation to quality assessment without human intervention; (2) Multi-dimensional evaluation - considering code correctness, visual fidelity, and interaction completeness; (3) Good scalability - supporting unified evaluation of different

types of software artifacts.

4.4.6. Technical Trends and Future Outlook

Through systematic analysis of three core areas in multimodal code generation, we identify four key technical trends:

Widespread Application of Agentic Workflows Table 15 shows that “plan→execute→observe→reflect” loop has become the mainstream paradigm for solving complex multimodal tasks. From nvAgent’s multi-agent collaboration to Agent-E’s hierarchical architecture to Frontend Diffusion’s three-stage agent chain, all reflect the core value of the agent paradigm in handling multi-step, multi-modal information fusion tasks.

Maturation of Self-Correction and Iterative Optimization Mechanisms The growing ability of models to autonomously detect and rectify errors has emerged as a pivotal technology for enhancing robustness, as evidenced by self-correction mechanisms like UICoder’s “compile–render–CLIP” triple feedback [1086], DesignCoder’s browser screenshot self-inspection [173], ChartIR’s structured instruction iteration [1133], and ReLook’s “generate–diagnose–refine” loop [574]. These approaches collectively underscore the critical role of iterative self-improvement, marking a significant paradigm shift from “one-shot generation” to “iterative optimization.”

Standardization of Code-in-the-Loop Evaluation Incorporating compilers and rendering engines as components of evaluation environments to achieve immediate, automated functional and visual feedback is becoming an important development direction in this field. Artifacts-Bench’s programmatic rendering evaluation and UICoder’s rendering feedback mechanisms both reflect this trend, not only improving evaluation objectivity but also providing reliable feedback signals for model self-correction.

Widespread Adoption of Hierarchical Generation Strategies Facing complex multimodal tasks, decomposing the generation process into multiple levels or stages has become the mainstream approach. UICopilot’s two-stage generation, DesignCoder’s hierarchical structure awareness, and WebDreamer’s world model planning all effectively alleviate long sequence generation difficulties and improve generation quality and controllability.

Despite substantial progress, several fundamental challenges persist across the multimodal code generation landscape [458, 558, 710]. Fine-grained UI hierarchy understanding and interaction semantics remain difficult, particularly for complex DOM or canvas manipulations. Multi-step perception-to-action planning still suffers from robustness issues in real-world deployment. Analyses of multimodal hallucination and grounding errors continue to motivate the development of domain-aligned data, structured representations, and executable-by-design evaluations. The field is converging toward evaluation paradigms that emphasize verifiable outputs (such as chart or code regeneration) rather than purely perceptual metrics, particularly as applications move toward professional software engineering settings. This shift reflects a broader maturation from proof-of-concept demonstrations to production-ready systems.

Looking ahead, multimodal code generation will achieve deepened development in three dimensions: **Stronger Agent Capabilities** - including significant improvements in long-term planning, environmental adaptation, and error self-healing abilities; **More Complete Evaluation**

Systems - building multi-dimensional, automated, and low-cost Code-in-the-Loop standards;
Richer Application Scenarios - expanding from web development to multi-platform including mobile and desktop, and deep integration with human-computer collaboration.

Multimodal code generation is gradually converging toward the grand vision of “digital world embodied intelligence” [138] - enabling AI to observe, think, program, and act, laying a solid foundation for the comprehensive release of next-generation software productivity. As technology continues to mature and application scenarios continue to expand, this field will become an important bridge connecting human creativity and machine execution power, driving fundamental transformation in software development paradigms.

4.5. Task-based Overview of Reinforcement Learning in Code Intelligence

4.5.1. Reinforcement Learning (RL) Algorithms

For code LLMs, RL algorithms [1269] follow the specific pattern (compiles→runs→passes tests) by incorporating executable program-level rewards, such as unit-test pass rates, compiler/runtime errors, static-analysis flags, and verifier/critic feedback. This closes the loop between generation and external tools and directly optimizes correctness and reasoning depth. The proximal policy optimization (PPO) methods enable online exploration with KL-regularized policy updates that balance diversity against fidelity to a supervised prior, often coupled with execution-guided rewards and repair loops (generate→run→debug) to boost pass@k and reduce hallucinations [548, 870, 1174, 1193, 1314]. The direct Preference optimization (DPO) circumvents the need for online rollouts by learning from pairwise preferences distilled from execution-graded samples (success vs. failure), yielding stable alignment for code style and step-by-step reasoning without the variance of policy-gradient estimates [732, 827, 1345]. Reinforcement Learning from AI Feedback (RLAIF) further leverages stronger verifier/critic LMs to supply dense, structured signals (critiques, edits, step-level checks), enabling self-verification and iterative refinement that particularly benefit long CoT traces in math and programming [348, 1314, 1345]. Large-scale post-training (e.g., o1 and R1) integrates these ingredients (execution-aware rewards, verifier guidance, and preference optimization) to elicit robust tool use, decomposition, and self-correction behaviors that translate into sizable accuracy gains on code-generation and reasoning benchmarks [363, 758, 1174].

Proximal Policy Optimization Methods and Variants

- PPO [870] is the standard policy-gradient algorithm for RL-based LLM fine-tuning. It balances policy improvement and stability by constraining each update via a clipped surrogate objective:

$$L_{\text{PPO}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right], \quad (1)$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio, and \hat{A}_t the advantage estimate. Clipping with $\epsilon \in [0.1, 0.2]$ stabilizes training by preventing large updates. For advantage estimation, PPO typically uses Generalized Advantage Estimation (GAE) [869]:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}, \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t), \quad (2)$$

where λ controls the bias-variance trade-off. In practice, PPO is combined with KL regularization [91, 992] to keep π_θ close to the reference model, preventing reward hacking

(e.g., verbosity, repetition). This combination underpins RLHF pipelines such as Instruct-GPT [768], which fine-tuned GPT-3 [123] with a reward model trained from human labels, yielding significant alignment improvements. PPO has since been widely adopted (e.g., WebGPT [718]). However, vanilla PPO struggles on sparse-reward reasoning tasks due to *value collapse* (when the value network becomes degenerate and outputs nearly constant predictions across different states, losing its ability to meaningfully distinguish state values and guide policy improvement), where critic training fails. For example, Yuan et al. [1236] observed near-zero accuracy on long-CoT math benchmarks. To overcome this, several PPO-inspired variants were proposed.

- **Group Relative Policy Optimization (GRPO)** [880] avoids value learning by sampling G outputs for a prompt and computing relative advantages $A_i = r_i - \mu_{\text{group}}$. This yields stable updates without a critic and reduced memory overhead, proving effective in mathematical reasoning tasks. Deriving from similar ideas, there are akin research that try to improve the group baseline [581], rollout efficiency [572], and group diversity [1313].
- **Dr. GRPO** [636] addresses a critical bias found in GRPO used in reinforcement learning for LLMs. Standard GRPO suffers from an optimization bias that artificially inflates response length during training, particularly for incorrect outputs, leading to inefficient token usage. Dr. GRPO corrects this bias by providing a more balanced optimization approach that maintains reasoning performance while significantly improving token efficiency.
- **Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO)** [1225] improves PPO/GRPO with engineering refinements: (i) *Clip-Higher* to maintain entropy, (ii) *Dynamic Sampling* to adapt G , (iii) *Token-Level Loss* for long-horizon credit assignment, and (iv) *Overlong Reward Shaping* to penalize verbosity. DAPO achieved 50% accuracy on AIME 2024 with a 32B model, surpassing DeepSeek-R1 at half the training cost [935].
- **Value-based Augmented PPO (VAPO)** [312, 1241] tackles three critical challenges that arise in value model based reinforcement learning. These challenges include value model bias over long sequences, heterogeneous sequence lengths during training, and sparse reward signals in verifier based tasks. To address these issues, VAPO integrates several key techniques including length adaptive Generalized Advantage Estimation, value pre-training to reduce initialization bias, and enhanced strategies for balancing exploration and exploitation during training.
- **REINFORCE++** [408], a critic-free reinforcement learning framework, is designed to improve the efficiency and stability of reinforcement learning from human feedback (RLHF) for LLMs. While current methods like PPO require computationally expensive critic networks, and existing critic-free alternatives (e.g., GRPO [300]) suffer from biased advantage estimation due to prompt-level normalization, REINFORCE++ addresses these issues through global advantage normalization by normalizing advantages across entire global batches rather than small, prompt-specific groups. REINFORCE++ is proposed for broad RLHF applications and complex reasoning tasks, demonstrating superior stability and performance compared to existing methods.

Direct Preference Optimization (DPO) and Variants

- While PPO-based RLHF has been highly successful, it is resource-intensive, requiring online sampling, reward modeling, and delicate hyperparameter tuning. **DPO** [827] simplifies this pipeline by formulating preference learning as a direct policy optimization problem without an explicit reward model or on-policy exploration. Given a dataset of preference pairs (x, y^+, y^-) , DPO trains the policy to assign higher probability to the preferred response, bypassing the intermediate reward modeling step.

Formally, for a prompt x with preferred y^+ and dispreferred y^- , traditional RLHF would train a reward model $r(x, y)$ and then optimize via RL so that $r(x, y^+) > r(x, y^-)$. DPO collapses this into a direct adjustment of π_θ using a Bradley–Terry preference model. Under the assumption that the optimal policy π^* induces the true reward up to a scaling β , one obtains the contrastive loss:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\log \sigma\left(\beta [\log \pi_\theta(y^+|x) - \log \pi_\theta(y^-|x)]\right), \quad (3)$$

where σ is the sigmoid and β controls preference sharpness. Optimization is purely offline, resembling supervised fine-tuning with implicit labels. DPO avoids training a reward or value model, reduces memory overhead, and fully leverages offline preference data. This simplicity has driven strong interest in the open-source community [968, 1211, 1298]. However, one work [441] finds PPO with a large reward model can still beat DPO in some scenarios. Likewise, another work [1328] reports no gains of DPO over SFT for Starling-7B, whereas PPO with a tuned reward model yielded stronger results [281, 868, 870, 1021]. These findings suggest that the effectiveness of DPO depends on the base model, data distribution, and hyperparameter choices.

- **DPO-VP** [986] is a notable extension, which integrates verifiable signals (e.g., correctness in math/code). By treating correct outputs as preferred, DPO-VP directly optimizes against binary feedback. Compared to PPO baselines, a small LLM trained this way achieves better performance across five reasoning benchmarks, but with much lower compute. Tu et al. [986] further propose a multi-round framework where the policy generates candidates, verifiable signals or PRM/ORM labels, and DPO refines the model iteratively. This loop resembles highly off-policy RL but consistently uses the DPO loss. DPO occupies a middle ground between supervised fine-tuning and full RL, remaining close to SFT in implementation while still capturing reward-driven optimization effects. It incorporates human feedback without the inner loop of RL [441]. Current work explores hybrid strategies, where DPO provides a strong offline initialization, followed by PPO or other RL methods for further gains.
- **CodeDPO** proposed by Miao et al. [692] contribute a pipeline to collect preference pairs for model optimization. CodeDPO proposed by [1270] integrates preference learning into code generation models to optimize both the correctness and execution efficiency of generated code. The approach uses a self-generation and validation mechanism where code snippets and test cases are simultaneously created and evaluated through a PageRank inspired algorithm that iteratively updates credibility scores, assuming that tests executable by multiple code snippets are more reliable and code passing more tests is more likely correct.

Reinforcement Learning from AI Feedback (RLAIF) A central bottleneck of RLHF is the cost of collecting high-quality human feedback. **RLAIF** addresses this by replacing human supervision with AI-generated feedback—such as outputs from LLMs, symbolic verifiers, or hybrid systems [18, 521]. The key motivation is that advanced LLMs can often evaluate responses comparably to humans, while many domains (e.g., code or math) admit automated correctness checks [155, 347, 538, 822, 974, 977]. This substitution greatly improves scalability, especially where exhaustive human annotation is infeasible.

Lee et al. [522] first proposed training reward models on preference labels produced by off-the-shelf LLMs rather than humans, coining the term RLAIF. Subsequent comparisons showed that strong AI labelers can yield performance comparable to RLHF. For instance, RLAIF-trained policies matched RLHF on summarization and dialogue [399, 1334], and even same-model feedback improved policy quality [521]. Notably, Ivison et al. [441] reported that querying an AI

judge directly during PPO updates (skipping reward model training) outperformed two-step pipelines [501, 667], suggesting that learned reward models may introduce distortions.

RLAIF has been applied across reasoning and code domains. In mathematics, algebra solvers and verifier models provide correctness signals, underpinning recent reasoning systems such as DeepSeek-R1 [881, 904, 1028]. In code generation, compilers and unit tests enable automatic verifiable rewards. Ma et al. [671] showed that even single-example RL with unit-test feedback can nearly double performance on math benchmarks. Beyond symbolic evaluators, LLM-based critics supply structured feedback through self-refinement, CoT review, or constitutional principles, as in Skywork-OR1 [387], CRITIC [342], and Constitutional AI [92]. Together, these approaches establish RLAIF as a unifying paradigm encompassing symbolic signals, programmatic evaluators, and auxiliary LLM critics.

Despite its promise, RLAIF faces key challenges. Feedback quality remains a concern, since AI evaluators may propagate biases or blind spots, leading to reward hacking; mitigations include diverse evaluators, human oversight, or periodic evaluator refreshing. Computational cost is another limitation. Querying large models is expensive, motivating reward-model distillation and lightweight proxies [521, 522]. RLAIF demonstrates that scalable alignment is achievable by replacing or augmenting human supervision with automated feedback, but ensuring robustness and efficiency remains an open problem.

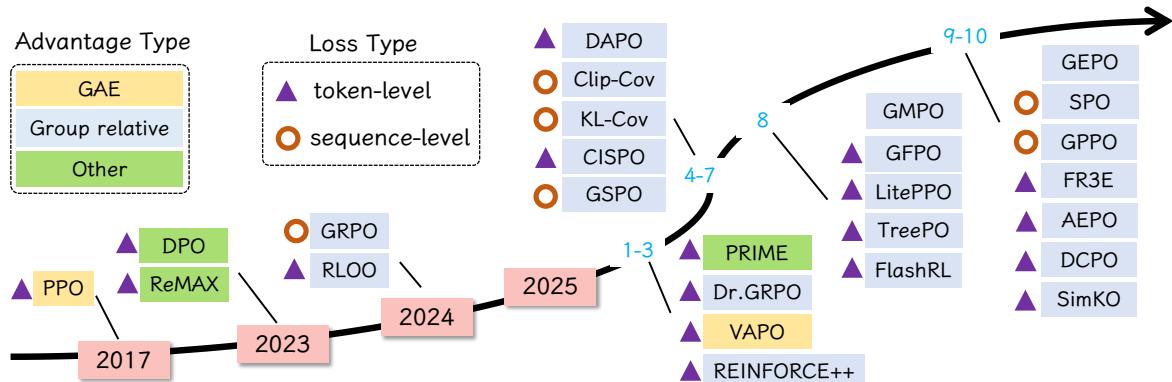


Figure 22. Overview of recent reinforcement learning algorithms for alignment.

4.5.2. RL for Code Generation

Figure 23 shows that reinforcement learning techniques are strategically adapted to address the distinct requirements of diverse code-related tasks. Code generation methods like CodeRL employ reward-based training for quality optimization, while completion tasks use incremental prediction approaches such as RLCoder. Understanding tasks integrate RL with attention mechanisms for summarization and documentation, whereas software engineering applications customize RL for multi-step reasoning in issue resolution and refactoring with task-specific rewards. Security and testing domains leverage RL for intelligent exploration, where fuzzing methods enable test input generation while bug repair systems use trial and error learning for defect localization and automated fixing.

Code Synthesis Code synthesis generates executable programs from natural language descriptions, particularly challenging in competitive programming tasks. CodeRL [513] pioneered the application of PPO-based approaches by treating the code-generating LM as an actor net-

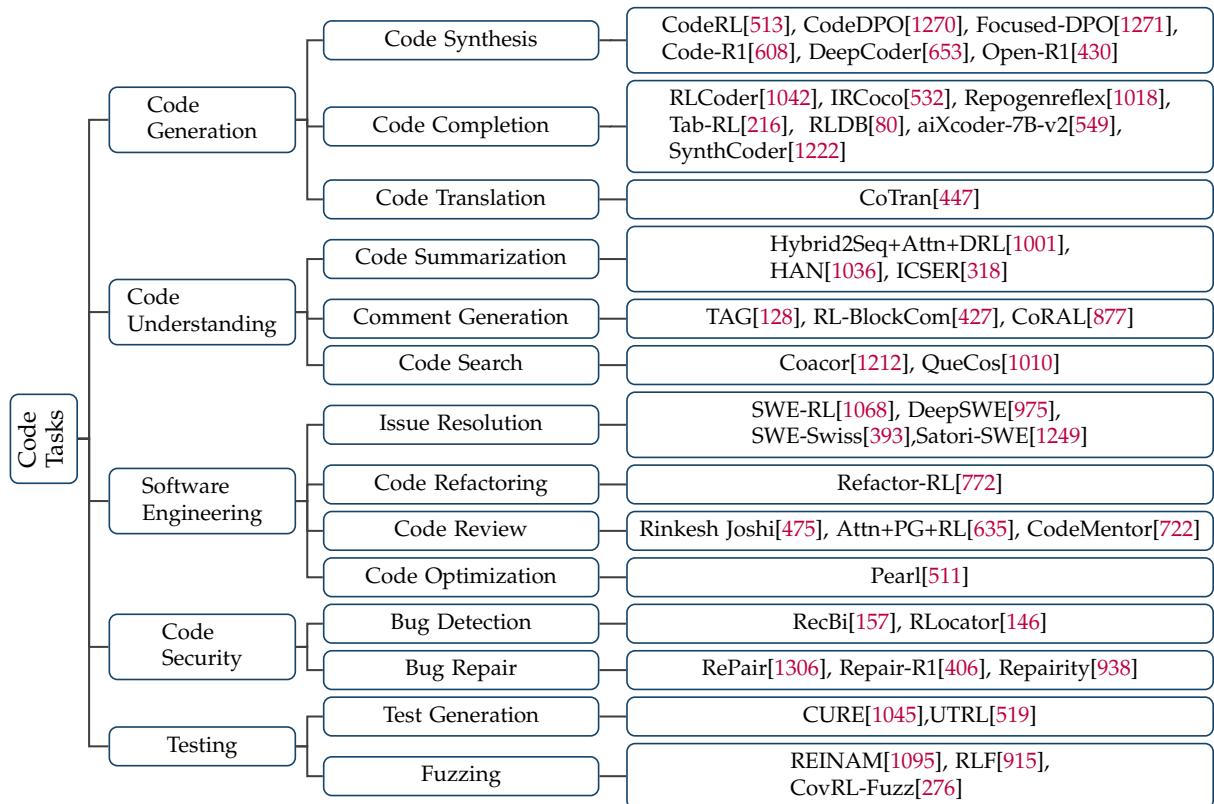


Figure 23. Taxonomy of coding tasks addressed by reinforcement learning methods

work with a critic network providing dense feedback signals, achieving state-of-the-art results on APPS [396] and MBPP [82] benchmarks. The field then shifted towards Direct Preference Optimization (DPO) methods. CodeDPO [1270] introduces a self-generation and validation mechanism with PageRank-inspired algorithms to create preference datasets optimizing both correctness and efficiency, while Focused-DPO [1271] further refines this by targeting error-prone points through error point identification. More recently, RLVR has emerged as a breakthrough approach that substantially enhances code generation capabilities. Code-R1 [608] demonstrates the importance of reliable reward signals, while the Open-R1 initiative [11, 13] releases validated datasets including Mixture-of-Thoughts with 350K reasoning trajectories and implements different reward functions with multiple sandbox providers. DeepCoder [653] showcases the effectiveness of these methods, achieving outstanding performance on LiveCodeBench [444] through distributed RL training and matching o3-mini[742] with just 14B parameters. The progression from PPO to RLVR represents a fundamental shift in how models learn to generate complex code, with RLVR’s verifiable rewards significantly improving performance (a topic we will explore further in subsection 4.6).

Code Completion Code completion has emerged as a prominent application of reinforcement learning in code generation, with recent advances addressing fundamental limitations of supervised learning approaches. RLCoder [1042] introduces an RL framework for repository-level completion that learns to retrieve useful content without labeled data, achieving 12.2% improvement on CrossCodeEval [254] and RepoEval benchmarks [848]. IRCoco [532] through immediate rewards-guided deep RL, providing instant feedback for dynamic context changes during continuous code editing. RepoGenReflex [1018] combines Retrieval-Augmented Gen-

eration (RAG) with Verbal Reinforcement Learning to dynamically optimize retrieval and generation processes. The aiXcoder-7B-v2 [549] addresses LLMs’ tendency to ignore long-range contexts through explicit RL-based supervision signals, achieving up to 44% improvement in exact match scores. SynthCoder [1222] integrates multiple optimization techniques including Curriculum Learning and Direct Preference Optimization with rejection sampling, establishing state-of-the-art performance on Fill-in-the-Middle tasks. In production environments, Cursor Tab [216] demonstrates the effectiveness of online RL by processing over 400 million daily requests and continuously updating models based on user feedback. While Augment Code’s RLDB [80] achieves performance gains equivalent to doubling model size by learning directly from developer-IDE interactions. These RL-based approaches collectively demonstrate significant advantages over traditional supervised methods by addressing exposure bias, improving context utilization, and enabling real-time adaptation to developer behaviors.

Code Translation Code translation converts programs between languages while preserving functionality. Feedback-driven approaches leverage execution feedback to guide model training: CoTran [447] pioneers reinforcement learning with compiler and symbolic execution feedback for Java-Python translation, while error correction methods like Rectifier [1217] utilize micro models to repair common translation errors across LLMs. kNN-ECD [1153] employs k-nearest-neighbor search with error correction sub-datastores to enhance TransCoder-ST translations. Reasoning-enhanced methods improve translation through intermediate representations: Explain-then-Translate [943] demonstrates that self-generated natural language explanations as intermediate steps achieve 12% average improvement in zero-shot scenarios across 19 languages. Retrieval-augmented approaches dynamically incorporate contextual examples: RAG-based few-shot learning [107] retrieves relevant translation pairs to guide models, while task-specific embedding alignment [108] optimizes retrieval quality for Fortran-C++ translation. Multi-agent systems decompose translation into specialized sub-tasks: TRANSAGENT [1238] employs four collaborative agents for initial translation, syntax fixing, code alignment, and semantic error correction, achieving superior performance through execution-based error localization. Repository-level translation addresses real-world complexity: AlphaTrans [436] introduces neuro-symbolic compositional translation with program transformation and multi-level validation for entire Java-to-Python projects. These diverse methodologies collectively advance code translation toward practical, scalable, and reliable automation.

4.5.3. *RL for Code Understanding*

Code Summarization Code summarization benefits from reinforcement learning to address the exposure bias issue inherent in traditional encoder-decoder frameworks. The Dual Model [1061] incorporates abstract syntax tree structures into an actor-critic network, using the critic to evaluate reward values of possible extensions for global guidance. HAN [1036] extends this approach with hierarchical attention mechanisms that integrate multiple code features including type-augmented ASTs and control flows, demonstrating substantial improvements in BLEU scores. Hybrid2Seq+Attn+DRL [1001] similarly employs deep reinforcement learning to mitigate the train-test discrepancy where models transition from ground-truth word training to full sequence generation during inference. ICSER [318] introduces a two-stage paradigm that first identifies model focuses through interpretation techniques, then reinforces the model to generate improved summaries, significantly enhancing DeepCom’s [414] performance. Collectively, these RL-based approaches demonstrate how actor-critic architectures and reward-guided training overcome the limitations of maximum likelihood training in capturing code semantics and structural information.

RL for Comment Generation Unlike code summarisation, comment generation targets developer-facing guidance tied to specific code spans and workflows as separate artifacts (e.g., review notes, API docs). Comment generation leverages reinforcement learning to produce more accurate and useful code documentation. TAG [128] introduces a Type Auxiliary Guiding framework that treats source code as an N-ary tree with type information, employing hierarchical reinforcement learning to handle the dependencies among type information for adaptive summarization. RL-BlockCom [427] focuses specifically on block comments within methods, combining actor-critic algorithms with encoder-decoder architectures to achieve a better performance through statement-based AST traversal. CoRAL [877] advances the field by designing reward mechanisms that consider both semantic similarity to expected comments and their utility as inputs for code refinement tasks, ensuring that generated comments are both meaningful and actionable. These approaches demonstrate how RL enables models to generate contextually relevant comments that better serve practical software development needs.

Code Search Code search leverages reinforcement learning to bridge the semantic gap between natural language queries and source code. CoaCor [249, 435, 873, 1212] adopts a novel perspective by training code annotation models to generate descriptions that enhance code retrieval performance, using RL to explicitly encourage annotations that improve distinguishability of relevant code snippets. QueCos [1010] addresses the semantic distance between user queries and code descriptions where code search performance serves as the reward signal for producing accurate query enrichment. Both approaches demonstrate that RL-guided semantic alignment substantially outperforms traditional matching-based methods in bridging the knowledge gap between natural language and code.

4.5.4. *RL for Software Engineering*

Issue Resolution Issue resolution in software repositories represents a complex challenge where RL has shown remarkable success in training models to autonomously fix real-world bugs. SWE-RL [1068] advances LLM reasoning through reinforcement learning on open-source repository, demonstrating that RL can effectively guide models through the intricate process of understanding and resolving GitHub issues. DeepSWE [975] scales RL training to create a fully open-sourced coding agent that achieves state-of-the-art performance on SWE-bench [928] through extensive reinforcement learning. SWE-Swiss [393] employs a sophisticated two-phase training strategy that first embeds localization, repair, and unit testing capabilities via multi-task SFT, then sharpens repair skills through targeted RL with direct feedback from test environments, achieving strong performance on SWE-bench Verified with just a 32B model. Satori-SWE [1249] introduces evolutionary test-time scaling (EvoScale) that treats generation as an evolutionary process, using RL to train models to self-evolve and improve their own generations across iterations, enabling the 32B model to match 100B+ parameter models while using fewer samples. These approaches demonstrate how RL transforms issue resolution from requiring human intervention to autonomous problem-solving at scale.

Code Refactoring Code refactoring [208] is the task of restructuring existing code to improve its internal structure, readability, and maintainability without altering its external behavior. Refactor-RL [772] addresses the limitations of traditional extract method refactoring approaches that require developers to manually identify refactoring boundaries and lack semantic understanding for meaningful method naming. The approach fine-tunes sequence-to-sequence models and aligns them using PPO, utilizing code compilation success and refactoring presence

as reward signals for code-centric optimization. This RL-aligned approach achieves 11.96% and 16.45% improvements in BLEU and CodeBLEU scores respectively over supervised fine-tuning alone, while increasing successful unit test passes from 41 to 66, demonstrating the effectiveness of reinforcement learning in producing functionally correct refactorings that capture both syntactic and semantic aspects of code transformation.

Code Review Code review automation benefits from reinforcement learning in predicting pull request outcomes and generating review artifacts. Joshi and Kahani [475] formalize PR outcome prediction as Markov Decision Processes with specialized reward functions to address data imbalance, achieving strong G-mean scores using PR features and even higher scores when focusing on PR discussions, effectively modeling both single-stage and multi-stage review processes. Attn+PG+RL [635] tackles the problem of missing PR descriptions by integrating pointer generator with direct ROUGE optimization through reinforcement learning, addressing out-of-vocabulary words in software artifacts while bridging the gap between training loss and evaluation metrics. CodeMentor [722] combines few-shot learning with RLHF, allowing domain experts to assess generated code and enhance model performance, achieving substantial improvements in code quality estimation, review generation, and bug report summarization. These approaches demonstrate how RL enables more effective collaboration in pull-based development by automating critical review processes.

Code Optimization Code optimization leverages reinforcement learning to automate complex compiler transformations. Pearl [511] introduces a deep RL framework that trains agents to select optimal sequences of polyhedral transformations, using a novel action space representation that determines both which optimization to apply and where in the loop nest to apply it. To address the data-intensive nature of compiler optimization where experiments typically require weeks, Pearl accelerates training through execution time memoization and actor-critic pre-training. Implemented in the Tiramisu compiler, Pearl achieves significant geometric mean speedups over state-of-the-art compilers while being the first RL-based system to support general tensor-manipulating loop nests with generalization to unseen programs, demonstrating how reinforcement learning can effectively navigate the complex optimization space previously requiring extensive manual tuning.

4.5.5. *RL for Code Security*

Bug Detection Bug detection [368, 1320] in compilers and software systems presents significant challenges due to limited debugging information and the vast search space for bugs. RecBi [157] pioneers reinforcement learning for compiler bug isolation by augmenting traditional mutation operators with structural ones to transform bug-triggering test programs into passing variants, then using RL to intelligently guide the generation of diagnostic test programs that effectively isolate bugs through execution trace analysis. RLocator [146] formulates bug localization as a Markov Decision Process to directly optimize ranking metrics rather than using similarity-based heuristics, outperforming state-of-the-art tools FLIM and BugLocator. These RL-based approaches demonstrate how direct optimization of evaluation measures and intelligent search guidance substantially improve bug detection effectiveness compared to traditional static analysis methods.

RL for Bug Repair Automated program repair [293, 618, 664, 970, 1291] has been significantly advanced by RL, enabling smaller models to achieve competitive performance. RePair [1306]

introduces process-based supervision, using a reward model as a critic to iteratively optimize repair policies, allowing LLMs to approach the performance of larger commercial LLMs. Repair-R1 [406] shifts the paradigm by first generating discriminative test cases and then using RL to co-optimize both test generation and bug fixing, leading to substantial improvements in repair and test generation success rates. Repairity [938] bridges the performance gap via a three-stage methodology that extracts reasoning traces from closed-source models, transfers knowledge via supervised fine-tuning, and applies LLM-guided reinforcement learning, nearly closing the gap with a leading commercial model. Collectively, these approaches demonstrate how reinforcement learning facilitates effective program repair at a smaller scale through intelligent feedback and reasoning transfer.

4.5.6. *Code Testing*

Test Generation Test generation [349, 882] has emerged as a crucial component for training and evaluating code generation models, with reinforcement learning approaches enabling the creation of high-quality test suites that expose subtle bugs and edge cases. CURE [1045] proposes a co-evolution framework where coding and unit test generation capabilities improve through their interaction outcomes without ground-truth supervision, achieving significant improvement in code generation accuracy. UTRL [519] trains test generators and code generators adversarially, where the test generator maximizes discrimination reward by exposing faults while the code generator maximizes code reward by passing tests, with Qwen3-4B trained via UTRL outperforming GPT-4.1 [753] in test quality. These approaches collectively demonstrate how reinforcement learning transforms test generation from a manual bottleneck into an automated process that significantly enhances both model training and evaluation reliability.

Fuzzing Fuzzing [421, 1192, 1296] is automated software-testing technique that feeds random or malformed inputs to programs to uncover bugs and security vulnerabilities. Fuzzing with reinforcement learning addresses the fundamental challenges of generating effective test inputs and navigating complex program behaviors. REINAM [1095] pioneers RL-based input-grammar inference without seed inputs, formulating grammar generalization as an RL problem and using symbolic execution engines to iteratively generate inputs that achieve superior precision and recall over existing approaches. RLF [915] tackles smart contract vulnerability detection by modeling fuzzing as a Markov Decision Process, designing reward mechanisms that consider both vulnerability detection and code coverage to guide generation of specific transaction sequences, detecting more vulnerabilities than state-of-the-art tools. CovRL-Fuzz [276] combines LLMs with coverage-guided reinforcement learning for JavaScript interpreter testing, integrating coverage feedback through TF-IDF weighted coverage maps to calculate fuzzing rewards, successfully identifying 58 real-world security bugs including 50 previously unknown bugs. These RL-based approaches collectively demonstrate how reinforcement learning transforms traditional random test generation into intelligent, adaptive exploration strategies that learn from program behaviors to discover vulnerabilities more effectively.

4.6. Applying Reinforcement Learning with Verifiable Rewards

Reinforcement Learning with Verifiable Rewards (RLVR) is an emerging paradigm designed to enhance reasoning capabilities in LLMs, particularly for domains such as mathematics and program synthesis. Formally, RLVR can be viewed as a reinforcement learning framework defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}_v, \pi_\theta)$, where \mathcal{S} represents the state space (e.g., the reasoning context or intermediate steps), \mathcal{A} denotes the action space (i.e., token generation or chain-of-thought

continuation), and π_θ is the model’s policy parameterized by θ . The distinctive component is the verifiable reward function \mathcal{R}_v , which is derived from an external, deterministic verifier. Instead of relying on learned reward models or heuristic preference scores, \mathcal{R}_v directly measures task correctness — for example, whether a generated mathematical proof reaches the correct answer or whether a piece of code passes all unit tests. The reward is typically binary (pass/fail), providing a clear and trustworthy signal that aligns precisely with task objectives.

The recent introduction of GRPO [881] and its large-scale application in DeepSeek-R1 [237] has driven RLVR to the forefront of research in reasoning LLMs. This family of algorithms and its application could be referred to a certain branch of RL algorithm ([subsubsection 4.5.1](#)), but focusing more the recently trending efficient PPO-variants, *i.e.*, those designs pruning the critic model. GRPO modifies the standard PPO framework by computing advantage estimates within a group of generated samples rather than training a separate value model. This “group normalization” approach eliminates the need for a learned critic network, significantly improving training efficiency and stability while maintaining alignment with the verifiable reward. Following DeepSeek-R1’s success in applying GRPO to mathematical reasoning — achieving high-quality long-chain reasoning purely from verifiable signals — a series of follow-up works have adopted the RLVR paradigm for both mathematical and code reasoning tasks, confirming its scalability and generality. These properties make RLVR particularly well-suited for reasoning-intensive domains such as algorithmic problem solving and code synthesis, where correctness can be explicitly checked by a computational oracle.

4.6.1. RLVR-Suitable Datasets for Code Tasks

A variety of specialized coding datasets have been developed to facilitate RLVR training. These datasets construct programming problems along with deterministic “verifiers” (e.g. input–output test cases or unit tests), ensuring the model outputs can be automatically checked for correctness. By providing clear pass/fail reward signals, they enable reliable reinforcement learning for code generation tasks. Notable representatives include the following datasets.

CodeContests A large competitive-programming dataset ($\approx 13.3K$ problems) for training AlphaCode models [229]. It compiles coding challenges from platforms including AtCoder, CodeChef, Codeforces, and HackerEarth. Each problem is accompanied by multiple human reference solutions and paired input–output test cases. These comprehensive test suites allow verifiable evaluation of a model’s code correctness on each problem, making CodeContests an early cornerstone for RLVR in code generation.

TACO (Topics in Algorithmic CODing) An algorithmic code dataset introduced by BAAI in late 2023, consisting of 25,433 training problems and 1,000 test problems up to 1.55 million total solution codes [563]. TACO aggregates tasks from CodeContests, APPS, CodeChef, Codeforces, GeeksforGeeks, and HackerRank. Covering common competitive-programming topics including sorting, dynamic programming, and graph algorithms. Each problem is paired with a set of I/O examples that serve as rigorous test cases. By focusing on verifiable algorithmic tasks, TACO provides a rich resource for training and evaluating code LLMs with automatic reward signals.

Eurus-2-RL A mixed-domain RL training set from the PRIME-RL project [211] containing 27K coding problems ($\approx 450K$ math problems) all equipped with verifiers. The coding subset

aggregates contest-style programming challenges sourced from APPS, CodeContests, TACO, Codeforces, and similar platforms, often translated into multiple programming languages. Each coding task is packaged with unit tests or I/O pairs that function as verifiable reward signals. Eurus-2-RL thus provides a large-scale, automatically-checkable corpus for jointly training models on code and math via RLVR.

IOI (International Olympiad in Informatics) Dataset A collection of recent IOI competition problems (years 2020–2024) with their official input files, output files, and checker programs by using the `testlib` system⁵. Although relatively small (229 problems), these tasks are advanced algorithmic challenges used in the IOI, each with a thorough set of secret test cases and an official verifier. This dataset is valuable for RLVR because the high difficulty of IOI problems demands complex reasoning, and the provided test suites offer an authoritative pass/fail reward signal for each generated solution.

ACECode-87K A synthetic coding dataset of roughly 87,000 problems created via automated test-case generation [1251]. Starting from a seed corpus of coding questions, the curators used GPT-4o-mini to “imagine” 16 diverse test cases per problem and filtered out any invalid or redundant cases. The result is a large collection of (problem, tests, expected outputs) triplets that are fully verifiable. ACECode-87K was primarily developed for training code LLMs with RL and the extensive test sets enable robust binary reward signals for code generation. This dataset demonstrated that synthetic but reliable test suites can substantially boost reward model training and subsequent RL fine-tuning of coder models.

SYNTHETIC-1 A massive open-source reasoning dataset comprising over 1.4 million tasks spanning mathematics, programming, software engineering, and more [684]. The coding subset alone contains approximately 144K program-synthesis and code-understanding problems, largely derived from public sources (e.g. APPS, CodeContests, Codeforces, and TACO). Crucially, each coding task includes a set of unit tests or I/O pairs as a verifier, often with the problem rephrased or ported to multiple languages for diversity. The broad coverage and automatic checkers in SYNTHETIC-1 make it suitable for both supervised fine-tuning and RL training of code-focused LLMs. Its diverse, verified problem set helps models learn generalizable reasoning strategies under the RLVR paradigm.

KodCode The largest fully synthetic coding problem dataset to date, introduced in 2025, offering about 48.4K (question, solution, test) triplets [1151]. KodCode is composed of 12 distinct themed subsets, ranging from basic algorithmic puzzles and classic data-structure problems to complex domain-specific programming challenges. Every example in KodCode is equipped with a thorough set of unit tests or I/O checks, ensuring each generated solution can be definitively verified. By emphasizing a wide diversity of task types including unusual or specialized coding scenarios while retaining automatic verifiability, KodCode serves as a challenging benchmark for code LLMs and a rich training ground for RLVR and supervised learning alike.

HardTests A competitive-programming dataset (about 47.1K problems) augmented with high-quality, synthesized test cases to reduce false positives in evaluation. Curated via an automated pipeline [395], HardTests took coding problems from a broad array of online judges

⁵<https://github.com/MikeMirzayanov/testlib>

(Codeforces, AtCoder, Luogu, LeetCode, etc.) and generated extra thorough test inputs for each, including tricky corner cases. All proposed solutions were executed against these tests to validate correctness. The resulting dataset provides more stringent verifiers for each problem, making reward signals in RL training far more precise. By catching edge-case failures, HardTests helps ensure that an RL-trained model’s improvements reflect genuine problem-solving ability rather than exploitation of weak test suites.

Code-R1-12K A curated RL training dataset consisting of 12k coding problem instances with associated tests, assembled as part of the open Code-R1 project [608]. It is built from 2k hand-picked LeetCode problems with reliable built-in unit tests and 10k additional verified problems filtered from the TACO corpus. The selection emphasizes low-noise, high-confidence reward signals and only problems with well-designed test cases and unambiguous correctness criteria are included. Code-R1-12K was used to train small code LLMs with the GRPO algorithm, showing that even a relatively compact, clean dataset can yield significant gains in code generation performance under RLVR.

LeetCodeDataset A comprehensive Python coding dataset covering about 90% of all LeetCode problems ($\approx 3.1K$ unique questions, as of mid-2024) [1116]. Each problem entry is enriched with metadata (difficulty level, topic tags, etc.) and paired with an extensive set of more than 100 test cases of varying difficulty. Notably, the dataset is temporally split into “pre-July 2024” and “post-July 2024” subsets to prevent data leakage between training and evaluation—ensuring that models can be trained on older problems and fairly tested on newer ones. This temporal split and the rich test suites make LeetCodeDataset ideal for verifiable-reward training for an RL agent can be rewarded for passing all tests on unseen problems, safe in the knowledge that it hasn’t memorized those solutions from training data.

CodeContests+ An enhanced version of the original CodeContests dataset, augmented by ByteDance’s SeedLab [1055] with additional high-quality test cases generated by an LLM-based agent. For each of the 11.4K competitive programming problems in CodeContests, the agent synthesized new challenging inputs and verified the correct outputs (through a sandbox executor or checker). These augmented test suites either supplement or replace the original ones, substantially increasing the true-positive rate of solution checking (i.e. reducing cases where an incorrect program might accidentally pass). Importantly, the underlying problem statements remain unchanged (only the evaluation criteria are strengthened). CodeContests+ thereby improves the accuracy and reliability of RL reward signals, without altering task content, enabling more trustworthy training and evaluation of code LLMs.

SYNTHETIC-2 The successor to SYNTHETIC-1, this dataset contains an even larger collection of automatically generated reasoning tasks (over 4 million verified problem-solving traces spanning coding, mathematics, logical puzzles, and more). Produced via a massive distributed inference pipeline (leveraging multiple LLMs in parallel), SYNTHETIC-2 [805] includes many problems beyond traditional competitive programming, such as code comprehension challenges, debugging tasks, and software engineering scenarios, all paired with known-correct answers or test-case validators. Every task is associated with a deterministic check (unit tests, proofs, or solutions) to serve as a reward signal. Although the coding subset ($\approx 61K$) is only a portion of the full release, the sheer scale and variety of SYNTHETIC-2 make it one of the largest corpora

for training LLMs in a verification-driven manner. It provides an unprecedented opportunity for both supervised fine-tuning and RL to push model reasoning capabilities to new heights.

Klear-CodeTest A dataset of 27,965 competitive-programming problems curated by the Kwai Klear team [297], each equipped with rigorously synthesized and validated test cases. Klear-CodeTest was constructed using a generator-and-checker framework. An LLM first proposes candidate test inputs for a given coding problem, then multiple solution attempts (and a reference solution) are run to check whether each test distinguishes correct code from incorrect. By iterating and filtering out redundant or ineffective cases, the pipeline produces a comprehensive test suite that covers the problem’s edge cases. The final dataset spans Codeforces, verified TACO/CodeContests problems, and other sources, with every example ready for automatic verification via either I/O comparison or a custom checker. Klear-CodeTest is intended for scalable RL training and robust evaluation of code LLMs, ensuring models learn to handle the full complexity of coding problems rather than overfitting to simplistic tests.

Collectively, the above datasets provide a foundation for verifiable-reward learning in code intelligence. They cover various programming challenges, each bundled with an oracle for correctness (test cases or checkers), so that a reinforcement learning agent can reliably gauge success. This abundance of automatically-checkable data enables large-scale RLVR training without human-in-the-loop labeling, and also supports rigorous evaluation of a model’s coding ability. By leveraging these resources, recent works have demonstrated substantial improvements in code generation accuracy and reasoning depth purely from self-play with verifiable rewards.

4.6.2. Representative RLVR-Trained Open-Source Code LLMs

Several open-source LLMs for code have recently been trained with reinforcement learning on verifiable tasks. Below, we highlight a number of such models, noting their base model, the scale and source of their training data, and the specific RL algorithms used. These examples illustrate the diversity of approaches in RLVR from standard PPO to GRPO and new techniques such as GPPO, as well as differences in whether a learned reward model or direct test-case feedback is employed. Despite varying in size and dataset scale, all these models report significant performance gains from verifiable reward optimization — often matching or surpassing much larger pretrained counterparts.

AceCoder [1250] is a code-centric LLM based on models such as Llama-3.1-8B or Qwen2.5-7B trained via a novel automated RL pipeline with REINFORCE++ algorithm. The model was trained on the AceCoder-87K synthetic dataset with GPT-generated tests, including a curated “hard subset” of 22K challenging problems with 16 test cases each. First, AceCoder constructs a reward model by generating preference pairs from test-case pass rates and training via Bradley–Terry loss. Then, the policy model is fine-tuned with either the learned reward model or direct binary test outcomes as the reward signal similar to DeepSeek-R1 [237]’s strategy. AceCoder improved pass rates by approximately 10–25% on HumanEval, MBPP, BigCodeBench, and LiveCodeBench compared to the supervised baseline. Notably, even with only 80 optimization steps on the base model, following the R1-style “zero-shot” RL setup, a 7B AceCoder model achieved over a 25% increase on HumanEval-plus and a 6% increase on MBPP-plus.

Table 16. Summary of representative open-source datasets for reinforcement learning with verifiable feedback in code intelligence tasks.

Date	Name	Type	#Sample	Test Format	Source	Link
2022.07	CodeContests	Algorithm	13.3k	IO pairs	Aizu AtCoder CodeChef CodeForces HackerEarth	 
2023.12	TACO	Algorithm	25.4k	IO pairs	CodeContests APPS CodeChef CodeForces GeeksforGeeks HackerRank	 
2025.01	Eurus-2-RL	Algorithm	26k	IO pairs	APPS CodeContests TACO Codeforces	 
2025.02	IOI	Algorithm	229	stdio and checker ⁶	IOI 2020-2024	 
2025.02	ACECode	Algorithm Code Understanding Debug	12.2k	unit	Evol OSS Stack Python	 
2025.02	SYNTHETIC-1	Algorithm SWE Code Understanding	275k	IO pairs	Apps Codecontests Codeforces TACO	
2025.03	KodCode	Algorithm	48.4k	pytest/IO pair	Leetcode Codeforces Apps Taco Code Contests Evol others	 
2025.03	HardTests	Algorithm	47.1k	IO pair	Luogu Codeforces AtCoder SPOJ CodeChef LeetCode GeekForGeeks others	 
2025.03	Code-R1-12K	Algorithm	12k	IO pairs/unit	LeetCode TACO-verified	 
2025.04	LeetCodeDataset	Algorithm	2.6k	unit	LeetCode	 
2025.06	CodeContests+	Algorithm	11.4k	stdio and checker	CodeContests	
2025.07	SYNTHETIC-2	Algorithm Code Understanding	61k	IO pairs	PRIME RL CodeForces	
2025.08	Klear-CodeTest	Algorithm	28k	IO Pairs/checker	Codeforces TACO-verified CodeContests	 

Open-R1 [430] is an open reproduction of the DeepSeek-R1 methodology, applied to code reasoning tasks and trained on Qwen2.5 series with extended context, RoPE 300k as the starting point. The training data comprises a collection of high-quality coding problems, including the IOI dataset, Codeforces tasks, and others, which is often supplemented with mathematical reasoning data to approximate the original R1 training distribution. The 7B-scale Open-R1 models, trained on a few thousand code problems with test oracles, demonstrated clear improvements over instruction-tuned baselines on coding benchmarks. Such an initiative showcases that

Table 17. Summary of representative open-source Code LLMs trained via reinforcement learning with verifiable rewards (RLVR). For each model we report the base model, datasets, RL algorithm, training length, and public links.

Date	Model	Base Model	Data	Algorithm	Length	Link
2025.03	AceCoder[1251]	Qwen2.5-Coder-7B-Instruct	AceCode	Reinforcement++	4K	
2025.03	Open-R1[430]	Qwen2.5-Math-7B	IOI CodeForces	GRPO	32K	
2025.03	Skywork-OR1[387]	DeepSeek-R1-Distill-Qwen-7B/DeepSeek-R1-Distill-Qwen-32B	LeetCodeDataset TACO	GRPO	16K→32K	
2025.03	Code-R1[608]	Qwen2.5-7B	Code-R1-12K	GRPO	6K	
2025.03	DeepCoder[653]	DeepSeek-R1-Distilled-Qwen-14B	LeetCodeDataset TACO SYNTHETIC-1	GRPO+	16K→32K	
2025.03	Seed-Coder[872]	Seed-Coder-8B-Base	CodeContests ICPC problems CodeForces LiveCodeBench	GRPO	16K→32K	
2025.03	AceReason [170]	DeepSeek-R1-Distill-Qwen-7B/DeepSeek-R1-Distill-Qwen-14B	AtCoder LeetCode Aizu	GRPO	24K→32K	
2025.03	Klear-Reasoner[917]	Qwen3-8B-Base	CodeSub-15K	GPPO	32K	

R1-level performance improvements can be achieved and reproduced openly and group-based RLVR consistently enhances code generation accuracy without proprietary data or methods.

Skywork-OR1 [387] is a series of models trained via large-scale GRPO starting on weights distilled from DeepSeek-R1-Distill-Qwen models. The training corpus included substantial verified coding and math datasets (e.g., LeetCodeDataset, TACO) with an emphasis on long-chain tasks. Models underwent multi-stage RL, with context length increasing from 16K to 32K tokens. Their 32B model achieved 63% pass@1 on LiveCodeBench (Aug 2024–Feb 2025) and surpassed DeepSeek-R1 on AIME 2024/25 [3, 4].

Code-R1 [608] demonstrates the efficacy of GRPO at a small scale. Fine-tuning a 7B Qwen2.5 model on just 12K high-quality problems yielded a 5-6% pass@1 gain on HumanEval+ over its SFT base, proving RLVR’s value even on a budget.

DeepCoder [653] is a 14B model trained on 24K vetted problems using GRPO+, an enhanced algorithm removing entropy/KL terms. DeepCoder-14B achieved 60.6% pass@1 on LiveCodeBench, matching OpenAI’s much larger 34B “o3-mini-2025” model [742] and showing mid-sized open models can rival proprietary systems.

Seed-Coder [872] employs a hybrid, two-stage RL pipeline: first using DPO to create an “Instruct” model, then applying a PPO-style algorithm on verifiable tasks to create a “Reasoning” variant. This model-driven curation approach outperformed larger models on multi-step tasks.

AceReason [170] introduces a two-stage PPO curriculum. Stage 1 trained only on mathematical prompts, which unexpectedly boosted both math and coding reasoning. Stage 2 applied code-only RL. This math-first curriculum proved highly effective for unlocking general reasoning.

Klear-Reasoner [917] is an 8B model trained on 45K expert-level tasks using gradient-preserving policy optimization (GPPO). This novel algorithm improved gradient utilization and mitigated premature entropy collapse, demonstrating the value of algorithmic innovation in RLVR.

In summary, these open-source models validate RLVR as a potent method for enhancing LLM reasoning. They showcase a diversity of successful strategies beyond the classic PPO, including value-free methods (GRPO) and gradient-preserving updates (GPPO), hybrid pipelines (DPO+PPO), and novel curricula (math-first-then-code). Two common themes emerge: the critical importance of high-quality, verified data (prioritized over quantity) and the use of strong, distilled base models to amplify RL gains. These openly-released models achieve SOTA results on benchmarks like LiveCodeBench, closing the gap with larger proprietary systems and providing a clear blueprint for advancing open-source reasoning AI.

4.6.3. Reward Shaping in Code Post-training

In the post-training phase of code LLMs, reward design serves as one of the core mechanisms to align model outputs with functional requirements and nuanced developer expectations. Effective reward formulation incentivizes code that is not only functionally correct but also secure, maintainable, and contextually aligned with real-world software engineering standards. This section establishes four foundational dimensions, including *correctness*, *task alignment*, *security*, and *structural quality*. These approaches serve as foundational pillars for reward shaping, introducing three complementary paradigms: human preference modeling for capturing subjective quality attributes (e.g., readability and idiomatic style), outcome-supervised reward modeling (ORM) for evaluating the final generated code against correctness and quality criteria, and process reward modeling (PRM) for enabling real-time, stepwise code correction during generation, as shown in [Figure 24](#). Together, these paradigms enable LLMs to produce production-ready code that meets both functional requirements and quality standards.

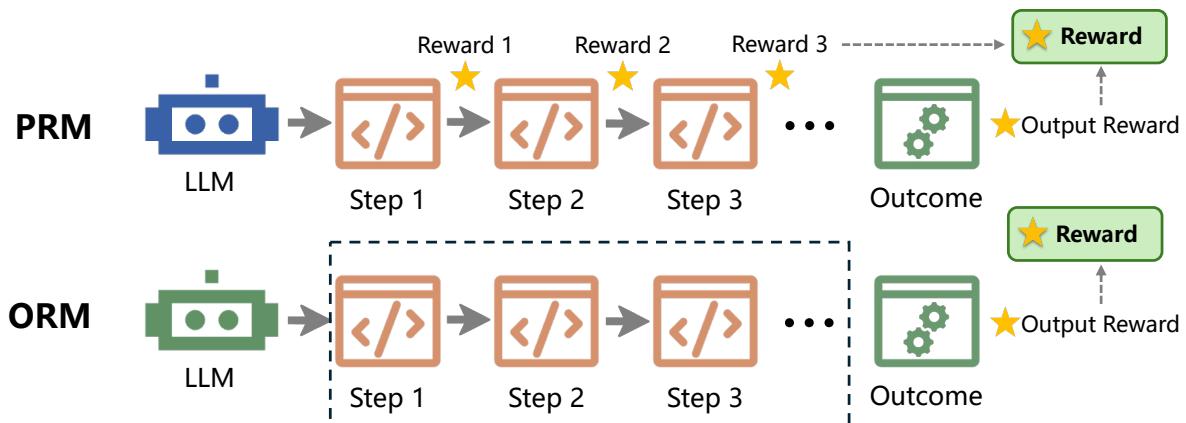


Figure 24. Comparison between ORM and PRM.

Correctness-Oriented Rewards Correctness is the most fundamental requirement for generated code. Correctness-oriented rewards ensure that code adheres to syntactic, semantic, and functional standards through static analysis and dynamic evaluation, as illustrated in [Figure 25](#).

- **Static Analysis-Based Rewards:** These are derived from static code analyzers that detect syntax errors, type mismatches, undefined variables, or style violations without executing the code. By integrating tools such as linters [482] or AST parsers [1084], rewards can be assigned

to syntactically valid and stylistically consistent code, providing immediate feedback during training.

- **Test Case-Based Rewards:** These evaluate functional correctness by executing the generated code against a set of input-output test cases [211, 608]. A positive reward is assigned only when the code passes all or a sufficient number of tests. Automated frameworks can generate unit tests from function signatures or natural language descriptions, enabling scalable and systematic reward computation for complex programming tasks.

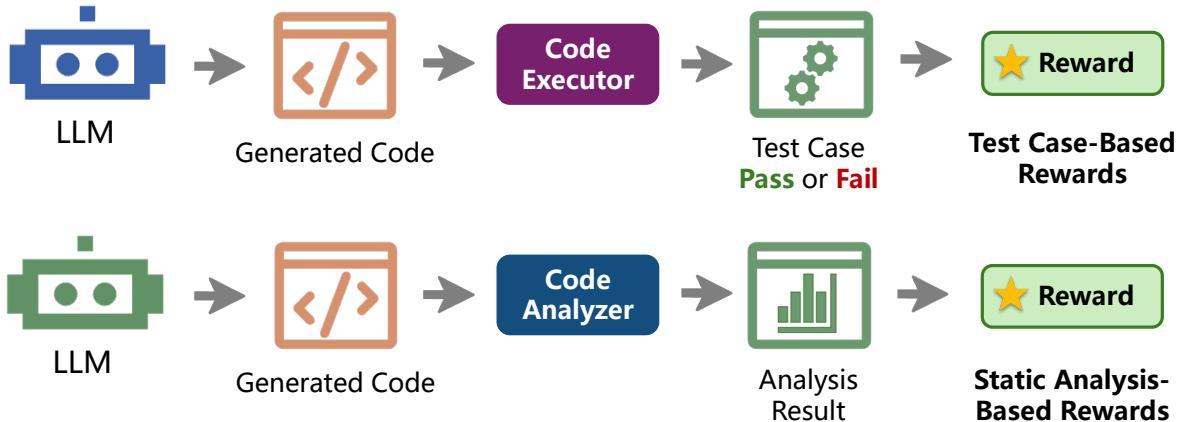


Figure 25. Comparison of typical reward designs.

4.6.4. Quality-Oriented Rewards

Beyond functional correctness, high-quality code must meet broader software engineering standards for real-world deployment. Quality-oriented rewards target attributes such as task relevance, security, and structural maintainability.

- **Task Alignment:** This dimension assesses whether the generated code fulfills the intent specified in the prompt. For example, implementing a sorting function for a query asking to “sort an array” receives a positive reward, whereas incorrect logic (e.g., summing instead of multiplying) is penalized. Semantic similarity between natural language instructions and generated code can be used to quantify alignment accuracy.
- **Security and Robustness:** These rewards encourage secure coding practices by discouraging common vulnerabilities:
 - *Vulnerability Avoidance:* Rewards are assigned for avoiding known security flaws such as SQL injection, cross-site scripting (XSS), or buffer overflows. Static analysis tools or taint tracking systems can detect such patterns and provide feedback signals.
 - *Access Control Enforcement:* Additional rewards are granted when permission checks and authentication mechanisms are properly implemented (e.g., validating user roles before file deletion).
- **Structural Quality:** These rewards promote long-term usability and maintainability by encouraging sound software design:
 - *Modularity and Reusability:* Code is rewarded if it is decomposed into well-defined functions or modules with clear responsibilities (e.g., separating input validation from business logic), enhancing readability and ease of maintenance.

- *Interface Compatibility*: Positive reinforcement is provided when generated code adheres to existing APIs, class hierarchies, or interface contracts, enabling seamless integration into legacy systems or multi-component architectures.

Rewards via Human Preference Modeling While correctness and quality-oriented rewards provide objective evaluation criteria, they often fail to capture nuanced aspects of code such as readability, clarity, and idiomatic style—qualities that are central to real-world developer preferences. Human preference modeling addresses this limitation by learning from explicit human judgments on code quality, enabling LLMs to align with subjective yet essential coding standards. This approach begins with the collection of high-quality preference data, where annotators rank or select among multiple code completions for the same prompt, typically comparing candidates in terms of efficiency, structure, or adherence to best practices. Specialized platforms like CodeRL and HumanEval-Preference facilitate systematic data gathering across diverse programming languages and task complexities. These preference pairs are then used to train a reward model via methods such as pairwise ranking loss or DPO, which learns to assign scalar rewards that reflect human judgment. To ensure robust generalization, recent models incorporate fine-grained attention mechanisms that focus on critical code segments, such as error-prone expressions or stylistic choices. Finally, the trained RM is integrated into post-training—serving as a reward signal in reinforcement learning (e.g., PPO or DPO)—to guide the LLM toward generating outputs that are not only functionally correct but also more readable and idiomatic, significantly improving alignment with human expectations.

Process Rewards Modeling Traditional reward mechanisms evaluate code only after full generation, which is inefficient for complex tasks where intermediate errors propagate irreversibly. Code Process Reward Modeling (PRM) addresses this limitation by providing granular, step-by-step feedback during the generation process, effectively turning the model into a “guided coder” that corrects itself in real time. PRM operates on the sequence of intermediate code states, assigning rewards at each token generation step based on contextual validity.

- **Design Principles**: PRM leverages two complementary signals: *local validity* (e.g., syntactic correctness of the current token) and *global trajectory coherence* (e.g., whether the partial code aligns with the problem’s high-level structure). For instance, when generating a function, PRM rewards tokens that correctly extend the function signature (e.g., `def calculate_sum(a: int, b: int) -> int:`) while penalizing tokens that introduce syntax errors or deviate from the expected API pattern.
- **Implementation Mechanisms**: PRM integrates with the LLM’s decoding process through two primary architectures:
 - *Token-Level PRM*: A lightweight neural network (e.g., a small MLP) processes the current token and preceding context to predict a per-token reward. This is computationally efficient and directly integrates into autoregressive decoding.
 - *State-Tracking PRM*: For complex tasks, PRM maintains an internal state (e.g., AST or control flow graph) to evaluate whether the partial code satisfies structural constraints (e.g., “all loops must have break conditions”). This requires symbolic reasoning modules but provides deeper contextual awareness.
- **Advantages Over End-of-Sequence Rewards**: PRM reduces the exploration space by immediately correcting invalid paths, accelerating convergence. Empirical studies show PRM can reduce the failure rate of code generation by **TODO XX%** compared to end-of-sequence

reward models, particularly for tasks requiring multi-step reasoning (e.g., implementing a state machine or handling edge cases). It also enables progressive learning, where the model gradually builds complex structures from validated sub-components.

In summary, reward shaping for code-focused LLMs in post-training combines explicit signals across multiple dimensions—*correctness*, *task alignment*, *security*, and *structural quality*—to ensure generated code is functionally sound and engineering-robust; furthermore, *human preference modeling* and *process reward modeling* enhance alignment with developer expectations and enable stepwise optimization, respectively, leading to more reliable, readable, and maintainable code generation.

5. Software Engineering Agents

Recent developments in large language models (LLMs) have contributed to the emergence of SWE Agents, including autonomous or semi-autonomous systems designed to support, automate, or enhance traditional software engineering workflows. These agents demonstrate potential for efficiency, accuracy, and scalability throughout the software development lifecycle. To systematically examine their capabilities and implications, this section organizes SWE Agents by their downstream tasks within the classical phases of the waterfall model [854] (to illustrate diverse agents in linear sequential software development), as shown in [Figure 26](#): requirements analysis and design, software development, testing, and deployment. This organizational framework not only illustrates the range of agent applications but also reveals phase-specific challenges and innovations. Furthermore, in addition to introducing existing agent frameworks, this review analyzes the training paradigms, including data curation, fine-tuning, and reinforcement learning strategies, which enable effective SWE Agents. By connecting task-oriented categorization with training methodologies, this section aims to support future research in developing more robust, adaptable, and reliable agents for software engineering.

5.1. SWE Agents Operate Across Lifecycles in Software Engineering

5.1.1. Requirements Engineering

Requirements engineering (RE) constitutes the essential discipline in software engineering that bridges stakeholder demands with technical implementations, ensuring the delivered system fulfills user expectations. The RE process generally unfolds through several interlinked stages, including **acquisition**, **examination and reconciliation**, **modeling and formalization**, and **assurance and confirmation**, where each stage addresses distinct aspects of capturing, refining, representing, and validating requirements. Beyond these discrete phases, emerging **multi-stage frameworks** integrate multiple RE activities within unified, agentic pipelines. [subsubsection 5.1.1](#) explores representative agentic approaches across these stages, culminating in integrated multi-stage systems that holistically automate RE lifecycles, as shown in [Figure 27](#).

Acquisition The acquisition phase focuses on identifying and extracting stakeholder expectations through diverse elicitation methods such as interviews, collaborative sessions, and observation. Agent-based user simulation has emerged as a powerful paradigm for this stage. **Elicitron** [1] generates diverse simulated user agents that autonomously interact with target products, logging their experiences and providing structured feedback. By interviewing and analyzing these simulated users, the system uncovers both explicit and latent requirements, significantly expanding coverage beyond conventional human-centered elicitation.

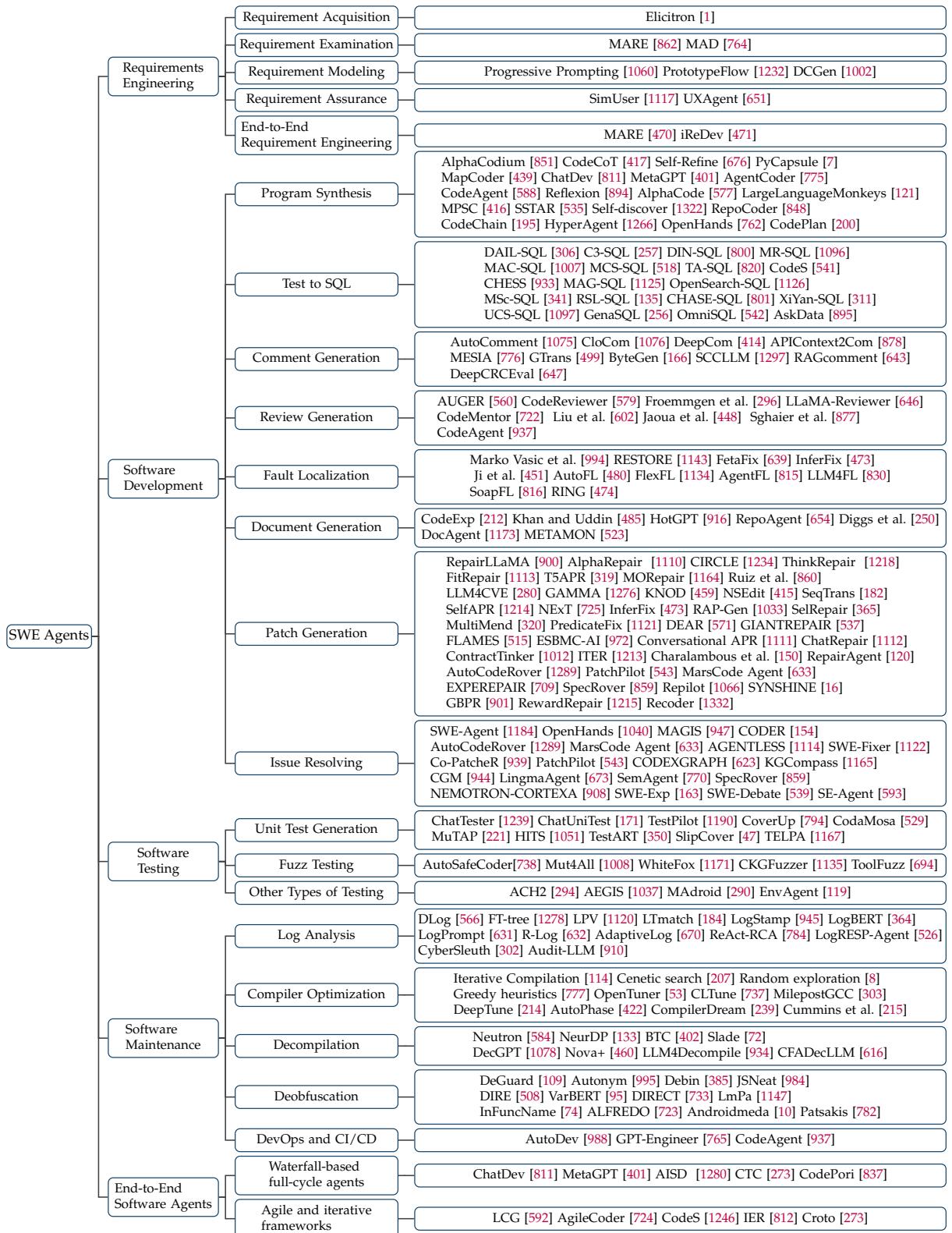


Figure 26. SWE-Agents in software engineering lifecycles.

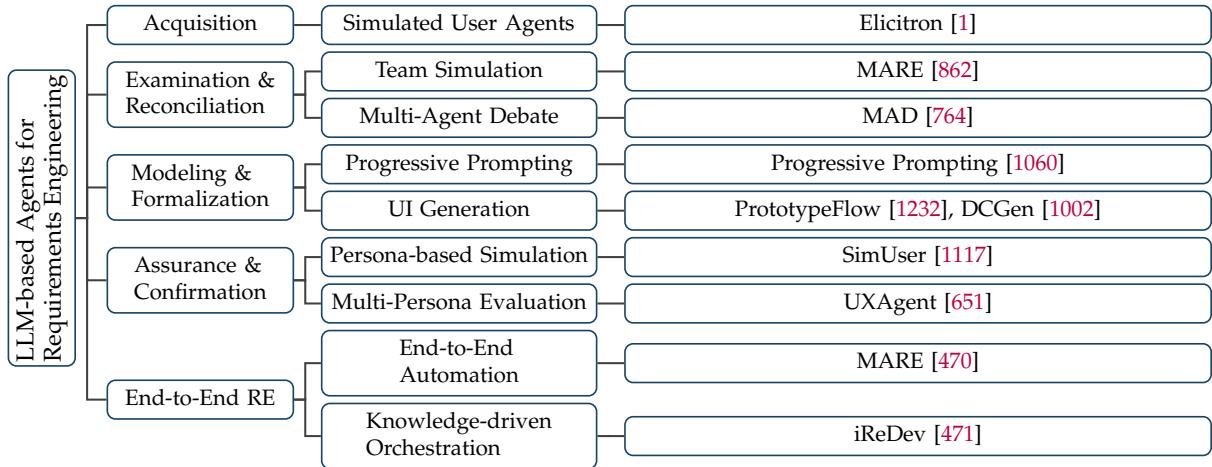


Figure 27. Taxonomy of LLM-based Agents for Requirements Engineering.

Examination and Reconciliation Following acquisition, the examination and reconciliation phase ensures that gathered requirements are coherent, feasible, and comprehensible. It often involves prioritization and conflict resolution among stakeholders. Multi-agent systems have proven effective in automating this analytical process. Sami et al. [862] simulate a virtual development team composed of agents with designated roles such as Product Owner and QA, collaboratively generating, evaluating, and ranking user stories. Meanwhile, Oriol et al. [764] introduce the **multi-agent debate (MAD)** mechanism, where opposing agents argue over requirement interpretations while a judge agent consolidates opinions into a refined and balanced specification. This adversarial collaboration enhances consistency and decision quality.

Modeling and Formalization In the modeling and formalization phase, requirements are translated into structured, machine-interpretable representations ranging from abstract UML models to concrete UI prototypes. Wei [1060] employ **progressive prompting** to iteratively map natural-language requirements to object-oriented code structures, facilitating traceability from text to implementation. For user-facing systems, **PrototypeFlow** [1232] coordinates multiple specialized design agents under a supervisory coordinator to transform textual requirements into coherent interface mockups. Similarly, **DCGen** [1002] utilizes multi-modal agent collectives that interpret UI screenshots and generate functional code, bridging the gap between visual and textual requirement representations.

Assurance and Confirmation This final stage validates that formalized requirements are both correctly specified and aligned with stakeholder intent to ensure that teams are building the system right and building the right system, where agent-based simulation plays a crucial role. **SimUser** [1117] models dual agents, where one emulates an application and another is a persona-based user to perform heuristic user experience (UX) validation. Extending this work, **UXAgent** [651] deploys thousands of diverse virtual users that autonomously navigate web interfaces and provide large-scale usability and satisfaction feedback, delivering an unprecedented level of automated assurance coverage.

End-to-End RE While the preceding sections address individual phases, the **end-to-end approach** integrates multiple RE processes into a continuous and feedback-driven pipeline. These frameworks manage an agent team, including collectors, analyzers, modelers, and validators,

within a unified environment that allows outputs from one stage to dynamically inform the next. For instance, **MARE** [470] establishes an agentic workspace where acquisition agents capture stakeholder input, modeling agents translate it into formal structures, and validation agents continuously test requirement coherence, thereby closing the loop between discovery and verification. Similarly, **iReDev** [471] introduces a knowledge-centric ecosystem of six domain-specific agents (including interviewers, analysts, and reviewers) that collaborate through shared event-based repositories. This design enables traceable transitions across stages, automated conflict detection, and consistent alignment with evolving stakeholder objectives. Collectively, multi-stage frameworks embody the agentic paradigm’s ultimate promise: end-to-end autonomy and adaptivity throughout the requirements lifecycle.

5.1.2. Software Development

5.1.2.1. Program Implementation

Program Synthesis Program synthesis agents extend beyond static prompting by incorporating multi-step reasoning, test-based verification, and feedback-driven refinement loops. They aim to construct full programs from scratch with minimal human intervention, often by simulating aspects of a human programmer’s workflow (planning, coding, testing, debugging).

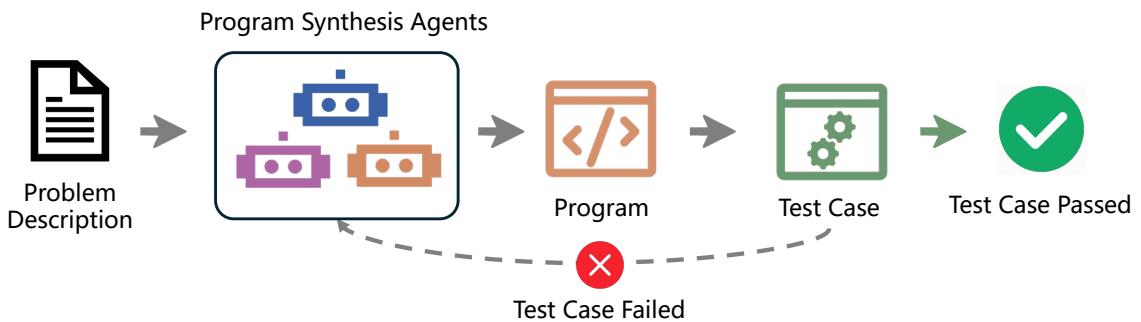


Figure 28. Overview of the program synthesis.

(1) Problem Definition Synthesize a program from a specification such that it passes a test suite. Formally, the input is a specification document (e.g. a natural language problem description, optional diagrams) and a set of test cases. The output is a program that satisfies the requirements and passes all tests, as illustrated in Figure 28. The agent must operate autonomously, without any ground-truth code, possibly through multiple internal reasoning and coding iterations.

(2) Architectures of Program Synthesis Agents Program synthesis agents can be organized in different ways depending on how they generate, test, and refine code. Recent research mainly explores two directions: single-agent systems, where one model iteratively improves its own outputs, and multi-agent systems, where several specialized agents collaborate to handle different tasks such as coding, testing, and debugging. The following discussion outlines these two main architectures.

- **Single-Agent Iterative Systems** Early approaches to agentic program synthesis predominantly rely on a single-agent paradigm, in which one large language model iteratively improves its own outputs through cycles of reasoning, self-reflection, and re-prompting. Within this setting, a unified prompt loop governs planning, coding, testing, and debugging, allowing the model to coordinate the entire workflow without external controllers. AlphaCodium [851] exemplifies this methodology by prompting the model to draft implementations, analyze potential flaws in natural language, and regenerate improved solutions while using structured YAML outputs to maintain consistency. Intermediate reasoning steps further reduce hallucination and specification drift, and automated test execution provides feedback that enhances overall performance. Similar strategies adopt the same self-improvement philosophy. CodeCoT [417] encourages the model to produce code and tests jointly to identify errors during generation, while few-shot self-refinement [676] demonstrates that iterative correction can be achieved with minimal supervision. These methods effectively address syntactic and shallow logical issues, yet they struggle with deeper semantic reasoning and domain-intensive tasks. Overall, single-agent architectures benefit from simplicity, internal continuity of reasoning, and low operational overhead, but offer limited support for richer multi-perspective analysis and collaborative problem solving.
- **Multi-Agent Pipelines** Recent work extends beyond single-agent loops by introducing dual and multi-agent pipelines that strengthen modularity, reliability, and division of labor in program synthesis. PyCapsule [7] exemplifies a dual-agent structure composed of a Programmer that generates and revises code and an Executor that runs the code and returns concrete feedback such as test failures or execution traces. This separation reduces hallucinated execution results and provides more grounded signals for iterative refinement. Later designs enhance the Executor with evaluative capabilities to better capture semantic issues that are not covered by basic tests. Building on this foundation, multi-agent systems introduce additional roles to emulate collaborative software development. MapCoder [439] coordinates a retriever, a planner, a coder, and a debugger in a structured workflow that retrieves examples, drafts plans, implements solutions, and performs testing. Similar systems such as ChatDev [811] and MetaGPT [401] extend this idea by simulating multi-role organizations that reveal both the benefits of natural language coordination and the challenges of communication overhead and error propagation. More recent approaches focus on streamlined collaboration with targeted specialization. AgentCoder [775] adopts a three-agent setup consisting of a coder, an independent test writer, and an executor that supports more rigorous self-evaluation. CodeAgent [588] further integrates auxiliary agents and external tools (documentation readers, dependency analyzers, and compilers), to enable deeper contextual reasoning and repository-scale analysis.

(3) Feedback as the Engine of Effective Code Search In agentic program synthesis, success depends not only on producing a single output but on how effectively the agent explores and refines the solution space. Feedback provides the mechanism that transforms generation into guided search. Through evaluation, critique, or self-testing, the model converts trial and error into informed iteration. Prior studies show that structured feedback significantly enlarges the effective action space of a language model, enabling correction of reasoning faults and gradual convergence toward functional correctness even with limited model capacity or compute [894]. Across systems such as AlphaCode [577] and PyCapsule [7], feedback-driven exploration has become a defining characteristic of agentic intelligence in program synthesis. Four complementary strategies illustrate this paradigm: parallel sampling, iterative self-refinement, hybrid search, and consistency-based re-ranking. Together, these approaches

provide distinct mechanisms through which agents probe, evaluate, and improve candidate programs.

- **Parallel Sampling and Selection** A common strategy for improving solution quality in code generation and program repair is to sample multiple candidate programs independently and select the most promising one, often referred to as best-of-N sampling. This approach leverages the stochasticity of large language models, as the probability of generating a correct solution increases with the number of samples. Prior work has characterized this trend through inference-time compute scaling, showing that broader sampling budgets systematically improve coverage of correct solutions [121]. In settings with reliable automatic verifiers, such as unit test driven coding benchmarks, expanded sampling frequently yields higher end-to-end success rates. However, naive sampling eventually encounters diminishing returns, particularly when verifiers are incomplete or when ranking depends on heuristic evaluators. Simple selection methods, including majority voting and reward model scoring, often plateau as sample sizes grow [121], indicating that unbounded sampling without principled selection is computationally inefficient. Consequently, practical systems combine moderate sampling budgets with more structured filtering mechanisms. Typical approaches include execution based validation, LLM based scoring, or hybrid pipelines that integrate sampling with feedback driven refinement. Early systems such as AlphaCode [577] relied on large sample pools followed by clustering and execution checks, whereas more recent models attain similar or better performance with far fewer samples by coupling sampling with stronger verifiers or iterative improvement procedures. These developments underscore the value of pairing sampling with robust and cost effective selection strategies.
- **Iterative Refinement** A central strategy in agentic program synthesis is to generate an initial solution and iteratively improve it using feedback. This feedback may arise from executing public test cases, from static diagnostics such as compiler errors or lint warnings, or from the model’s own critique. Empirical studies show that one to three refinement rounds usually yield the largest performance gains, after which improvements plateau or may even regress due to noisy or misleading feedback. PyCapsule [7], for example, reports that a small number of self-debug attempts corrects many errors, while additional iterations often provide diminishing returns. Similar observations appear in interactive repair settings. On the Commit0 benchmark, Zhao and et al. [1300] demonstrate that only a few repair rounds substantially increase the test pass rate, with later iterations offering limited benefit. Systems typically terminate once code passes available tests or after a fixed iteration budget. Complementary techniques encourage the model to articulate its reasoning before proposing fixes, as seen in CodeCoT [417] and self-refinement methods [676], which produce more targeted corrections. Some frameworks further cache previous mistakes to prevent oscillation and repeated ineffective edits [894, 1322].
- **Hybrid Search** A recent trend is to combine parallel generation with iterative refinement, capitalizing on both diversity and feedback. For example, Li and et al. [535] present a test-time scaling approach that first samples multiple candidates and then improves each one independently through a few rounds of self-debugging. After refinement, S^* employs a selection step where it pits the refined solutions against each other using additional tests. Notably, these tests can be AI-generated: S^* uses an LLM to synthesize adversarial inputs that differentiate between two candidate programs, then runs both programs to see which one produces correct or more robust outputs. This pairwise tournament of solutions yields a final winner that often inherits the strengths of many. By integrating both axes of search (breadth via sampling, depth via refinement), the hybrid strategy

achieved near-SOTA results on benchmarks like LiveCodeBench[443], while using far less compute than an equivalent large model. The advantage is especially pronounced when using smaller models – techniques such as S narrowed the gap to proprietary models by making better use of compute at inference time. A known drawback, however, is the added system complexity: hybrid search must coordinate multiple concurrent code-generation threads or agents and balance exploration depth against computational cost. Prior studies [443, 676, 811] note that excessive iterations or candidate sampling can lead to diminishing returns and higher resource overheads, highlighting the need for adaptive stopping criteria. Nonetheless, this approach embodies the general philosophy of “generate, then iterate,” which underlies many agentic systems.

- **Consistency-based Re-ranking** Consistency among multiple candidate outputs provides an implicit signal of correctness and offers an alternative to direct performance-driven selection. Inspired by self-consistency in chain-of-thought prompting, this line of work assumes that solutions independently derived in multiple coherent forms are more likely to be valid [1041]. In program synthesis, Multi-Perspective Self-Consistency (MPSC) [416] illustrates this idea by generating three complementary perspectives for each candidate: an implementation, a natural-language specification describing the intended behavior, and a set of unit tests. Consistency is assessed along two dimensions. The first dimension examines whether code execution aligns with the behavior implied by the specification. The second dimension evaluates whether execution results satisfy the expectations encoded in the tests. These relationships are modeled through a weighted graph in which edges reflect the strength of agreement, enabling candidate solutions to be ranked not only by external test results but also by internal coherence.

(4) Scaling to Repository-Level Generation As code-generation research moves from single-function synthesis toward full-repository reasoning, the central challenge shifts from producing correct code snippets to managing long-context dependencies, iterative feedback, and large-scale integration. The Commit0 framework [1300] catalyzed this transition by defining an interactive environment where agents must build complete libraries under test-driven development. While the benchmark itself remains difficult, it has spurred a new class of repository-level development agents that extend its core principles.

- **Interactive Development Loops** The prototype agent SDE-I [1300] introduces a three-stage compile–test–debug cycle, including drafting modules in topological order, running static analysis, and iteratively patching unit-test errors. This feedback-centric design inspired follow-up agents such as OpenHands [762] and CodePlan [200], which extend the loop with automatic tool calls and explicit execution reasoning. OpenHands enhances the refinement process by diversifying repair strategies instead of repeating previous fixes, while CodePlan introduces structured planning to better coordinate compilation, execution, and debugging. Together, these systems demonstrate how integrating multi-round, verbalized feedback and execution reasoning can substantially improve the robustness of automated debugging workflows.
- **Dependency-Aware and Retrieval-Augmented Generation** Commit0 [1300] highlights the difficulty of maintaining cross-file consistency, where agents must understand import graphs and API relations. Subsequent systems like RepoCoder [848] and CodeChain [195] explicitly model repository dependency graphs, generating modules in a dependency-sorted manner and retrieving relevant snippets for each component, which shortens context length by orders of magnitude while retaining semantic coherence.

- **Tool-Integrated and Memory-Enhanced Agents** To cope with the long-horizon nature of repository tasks, recent frameworks embed agents within real development environments. OpenHands [762] and HyperAgent [1266] employ multiple sub-agents for navigation, editing, and testing inside a sandboxed IDE. HyperAgent’s four-role team (planner, navigator, editor, executor) achieves state-of-the-art performance on RepoExec [516] by allocating different models to different functions (this idea derives from SDE-I’s separation of analysis and execution). These systems illustrate how scaling to repository level requires not only larger context windows but persistent memory and tool-aware interaction.

Text to SQL Within the broader landscape of software engineering agents, Text-to-SQL plays a critical role during the development phase, where it enables intelligent systems to transform natural-language requirements into structured database queries, as shown in Figure 29. By bridging human intent and data retrieval, it supports developers in automating data access, validating design assumptions, and constructing data-driven components without manual query crafting. As such, Text-to-SQL serves as both a reasoning benchmark and a practical development tool that unifies natural-language understanding, structured data querying, and autonomous decision-making. Over time, the field has evolved from early rule-based and template-guided systems to neural encoder-decoder models, and more recently to LLM-driven pipelines that incorporate schema linking, execution feedback, and multi-agent collaboration. This progression has shaped a rich research landscape that continues to expand in scope and sophistication.

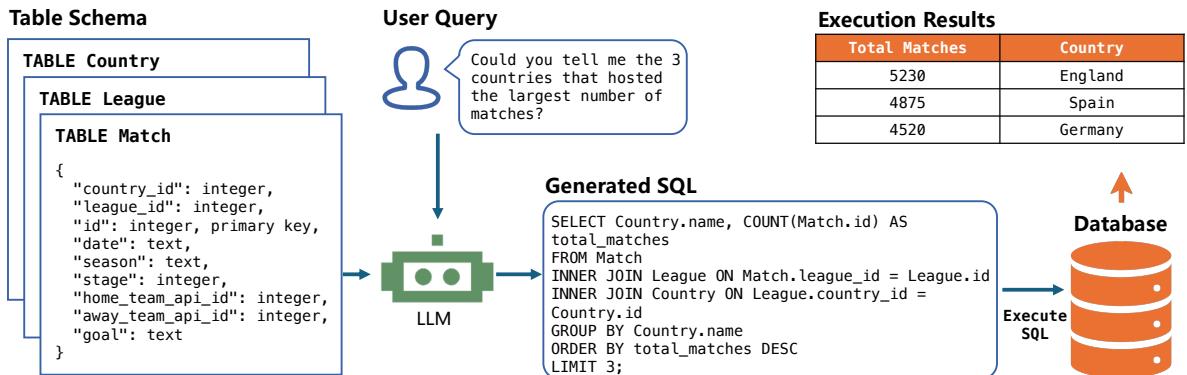


Figure 29. Illustration of text to SQL.

Pre-LLM Structure-aware Modeling Before the emergence of LLMs, systems paired BERT-style encoders with structure-aware decoders, such as abstract syntax trees [366, 1006, 1085] and predefined query sketches [391]. Encoder-decoder models also learned reusable NL→SQL patterns from supervised corpora [306, 433, 540, 553, 1315].

Prompting and In-context Learning Recently, some researchers have leveraged LLMs in text-to-SQL tasks to handle complex reasoning [1063, 1209]. A critical aspect of this approach is the design and utilization of prompts, which directly influence the accuracy of SQL generation by guiding LLMs effectively. For example, Tai et al. [932] improve the inference capabilities of LLMs using chain-of-thought prompting, including both the original chain-of-thought prompt and the least-to-most prompt. Further, Chang et al. [149] uncover the critical database knowledge and optimal representations for effective prompting through a comprehensive analysis.

DAIL-SQL [306] considers both questions and SQL queries to select few-shot examples, adopts an example organization strategy to balance quality and quantity, and utilizes code representation prompting for question representation. Additionally, C3-SQL [257] and DIN-SQL [800] have introduced innovative frameworks for database simplification, query decomposition, and prompt engineering. Overall, C3 proves that well-crafted zero-shot prompts plus execution voting are enough to reach SOTA, whereas DIN-SQL introduces a modular, difficulty-aware pipeline and a built-in self-repair loop to raise the ceiling for complex, few-shot Text-to-SQL.

Schema Grounding, Retrieval, and Agents To strengthen grounding and robustness, researchers have introduced schema linking to identify database tables and columns associated with natural language queries and have proposed complex and integrated prompting engineering methods. MR-SQL [1096] uses three independent retriever modules to pre-process the original information for accurate schema linking and to select few-shot examples with high reference significance. Li et al. [580] define a unified KGs-based schema for LLMs to generate SQL queries based on the evidence and retrieved information. MAC-SQL [1007] and MCS-SQL [518], centered on multi-agent collaboration, are designed to handle more intricate data scenarios and a broader range of error types for detection and correction. Beyond text-to-SQL generation, recent work has begun emphasizing the challenge of SQL debugging itself. BIRD-CRITIC [555] introduces a comprehensive benchmark and an associated training framework that highlight the limitations of current LLMs in diagnosing and refining complex SQL queries, while also demonstrating the potential of specialized open-source systems for strengthening this capability. TA-SQL [820] and CodeS [541] first pre-trains a code model to predict the SQL skeleton, then fine-tunes it to fill in tokens while using execution-checked data augmentation to multiply equivalent SQL variants, enabling accurate fine-tuning without any LLM prompting. CHESS [933] enables more accurate schema linking by retrieving relevant information from database catalogs and database values. Another approach, MAG-SQL [1125] features a multi-agent generative approach with soft schema linking and iterative Sub-SQL refinement. OpenSearch-SQL [1126] aligns the inputs and outputs of agents through the alignment module, reducing failures in instruction following and hallucination. MSc-SQL [341] mitigates the performance gap of smaller open-source models by sampling and comparing multiple SQL query results. RSL-SQL [135] robustly links the schema by first pruning it bidirectionally, then augments the slimmed schema with LLM-generated SQL components, asks the LLM to vote between full-schema and simplified-schema queries, and finally refines the chosen SQL through multi-turn self-correction until it executes successfully. CHASE-SQL [801] prompts multiple LLM agents to generate diverse SQL candidates via divide-and-conquer, query-plan reasoning and online synthetic examples, then trains a pairwise-selection agent to pick the best query. XiYan-SQL [311] integrates the ICL approach to maximize the generation of high-quality and diverse SQL candidates. To better retrieve the correct identifiers from the schema and transform the linguistic structure, UCS-SQL [1097] unites content and structure pipes to respectively extract key content and transform linguistic structure, thereby collaboratively enhancing SQL query generation. GenaSQL [256] introduces “Nrep” consistency, which leverages multiple representations of the same schema input to mitigate weaknesses in any single representation, making the solution more robust and allowing the use of smaller and cheaper models. OmniSQL [542] builds a scalable pipeline that bootstraps realistic relational databases from web tables, generates complexity-aware SQL, back-translates them into nine stylistically diverse natural-language questions, and synthesizes chain-of-thought solutions before fine-tuning open-source LLMs on the resulting quadruples to achieve state-of-the-art Text-to-SQL performance. AskData [895] explores the use of two standard and one newer metadata extraction techniques: profiling, query log analysis, and SQL-to-text generation using an LLM, which boosts BIRD [554] benchmark

accuracy by about 10 percentage points over the previous best.

5.1.2.2. Program Analysis

Table 18. Summary of code comment generation methods and capabilities

Work	Mapping-Based	RAG	Structured Methods	IR-based	ICL	Eval
AutoComment [1075]	✓	✗	✗	✗	✗	✗
CloCom [1076]	✓	✗	✓	✗	✗	✗
DeepCom [414]	✗	✗	✗	✗	✗	✗
APIContext2Com [878]	✗	✗	✓	✗	✗	✗
MESIA [776]	✗	✗	✗	✗	✗	✓
GTrans [499]	✗	✗	✓	✗	✗	✗
ByteGen [166]	✗	✗	✓	✓	✗	✗
SCCLLM [1297]	✗	✗	✗	✗	✓	✗
RAGcomment [643]	✗	✓	✗	✗	✓	✗
DeepCRCEval [647]	✗	✗	✗	✗	✓	✓

Mapping-Based: utilizes pre-existing code–comment alignments to guide generation.

RAG: employs retrieval-augmented generation to integrate external code or documentation knowledge.

Structured Methods: leverage program-structural signals (e.g., API context, ASTs) to enhance comment generation.

IR-based: operate on intermediate representations (e.g., bytecode, LLVM IR, Dex/Smali) for comment generation.

ICL: applies LLMs for comment generation via in-context learning (few-shot, tool-free adaptation).

Eval: designs dedicated benchmarks or metrics to assess the utility and quality of generated comments.

Comment Generation Comment generation [400, 644, 708] refers to the automated creation of natural language annotations that describe the purpose, functionality, or behavior of code segments. Its primary significance lies in improving software readability, maintainability, and collaboration efficiency, particularly in large-scale or legacy codebases where manual comment writing is labor-intensive and error-prone. By bridging the gap between source code and human understanding, effective comment generation can accelerate onboarding for new developers, assist in code reviews, and support various downstream tasks such as bug localization or API usage comprehension.

Mapping-Based Methods Early research in this area predominantly adopted mapping-based methods, which established direct correspondences between code fragments and descriptive text. These systems typically relied on databases of code–description pairs to retrieve the most relevant comment for a given code segment. AutoComment [1075] constructed such databases to retrieve suitable comments, while CloCom [1076] detected similar code segments across repositories and transferred corresponding annotations. While these methods prioritized accuracy and worked well within constrained settings, they suffered from limited semantic generalization and heavy dependence on pre-existing mappings.

Neural Seq2Seq Models The introduction of sequence-to-sequence (Seq2Seq) neural architectures marked a shift toward generative approaches, enabling models to learn comment patterns from large-scale annotated corpora. DeepCom [414] employed Seq2Seq networks for end-to-end code-to-comment generation, while Apicontext2com [878] incorporated predefined API context to improve the relevance of generated comments. MESIA [776] introduced a framework for evaluating the supplementary information contained in generated comments,

GTrans [499] enhanced Transformer architectures with graph neural networks to capture richer structural properties of code, and ByteGen [166] extended comment generation to bytecode, bypassing the need for original source code. These neural approaches improved semantic fluency and adaptability but continued to require large annotated datasets and faced challenges in domain transfer.

LLM- and Agent-Based Comment Generation The emergence of LLMs and software engineering agents has transformed comment generation from a static text-to-text task into an interactive, reasoning-driven process. LLM-based methods such as SCCLLM [1297] and RAG-Comment [643] leverage in-context learning and retrieval augmentation to produce more contextually grounded comments. Recent works integrate these capabilities into autonomous or collaborative agents that perform **self-reflection, multi-round reasoning, and tool-assisted documentation**. For instance, documentation-oriented agents in frameworks such as AutoDev [988] continuously analyze repository histories, link code changes to their rationales, and generate or revise comments as part of an end-to-end development pipeline. Multi-agent paradigms further extend this process: a reader agent summarizes code intent, a review agent verifies factual correctness, and a refiner agent improves stylistic quality and consistency.

Review Generation Automated review generation [893, 1089, 1254] refers to employing intelligent systems to automatically produce constructive, context-aware, and actionable comments for code changes. The input typically consists of diffs, code snippets, or function revisions, while the output is a coherent review comment identifying issues, suggesting improvements, or confirming correctness. Beyond accelerating human review and improving consistency, review generation agents also serve as autonomous participants in software quality assurance—capable of reasoning, critiquing, and coordinating with other agents within the development pipeline.

Task-Specific Pretraining and Fine-Tuning Early works approach code review generation through task-specific model design. AUGER [560] pre-trains a T5 model on Java code–comment pairs using a masked language modeling objective. CodeReviewer [579] further introduces diff tag prediction and denoising tasks, while the DIDACT framework [680] combines multi-task pretraining on billions of examples with specialized fine-tuning on real human comments. These models provide linguistic and semantic grounding for downstream reviewer agents.

LLM-Based Code Review Models Recent research has increasingly adopted general-purpose large language models for automated review generation. LLaMA-Reviewer [646] applies LoRA fine-tuning to adapt LLaMA for code diff analysis, while QLoRA-based models [375] and Carllm [1228] explore trade-offs between prompt-based adaptation and full fine-tuning. CodeMentor [722] employs RLHF to enhance contextual accuracy and reviewer tone. Collectively, these models demonstrate that lightweight adaptation of general-purpose LLMs can achieve competitive performance while maintaining scalability for downstream deployment within multi-agent systems.

Data-Centric and Hybrid Quality Enhancement Beyond model improvements, a complementary line of work enhances data pipelines and hybrid reasoning. Liu et al. [602] employ LLMs as classifiers to filter non-actionable review comments, improving dataset quality. Jaoua et al. [448] combine static analyzers and LLMs for hybrid data augmentation, while Sghaier

et al. [877] explore comment reformulation via LLM rewriting. These efforts emphasize the synergistic integration of structured rules and generative flexibility.

Agent-Based Review Frameworks The rise of software engineering agents has extended review generation from single-model prediction to multi-agent orchestration. In these systems, specialized reviewer agents simulate collaborative human review teams, coordinating through shared memory and role-based reasoning. Representative frameworks are summarized in [Table 19](#). For instance, **Hydra-Reviewer** [846] introduces a parallel multi-agent architecture where each agent focuses on one review dimension (e.g., logic, readability, security), combining their assessments via a dimensional classification schema. **CodeAgent** [937] models team collaboration with hierarchical roles (CEO, Coder, Reviewer, QA-Checker) to emulate human decision-making pipelines and prevent prompt drift during multi-turn discussions, achieving a 41% higher vulnerability detection rate than GPT-4. **DeputyDev** [487] scales this approach to industrial settings with expert-level agents specializing in security, maintainability, and performance, coordinated by a hybrid engine that merges and verifies outputs in near real-time. **iCodeReviewer** [791] dynamically routes requests among prompt experts specialized for specific vulnerability categories, guided by code-aware routing algorithms that enhance issue identification.

Table 19. Representative Agentic Review Generation Systems

System	Core Architecture	Agent Roles	Key Innovation
Hydra-Reviewer [846]	Parallel multi-dimensional analysis	Reviewers per dimension (logic, readability, etc.)	Multi-dimensional classification taxonomy
CodeAgent [937]	Simulated human collaboration	CEO, Coder, Reviewer, QA-Checker	Prompt drift prevention via QA consistency
DeputyDev [487]	Industrial-scale expert orchestration	Security, maintainability, performance agents	Hybrid engine for output merging
iCode-Reviewer [791]	Security-focused dynamic expert mix	Prompt specialists per vulnerability type	Code-based dynamic routing algorithm

Fault Localization Fault localization [481, 705, 1092, 1160] is the process of automatically identifying the specific locations in source code that are responsible for causing a program failure, typically by analyzing program execution data such as test case outcomes. As illustrated in [Figure 30](#), research on fault localization and automated repair has evolved into a rich and diverse area, spanning neural architectures, optimization-driven learning, external tool integration, and multi-agent collaboration. The overarching objective of these approaches is to enhance the precision, scalability, and adaptability of automated debugging, addressing varied scenarios such as variable misuse detection, large-scale code reasoning, and domain-specific model conversions.

End-to-End Neural Methods This line of research focuses on unified neural architectures that jointly address fault detection, localization, and repair within a single learning pipeline. Early work explored pointer network based models capable of identifying faulty statements

while simultaneously generating candidate repairs, illustrating the benefits of joint learning compared with enumerative approaches. Subsequent systems such as Restore [1143] incorporated feedback driven refinement, where failed validation attempts inform dynamic analysis and guide the search toward more effective patches. Domain-oriented frameworks like FetaFix [639] further extended neural repair to specialized settings, including the automated conversion of deep learning models, by identifying faults across representation frameworks such as ONNX and TVM and applying iterative correction. Taken together, these efforts highlight the strengths of integrated neural reasoning for aligning localization and repair. By coupling end-to-end learning with structured feedback, these methods capture complex bug semantics while maintaining efficiency across diverse program domains.

Optimization Methods Another research trajectory explores fine-tuning large language models with targeted and domain-specific datasets to enhance debugging performance. Approaches such as DeepDebug [261, 1319] synthesize artificial bugs from real-world commit histories to augment training data, thereby improving robustness in joint localization–repair tasks. InferFix [473] integrates static analysis and retrieval-augmented learning, combining a contrastively trained retriever with a generative LLM to unify detection, classification, and correction under a single framework. Other works [451] further demonstrates that line-aware fine-tuning can yield interpretable fault localization by training models to directly generate line-level predictions, circumventing traditional reliance on execution coverage. Overall, optimization-based methods highlight the importance of adaptive fine-tuning strategies that encode bug semantics and structural patterns into model representations, improving generalization across languages and repositories.

External Tool Calling To overcome inherent LLM constraints such as limited context windows and incomplete code understanding, recent studies integrate external tools into the debugging pipeline. AutoFL [480] exemplifies this direction by enabling LLMs to interact with repositories via tool calls, thereby supporting explainable fault localization grounded in code navigation and repository traversal. Building upon this idea, FlexFL [1134] introduces a two-agent framework that combines static fault localization signals with LLM-based refinement: one agent prioritizes suspicious code regions, while another performs contextual inspection and re-ranking through interactive reasoning. These methods bridge the gap between reasoning and execution, allowing models to iteratively refine localization hypotheses through tool-assisted verification and repository-scale exploration.

Multi-Agent Collaboration An emerging direction in fault localization employs multiple specialized agents that cooperate to decompose and coordinate debugging tasks. RING [474] exemplifies this trend by enabling language models to jointly conduct localization, ranking, and patch generation through inter-agent communication. LLM4FL [830] further develops a tri-agent architecture composed of context extraction, debugging, and reviewing agents, each responsible for subtasks such as reducing coverage data or reasoning over failure graphs. Reinforcement driven dialogue facilitates information exchange and consensus formation among agents. Compared with single-agent approaches, these multi-agent paradigms introduce complementary reasoning perspectives and structured role separation, resulting in improvements in both interpretability and robustness. Collectively, they mark a shift toward collaborative fault localization, where distributed expertise and coordinated decision making enhance the autonomy and adaptability of the debugging pipeline.

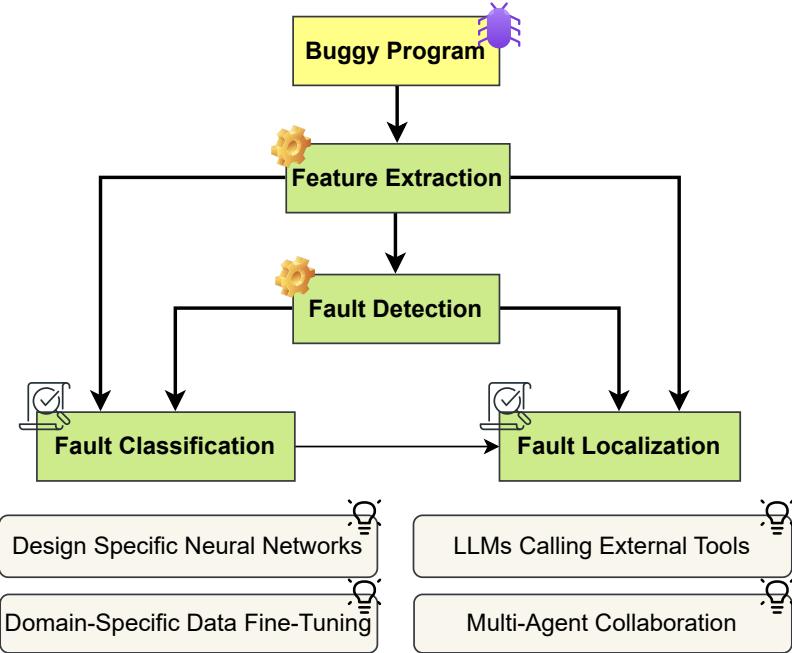


Figure 30. Taxonomy of End-to-end Fault Localization.

Document Generation In software engineering, document generation [498] refers to the automated production of comprehensive, structured, and often long-form textual documentation for software projects, APIs, or codebases. This task is essential for ensuring effective knowledge transfer, facilitating software maintenance, and enhancing usability for both internal and external stakeholders. High-quality documentation reduces the cognitive load on developers, accelerates onboarding, and supports compliance and quality assurance processes, making its automation a valuable goal in modern software development.

Early Model-Based Approaches Early work in this domain grappled with the inherent difficulty of producing long-form, contextually coherent explanations. CodeExp [212] formalized the code explanation generation task and proposed a multi-stage fine-tuning strategy based on a retrieve-then-generate mechanism. Their methodology first trained the model to retrieve relevant contextual snippets from the codebase, and subsequently used these snippets to condition the final explanation generation process. This approach aimed to improve the relevance and completeness of the output. Despite this advancement, these initial efforts were limited by model capabilities and often fell short in handling complex, large-scale codebases.

LLM-Driven Approaches The emergence of powerful LLMs such as GPT and Codex rapidly advanced the field. Khan and Uddin [485] demonstrate Codex’s potential in generating comprehensive documentation, Hotgpt [916] investigate a general-purpose, one-model-fits-all solution for documentation tasks, and Shekhar Dvivedi et al. [883] provide a comparative analysis of different LLM-based documentation generation approaches. These works confirmed that LLMs excel in producing linguistically fluent, cross-domain documentation, though they also revealed weaknesses in technical accuracy and the ability to keep pace with evolving codebases.

Agent-Based Approaches Recent work has increasingly explored agent-based architectures that integrate large language models into proactive and continuously evolving documentation pipelines. RepoAgent [654] presents an open-source framework that automatically generates, maintains, and updates documentation as code evolves. Diggs et al. [250] focus on bringing similar automation to legacy systems, while DocAgent [1173] introduces a collaborative multi-agent framework composed of specialized roles including Reader, Searcher, Writer, Verifier, and Orchestrator. METAMON [523] augments this paradigm by combining LLMs with metamorphic testing to identify inconsistencies between documentation and actual program behavior. Collectively, these agent-based systems offer promising directions for addressing long-standing documentation challenges. Automating the maintenance cycle, as in RepoAgent, helps keep documentation accurate and sustainable over time. Role specialization and collaborative workflows, as demonstrated in DocAgent, improve scalability and enable coverage of large and complex codebases. The incorporation of active verification, exemplified by METAMON, further ensures that generated documentation aligns with real code semantics. Together, these techniques suggest that coordinated agentic workflows can substantially enhance both the quality and reliability of software documentation.

5.1.1.3. Program Editing

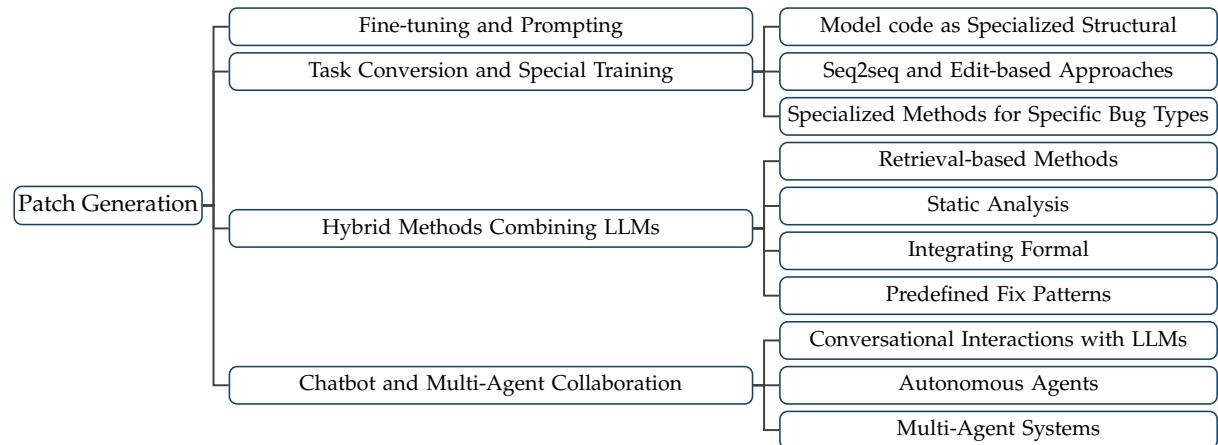


Figure 31. Taxonomy of End-to-end Patch Generation.

Patch Generation Patch generation [105, 591, 832, 1033] refers to the automated creation of code fragments that repair program vulnerabilities or logical errors. Its central objective is to automate the workflow traditionally executed by developers, spanning fault localization through patch synthesis, thereby improving software reliability and development efficiency. Recent years have witnessed substantial progress in this field, with a shift from template-based and example-driven repairs toward large language model-powered and agent-oriented frameworks. Current approaches can be grouped into several paradigms, including fine-tuning and prompting techniques, task reformulation and training strategies, static analysis and rule-based repair, retrieval augmented generation, and multi-agent or conversational architectures. Each direction introduces distinct mechanisms for reasoning, adaptation, and verification, collectively advancing automated patch generation toward more autonomous, interpretable, and scalable systems.

Fine-tuning and Prompting This line of work focuses on enhancing pre-trained language models for code through targeted fine-tuning or prompting to produce correct patches. Techniques such as RepairLLaMA [900], AlphaRepair [1110], and T5APR [319] leverage parameter-efficient fine-tuning, masking strategies, and multitask learning to adapt general-purpose models for automated program repair. Others, like CIRCLE [1234] and FitRepair [1113], adopt continual learning and retrieval-based prompting to support multilingual and cross-domain repair. Emerging models, such as MORRepair [1164] and ThinkRepair [1218], incorporate reasoning guidance, chain-of-thought prompting, and execution feedback, integrating interpretability into the repair process. Complementary work, including NExT [725] and LLM4CVE [280], further augments LLMs with execution traces and security-specific training, enabling them to reason over runtime behaviors and vulnerabilities. Collectively, these approaches highlight the growing capability of fine-tuned or prompted LLMs to internalize domain knowledge, generalize across languages, and perform repair through structured reasoning rather than surface pattern matching.

Task Conversion and Specialized Training Several studies improve repair precision by reframing the patch generation task into other well-studied modeling problems, such as sequence transformation, grammar prediction, or optimization. Recoder [1332] and NSEdit [415] reformulate repair as structured code editing, employing grammar-constrained decoding and pointer-based edit modeling to preserve syntax. RewardRepair [1215] integrates execution feedback into training through reinforcement-style objectives, guiding models toward semantically correct fixes. SeqTrans [182] and KNOD [459] incorporate data-flow information and domain-specific rule distillation to enhance generalization and interpretability. Gradient-based program repair (GBPR) [901] introduces a differentiable optimization framework that maps program behavior to a continuous search space, allowing gradient descent to identify repairs. Collectively, these task conversion approaches push the boundary of APR from discrete token prediction toward learning structured, semantically grounded transformations.

Static Analysis and Pattern-Guided Repair Another strand of research integrates symbolic reasoning, compiler diagnostics, or predefined fix patterns with LLM-based generation to guarantee correctness and robustness. Dear [571] and Synshine [16] combine deep learning with static analysis and compiler feedback to identify multi-location or syntactic faults, enhancing precision through diagnostic-guided patching. Gamma [1276] and Hybrid [537] leverage predefined templates or structural patch skeletons to constrain the search space, improving both validity and efficiency. FLAMES [515] bridges semantic search and LLM-driven repair, employing guided refinement based on test feedback. Similarly, hybrid systems such as ESBMC-AI [972] and ContractTinker [1012] integrate model checking and dependency analysis with iterative LLM reasoning to achieve verifiable and semantically consistent repairs. Overall, these methods demonstrate how formal reasoning and rule-driven heuristics can complement generative LLMs, yielding interpretable and verifiable patching pipelines.

Retrieval-Augmented Approaches Retrieval-based methods enrich buggy code contexts with relevant bug-fix examples to inform generation. InferFix [473] exemplifies this trend by coupling static analysis with retrieval-augmented prompting, while RAP-Gen [1033] employs hybrid lexical and semantic retrievers to guide patch synthesis. SelRepair [365] and MultiMend [320] propose dual retrieval and RAG selection mechanisms to enhance efficiency and multi-hunk repair capability. PredicateFix [1121] leverages static analysis predicates to retrieve targeted examples, reinforcing semantic alignment during patch validation. Collectively,

retrieval-augmented methods strengthen contextual grounding, enabling models to reason from prior experience while maintaining generality across projects and programming languages.

Conversational and Multi-Agent Frameworks Recent advances in patch generation increasingly adopt conversational paradigms and multi-agent collaboration to support interactive and adaptive repair. Frameworks such as Conversational APR [1111] and ChatRepair [1112] employ iterative dialogue between patch generation and validation, enabling large language models to refine candidate patches through natural language reasoning and feedback from test execution. Systems including ITER [1213] and RepairAgent [120] further integrate fault localization, patch synthesis, and validation into continuous feedback loops, coordinating multiple agents or tool calls to support dynamic refinement. At a broader scale, AutoCodeRover [1289], PatchPilot [543], and MarsCode Agent [633] extend these ideas to repository-level repair, combining multi-agent planning, project-wide code search, and execution-based validation to emulate collaborative development workflows. More recent frameworks such as ExpeRepair [709] and SpecRover [859] incorporate memory components, specification inference, and reviewer agents, supporting persistent learning and adaptive validation over extended interactions. Complementary approaches like Repilot [1066] explore coordination between language models and completion engines to enhance repair stability and patch consistency. Collectively, these conversational and multi-agent systems represent a shift from single-pass patch generation to interactive, collaborative, and self-improving repair ecosystems. They illustrate how coordinated agentic behavior, iterative reasoning, and structured feedback can substantially enhance autonomy and reliability in automated software maintenance.

Issue Resolving In recent years, the widespread adoption of large language models in software engineering has positioned agent-based automated code repair as a key direction for advancing intelligent software maintenance. Automated issue repair is inherently a complex decision-making process that spans the full workflow from understanding the problem to generating a patch [Figure 5.1.2](#), and typically involves several interconnected stages such as fault localization [Table 5.1.2](#), program editing [Figure 5.1.2](#), and validation [subsubsection 5.1.3](#). From a technical perspective, the field has evolved across multiple layers, beginning with foundational interface designs and progressing toward more sophisticated decision-making mechanisms. These developments have led to substantial advances across the entire pipeline and have gradually shaped a hierarchical technical architecture for agent-based repair, as illustrated in [Figure 32](#).

Foundational interface layer Effective automated issue repair first requires solving the interaction problem between LLMs and programming environments, since traditional text-only interfaces are insufficient for complex programming tasks. Foundational works such as SWE-Agent[1184] and OpenHands [1040] were the first to introduce the language-model-as-agent concept, treating LLMs as a new kind of end user and designing dedicated agent-computer interfaces (ACIs) around their capabilities and limitations, as shown in [Figure 33](#). ACIs simplify command structures, optimize environment feedback and history management, and integrate efficient editing and protective mechanisms (e.g., multi-line replacement, automatic rollback, dynamic history folding), substantially improving the operational efficiency and reliability of LLM agents in code creation, editing, and testing. This work provided critical technical foundations and design principles for subsequent agent-based automated repair systems, enabling LLMs to operate stably in complex programming environments.

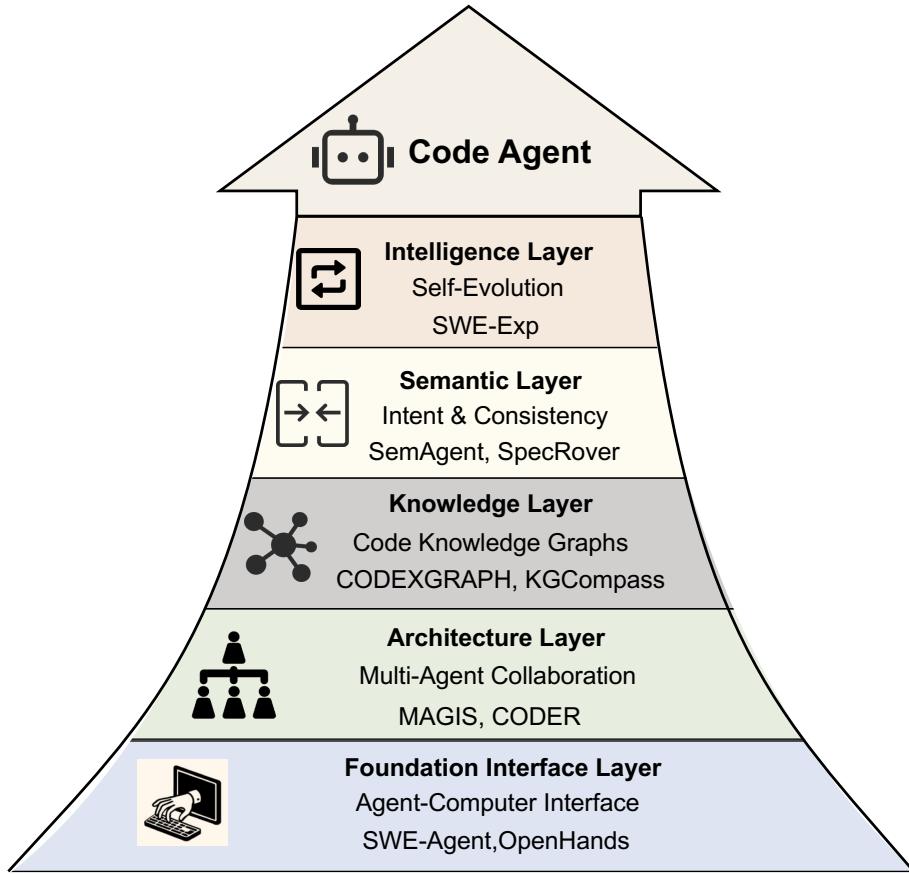


Figure 32. A hierarchical technical architecture for coding agents, illustrating the progression from the Foundation Interface Layer, through the architecture, knowledge, and semantic layers, to the culminating Intelligence Layer.

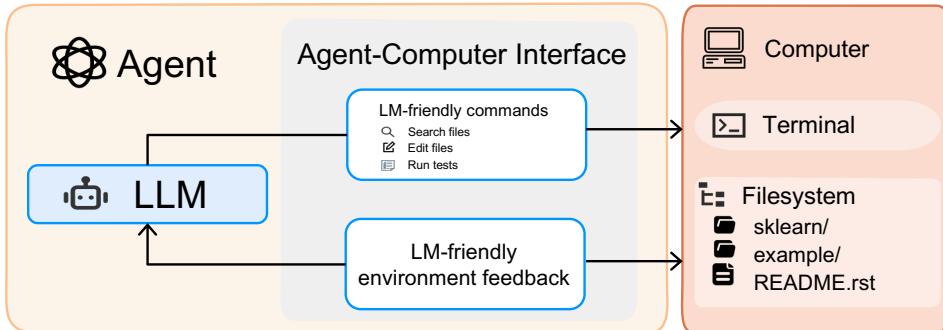


Figure 33. The paradigm of agent-environment interaction mechanisms for addressing issue resolution problems.

Architectural Layer After basic interaction capabilities are established, the primary challenge becomes handling complex software issues while maintaining system efficiency and practical usability. Real-world faults often involve multiple files and layered dependencies, which are difficult for a single agent to manage. Modular collaborative architectures therefore decompose the repair workflow into specialized subtasks such as code understanding, dependency analysis, fault localization, patch generation, and validation, leading to notable improvements in scalability and performance. Magis [947] illustrates this idea by assigning

responsibilities to four roles: manager, repository steward, developer, and quality assurance engineer, and by decomposing problems into role specific subtasks. With BM25 retrieval and memory mechanisms, Magis improves collaboration efficiency in complex multi file modification scenarios. Coder [154] employs a predefined task graph to coordinate roles including manager, reproducer, fault localizer, editor, and verifier, enabling efficient repository level repair. Systems such as AutoCodeRover [1289] and MarsCode Agent [633] adopt staged workflows for context retrieval and patch generation and use multi agent collaboration supported by code knowledge graphs, Language Server Protocol services⁷, and AutoDiff tools⁸ to achieve systematic repair. At the same time, simplified workflow designs reduce system complexity, improve resource utilization, and enhance execution stability, making automated repair more practical. A representative workflow example is shown in Figure 34. Agentless [1114] proposes an agentless automation approach that uses a simple three-stage flow (localization, repair, patch validation) and simplifies decision-making and tooling via hierarchical localization, context windows, and majority-vote mechanisms, significantly lowering system complexity while retaining efficiency. SWE-Fixer [1122] focuses on efficiently addressing software engineering issues on GitHub through coarse-to-fine retrieval strategies and structured outputs, improving maintainability. Co-PatcheR [939] adopts a collaborative small-model system that assigns localization, generation, and verification tasks to specialized small reasoning models, greatly reducing computational cost. PatchPilot [543] emphasizes stability, cost-effectiveness, and formal guarantees in the automated repair pipeline; its rule-based planning workflow uses self-reflection, hierarchical localization, and formal verification to ensure patch correctness and safety. This combination of modular collaboration and simplified workflows aims to improve execution efficiency through role specialization while preserving decision consistency, practicality, and deployability via process optimization.

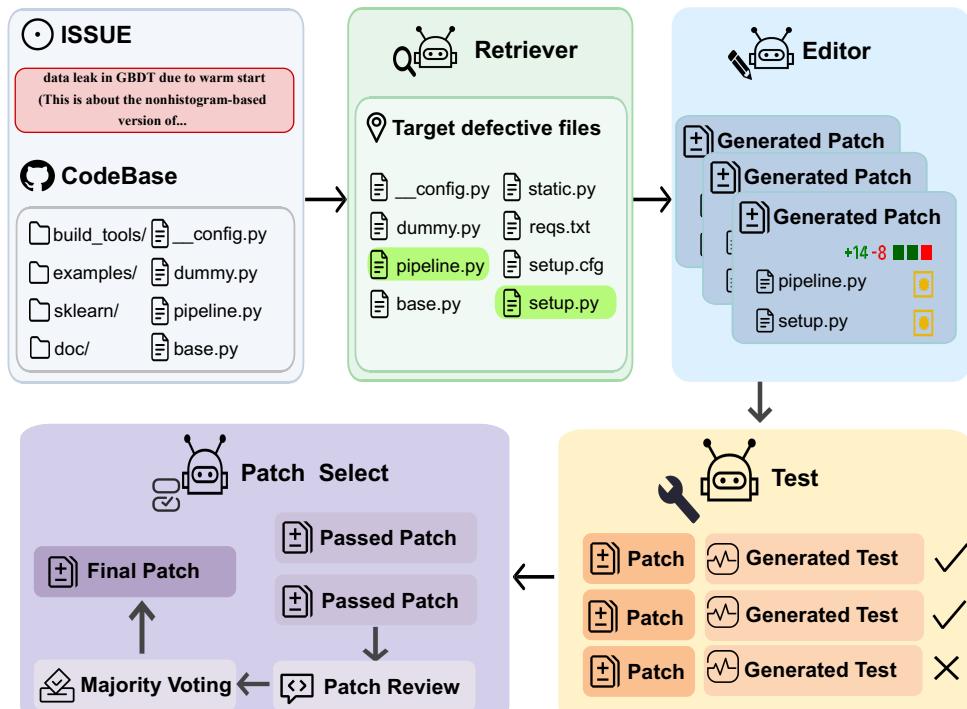


Figure 34. A typical workflow for addressing issue resolving problems.

⁷<https://microsoft.github.io/language-server-protocol/>

⁸<https://www.autodiff.org/>

Knowledge Layer As system architectures advance, deeper understanding of code semantics and dependency relationships becomes essential for improving repair quality. Structured knowledge modeling, particularly representing a codebase as a graph, has enabled significant progress in semantic comprehension. CodexGraph [623] integrates large language models with repository structure through a graph database interface, supporting complex graph queries and multi task operations and improving the system’s ability to capture intricate dependencies among code entities. KGCompass [1165] constructs repository level knowledge graphs and guides repair processes by precisely linking issue descriptions to relevant code components, which narrows the search space and enhances both localization accuracy and the interpretability of generated patches. CGM [944] incorporates semantic and structural properties of the codebase into model attention mechanisms and employs an agentless graph retrieval augmented generation framework to enable efficient subgraph retrieval and patch synthesis. LingmaAgent [673] combines knowledge graphs with Monte Carlo tree search to achieve global repository understanding and dynamic patch generation. Together, these methods convert abstract semantic relationships within the code into computable graph structures, enabling structured reasoning that substantially improves problem understanding and solution generation.

Semantic layer On top of knowledge representation, accurately understanding developer intent and producing high-quality patches is a frontier in technical development. High-quality patch generation requires not only syntactic correctness but also semantic consistency and alignment with developer intentions. Methods based on deep semantic analysis fuse multi-dimensional semantic signals to significantly improve patch accuracy and applicability. SemAgent [770] focuses on semantic-aware repair by performing semantic-driven fault localization and a two-stage repair architecture that combines problem semantics, code semantics, and execution semantics to produce complete and consistent patches. SpecRover [859] centers on specification inference: integrating program structure, behavior, and tests to infer developer intent and generate high-quality patches, emphasizing multi-stage workflows and evidence generation mechanisms to provide interpretable guarantees of patch correctness. Nemotron-CORTEXA [908] develops specialized code-embedding models and localization agents, leveraging ASTs and LSP-derived information to construct code knowledge graphs, enabling multi-step reasoning and high-precision localization; it increases the diversity and accuracy of generated patches through varied context and ensemble learning. By deeply understanding the root causes and developer intent, these approaches avoid common overfitting seen in traditional methods and produce more generalizable and robust repair solutions.

Intelligent Layer Once foundational repair capabilities are in place, a key objective becomes enabling systems to learn continuously and improve autonomously. Because software faults are diverse and highly dynamic, static repair strategies often struggle to generalize across evolving scenarios. Experience driven self evolution mechanisms address this limitation by learning from historical repair trajectories, refining decision strategies over time, and substantially enhancing system adaptability and robustness. SWE Exp [163] exemplifies this direction through multi dimensional experience repositories that capture reusable knowledge from both successful and failed repairs. It employs a dual agent architecture that separates strategic planning from tactical execution and augments decision making with Monte Carlo tree search to support continual and cross problem learning. SWE Debate [539] introduces a competitive multi agent debate framework that constructs fault propagation chains from code dependency graphs and organizes agents in staged debates to encourage diverse reasoning paths and more precise fault localization, addressing the limitations of single perspective analysis. SE Agent [593] applies

self evolution principles by systematically expanding the exploration space and leveraging cross trajectory insights through the operations of revise, reorganize, and refine to iteratively optimize reasoning processes and enable dynamic decision making. Together, these approaches transition software repair from a static and isolated workflow to a systematic, knowledge driven learning paradigm in which agents adapt, accumulate experience, and improve over time.

5.1.3. Software Testing

Software testing is a critical phase in the software development lifecycle that ensures the quality, reliability, and security of applications. As the complexity of software systems grows, traditional testing methods can be time-consuming and resource-intensive. The advent of LLMs has revolutionized this field, enabling the creation of autonomous agents that can automate and enhance various testing processes, from generating unit tests to performing security analyses and conducting fuzz testing. These AI-powered agents are transforming software testing from a manual, human-centric process into a more efficient and scalable system.

LLM-Driven Test Generation Frameworks In the field of software engineering, researchers have explored the use of LLM-based agents for automated test generation and dataset construction. Otter [17] and SPICE [106] respectively focus on two key aspects: automated test generation and large-scale dataset annotation. Specifically, Otter employs a locator–planner–generator architecture to automatically identify test targets and generate high-quality test cases, while SPICE constructs a scalable LLM-assisted annotation framework that integrates program analysis and human verification to build large, high-quality test datasets. Automated test generation has further promoted the development of the test-driven development (TDD) field. SWE-Flow [1272], centered on TDD, achieves high-quality data synthesis by automatically inferring incremental development steps, constructing runtime dependency graphs (RDG), and generating structured development plans. This framework supports controllable data generation and automatic documentation creation, greatly enhancing dataset structure and usability. Experiments demonstrate that SWE-Flow-based data synthesis significantly improves LLM performance in realistic software development tasks. Building upon these representative systems, subsequent studies have shifted attention from one-shot code generation to systematic test engineering — focusing not only on producing test cases, but also on improving their **quality, coverage, and executability**. This transition is most evident in the area of unit test generation.

Unit Test Generation Unit testing validates the smallest executable parts of software such as functions or classes. Recent research on LLM-based unit test generation [1239] has shifted from exploring **model capability** to building **systematic engineering** frameworks that ensure both correctness and meaningfulness of generated tests, as shown in Figure 35.

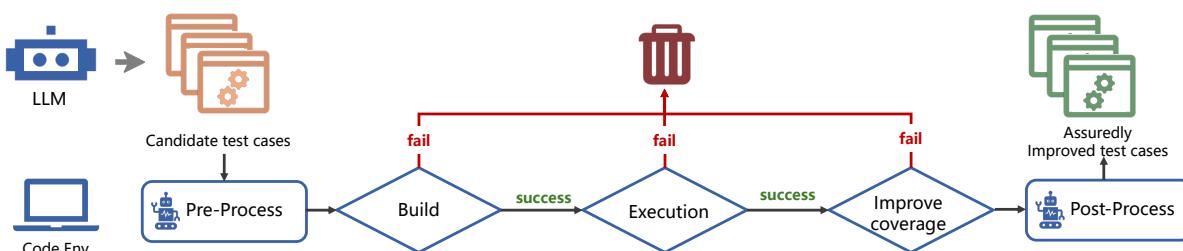


Figure 35. Overview of unit test generation.

From Generation Capability to Generation Quality Early studies such as AthenaTest and A3Test focused primarily on whether large language models could generate syntactically correct test code [31, 987]. With the emergence of more capable models, including ChatGPT and Codex [151, 755], research has shifted from basic code generation toward ensuring correctness, coverage, and semantic relevance. To improve semantic fidelity, ChatTester [1239] enhances assertion quality through intent inference and iterative refinement. TestPilot [1190] increases reliability by incorporating prompt refiner modules that leverage documentation and error guided regeneration. Coverage oriented methods such as CoverUp [794] and CodaMosa [529] integrate coverage feedback into the generation loop to improve line and branch coverage. MuTAP [221] further introduces mutation based feedback to strengthen test effectiveness and fault detection. Collectively, these works illustrate a transition from static generation to adaptive and feedback driven refinement, highlighting that improvements in model capability must be matched by advances in test quality.

From Single-Step Generation to Multi-Step Iterative Processes As research on unit test generation has progressed, it has become evident that a single generation step is insufficient for producing reliable tests. To address hallucinations and make more effective use of feedback, recent work has shifted from one shot generation to multi step iterative pipelines. ChatUniTest [171] enriches contextual prompting by incorporating detailed class and dependency information to support iterative generation and refinement. TestART [350] adopts template based repair strategies to iteratively correct errors in generated tests. HITS [1051] introduces a structured multi step reasoning process that includes summarization, dependency extraction, slicing, and guided generation, coupled with self debugging feedback to improve coverage and robustness. Subsequent frameworks such as SlipCover [47], CoverUp [794], TestPilot [871], and TELPA [1167] further strengthen this paradigm by integrating coverage feedback, orchestration mechanisms, and execution driven refinement. Overall, recent systems no longer invoke large language models in a single step. Instead, they increasingly adopt semi autonomous pipelines that generate, evaluate, and repair tests iteratively, progressively improving coverage, correctness, and reliability.

Fuzz Testing While unit testing focuses on functional correctness, fuzz testing targets robustness and security by generating diverse, often malformed inputs. This direction extends the use of LLM agents from functional validation to vulnerability detection. Nunez et al. [738] introduces a multi-agent system integrating coding, static analysis, and fuzzing agents in continuous feedback loops for autonomous vulnerability detection and repair. To overcome manual mutator design limitations, Wang et al. [1008] presents a language-agnostic framework that synthesizes mutation operators using historical bug data. Similarly, Yang et al. [1171] leverage LLMs for scalable white-box fuzzing of deep learning compilers, while Xu et al. [1135] utilize code knowledge graphs for automatic fuzz driver generation. Further, Milev et al. [694] propose ToolFuzz, combining LLM-based query generation with fuzzing to test agentic tool reliability. Collectively, these studies demonstrate a gradual evolution from single-purpose fuzzing scripts to fully automated, cross-language LLM-driven fuzz testing ecosystems, capable of intelligent adaptation and learning.

Other Testing Agents Beyond unit and fuzz testing, large language model based agents have been extended to broader testing paradigms, including mutation testing, bug reproduction, interactive feature testing, and autonomous test execution, forming an increasingly comprehensive ecosystem for automated software testing.

Mutation Testing Recent work applies LLMs to mutation testing by replacing handcrafted mutation operators with model generated mutants, as demonstrated by LLMorpheus [973]. PRIMG [117] introduces mutation informed prioritization to guide efficient test evaluation, while ACH2 [294] utilizes mutation guided super bugs to stimulate high value test generation. These approaches strengthen the connection between test generation and fault detection and reinforce the broader trend toward feedback driven testing.

Automated Bug Reproduction Agent based testing has also been extended to automated bug reproduction. AEGIS [1037] combines structured context extraction with finite state optimization to reproduce software failures, and subsequent work [180] adapts this framework to industrial settings. These methods demonstrate that agentic coordination can enhance not only test creation but also debugging and verification workflows.

Multi-Agent Interactive Feature Testing Multi agent collaboration has further been applied to interactive feature testing. MAdroid [290] introduces an architecture composed of coordinator, operator, and observer agents that simulate realistic user interactions within graphical interfaces. This design enables automated evaluation of cross user workflows and expands the scope of agent based software testing.

Autonomous Test Execution Despite progress in generating and repairing tests, executing them across heterogeneous environments remains a major challenge. ExecutionAgent [119] addresses this by autonomously constructing execution environments and running test suites for complex software projects. Through the use of meta prompting, project artifacts such as configuration scripts, and iterative feedback for command execution and error recovery, the system demonstrates strong adaptability to diverse software stacks and highlights the feasibility of LLM driven automated test execution.

5.1.4. Software Maintenance

After software passes the testing stage, it is deployed into production, and often continuously maintained or updated [1293], as illustrated in Figure 36. Therefore, some research has concentrated on addressing tasks within deployment and operations.

Log Analysis Log analysis is a core task in software operation and maintenance. It focuses on transforming raw, unstructured logs into actionable insights and identifying anomalies or faults in system behavior. Research in this area has evolved from traditional deterministic and statistical approaches to neural methods, and more recently to agent-based systems that emphasize autonomous reasoning and adaptive decision making. This progression can be organized into two main categories: traditional log parsing foundations and agent-based log analysis.

Traditional Foundations for Log Parsing and Analysis Before the rise of neural and agent-based systems, research on log analysis was primarily grounded in deterministic and statistical techniques. These early foundations focused on identifying structural regularities within logs and developing parsers capable of handling large, heterogeneous datasets. Representative approaches can be grouped into two major lines of work, summarized as follows.

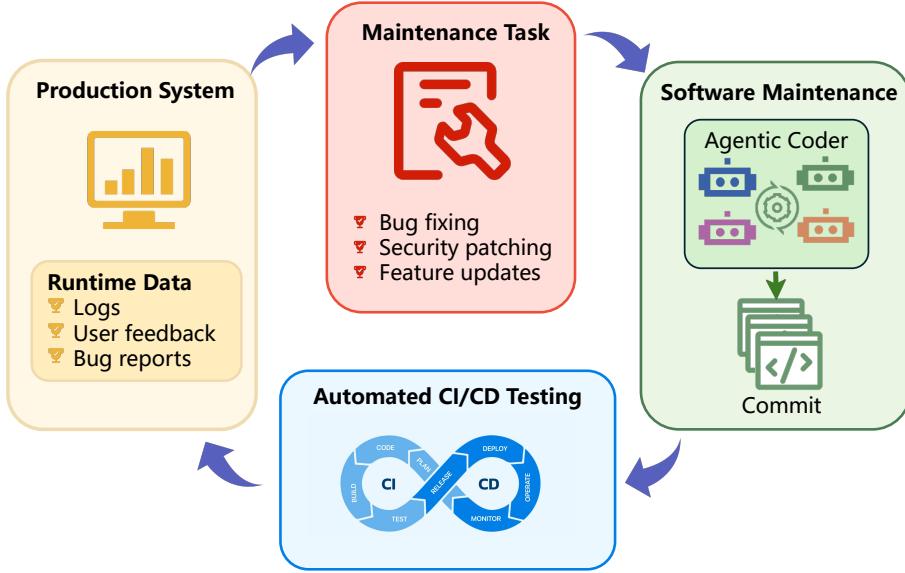


Figure 36. Overview of software maintenance across the full software life-cycle.

- **Pattern Mining and Clustering.** Early systems relied on deterministic pattern mining and clustering to extract structure from noisy log data. Methods such as DLog [566] and FT-tree [1278] identify invariant token sequences by constructing prefix or token-frequency trees. Clustering-based approaches, exemplified by LPV [1120], group logs with similar structures using hierarchical or density-based clustering, enabling unsupervised template discovery at scale.
- **Heuristic and Neural Parsing.** Subsequent work introduced heuristic matching techniques such as longest-common-subsequence alignment in LTmatch [184]. Neural approaches emerged with models like LogStamp [945], which frame log parsing as a token classification task using pretrained encoders. Advanced neural systems such as LogBERT [364] and LogPrompt [631] incorporate self-supervised learning and prompt tuning to improve adaptability across log formats. Although these methods significantly improved precision in structured environments, they lack dynamic reasoning and cross-context adaptability, motivating the emergence of more flexible agent-based paradigms.

Agent-based Log Analysis Agent-based approaches redefine log analysis as an iterative reasoning task in which autonomous agents interpret system states, generate hypotheses, interact with external tools, and refine conclusions based on feedback. Compared with static parsers, these systems emphasize adaptability, cognitive reasoning, and coordinated decision making.

- **Single-Agent Reasoning.** Frameworks such as R-Log [632] reformulate log analysis from a static mapping problem to a reasoning-centric pipeline in which the agent produces intermediate explanations before synthesizing conclusions. By incorporating reinforcement learning in simulated O&M environments, the system learns to optimize factual soundness and reasoning coherence across diverse log scenarios. AdaptiveLog [670] focuses on efficiency and accuracy by orchestrating a lightweight language model with a more capable one. Using Bayesian uncertainty estimation, the system selectively escalates challenging cases to the stronger model, achieving an effective balance between computational cost and analytical fidelity. The ReAct-RCA framework [784] employs a

Thought–Action–Observation loop in which the agent iteratively interacts with domain knowledge bases and telemetry data. During root-cause analysis, the agent continuously refines intermediate hypotheses, enabling a more adaptive and process-oriented form of operational intelligence.

- **Multi-Agent Collaboration.** Building on single-agent reasoning, recent frameworks employ multiple specialized agents to address heterogeneous log sources and complex operational contexts. LogRESP-Agent [526] integrates recursive plan–act–observe reasoning with retrieval-augmented generation, enabling context-aware anomaly investigation and template-free parsing across varied log datasets. CyberSleuth [302] organizes sub-agents dedicated to network-level and application-level forensics. By incorporating targeted web searches to ground reasoning in external vulnerability knowledge, the system minimizes hallucination and strengthens analytical robustness. Audit-LLM [910] advances this paradigm through a structured decomposition of detection tasks into three agents: a *Decomposer* that identifies subtasks, a *Tool Builder* that configures external utilities, and an *Executor* that performs the final analysis. These agents interact through an Evidence-Based Multi-Agent Debate mechanism, where candidate conclusions are iteratively challenged and validated to enhance decision fidelity.

Overall, the evolution of log analysis reflects a clear trajectory from deterministic parsing toward neural and agent-based reasoning. Traditional methods established the foundations for structuring and clustering logs, while modern agent-based systems introduce dynamic reasoning, tool interaction, and multi-agent coordination. Together, these approaches represent a shift toward more adaptive, interpretable, and context-aware operational intelligence.

Compiler Optimization Compiler optimization seeks to generate efficient executables by identifying transformation strategies that improve performance and portability across diverse architectures. Traditional compiler techniques laid the foundation for this task, and growing system complexity has gradually motivated a shift toward learning based and agent driven approaches.

Traditional Foundations Before the development of intelligent or learning based optimizers, compiler optimization primarily relied on empirical exploration and hand crafted heuristics. Representative methods include the following:

- **Iterative Compilation.** Early research explored compiler strategy spaces through repeated compilation, execution, and profiling, enabling systems to search for high quality optimization configurations [114]. Search based variants, including genetic algorithms [207], random exploration [8], and greedy heuristics [777], further expanded this paradigm. Frameworks such as OpenTuner [53] and CLTune [737] provided general infrastructures for empirical tuning, although these methods often lack adaptivity and incur substantial cost.
- **Machine Learning for Compilers.** To reduce search overhead, supervised learning techniques were introduced to predict optimization sequences or compiler flags directly from program features. Systems such as MilepostGCC [303] and DeepTune [214] demonstrate that predictive models can guide optimization more efficiently than exhaustive exploration. However, these methods remain static and do not dynamically reason about optimization choices in context.

Agent Based Compiler Optimization With the growth of intelligent decision making techniques, compiler optimization has increasingly been reframed as a sequential reasoning problem. Instead of relying on fixed heuristics, agent based systems explore, evaluate, and refine optimization strategies through interaction and feedback. Key directions include:

- **LLM Augmented Optimization.** Recent work investigates large language models as meta optimizers capable of reasoning about code semantics and compiler configurations. Cummins et al. [215] demonstrate that language models can infer optimization flags from unoptimized assembly, revealing the potential of natural language guided optimization. These systems integrate in context reasoning and retrieval to dynamically adapt compiler strategies in a more interpretable manner.
- **Reinforcement Learning Agents.** Reinforcement learning based frameworks treat the compilation pipeline as a Markov Decision Process, where optimization passes correspond to actions and performance signals provide rewards. AutoPhase [422] shows that learned policies can outperform manually crafted pass orderings and better adapt to program characteristics.
- **World Model Driven Optimization.** Building on reinforcement learning, Compiler-Dream [239] introduces a world model that simulates compiler behavior. By training a predictive model to approximate compilation outcomes, the system enables broad strategy exploration in a simulated environment and supports generalization across languages and architectures through offline reinforcement learning.

Overall, compiler optimization has evolved from heuristic and search based exploration toward learning guided prediction and, more recently, agent driven adaptive reasoning. This progression reflects a broader trend toward compilers that continuously refine and generalize optimization strategies through interaction, feedback, and higher level semantic understanding.

Decompilation Decompilation aims to recover high-level source code from low-level representations such as assembly or bytecode. It is a fundamental task in reverse engineering, software maintenance, and security analysis. Traditional rule-based tools often struggle to reconstruct high-level semantics, especially when facing compiler optimizations. Recent advances in neural models and large language models have substantially reshaped this field by framing decompilation as a code translation and semantic reconstruction task.

Neural and Learning-Based Decompilation Early learning-based systems adopt neural machine translation techniques to translate low-level programs into high-level languages. Neutron [584] applies attention-based sequence modeling to generate human-readable source code while preserving functional behavior. To better handle compiler-optimized binaries, NeurDP [133] uses graph neural networks to map low-level representations into intermediate forms that bridge the semantic gap between binary and source code. Transformer-based approaches further demonstrate strong adaptability; BTC [402] treats assembly and source languages as plain text, enabling retargeting across different programming languages, while Slade [72] infers types and reconstructs coherent code structures using a sequence-to-sequence Transformer.

LLM-Augmented Decompilation With the rise of large language models, decompilation has increasingly been approached as a semantic reasoning and code reconstruction task.

DecGPT [1078] proposes a hybrid workflow that uses pretrained models to repair syntax and memory issues in decompiled C programs, improving their correctness and executability. Nova+ [460] enhances LLM training through hierarchical attention and contrastive objectives that better capture assembly-level semantics. LLM4Decompile [934] introduces a family of open-source models trained specifically for decompilation, along with end-to-end variants capable of producing high-level code directly from binary input.

Agent-Based Decompilation Recent work frames decompilation as an agentic reasoning task in which models interact with structured program information and external tools to improve reconstruction quality. CFADecLLM [616] exemplifies this direction by combining traditional disassembly outputs, such as control flow graphs, with natural language or structured token representations. Through a two-stage information fusion mechanism and role-guided prompting, the system integrates both instruction sequences and structural program analysis into a unified reasoning process. This approach leverages the strengths of program analysis techniques and the generative capabilities of language models, enabling more robust and semantically faithful recovery of high-level code.

Summary Overall, the field has progressed from rule-based heuristics to neural translation models, and from general-purpose LLMs to agent-based systems that incorporate structured program analysis. This evolution reflects a broader trend toward decompilers that not only translate low-level code but also reason about program semantics, control flow, and high-level intent.

Deobfuscation Deobfuscation refers to the reverse process of renaming identifiers, where meaningful identifier names are recovered from obfuscated programs. DeGuard [109] is proposed for deobfuscating Android APKs based on probabilistic learning of large code bases. The key idea is to learn a probabilistic model over thousands of non-obfuscated Android applications and to use this probabilistic model to deobfuscate new, unseen Android APKs. Autonym [995], based on statistical machine translation (SMT), recovers some original names from JavaScript programs minified by the highly popular UglifyJS. Debin [385] handles ELF binaries on three of the most popular architectures: x86, x64, and ARM through machine learning. JSNeat [984] follows a data-driven approach to recover names by searching for them in a large corpus of open-source JS code. In addition to the aforementioned traditional approaches, recent studies have incorporated neural networks. DIRE [508] uses both lexical and structural information recovered by the decompiler for variable name recovery. Artuso et al. [74] investigate the problem of automatically naming pieces of assembly code. They justify training by assuming function names reflect semantics, using debug symbols as labels, and evaluate via standard NLP metrics against original names. VarBERT [95] is an early method to integrate a transformer-based model. It infers variable names in decompiled code based on Masked Language Modeling, Byte-Pair Encoding, and neural architectures such as Transformers and BERT. Similar to VarBERT, DIRECT [733] is another transformer-based architecture customized specifically for decompiling binary executables to high-level code. More recent works apply LLM to handle the decompilation task. For example, LmPa [1147] leverages the strengths of pre-trained generative models (CodeGemma-2B, CodeLlama-7B, and CodeLlama-34B) while mitigating model biases while mitigating model biases by aligning output distributions with developer symbol preferences and incorporating contextual information from caller/callee functions during both training and inference.

For multi-layer obfuscation and malicious code analysis scenarios, agents achieve adaptive deobfuscation through semantic recovery and tool collaboration. Traditional deobfuscation tools typically handle only specific types of obfuscation and struggle to cope with mixed and overlapping obfuscation techniques. Agents leverage the semantic understanding capabilities of large models, combined with tool assistance, to automatically identify and remove multiple obfuscation patterns: analyzing obfuscated code segments, identifying redundant patterns that do not affect business logic (such as useless loops or meaningless operations) and removing or simplifying them, while inferring more meaningful identifier names and adding comments based on context. The ALFREDO framework [723] employs a classifier to identify obfuscation types, then invokes corresponding deobfuscation tools or strategies for different obfuscation types (such as calling Ghidra to extract low-level logic, or letting LLM attempt code reconstruction), gradually eliminating obfuscation components in the code through iterative loops. This multi-step reasoning and tool-using framework can process mixed and overlapping obfuscations in a single workflow. Androidmedia [10] employs a context-based semantic reasoning approach, analyzing code context, method call relationships, and variable usage patterns, utilizing LLM's semantic understanding capabilities to infer more meaningful identifier names and generate comments. In malicious software analysis scenarios, agents combine pattern recognition and semantic recovery [782]: first identifying obfuscation patterns (such as control flow flattening, false branch insertion, etc.), then gradually restoring the original logic structure based on code semantics and contextual information. This class of Agentic frameworks adopts multi-step iterative methods: identifying obfuscation types, invoking corresponding deobfuscation tools or semantic reconstruction modules, and continuously optimizing deobfuscation results through feedback loops.

DevOps and CI/CD In modern DevOps, agents act as *autonomous collaborators* that bring adaptivity and intelligence to continuous integration and deployment (CI/CD) pipelines. While traditional pipelines execute sequentially according to predefined scripts, agent-based DevOps frameworks dynamically adjust their workflows based on real-time context, execution feedback, and learned experience, transforming automation from static scripting into interactive decision-making.

AutoDev [988] exemplifies this paradigm through a conversational multi-agent architecture. It integrates a *Conversation Manager* for maintaining dialogue context, an *Agent Scheduler* for task decomposition and delegation, and a unified tool library for interacting with IDEs, build systems, and test frameworks. The scheduler decomposes complex goals into subtasks—such as code editing, testing, and deployment—and dispatches them to specialized agents running in isolated Docker sandboxes. Execution results are continuously fed back into the conversation loop, allowing agents to iteratively plan, act, and refine their strategies. This architecture reframes DevOps automation as a problem of multi-agent coordination and dialogue-driven orchestration.

Within CI/CD pipelines, agents enhance control and decision-making through multiple mechanisms. They identify flaky tests via log analysis and selectively re-run or skip them; construct code–test dependency graphs to enable incremental test selection; and apply reinforcement learning or heuristic strategies to optimize concurrency and resource allocation. During deployment and monitoring, agents employ anomaly detection and priority assessment to trigger rollback actions or notify engineers of critical failures, integrating multimodal signals such as logs, metrics, and code changes for holistic evaluation.

GPT-Engineer [765] further extends DevOps automation toward development itself, en-

abling end-to-end workflows from natural-language requirements to executable code through iterative generation, testing, and self-improvement. **CodeAgent** [937] advances this paradigm through the *Code-as-Action* framework, where agents generate and execute Python code as dynamic actions to perform editing, analysis, and deployment tasks. In CI/CD contexts, CodeAgent integrates static analysis, dependency tracking, and test coverage estimation for intelligent test selection, while reinforcement learning models guide deployment adjustments and post-release optimization.

Collectively, these agentic DevOps frameworks transform software delivery from static, rule-driven automation into adaptive ecosystems capable of autonomous reasoning, coordinated task execution, and continuous self-optimization across distributed software infrastructures.

5.1.5. End-to-End Software Agents

Building on repository-level reasoning, a recent wave of research [1, 273, 273, 401, 592, 724, 811, 812, 837, 1246, 1280] extends agentic coding beyond implementation toward the full software development life cycle (SDLC)—covering requirements elicitation, design, implementation, testing, and maintenance. These systems aim to realize end-to-end software development through coordinated multi-agent workflows that emulate human engineering teams.

Waterfall-based full-cycle agents Early end-to-end frameworks such as Self-Collaboration, ChatDev [811], and MetaGPT [401] model a sequential, role-specialized workflow inspired by the waterfall process, where designated agents (e.g., CEO, CTO, Developer, Tester) collaborate through standardized operating procedures (SOPs) to complete projects from requirement analysis to final implementation. Subsequent systems including AISD [1280], CTC [273], and CodePori [837] extend this paradigm into iterative requirement–design–implementation–testing cycles, integrating execution feedback to refine upstream design decisions and improve code correctness. These works demonstrate that structured role assignment and hierarchical coordination can enable LLMs to manage complex multi-stage engineering tasks autonomously.

Agile and iterative frameworks Complementary to waterfall-style orchestration, other frameworks adopt agile or test-driven methodologies to enhance adaptability and incremental refinement. LCG [592], AgileCoder [724], and CodeS [1246] organize multiple agents around sprint-style development loops, combining planning, coding, and testing within each iteration. Systems such as Iterative Experience Refinement [812] and Cross-Team Collaboration [273] further incorporate experience reuse and inter-agent learning to stabilize long-horizon collaboration. By embedding feedback and shared memory, these agile frameworks approximate the iterative development patterns of human teams while retaining automation benefits.

5.2. General Code Agents in Software Engineering

As large language models continue to make breakthroughs in code generation, understanding, and reasoning capabilities, researchers have begun exploring how to build general code agents that can span all stages of software development. Unlike specialized agents that focus on only a single aspect (such as requirement analysis, testing, or debugging), these systems aim for full-process intelligent collaboration, seeking to take complete responsibility from task planning and code generation to debugging and deployment in real software engineering environments. Through multi-turn interactions, context tracking, and tool invocation, they achieve autonomous understanding and execution of complex engineering tasks, providing

developers with continuous and unified intelligent support. The following will introduce several representative general code agent systems.

- **CodeAct** [1039]: CodeAct extends traditional text and JSON-formatted actions to executable python code, supporting dynamic code generation and multi-turn interaction through an integrated python interpreter. This framework leverages the control flow features of programming languages, enabling the agent to store intermediate results, combine multiple tools, and possess self-debugging capabilities.
- **OpenHands** [1040]: OpenHands 3adopts an event-stream architecture to manage agent-environment interactions and ensures code execution security through a Docker sandbox environment. It provides diverse execution environments, including a bash terminal, a Jupyter IPython server, and a playwright-based browser, and supports a multi-agent collaboration mechanism
- **OpenCode** [760]: OpenCode aims to provide a controllable and extensible intelligent programming environment. Its core adopts a **multi-agent architecture**, consisting of a *Plan Agent* responsible for analysis and planning, a *Build Agent* for executing modifications, and a *General Agent* for auxiliary queries. This design enables the **decoupling of reasoning and action** while ensuring safety and controllability.
- **Aider** [949]: Aider emphasizes human-AI collaboration rather than full autonomy. It achieves excellent performance on the SWE-Bench benchmark through precise static code analysis and an efficient LLM-based code editing mechanism. The framework incorporates multi-level linting and automated test repair logic, simulating the pair-programming workflow of actual developers.
- **Augment** [951]: Augment is a code agent designed for professional software engineering scenarios, targeting **end-to-end intelligent collaboration** from code comprehension to execution. It possesses proactive task execution and contextual management capabilities. Its core architecture incorporates a powerful **repository-level semantic understanding engine (context engine)** that indexes and retrieves functions, dependencies, and documentation within large codebases to maintain global consistency across multi-file and multi-module environments. In addition, Augment introduces **memory** and **rule** systems to learn developer habits and project conventions, enabling personalized and continuous collaboration. With its comprehensive codebase understanding, executable capabilities, and extensible design, it represents a robust paradigm for industrial-scale code agents.
- **Trae Agent** [310]: Trae Agent is an open-source, general-purpose code agent designed for software engineering workflows, notably capable of natural-language command execution, file editing, bash invocation, and large-codebase reasoning. Trae Agent formulates repository-level issue resolution as an optimal-search problem in a large ensemble space, and achieves strong results by combining generation, pruning, and selection modules.
- **Refact Agent** [964]: Refact Agent is designed to achieve **full-process automation** from requirement parsing and code generation to debugging and deployment. The system can deeply analyze entire code repositories, synchronize with development environments in real time, and perform multi-step tasks through integration with terminal commands, version control systems (e.g., Git), and CI/CD pipelines. Its core features include support for **over 25 programming languages**, the adoption of **retrieval-augmented generation (RAG)** techniques for project-specific contextual understanding, and **self-hosted deployment** options to ensure data privacy and security.

5.3. Training Techniques for SWE Agents

5.3.1. Fine-tuning SWE Agents

SFT is a critical process for adapting pre-trained language models into specialized SWE agents capable of performing specific and nuanced tasks. In practice, LLM is fine-tuned only with successful trajectories by filtering failed samples (cannot pass the unit tests) for better performance, which is denoted as the rejection sampling fine-tuning (RFT) [1237]. The common methodologies observed across the literature can be synthesized into three core areas: the strategic creation of training data, the design of sophisticated training objectives, and the adoption of scalable and adaptive training frameworks.

Strategic Curation and Synthesis of High-Fidelity Training Data The foundation of any effective fine-tuned agent lies in the quality and relevance of its training data. A prominent trend is the move beyond using raw datasets toward strategic data refinement and synthesis, which involves meticulously filtering noise, verifying correctness through execution, and creating complex, structured instances from the ground up. This data-centric approach ensures agents learn from clear, relevant, and representative examples. Key strategies include:

- **High-Fidelity Data Enhancement and Filtering** State-of-the-art methods actively enhance existing corpora by filtering noise and augmenting them with high-quality synthetic data. This includes using LLMs as classifiers to systematically remove non-actionable comments from code review datasets [602], creating cleaner training signals. Concurrently, datasets are augmented by synthesizing artificial but realistic bugs from real-world commit histories, providing rich training pairs for fault localization and repair tasks [261]. Furthermore, hybrid augmentation methods enrich datasets by combining the formal rigor of static analyzers with the generative flexibility of LLMs, producing training examples that are both semantically diverse and structurally sound [448].
- **Execution-Driven and Verifiable Data Augmentation** To guarantee the semantic correctness of training data, many frameworks incorporate an active verification loop. The CodeS framework, for instance, employs an execution-checked augmentation strategy that generates multiple equivalent SQL variants and verifies their functional correctness before adding them to the training set [541]. A broader approach is seen in SPICE, which integrates program analysis with human verification to construct large-scale, high-quality test datasets [106]. The principle of using execution feedback extends to shaping training objectives themselves; RewardRepair integrates execution outcomes into training via reinforcement-style objectives, effectively using runtime success as a verified signal to guide the model toward correct fixes [1215].
- **End-to-End Structured Data Synthesis** The most sophisticated approaches involve synthesizing entire complex datasets from scratch to mirror real-world software engineering processes. The OmniSQL pipeline is a prime example, as it bootstraps relational databases, generates complexity-aware SQL, back-translates it into diverse natural-language questions, and synthesizes chain-of-thought solutions to create comprehensive training quadruples [542]. Similarly, SWE-Flow achieves high-quality data synthesis for test-driven development by automatically inferring incremental development steps and constructing runtime dependency graphs to generate structured plans and code [1272]. This philosophy of building large, structured corpora is also seen in frameworks like DIDACT, which constructed a massive, billion-example dataset specifically for multi-task pretraining on code, providing a strong foundation for various downstream tasks [680].

Refining Model Behavior through Structural Objectives The efficacy of fine-tuning is also determined by the specific algorithms and learning objectives used to guide the model’s adaptation. Research has progressed from generic sequence-to-sequence modeling toward more sophisticated objectives that explicitly incorporate the structural and semantic properties of code, leading to more reliable and accurate agents.

- **Multi-Task and Curriculum-Based Learning** To build a strong semantic foundation, agents are often pre-trained or fine-tuned on a curriculum of related tasks. In review generation, for instance, models are trained on a combination of objectives, including masked language modeling, diff tag prediction, and code denoising, before being fine-tuned on human comments [560, 579, 680]. This multi-task approach equips the model with a more robust and generalized understanding of code changes, which directly benefits the final review generation task. Similarly, in patch generation, multitask learning is used to adapt a single model to the varied demands of automated program repair [319].
- **Incorporating Structural Constraints** A critical advancement is the design of training objectives that respect the inherent structure of code. To ensure syntactic validity in patch generation, the task is often reframed as a structured editing problem that uses grammar-constrained decoding or pointer-based models to perform precise, syntactically correct modifications [415, 1332]. In the text-to-SQL domain, models may first be trained to predict a high-level SQL “skeleton” before being fine-tuned to fill in the specific table and column names, decomposing the task in a structure-aware manner [541]. Other methods explicitly incorporate data-flow information into the learning process of model or use graph neural networks to process code’s structural representations [133, 182].
- **Specialized Learning Formulations** Researchers have also explored novel learning formulations tailored to specific SWE challenges. In fault localization, contrastive learning is employed to train a retriever to distinguish between relevant and irrelevant code snippets for a given bug, which sharpens the model’s ability to identify contextually important information for repair [473]. A similar contrastive approach is used in decompilation to train models to learn the subtle semantic nuances of assembly optimizations, which is difficult with traditional loss functions [460]. These specialized objectives guide the model to learn more effectively from the unique characteristics of the problem domain.

Evolving Paradigms for Scalable and Continuous Adaptation The overarching strategies, or paradigms, for applying fine-tuning have also matured, with a growing emphasis on computational efficiency, adaptability to new contexts, and the capacity for continuous improvement over time.

- **Parameter-Efficient Fine-Tuning (PEFT)** The immense size of LLMs makes full fine-tuning computationally prohibitive. Consequently, PEFT methods such as low-rank adaptation (LoRA) [405] and QLoRA [245] have become standard practice [1336]. These techniques allow for the adaptation of very large models by training only a small fraction of their parameters. This approach is widely used across tasks like code review generation [375, 646] and patch generation [900], enabling the creation of highly specialized agents in a scalable and cost-effective manner.
- **Multi-Stage and Continual Learning Frameworks** For complex tasks that require deep reasoning, a multi-stage training process is often more effective. In document generation, a “retrieve-then-generate” fine-tuning strategy first trains the model to identify relevant contextual information from a codebase before training it to synthesize the final explanation, mirroring a more effective human-like workflow [212]. To ensure agents remain

effective over time, continual learning paradigms are employed, allowing a model to adapt to new programming languages, libraries, or evolving codebases without catastrophically forgetting previously learned knowledge, a key requirement for long-term multilingual program repair [1234].

- **Direct Task-Specific Adaptation** While more complex paradigms are emerging, the foundational approach of directly fine-tuning a general model for a single, well-defined task remains highly effective. This is demonstrated across numerous domains where models like T5 or LLaMA are successfully adapted into specialized agents for code review [1228] or automated program repair [1110]. The success of this direct approach is heavily contingent on the quality of the curated training dataset, reinforcing the central importance of the data-centric strategies discussed earlier.

5.3.2. Reinforcement Learning for SWE Agents

RL provides a distinct paradigm for training SWE agents, shifting the focus from mimicking static and expert-annotated datasets to learning optimal behaviors through direct interaction with an environment. Unlike SFT, which relies on pre-existing ground-truth data, RL allows agents to improve their decision-making policies by receiving feedback (in the form of rewards) for the outcomes of their actions. This approach is particularly well-suited for complex and multi-step tasks in software engineering where the notion of the correctness of a trajectory is ill-defined, but the quality of a final outcome (e.g., a passing test suite or an optimized program) is empirically measurable. Existing works highlight multiple key strategies for applying RL to enhance agent capabilities, centering on the design of reward functions, the formulation of the learning environment, and the algorithms used to drive policy improvement.

RL Algorithm Selection

- **Policy Gradient Methods (e.g., REINFORCE [407, 581] and A2C/PPO [870])** Policy gradient methods directly optimize the decision-making policy and are well-suited for tasks with large, complex, or continuous action spaces where a direct mapping from state to action is needed. This makes them ideal for sequential decision-making problems common in software engineering. For instance, in compiler optimization, the task of selecting an optimal sequence of transformation passes from a vast library of options is a perfect use case. This is exemplified by AutoPhase [422], which employs policy gradients to learn how to arrange and combine optimization passes for the LLVM compiler. The same algorithmic principle would be effective in learning multi-step debugging and repair strategies within agentic frameworks like ITER [1213] and RepairAgent [120], where the agent must learn a policy to decide the next action (e.g., rerun tests, attempt a different patch, refine localization) based on the current state of the repair process.
- **Offline Reinforcement Learning** Offline RL algorithms are designed to learn from a fixed, pre-collected dataset of interactions without actively exploring the environment. This approach is critical for SWE tasks where live interaction with the environment is prohibitively expensive, slow, or risky. The foremost example is compiler optimization, where compiling and benchmarking thousands of program variants is infeasible. CompilerDream [239] explicitly uses offline RL to learn from a world model trained on existing compilation data, thereby avoiding costly real-world interactions. This paradigm is also highly relevant for automated program repair, where running a full test suite after every minor change is a major bottleneck. An offline RL agent could learn from a large historical dataset of bug reports, code changes, and test outcomes from systems like RewardRepair [1215] or the

iterative attempts logged by Reflexion [894] to derive an effective repair policy without needing extensive live testing during training.

- **Value-based and Preference-based Methods (e.g., Q-Learning and RLHF)** This class of algorithms is optimal when the reward signal is not easily defined by a simple and objective metric but is instead based on qualitative or preferential feedback. Reinforcement learning from human feedback (RLHF) is the most prominent example, where a reward model is trained to reflect human preferences. This is essential for tasks involving human-computer interaction or subjective quality assessment, such as automated code review. CodeMentor [722] uses RLHF to align its generated reviews with human expectations for tone and contextual relevance. This approach is equally applicable to refining the quality of reasoning in other domains; for instance, the reward for generating high-quality, factual reasoning traces in R-Log [632] could be learned from human ratings. Similarly, in multi-agent debate frameworks like LLM4FL [830] or SWE-Debate [539], a preference-based reward model could be trained to score the persuasiveness and correctness of agents' arguments, guiding them toward more effective collaborative problem-solving.

Optimizing for Task Success with Execution-Based Rewards A primary application of RL in SWE agents is to directly optimize for tangible, verifiable outcomes. In this setup, the environment is often a simulated or real software project, and the agent's actions (e.g., generating a patch, selecting a compiler flag) are evaluated based on their real-world effect. The reward signal is typically sparse but unambiguous, tied directly to the success or failure of the task. This aligns the agent's learning objective with the ultimate goal of the SWE task itself.

- **Automated Program Repair** In this domain, the reward is directly correlated with the successful validation of a generated patch. Frameworks like RewardRepair [1215] integrate execution feedback from test suites into the training process using reinforcement objectives, rewarding the model for generating fixes that pass tests. This principle of learning from trial-and-error is central to agentic systems like Reflexion [894], which use feedback to convert failed attempts into informed iterations and cache past mistakes to avoid repeated failures, effectively learning a policy to escape negative reward cycles. Similarly, agentic frameworks such as ITER [1213] and RepairAgent [120] operate on a continuous feedback loop of generation and validation, where the successful outcome of the validation phase serves as a positive reward signal that reinforces the agent's repair strategy.
- **Compiler Optimization** RL agents learn to navigate the vast search space of compiler transformations by receiving rewards based on program performance. Systems like AutoPhase [422] model the selection of optimization passes as a Markov Decision Process and use policy gradient algorithms, where the agent is rewarded for sequences of passes that result in faster execution times. Further, CompilerDream [239] employs a world-model-based approach with offline reinforcement learning, allowing the agent to learn an effective optimization policy by exploring in a simulated environment, where rewards are based on predicted performance outcomes, thus drastically reducing the cost of real-world compilation and profiling. The foundational concept of iterative compilation [114], which relies on a cycle of compiling, executing, and profiling, establishes the feedback mechanism that these modern RL agents formalize and automate to learn their policies.
- **Security and Vulnerability Management** The feedback loops inherent in security tasks provide clear reward signals for RL agents. The multi-agent system for securing code introduced by Nunez et al. [738] uses continuous feedback loops between coding, analysis, and fuzzing agents; an RL framework can reward the system for autonomously detecting and repairing vulnerabilities identified by the fuzzer. In the forensic analysis system

CyberSleuth [302], an agent could be rewarded for correctly identifying threats or linking log data to known CVEs, thereby learning an efficient investigation policy. In multi-agent debate systems like Audit-LLM [910], RL could improve performance by rewarding agents that successfully challenge opponents or contribute to accurate threat detection.

Refining Qualitative Behaviors and Multi-Agent Collaboration Beyond simple task success, RL is also employed to shape more nuanced and qualitative aspects of an agent’s behavior, such as the clarity of its reasoning, the helpfulness of its feedback, and its ability to collaborate effectively with other agents. In these cases, the reward mechanism is often more complex, sometimes incorporating human feedback or rewarding intermediate reasoning steps to encourage more robust and interpretable decision-making processes.

- **Aligning with Human Preferences and Quality Standards** Reinforcement Learning from Human Feedback (RLHF) is a key technique for training agents to produce outputs that are not just technically correct but also useful and well-regarded by developers. CodeMentor [722], for instance, employs RLHF to improve the contextual accuracy and tone of its automated code reviews, learning a reward model from human ratings of its suggestions. This same principle can be applied to data curation; for example, an agent could be trained using RLHF to filter non-actionable review comments, learning from human feedback which comments are considered valuable, a process explored in the work by Liu et al. [602]. Similarly, in the context of reformulating comments for clarity, as studied by Sghaier et al. [877], RLHF can provide the necessary reward signal to guide an agent toward producing more helpful and understandable text.
- **Enhancing Reasoning and Interpretability** RL can be used to reward the cognitive process of an agent, not just its final output, leading to more transparent and reliable systems. The R-Log [632] framework exemplifies this by using reinforcement learning in a simulated environment to directly reward an agent based on the factual soundness and quality of its intermediate reasoning traces during log analysis. This focus on the thought process is also seen in the ReAct-RCA framework [784], where an agent’s *Thought–Action–Observation* loop could be optimized with RL to reward thought patterns that lead to more efficient root-cause analysis. This approach could also enhance models like ThinkRepair [1218], which uses chain-of-thought prompting; an RL layer could be added to reward the generation of more logical and effective reasoning chains that result in successful program repairs.
- **Optimizing Multi-Agent Dialogue and Collaboration** In systems with multiple interacting agents, RL provides a mechanism for learning effective communication and collaboration strategies. The LLM4FL [830] framework utilizes a reinforcement-based dialogue to enable agents to dynamically exchange feedback and achieve consensus during fault localization. This concept is extended in debate-based frameworks like Audit-LLM [910] and SWE-Debate [539], where the competitive or collaborative debate provides a natural environment for RL. Agents can be rewarded for constructing persuasive arguments, successfully identifying flaws in others’ reasoning, or contributing evidence that moves the group toward a correct and robust conclusion, thereby learning an optimal policy for collaborative problem-solving.

Learning Sequential Decision-Making in Dynamic Environments Many software engineering tasks are inherently sequential and take place in dynamic, stateful environments. RL is a natural fit for these problems, as it allows agents to learn complex, multi-step policies for navigating these environments. By modeling the task as a Markov decision process (MDP),

agents can learn to make a sequence of decisions that maximizes a cumulative reward, enabling them to handle long-horizon tasks like interactive debugging, strategic planning, or managing CI/CD pipelines.

- **Strategic Exploration of Code and Solution Spaces** RL, particularly in conjunction with search algorithms, enables agents to learn how to explore vast and complex solution spaces more efficiently. The SWE-Exp [163] framework explicitly uses Monte Carlo tree search (MCTS), a method with strong ties to RL, to enhance decision-making and learn from both successful and failed repair trajectories, effectively building an experience-driven policy. This use of MCTS is also seen in LingmaAgent [673], which combines it with knowledge graphs to inform its search for dynamic patch synthesis. The self-evolution principles of SE-Agent [593], which systematically expands its exploration and optimizes reasoning trajectories based on past insights, represent a high-level form of policy improvement that is central to the reinforcement learning paradigm.
- **Interactive and State-Aware Repair Processes** Automated repair is not a one-shot task but an interactive process of hypothesis, testing, and refinement. Frameworks such as Reflexion [894] embody this by turning trial-and-error into informed iteration and using memory to avoid past mistakes, which is analogous to an RL agent learning a policy from environmental feedback (e.g., negative rewards on failed patch attempts). Conversational APR systems [1111, 1112] create an interactive loop where an RL agent could learn an optimal dialogue policy, deciding when to attempt a new patch versus when to ask for more information to maximize its long-term success rate. Likewise, the continuous feedback loop in RepairAgent [120], which integrates localization, generation, and validation, defines a stateful environment where an RL agent could learn a policy to dynamically decide the next best action (e.g., re-run localization, attempt a different kind of patch) based on the history of its interactions.
- **Navigating CI/CD and DevOps Pipelines** RL can be used to train agents that intelligently manage and optimize the long-horizon processes of continuous integration and deployment. CodeAgent [937] applies reinforcement learning models to guide deployment adjustments and post-release optimizations, learning from real-time feedback to improve pipeline performance over time. The AutoDev [988] framework, with its *Agent Scheduler* for task decomposition, provides an ideal setting for an RL-trained meta-agent that learns the optimal policy for delegating subtasks to specialized agents based on the evolving state of the project. Similarly, the ExecutionAgent [119], which handles test execution through iterative feedback loops and error recovery, leverages RL to learn the most effective sequence of recovery commands to try when encountering different types of build or test failures.

Implementation Details and Synergies with Supervised Fine-Tuning The successful application of RL in SWE agents is rarely a standalone process. It is almost always preceded by and intertwined with SFT. This symbiotic relationship is critical for stabilizing training, improving sample efficiency, and ensuring that the exploration of agent is grounded in a solid foundation of domain knowledge. The key implementation considerations revolve around the necessity of SFT as a pre-training step, the strategic balance between SFT and RL training phases, and the potential for cyclical, data-driven improvement.

- **SFT as a Necessary Foundation for RL** Attempting to train an SWE agent with RL from a general-purpose, non-fine-tuned model is often intractable. The action space (e.g., the space of all possible code edits or review comments) is vast and sparsely rewarded. SFT

serves as a critical bootstrapping phase that initializes the agent’s policy with a strong prior based on high-quality, human-generated examples. This pre-training teaches the model the fundamental syntax, semantics, and common patterns of the target task, dramatically constraining the search space for the subsequent RL phase. For example, in automated program repair, a model is first fine-tuned on large datasets of bug-fix commits, as seen in approaches like AlphaRepair [1110] and T5APR [319]. Only after this SFT phase does it become feasible to apply an RL objective, like in RewardRepair [1215], to optimize for the specific goal of passing a test suite. Without the initial SFT, the agent would generate syntactically invalid or semantically nonsensical code, making it nearly impossible to discover a rewarding trajectory.

- **Balancing SFT and RL Training Ratios** The allocation of training resources (in terms of data volume and training epochs) between SFT and RL is a critical hyperparameter that balances knowledge acquisition with goal-oriented optimization. The standard and most effective methodology is a multi-stage approach: a comprehensive SFT phase followed by a more targeted RL phase. An extensive SFT phase, such as the one enabled by the massive, multi-task DIDACT [680] dataset for code review or the structured data synthesis in OmniSQL [542] for Text-to-SQL, builds a robust and generalized base model. The subsequent RL phase (e.g., using RLHF as in CodeMentor [722]) can then be shorter, as its primary role is to refine the model’s behavior and align it with specific success metrics rather than teaching it the task from scratch. Over-relying on SFT risks creating a rigid policy that mimics the training data too closely and struggles to explore novel solutions, while insufficient SFT leads to an inefficient and unstable RL process. The optimal ratio ensures the agent begins exploration from a high-quality starting point without being overly constrained by it.
- **Iterative Cycles of RL and SFT for Continual Improvement** The most advanced training paradigms treat the relationship between SFT and RL not as a linear and one-time sequence, but as a continuous and iterative cycle. In this paradigm, the high-quality and successful trajectories discovered by the RL agent are used to create new and high-fidelity data for a subsequent round of SFT. This process distills the knowledge gained through environmental interaction and exploration back into the model’s parameters, creating a self-improving loop. For instance, the experience repository in SWE-Exp [163], which stores successful and failed repair trajectories, provides a perfect source of data for this cycle. The effective patches discovered through its exploration can be added to an SFT dataset to improve the next iteration of the base model. Similarly, the self-evolutionary process of SE-Agent [593], which revises and optimizes reasoning trajectories, generates improved problem-solving strategies that can be used as high-quality examples for fine-tuning. This hybrid approach leverages RL for discovery and SFT for knowledge consolidation, enabling the agent to learn and adapt over time in a scalable and data-efficient manner.

5.4. Future Trends: Towards Integrated and Autonomous Software Engineering Ecosystems

The development of SWE Agents, as described in this section, marks a clear technological trajectory from task-specific automation toward more autonomous and integrated systems that span the entire software development lifecycle. While current research has made significant progress in various domains such as requirements engineering, code generation, and testing, there remains ample opportunity for further integration and improvement. The following trends represent potential directions that may influence the next generation of SWE Agents, as they evolve from specialized tools toward more comprehensive and capable SWE partners.

From Specialized Agents to Full-Lifecycle Orchestration Current SWE agents often operate in independent stages, optimized for discrete phases of the software lifecycle. For instance, some agents excel at requirements acquisition, while others like Otter focus on test generation [17]. A significant future direction will be the development of integrated, full-lifecycle agentic frameworks. These systems will orchestrate workflows that seamlessly transition from one phase to the next: an agent could take a high-level user need, engage in a multi-agent process to refine requirements, generate the corresponding code and documentation, create robust test suites for validation [1239], and finally, manage deployment and maintenance by analyzing runtime logs [631]. End-to-end frameworks such as ChatDev [811], MetaGPT [401], and AgileCoder [724] have already begun to model the complete software development process, providing a practical blueprint for this vision of cross-lifecycle integration.

Deep Contextual Understanding and Long-Term Memory via Structured Knowledge A persistent challenge for SWE agents is the limited context window of LLMs, which hinders their ability to reason about large, complex codebases. While techniques like repository-aware indexing and dependency graph analysis have provided initial solutions, the future lies in agents that can build and maintain persistent, dynamic knowledge graphs of software projects. Moving beyond the static representations seen in systems like CodexGraph [623], these agents will develop a dynamic “mental model” of a repository, encompassing not only its static structure but also its evolutionary history, runtime behavior, and implicit design principles. For instance, KGCompass [1165] accurately links issue descriptions to code entities via a knowledge graph, while CGM [944] combines graph retrieval with generation. This enables agents to perform complex, repository-level tasks with a level of understanding that may eventually rival, and perhaps exceed, that of human developers.

From Collaboration to Self-Evolution: The Rise of Multi-Agent Ecosystems To address the increasing complexity of software engineering tasks, future research is shifting from single-agent systems to collaborative multi-agent ecosystems, emphasizing role specialization and dynamic interaction to enhance problem-solving capabilities. In these ecosystems, different agents assume specific roles (e.g., planner, developer, tester), mimicking the collaborative dynamics of human teams. Frameworks such as CodeAgent [937] and MAGIS [947] efficiently handle complex, multi-file tasks through clear role division. Looking further, agents will gain the ability to self-evolve. Inspired by the *intelligent layer* of agent design, frameworks like SWE-Exp [163], SWE-Debate [539], and SE-Agent [593] explore mechanisms for learning from historical experience and internal debates, allowing agents to continuously refine their strategies. This evolution from collaboration to self-iteration signals a shift from passive executors to intelligent ecosystems capable of autonomous learning and adaptation.

Synergistic Human-Agent Collaboration: Building Trustworthy AI Pair Programmers While the pursuit of full autonomy is a driving force in agent research, the most practical and impactful future will likely involve synergistic human-agent collaboration. Rather than replacing developers, agents will evolve into proactive, intelligent pair programmers. Frameworks like Aider [949], which emphasize human-AI collaboration over complete autonomy, are at the forefront of this trend. Future systems will feature mixed-initiative interaction models, where the agent handles the laborious and repetitive aspects of coding, testing, and debugging, while the human developer provides high-level strategic direction, resolves ambiguities, and validates critical decisions. Conversational frameworks such as ChatRepair [1112] and AutoDev [988]

create interactive feedback loops that allow agents to iterate under human guidance, promising to enhance developer productivity and creativity without sacrificing control or oversight.

Trust, Security, and Verifiability by Design As SWE agents gain more autonomy and are granted greater access to production systems, ensuring their actions are safe, secure, and verifiable becomes a paramount concern. While the current use of sandboxed environments for code execution is a necessary first step, future work must integrate security and verification as first-class citizens in the agent’s reasoning process. This includes the development of agents that can proactively identify and mitigate security vulnerabilities during code generation, as explored in multi-agent fuzz testing systems [738]. Furthermore, future agents will draw inspiration from hybrid systems like ESBMC-AI [972] and ContractTinker [1012], which combine formal methods like model checking with the generative capabilities of LLMs to provide verifiable correctness guarantees. This “security-by-design” philosophy will be essential for deploying autonomous systems in safety-critical environments.

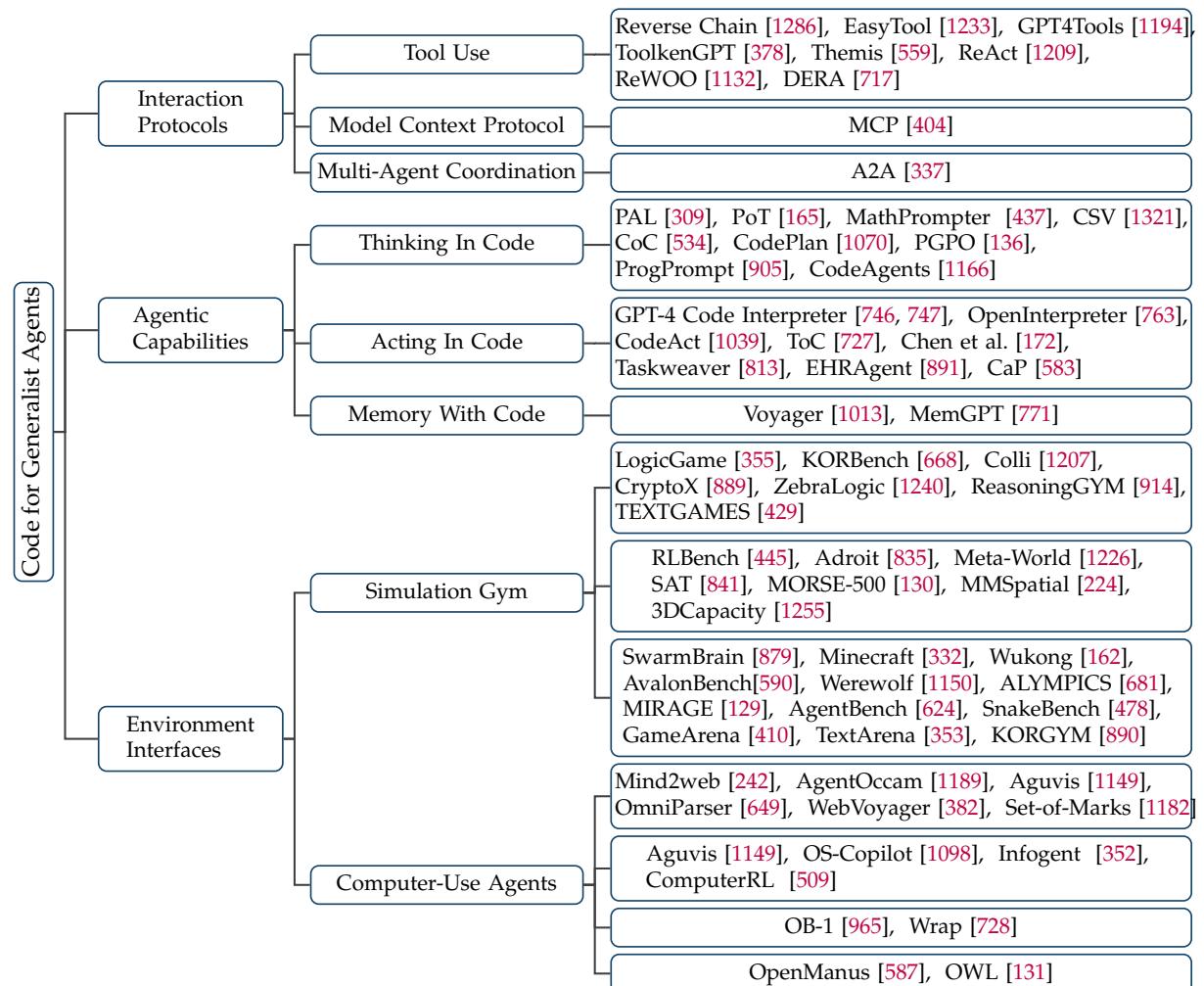


Figure 37. Taxonomy of Code for Generalist Agents.

6. Code for Generalist Agents

Using code as a universal medium allows AI agents to both reason about problems and execute actions across many different tasks and environments, rather than being limited to a single specialized function. The integration of code has become a pivotal paradigm in the development of generalist agents, enabling the combination of cognitive reasoning with executable actions across diverse environments. In [Figure 37](#), this section synthesizes recent progress in utilizing code beyond its conventional role, highlighting three key dimensions in agent architecture:

- **Interaction Protocols:** Code establishes structured communication frameworks, including tool-use patterns (ReAct [1206], ReWOO [1132], DERA [717]), the model context protocol (MCP) [404], and multi-agent coordination schemes (A2A) [337], enabling precise tool invocation, state management, and inter-agent collaboration.
- **Agentic Capabilities:** Code-driven approaches such as CodeAct [1039], Smolagents [431], and Open Interpreter [763], along with emerging CodePlanning techniques, empower agents to generate and execute code for complex logic, data manipulation, and software engineering tasks, thereby enhancing autonomy and operational efficiency.
- **Environment Interfaces:** Code underpins both simulated environments (e.g., CodeGYM [1281] for puzzle and spatial reasoning tasks, GameArena [410] for strategic planning) and real-world interfaces (GUI and terminal-based agents), offering scalable, verifiable platforms for agent training, evaluation, and deployment.

Through these three dimensions, we examine how code contributes to the development of adaptable, tool-augmented agents capable of solving open-ended tasks.

6.1. Code as Interaction Protocols

6.1.1. Tool Use

In LLM-based tool-agent workflows [342, 378, 561, 694, 781, 817, 1210], as shown in [Figure 38](#), each tool is fundamentally embodied in code, which serves as the formal substrate that defines both the syntactic interface and the semantic logic of tool functionality. Function calling operates as the central mechanism in this process, providing LLMs with a structured interface for interaction with executable code or external services. During invocation, the LLM extracts the necessary parameters from user input based on the description of the tool and sends a corresponding request to the tool server. The primary objectives are to accurately extract parameter content and format, ensure the completeness of required parameters, follow the predefined output specifications of the tool, and validate that parameter values remain within the acceptable range. Methods for function calling can be broadly categorized into tuning-free and tuning-based approaches, depending on whether the model parameters are fine-tuned.

Tuning-Free Approaches Tuning-free approaches leverage the in-context learning capabilities of LLMs, typically using few-shot demonstrations or rule-based refinements, to enhance parameter extraction and tool alignment. For example, Reverse Chain [1286] adopts a reverse reasoning strategy that first identifies the appropriate target tool for a given task and then fills in the relevant parameters; when certain parameters are unspecified, auxiliary tools are invoked to supplement them. Similarly, EasyTool [1233] strengthens the understanding of tool functions and parameter requirements by prompting ChatGPT to rewrite lengthy tool descriptions into concise, function-oriented guidelines.

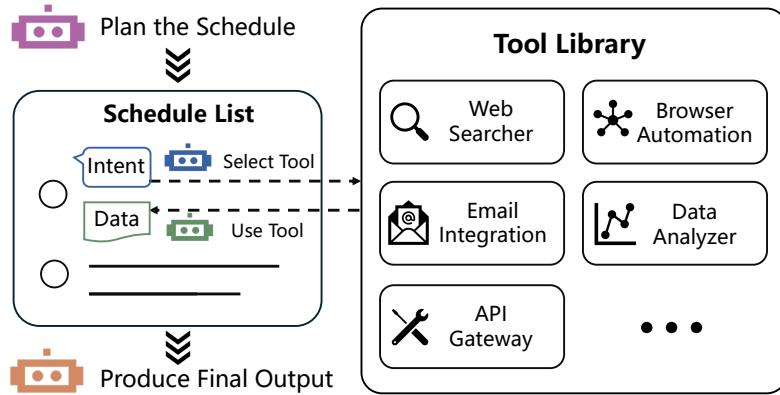


Figure 38. Workflow of LLM-based tool-using.

Tuning-Based Approaches Tuning-based approaches enhance function calling through parameter fine-tuning with dedicated tool-learning datasets. GPT4Tools [1194] integrates tool-use capabilities into open-source LLMs via LoRA-based fine-tuning, leveraging instruction datasets of tool usage automatically generated by ChatGPT. ToolkenGPT [378] introduces specialized tokens, referred to as toolkens, which act as triggers for tool invocation: when a toolken is predicted, the model switches to a specialized mode for generating input parameters and subsequently incorporates tool outputs into the generation process. Themis [559] further improves interpretability in reward modeling by autoregressively integrating reasoning and tool use, dynamically determining which tools to invoke, how to assign parameters, and how to fuse the resulting outputs into ongoing reasoning.

Tool Calling as the Engine of Agent Frameworks Function calling serves as a foundational mechanism in various tool-augmented agent paradigms. The ReAct framework [1209] implements function calls during the action phase, where the model explicitly outputs the tool name and parameters to initiate invocation, followed by an observation phase in which tool responses are integrated to guide subsequent reasoning. ReWOO [1132], by contrast, features a two-stage structure: a planning phase that compiles a structured list of required function calls, and an execution phase in which Worker agents invoke functions, carry out tasks, and return the results. To handle uncertainty and invocation failures, DERA [717] introduces a cyclical interaction protocol (Execute → Pause → Dialog → Resume). When confronted with ambiguous instructions, the model suspends execution, engages in clarifying dialogue with the user, and resumes function calls once sufficient information is acquired, thereby enhancing both robustness and interactivity.

6.1.2. Model Context Protocol

The model context protocol (MCP) [62, 404, 876] is a standardized communication framework designed to coordinate interactions between models and external tools. In contrast to autonomous tool use that depends entirely on internal reasoning, MCP introduces structured message formats, explicit invocation semantics, and well-defined mechanisms for managing context states. Its operational cycle consists of four stages: request, response, state update, and re-invocation. Together, these stages form a closed-loop process that couples tool execution with continuous context management, thereby improving the reliability, interpretability, and scalability of multi-turn task completion.

6.1.3. Multi-Agent Coordination

Agent-to-Agent (A2A) [337, 372, 496, 638, 998] is a collaborative pattern in which agents communicate directly to accomplish complex tasks. Rather than relying on a single agent's autonomous reasoning, A2A decomposes tasks among specialized agents that coordinate tool invocation and information processing through message passing and shared context. Its workflow involves task delegation, information exchange, result integration, and iterative refinement. Agents in this paradigm can invoke external tools and interact with peers to fill knowledge gaps or compensate for limitations, thereby improving both the efficiency and accuracy of problem-solving.

6.2. Code as Agentic Capabilities

AI agents have gained substantial research interest, accompanied by increasing efforts to characterize their underlying capabilities [100, 218, 855]. Yao et al. [1209] propose the ReAct paradigm, which structures an agent's behavior into three stages: thought (reasoning, planning, and decision-making), action (executing decisions), and observation (receiving environmental feedback for subsequent decisions). Building on this, Xi et al. [1107] outline an LLM-based agent architecture composed of three high-level components: brain (central cognitive processing), perception (interpreting environmental signals), and action (performing environment-altering operations). Wang et al. [1023] further refine agent functionality into four modules: profile (representation of self, environment, goals, and constraints), memory (storing and recalling past experiences), planning (generating action sequences to achieve goals), and action (executing planned behaviors). From a neuroscience-inspired perspective, Liu et al. [597] argue that agent capabilities encompass seven fundamental aspects, including cognition (knowledge acquisition and reasoning), memory, world model, reward mechanisms, emotion modeling, perception, and action systems, together forming the core substrate for adaptive and autonomous intelligence.

This section aims to explore the application of code-based LLMs in AI agents. By integrating the classic agent paradigm proposed in ReAct [1209] with the definitions and classifications from related works, this section analyzes the role of code-based LLMs within the agent architecture from three perspectives: thinking, acting, and memory.

6.2.1. Thinking in Code

When tackling complex reasoning tasks, LLMs benefit significantly from code generation techniques, which enhance both precision and efficiency [237]. The concept of reasoning with code initially emerged in the domain of mathematical problem-solving. Gao et al. [309] propose program-aided language models (PAL), which leverage few-shot learning and chain-of-thought (CoT) prompting to generate Python code that executes intermediate reasoning steps involving mathematical operations, symbolic manipulation, and algorithmic logic. This approach enables more accurate outcomes through execution-based verification. Building on this foundation, subsequent studies have explored alternative strategies for code-based mathematical reasoning, including program of thoughts (PoT) [165], which replaces natural language CoT with programmatic reasoning, MathPrompter [437], which optimizes PAL-style prompting through refined mathematical templates, and code-based self-verification (CSV) [1321], which improves reliability by verifying answers via reverse reasoning.

Beyond mathematical problem-solving, Li et al. [534] introduces the chain of code (CoC) reasoning framework, which extends code-based reasoning to both numerical and general semantic tasks. In this approach, LLMs generate and execute code snippets under prompt guidance, facilitating structured problem-solving. For numerical tasks, CoC enables precise computation

through executable code. For semantic reasoning, it leverages pseudocode composed of data structures, conditionals, and loop constructs to model abstract reasoning processes. While high-level control flow is implemented via code logic, certain decision functions still rely on commonsense semantic reasoning. By incorporating variables and structured control logic, CoC substantially improves the efficiency and accuracy of solving complex reasoning tasks.

In addition to supporting single-task reasoning, code generation techniques also play a pivotal role in multi-task planning. Wen et al. [1070] constructed a dataset comprising two million training instances, where models learn to generate code that bridges the input and output, thereby capturing implicit planning trajectories. Compared to traditional LLM training approaches, CodePlan [93] formulates task planning explicitly in code, which reduces ambiguity from natural language instructions and enhances performance in complex multi-hop reasoning tasks. Cao et al. [136] propose Planning-Guided Preference Optimization (PGPO), a framework that integrates pseudocode-based task planning into ReAct. By leveraging specialized reward functions and preference learning, PGPO effectively replaces natural-language-based planning with executable reasoning steps. Furthermore, Singh et al. [905] demonstrates that prompts expressed directly as code improve task planning in both virtual household environments and robotic manipulation settings, highlighting the versatility of code-driven planning in diverse domains.

Compared to single-agent workflows, multi-agent systems offer superior capabilities in collaboration and problem-solving efficiency. Yang et al. [1166] investigates code planning strategies tailored to multi-agent scenarios. In their framework, a dedicated planner generates high-level pseudocode plans in Python style, comprising variable instantiations, conditional logic, and iterative structures. These plans are produced through stepwise decomposition of complex tasks and are annotated with natural language comments to guide decision-making by large language models. The planner then transforms the plan into executable CodeAct instructions, which are passed to either the ToolCaller module [1166] or the Python interpreter for execution. This code-based multi-agent design facilitates the modular reuse of agent components with similar functionalities, thereby enhancing both system scalability and token efficiency.

6.2.2. Acting in Code

When used within an agent’s execution module, code must interface with system-level tools via standardized agent protocols. With the advent of ChatGPT, OpenAI introduced an enhanced variant of GPT-4, known as the GPT-4 Code Interpreter or GPT-4 Code [746, 747], which enables models to execute code for advanced reasoning tasks. Complementing this, Open Interpreter [763] presents an open-source framework that allows language models to execute code locally across multiple languages, including Python, JavaScript, and Shell. It equips function-calling LLMs with an `exec()` interface, which accepts two arguments: the target programming language (e.g., Python or JavaScript) and the corresponding code snippet, thereby enabling grounded execution within the agent workflow.

Wang et al. [1039] introduces the concept of CodeAct, which enables interactive operations by generating executable Python code. Unlike traditional action formats such as text or JSON, CodeAct eliminates the need for custom tool wrappers by directly integrating code execution with the Python interpreter. This approach offers richer support for control flow and data management, allowing intermediate results to be stored as variables and reused across steps, thereby reducing the overall number of execution actions.

Building on this foundation, Ni et al. [727] proposes ToC, an end-to-end framework for

code generation and execution. ToC constructs a hierarchical code tree by considering the global solution structure of a given problem. It further incorporates self-supervised feedback mechanisms that evaluate the success of code execution and leverages answer validation through majority voting. Compared to CodeAct, ToC achieves better performance in accuracy while reducing execution rounds, demonstrating enhanced efficiency and robustness in complex problem-solving scenarios.

In studying the decision-making processes of code-capable agents, Chen et al. [172] examines how LLMs with varying reasoning capabilities decide whether to invoke the Code Interpreter and how this behavior affects task accuracy. Experimental results on mathematical reasoning tasks reveal a counterintuitive trend: less capable models, such as GPT-3.5 [151], are more inclined to use the Code Interpreter and consequently achieve higher accuracy. In contrast, more capable models, such as GPT-4o [749], tend to rely on their internal reasoning abilities and prefer textual reasoning, often exhibiting overconfidence that leads to increased errors on moderately difficult problems—a phenomenon termed the inverse scaling effect. However, when constrained to rely exclusively on either textual or code-based reasoning, neither modality achieves optimal performance. To mitigate this, the authors propose three decision-making strategies to enhance LLM reasoning accuracy. The first encourages consistent use of code-based reasoning to improve reliability. The second introduces a parallel reasoning mechanism, wherein both textual and code-based reasoning are executed independently, and their outputs are aggregated. The third employs confidence-based selection, allowing the model to assess its confidence in each reasoning modality and proceed with the one deemed more reliable. All three strategies yield notable improvements, underscoring the importance of dynamic reasoning modality selection in code-agent systems.

To operationalize the benefits of code-based reasoning, several studies examine concrete use cases where code generation functions as the primary execution mechanism for agents. Qiao et al. [813] proposes a code-first framework tailored for complex data processing tasks, wherein each user request is translated into executable code and user-defined plugins are treated as callable functions. For example, the system conducts anomaly detection on time series data stored in SQL databases by automatically generating the required code. In the healthcare domain, EHRAgent [891] addresses the inefficiencies of conventional workflows, which require clinicians to relay their needs to software engineers for implementation. It enables medical personnel to directly generate executable code via LLMs, facilitating multi-table reasoning over Electronic Health Records (EHR) and supporting a variety of clinical tasks, thereby lowering the technical barrier for healthcare professionals. Similarly, Liang et al. [583] explores the use of LLM-generated code to control robotic systems in embodied intelligence scenarios, allowing agents to convert high-level instructions into precise, executable actions.

6.2.3. Memory With Code

In response to the context length limitations of LLMs, researchers have explored various storage strategies, among which code-based storage has demonstrated notable effectiveness. Wang et al. [1013], Xu et al. [1146] exemplifies this through a framework in which agents autonomously acquire skills by generating and iteratively refining code through interactions with the environment in Minecraft. Validated skills are stored as executable code in a dedicated library and later retrieved directly for reuse, thereby eliminating the need for repeated model inference. Similarly, Packer et al. [771] proposes a virtual memory-inspired context management framework that extends the effective memory of LLMs. This is achieved by designing specialized read and write function calls to dynamically manage both internal (model-resident) and external context,

enabling real-time modification of contextual content during interactions.

6.3. Code as Environment Interfaces

6.3.1. *Code as Simulation Gym*

With the advancement of LLM reasoning capabilities and the increasing diversity of reasoning tasks, evaluating and cultivating long-term planning skills has become a critical challenge. Traditional single-turn reasoning benchmarks, however, are insufficient for assessing or training such capabilities. To address this limitation, Code as Simulation Gym (CodeGYM) has been proposed, building on the Gymnasium [982] reinforcement learning platform. CodeGYM constructs a variety of complex task environments that allow models to engage in continuous interaction, maintain and update state, and receive feedback or rewards. Existing CodeGYM implementations generally fall into three categories: puzzle solving (e.g., logical and mathematical problems requiring step-by-step reasoning), spatial reasoning (e.g., navigation and manipulation tasks in grid-based or continuous environments), and GameArena (e.g., multi-agent or strategy-based game scenarios).

Dynamic PuzzleGYM The Dynamic PuzzleGYM framework collectively refers to a family of rule-based puzzle generation systems designed to automatically construct large-scale, verifiable reasoning datasets [225]. These systems leverage random seeds and initialization parameters to programmatically generate problem instances by combining known conditions, background information, and task formulations, along with their corresponding solutions. Early implementations such as LogicGame [355] and KORBench [668] present diverse logic puzzles that require models to interpret initial states and apply specified rules to derive solutions. More advanced platforms, including Colli [1207], CryptoX [889], and ZebraLogic [1240], employ transformation strategies and constraint mechanisms to produce complex combinatorial reasoning tasks. Similarly, ReasoningGYM [914] and TEXTGAMES [429] offer extensive single-turn game puzzles, with ReasoningGYM further integrating reward interfaces for reinforcement learning. Compared to conventional QA datasets, CodeGYM-generated puzzles exhibit better scalability, lower overlap with pretraining corpora [668, 914], and stronger suitability for the data requirements of modern LLM training [914].

Synthetic SpatialGYM Synthetic SpatialGYM is a synthetic data generation framework designed to create customized datasets for training and evaluating LLMs on spatial reasoning tasks [179], thereby eliminating the high costs associated with real-world image collection. It programmatically generates spatial reasoning datasets that meet specific visual and structural requirements. Early implementations include robotic learning benchmarks (e.g., RL-Bench [445]), real-world manipulation scenarios (e.g., Adroit [835]), and multi-task environments (e.g., Meta-World [1226]). With the advancement of LLMs, specialized CodeGYM-style platforms have emerged to support spatial reasoning: SAT [841] produces static and dynamic 3D question–answer pairs, while MORSE-500 [130] generates spatial datasets using tools like Manim, Matplotlib, and MoviePy. These synthetic benchmarks facilitate the development of spatial perception and reasoning in vision–language models, supporting downstream applications in embodied intelligence, healthcare, and automation [224, 1255].

GameArena GameArena refers to a class of CodeGYM platforms designed to evaluate and train the long-term planning and strategic reasoning capabilities of large language models

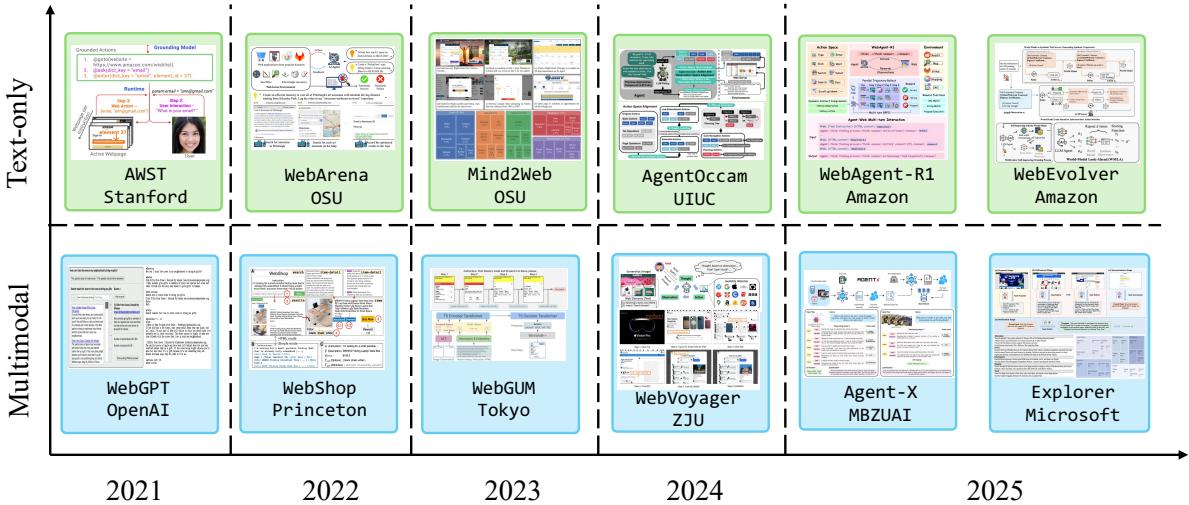


Figure 39. Evolution of GUI agents for website.

through interactive game-based environments. These platforms typically expose observation–action interfaces, enabling models to engage in multi-turn decision-making and receive rewards upon achieving predefined terminal goals. Early systems built on existing commercial games, including StarCraft II [879], Minecraft [332], and Black Myth: Wukong [162], served as interactive testbeds for agent behavior. Subsequent work expanded the paradigm to social deduction games such as Avalon [590] and Werewolf [1150], as well as narrative-driven role-playing and mystery-solving settings [129, 681], thereby emphasizing models’ abilities in strategic interaction and social inference. With continued progress in LLM capabilities, several dedicated GameArena-style benchmarks, such as AgentBench [624], SnakeBench [478], GameArena [410], TextArena [353], and KORGYM [890], have been introduced to provide tailored game scenarios and structured feedback loops, supporting comprehensive evaluation of model planning, adaptation, and collaboration skills.

6.3.2. Computer-Use Agents

As LLMs and vision-language models (VLMs) [87, 89] gain stronger reasoning capabilities, agents are increasingly able to operate autonomously in digital environments. This progress has led to the emergence of computer-use Agents, typically categorized into three types: (1) GUI agents, which interact through graphical interfaces; (2) Terminal agents, which operate via command-line environments; and (3) Cross-environment agents, which integrate multiple modalities and systems to support more complex tasks.

GUI Agents GUI agents interact with their environments by analyzing screen content and generating structured action commands in JSON format or executable code to automate tasks. As shown in Figure 39, the evolution of website agents from 2021 to 2025 spans from early text-only systems to advanced multimodal frameworks. Initial efforts include text-based agents such as Stanford’s AWST [1139], while recent developments feature sophisticated models like Amazon’s WebEvolver (2025). Multimodal agents began with OpenAI’s WebGPT [718] and were subsequently extended by systems such as WebShop [1205] (Princeton), WebGUM [305] (University of Tokyo), Agent-X [305] (MBZUAI), and Explorer [145] (Microsoft). These developments have driven advances in two core modules of GUI agents: perception, which determines how interface information is interpreted, and interaction, which governs how actions are executed.

Perception The perception module plays a critical role in GUI agents by processing both the current screen state and historical context to enable step-wise reasoning. Since GUI environments often include noisy or irrelevant elements, perception strategies focus on extracting task-relevant information to improve decision-making. These strategies can be broadly categorized into text-based, image-based, and multimodal-based approaches, each leveraging different input modalities to support accurate and efficient reasoning.

- **Text-based:** These methods rely on HTML structures or accessibility trees (a11y trees) to represent the interface. To handle the excessive number of web elements, MindAct [242] introduces a two-stage reasoning framework that first uses a lightweight model to filter out irrelevant elements, followed by an LLM to select the final targets. AGENTOCCAM [1189] further simplifies perception by merging functionally descriptive and interactive elements with shared tags. It also reduces the complexity of the historical context by leveraging the tree structure to retain only task-relevant nodes.
- **Image-based:** These approaches focus on interpreting visual content directly, which is particularly useful when textual structures (e.g., HTML or a11y trees) are incomplete or unavailable. Aguvis [1149] improves cross-platform generalization by unifying the action space and collecting diverse GUI screenshots, enabling agents to reason over visual states more effectively. OmniParser [649] further enhances visual grounding by parsing interface images into structured element maps, allowing models like GPT-4V to more accurately align visual regions with actionable targets and interface semantics.
- **Multimodal-based:** Building on the strengths of both text- and image-based inputs, multimodal perception further enhances the agent’s ability to understand complex interfaces. WebVoyager [382] exemplifies this direction with an end-to-end framework that jointly encodes visual and textual information to follow user instructions. By applying the Set-of-Marks technique [1182], it overlays interactive annotations on screenshots, effectively foregrounding key interface elements for downstream reasoning.

Interaction The interaction module determines the agent’s action space, accepts valid action commands, and executes them to manipulate the external environment. To accommodate diverse interaction demands and enhance operational flexibility, two main strategies have emerged: human simulation and tool-based interaction. Human simulation methods imitate user operations, including mouse clicks and keyboard input, to trigger GUI behaviors. For example, Aguvis [1149] generates and executes Python code via the pyautogui library to simulate such interactions. In contrast, tool-based approaches directly invoke APIs or execute scripts to achieve higher efficiency. OS-Copilot [1098] introduces a tool-specific code generator to directly perform environment-level interactions. Infogent [352] expands the agent’s interaction scope by integrating external APIs like Google Search. ComputerRL [509] further proposes an API-GUI hybrid paradigm, where LLMs automatically synthesize API code and corresponding test cases, forming a large-scale, programmatically generated API ecosystem. This framework integrates GUI-level human simulation to balance the efficiency of direct API execution with the flexibility of interface-level control.

Terminal Agents Terminal Agents interact with environments via terminal or shell interfaces, where they translate user instructions into command-line code and execute it to accomplish tasks. OB-1 [965] builds an agent ensemble equipped with shared memory, feedback, and incentive mechanisms. It uses persistent memory blocks to store tasks, notes, and to-do items over extended periods, supporting editable histories for updating prior reasoning and annotations. An adaptive command timeout mechanism balances speed and reliability by distinguishing

between fast shell checks and time-intensive operations such as kernel compilation, enabling OB-1 to handle long-running, error-prone tasks effectively. Similarly, Wrap [728] maintains an editable to-do list, dynamically updating it as tasks progress or when deviations arise due to unforeseen challenges or newly acquired information. To enhance robustness, it incorporates a fallback mechanism that retries failed requests—caused by issues like service interruptions, rate limits, or incorrect tool invocation—using alternative models.

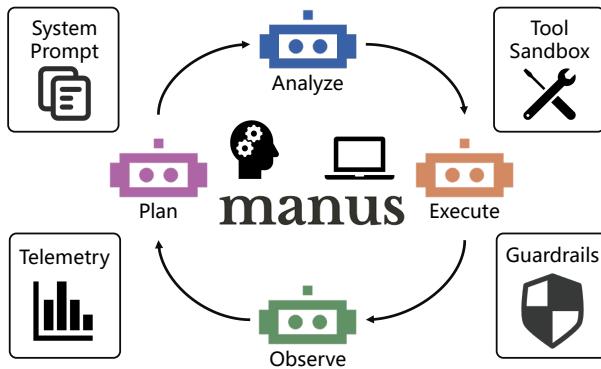


Figure 40. Overview of the system architecture of Manus.

Cross-Environments Agents Cross-environment agents integrate with various systems via the code-act mechanism, enabling interaction across browsers, terminals, code executors, and file systems, thereby broadening their capabilities. Manus [959] uses a browser framework and APIs to retrieve online data, executing code in a sandboxed VM to generate HTML-based responses, as illustrated in Figure 40. OpenAI Deep Research [960] searches and interprets large-scale web content, including text, images, and PDFs, to generate comprehensive reports. OpenManus [587] combines LLM-driven planning with tool execution in a multi-agent system. A planning agent decomposes tasks and tools like GoogleSearch, BrowserUse, and PythonExecute are invoked step-by-step using the ReAct [1206] paradigm with error reflection. OWL [131] builds a role-based multi-agent framework that integrates specialized agents, supports multiple search engines and file formats, and enhances performance through multimodal and text-based toolkits.

7. Safety of Code LLMs

Code LLMs are revolutionizing software development by augmenting human creativity and efficiency. However, as the coding abilities continue to advance, the security of their generated code has increasingly become a matter of public concern [491, 1203]. This emerging field of Code LLM safety encompasses multiple dimensions of security challenges, from inherent vulnerabilities in pre-training data to sophisticated adversarial attacks during deployment. Empirical studies identify, assess, and stress-test these risks at scale (e.g., CodeQL [322], CodeSecEval [789], and HumanEval [161]) and systematic evaluations reveal that open-source models such as Qwen3 [1163] and DeepSeek-R1 [237] frequently output insecure solutions [363, 628, 1163]. Moreover, advanced adversarial prompting can induce functionally correct yet subtly vulnerable code beyond conventional prompt engineering [1083]. The security landscape of Code LLMs presents unique challenges distinct from traditional software vulnerabilities, as these models can propagate and amplify insecure coding patterns learned from their training data across countless applications. As shown in Figure 41, in this section, we provide a comprehensive survey of code

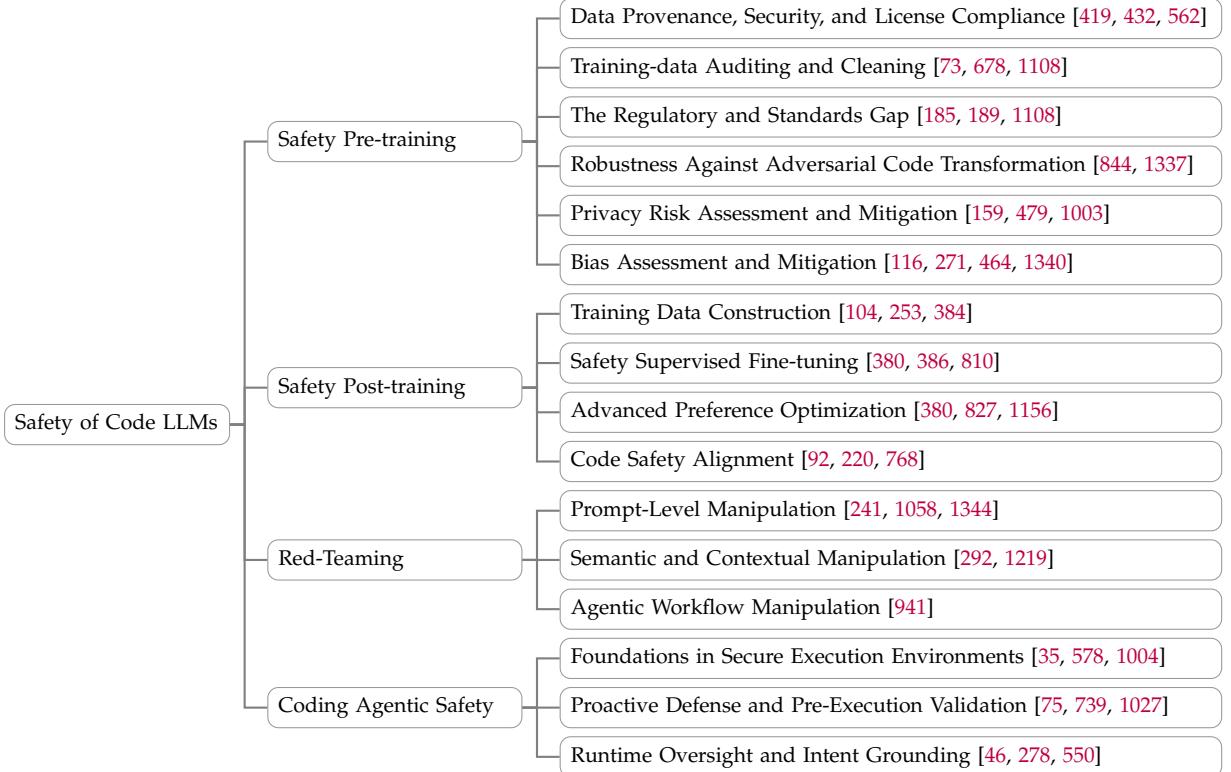


Figure 41. A taxonomy of key dimensions for ensuring the safety of code LLMs. The framework covers four main areas: safety pre-training, safety post-training, red-teaming, and coding agentic safety.

LLM safety challenges and mitigation strategies, organizing our discussion around four critical aspects: (1) Safety pre-training; (2) Safety post-training; (3) Red-team evaluation; and (4) Coding agentic safety. Each dimension presents distinct security considerations and requires tailored approaches to ensure the safe deployment of Code LLMs in real-world applications.

7.1. Safety Pre-training for Code LLMs

The pre-training phase establishes the foundational security characteristics of code LLMs, making it a critical stage for embedding safety considerations into model behavior. At root, the foundation of model insecurity lies in pretraining data [702, 1020]. Code LLMs are trained on public repositories where insecure code patterns are common. Consequently, they are most likely to output the code that is functionally correct but insecure (path of least resistance). Unlike natural language models that primarily learn linguistic patterns, Code LLMs must navigate the complex landscape of programming languages, APIs, and security vulnerabilities present in their training corpora. These models learn by imitation from large-scale, largely uncurated public code repositories (e.g., GitHub), absorbing both functional idioms and prevalent vulnerability patterns that have accumulated over decades of software development. The scale of this challenge is substantial—modern Code LLMs are trained on terabytes of code spanning hundreds of programming languages and millions of projects, each potentially containing security flaws that the model may learn to replicate. Consequently, they often emit code that is functionally correct yet insecure, a behavior readily elicited by both direct and indirect prompting [322, 363, 628, 789, 1020, 1163]. Furthermore, advanced adversarial prompts such as DeceptPrompt further bypass standard guardrails and surface subtle flaws [686, 1083], inducing

risks to Code LLMs and the related code generation systems. Recent empirical evidence has extensively documented significant security vulnerabilities in Code LLMs. Standardized benchmarks and stress tests consistently reveal that these models frequently produce insecure code generations [322, 789, 1020, 1227]. These vulnerabilities are not isolated incidents but rather indicate systemic weaknesses rooted in the pre-training process, as demonstrated in studies of prominent open-source Code LLMs like Qwen3 and DeepSeek-R1, which repeatedly offer insecure solutions across a range of tasks [363, 628, 1163]. The pre-training safety challenge is further compounded by the threat of data poisoning and targeted attacks; for example, adversarial prompting techniques like DeceptPrompt can induce a model to generate code that is functionally correct but contains subtle, exploitable vulnerabilities, effectively bypassing conventional prompt filtering mechanisms [1083]. Understanding and addressing these pre-training safety concerns is essential for developing Code LLMs that can be trusted in production environments.

Basically, current safety pre-training incorporates interventions across two primary dimensions, the data pipeline and the objective/learning signal, thereby enabling base models to inherently embed safe coding practices from initialization [497, 678]. This dual-axis approach represents a paradigm shift from traditional safety measures, recognizing that security considerations must be woven into the fabric of model training rather than applied as an afterthought. The data pipeline dimension focuses on curating and filtering training corpora to minimize exposure to vulnerable code patterns, while the objective/learning signal dimension modifies the training process itself to prioritize secure coding practices alongside functional correctness. Drawing from existing literature, security should be established as a primary objective during the pre-training stage of Code LLMs [702, 1020]. This proactive approach acknowledges that once models internalize insecure patterns during pre-training, subsequent alignment efforts face an uphill battle against deeply embedded behaviors. Concretely, objective-level measures include training with synthetic refusal exemplars and adversarial augmentation, while the data axis spans license-compliant sourcing, corpus auditing/cleaning, privacy protection, bias assessment, and robustness to code-specific adversaries. On the objective side, synthetic refusal exemplars teach models to recognize and decline requests that could lead to security vulnerabilities, effectively building in a security-aware decision-making process from the ground up. Adversarial augmentation exposes models to carefully crafted malicious inputs during training, enhancing their ability to detect and resist exploitation attempts in deployment. Meanwhile, the data pipeline interventions form a comprehensive defense system: license-compliant sourcing ensures legal and ethical use of training data while potentially filtering out code from sources known for poor security practices; corpus auditing and cleaning systematically identify and remove code samples containing known vulnerabilities or dangerous patterns; privacy protection mechanisms prevent the model from memorizing and reproducing sensitive information such as API keys or credentials; bias assessment ensures fair and non-discriminatory code generation across different contexts and user groups; and robustness measures specifically target code-domain adversaries who might exploit syntactic peculiarities unique to programming languages. Together, these multifaceted interventions create a robust foundation for secure code generation that addresses both the quality of training data and the learning dynamics of the model itself. While academia and industry have made substantial progress and proposed valuable work on the security of foundation Code LLM, the domain is still beset by numerous unresolved problems. In this work, we aim to summarize and discuss the following aspects:

7.1.1. Data Provenance, Security, and License Compliance

The safety and trustworthiness of Code LLMs are fundamentally rooted in the security and legitimacy of their pre-training corpora. As these models are trained on datasets reaching

the trillion-token scale, establishing robust data governance frameworks becomes a critical prerequisite. A primary concern is mitigating legal and ethical risks, particularly the inadvertent inclusion of code governed by restrictive or copyleft licenses, which could lead to widespread license violations in model-generated code [419]. To address this, pioneering efforts like the Big-Code project have developed systematic approaches for data curation. Their resulting dataset, The Stack, was meticulously filtered down to 6.4TB of permissively licensed code from a raw corpus of over 102TB, leveraging an automated pipeline for license detection and filtering [432]. This initiative set a precedent for ethical data sourcing in code generation. The subsequent development of the StarCoder models further refined this approach by implementing a comprehensive project-level data governance strategy, which included mechanisms for attribute-based access control and clear documentation of data provenance [562]. Despite these advances, significant technical challenges persist. Automated license detection tools, while effective at scale, are inherently imperfect and can suffer from both false positives (incorrectly excluding compliant code) and false negatives (failing to filter out non-permissive code). Furthermore, the complex issue of code de-duplication (identifying and removing functionally identical or near-identical code snippets that may exist under different licenses) remains an open research problem [41]. These challenges necessitate a dynamic approach to data management, requiring continuous auditing and vigilant monitoring of data sources, especially during periodic data refresh cycles, to uphold the legal and ethical integrity of the training corpus.

7.1.2. *Training-data Auditing and Cleaning*

After ensuring data source security, the safety of corpora employed in pre-training is equally paramount, which is the “first-line” of defense against model-inherent risks and prompt-poisoning [73, 678]. Several methods have been proposed to filter out the harmful contents and ensure the pre-training corpora safety of Code LLMs. For instance, one common method is to use heuristic filters, which employ pattern rules to detect known exploits and unsafe APIs. Another approach involves deploying high-capacity classifiers that function as security and abuse detectors at a large scale. Additionally, synthetic refusal exemplars are utilized to convert dangerous queries into safe denials, which in turn helps to steer the model’s behavior. From an operational perspective, it is also crucial to balance false positives and negatives and to maintain a small human spot-check rate in order to calibrate precision and recall under data drift [73]. A further critical factor contributing to pre-training insecurity is the profound lack of common standards for data security and hygiene within the AI ecosystem. This stands in stark contrast to the maturing field of software supply chain security, where concepts like the Software Bill of Materials (SBOM) and Cyber Resilience Act (CRA) are becoming indispensable for transparency and risk management [185, 1108]. For Code LLM training corpora, no analogous, widely adopted standard exists. Consequently, there is no systematic requirement for dataset creators to document or screen for critical security attributes, such as the prevalence of code snippets containing known vulnerabilities, e.g., Common Weakness Enumeration (CWE), embedded malicious payloads, or exposed secrets. This regulatory and standards gap forces downstream model developers to either implicitly trust the opaque curation process of data providers or implement their own costly and often inconsistent validation protocols. This systemic failure to standardize data-level security verification constitutes a fundamental vulnerability in the AI supply chain, creating a direct pathway for security flaws to be deeply embedded within the foundational models themselves.

7.1.3. The Regulatory and Standards in Data Security

A fundamental factor contributing to pre-training insecurity of Code LLM is the profound vacuum of both common standards and enforceable regulations for data security and hygiene within the AI ecosystem. The lack of technical standards is readily apparent when contrasted with the maturing field of software supply chain security, where concepts like the Software Bill of Materials (SBOM) and Cyber Resilience Act (CRA) are becoming indispensable for transparency and risk management [185, 1108]. For Code LLM training corpora, no analogous, widely adopted standard exists to systematically screen for critical security attributes like known vulnerabilities (e.g., Common Weakness Enumeration (CWE) [189], embedded malicious payloads, or exposed secrets. This technical standards gap, however, is largely a symptom of a deeper and more critical issue: the absence of a specific, legally-binding international regulatory framework for the security of AI-generated code. As our analysis indicates, as of 2025, no direct regulations govern this domain. While broad AI governance frameworks like the EU AI Act and the US Executive Order on AI establish high-level, risk-based principles, they lack the granular, executable guidance necessary for developers to ensure the security of code generation systems. This regulatory ambiguity forces downstream developers into a reactive posture, relying on costly and inconsistent ad-hoc validation protocols. More importantly, it disincentivizes the cross-industry investment required to build the very security benchmarks and standardized tools that are desperately needed. This systemic failure to establish a clear regulatory foundation thus acts as a primary impediment, directly hindering the safe, reliable, and trustworthy development of Code LLMs.

7.1.4. Robustness Against Adversarial Code Transformations

A critical dimension of Code LLM safety is robustness against adversarial attacks that specifically leverage the unique properties of source code. Unlike natural language, the formal syntax and structure of programming languages create a distinct and potent attack surface: semantics-preserving syntactic transformations. Research has demonstrated that simple malicious prompts rejected by natural-language safety filters can be successfully disguised by applying code-specific obfuscations, rendering keyword-based defenses and simple classifiers ineffective [844, 1199, 1337, 1338]. These transformations exploit the one-to-many relationship between a program's semantic intent and its syntactic representation. Attack vectors in this category are diverse, ranging from simple identifier obfuscation and string literal encoding (e.g., using Base64 or Hex) to more complex structural manipulations like control-flow flattening, opaque predicate insertion, and the injection of polymorphic or metamorphic code snippets. The core challenge is that these transformed inputs are functionally identical to their malicious, unobfuscated counterparts, yet appear vastly different at the token level. The primary defense paradigm emerging to address this vulnerability is adversarial training through data augmentation. This technique involves enriching the training dataset with adversarially generated examples—malicious code snippets that have been automatically obfuscated. The objective is to force the model to learn deeper, more abstract representations of code functionality that are invariant to syntactic variations. However, the success of this approach is contingent upon solving the fidelity-diversity dilemma in the augmentation generation process, where the fidelity-diversity dilemma refers to the conflict where increasing the variety and complexity of generated data augmentations (diversity) heightens the risk of corrupting the original sample's meaning and functional correctness (fidelity). This scenario gives rise to a pressing demand: developing scalable, diverse, and provably correct code transformation engines for adversarial training. Failure to balance these factors can lead to significant distributional drift, where the model's performance on benign, real-world code degrades, or it fails to generalize its robustness

to novel adversarial strategies, a challenge extensively discussed in recent work [844, 1199].

7.1.5. Privacy Risk Assessment and Mitigation in Pre-training Data

The pre-training corpora for Code LLMs, often scraped from public repositories like GitHub, present significant privacy challenges. Although seemingly public, this source code can inadvertently contain a vast amount of sensitive information, including Personally Identifiable Information (PII) such as developer names and emails in comments, hardcoded credentials like API keys and passwords, and proprietary algorithms [159]. Addressing these risks before training is a critical step in the safety pipeline for Code LLMs.

A foundational and pragmatic approach to mitigate PII leakage involves a multi-step data sanitization pipeline: (1) curate a high-quality, PII-labeled subset of the code data to act as a ground truth; (2) train a dedicated high-recall detector for PII and other structured secrets (e.g., API keys), optimizing for strong F1 scores on these specific classes; and (3) systematically scan the entire pre-training corpus and mask or redact any detected sensitive information. This “detect-and-mask” strategy is a crucial first line of defense. However, its efficacy is contingent on the detector’s comprehensiveness, as novel or complex secret formats may evade detection.

Beyond direct PII removal, research has shown that data duplication is a major catalyst for privacy risks. Models are significantly more likely to memorize and regenerate data snippets that appear multiple times in the training set. Therefore, a critical and highly effective mitigation strategy is the aggressive deduplication of the training corpus. As demonstrated, this single preprocessing step can substantially reduce the likelihood of the model memorizing and leaking specific, unique sequences from the training data, thereby mitigating a significant portion of privacy risks [479]. Combining data deduplication with PII sanitization creates a much more robust defense against inadvertent memorization.

Despite these preprocessing efforts, risks remain, as sensitive information is not just stored explicitly but can also be encoded implicitly within the model’s embeddings [1003]. This necessitates a discussion of in-training protection mechanisms and risk assessment. After data sanitization, it remains imperative to assess residual risks through rigorous evaluation, for instance, by using held-out probes to measure memorization and re-identification potential. Membership inference attacks and targeted extraction queries are common methods to audit the privacy posture of a trained model.

It is also important to consider the broader context in which these models are deployed. While our focus is pre-training, the privacy vulnerabilities can manifest during inference. For instance, side-channel attacks targeting the KV-cache can leak information about previous user interactions, posing a downstream risk that is indirectly influenced by the model’s training [660]. Furthermore, emerging architectures like RAG shift the privacy burden from the model’s parameters to the external knowledge base, introducing new challenges in ensuring the privacy of the retrieval corpus and the decoding process itself [425, 1016].

In summary, while a pipeline of data sanitization and deduplication provides a strong foundation for privacy protection in the pre-training of Code LLMs, it is not a complete solution. The primary limitations include the impossibility of perfect PII detection, the inherent trade-off between aggressive data cleaning and model utility, and the fundamental nature of information leakage through model embeddings. A holistic approach to privacy must therefore integrate robust data preprocessing with rigorous post-training audits and an awareness of downstream vulnerabilities in deployment architectures. A comprehensive framework for envisioning and mitigating these multifaceted risks across the entire product lifecycle is essential for building

truly safe and trustworthy systems [520].

7.1.6. Bias Assessment and Mitigation

The safety of code generation extends beyond security vulnerabilities to encompass fairness and the mitigation of social biases. Biases in Code LLMs can manifest subtly within generated code, such as in variable names, comments, and even algorithmic logic, thereby perpetuating harmful societal stereotypes [116]. The primary methodology for evaluating such biases involves the use of specialized test harnesses. These frameworks systematically generate prompts embedded with sensitive demographic attributes (e.g., gender, race, or religion) and analyze the model's outputs to detect prejudiced associations [271, 464, 1340]. For instance, the FairCoder benchmark [271] provides a structured set of scenarios to probe for these biases in code-related tasks. Rather than relying on simple keyword matching, a more robust analysis technique involves parsing the Abstract Syntax Trees (ASTs) of the generated code, which allows for a deeper, semantic understanding of potential biases embedded in the code's structure and identifiers [161, 857].

To quantitatively measure the extent of bias, several metrics have been proposed and are actively used in the literature. These include:

- **Core Bias Score (CBS):** This foundational metric, introduced by Liu et al. [627], measures the prevalence of biased outputs by calculating the frequency with which a model generates code containing stereotypes when given demographically sensitive prompts.
- **CBS_U@K:** To evaluate the stability of non-biased responses, the Consistent Unbiased Score across K runs measures the consistency of a model in providing unbiased outputs over multiple generations for the same prompt [596].
- **CBS_I@K:** Conversely, the Consistent Biased Score across K runs assesses the stability of biased outputs, which is crucial for understanding the robustness of a model's stereotypical associations [464]. This metric is also particularly relevant when investigating biases within LLM-based evaluators themselves, a phenomenon where the judge model may favor outputs that align with its own internal biases [596].

Current research has consistently demonstrated the existence of significant social biases across various Code LLMs, including those from major providers [271, 627, 1340]. A critical and recurring finding is the stark divergence between conversational safety and code-generation safety [419, 857]. This indicates that safety fine-tuning performed on general conversational data does not reliably transfer to the domain of code generation. Models that are aligned to refuse inappropriate requests in natural language can still produce code that reflects deep-seated societal biases. This highlights a significant limitation in current alignment approaches, suggesting that domain-specific safety protocols are essential for pre-training and fine-tuning Code LLMs. While assessment methodologies are becoming more sophisticated, research into effective mitigation remains an emerging and challenging area. Initial efforts have explored techniques like model editing to directly modify model parameters and erase specific gender-based associations, but developing scalable and robust mitigation strategies that do not compromise the model's primary coding capabilities is an open problem [688].

In a nutshell, Code LLM are insecure by default. Their training on public corpora, where flawed code is pervasive, predisposes them to follow the path of least resistance, resulting in code that works but is unsafe. Safety thus becomes an external goal that competes with the model's primary objective of imitation, weakening any security assurances. This inherent conflict explains why more powerful models often prove more adept at reproducing insecure

Table 20. Security-oriented feature matrix for post-training methods (✓ present, ✗ absent).
 Legend: ✓ present/typical; ✗ not primary or uncommon. H labels: heavy human labeling; AI judge: scalable AI feedback; Tools verif.: SAST/tests/compiler or analyzers as verifiable signals; Token-level: supervision at token/diff granularity; Struct-aware: AST/CFG/DFG or rule-based structure; Guardrail (infer): acts at inference/runtime; Over-refusal: notable risk of over-refusal; CI/CD: fits easily into CI/CD pipelines.

Method	H labels	AI judge	Tools verif	Token-level	Struct-aware	Guardrail(infer)	Over-refusal	CI/CD
<i>SFT / PEFT</i>								
SFT: vuln-fix	✓	✗	✗	✗	✗	✗	✗	✗
SFT: safety inst.	✓	✗	✗	✗	✗	✗	✗	✗
SFT: tools-in-loop	✗	✗	✓	✗	✓	✗	✗	✓
PEFT (Safe LoRA)	✗	✗	✗	✗	✗	✗	✗	✗
<i>Preference learning</i>								
Preference learning (DPO/IPO/KTO)	✓	✗	✗	✗	✗	✗	✗	✗
Localized preference (token-level)	✗	✓	✓	✓	✗	✗	✗	✗
Structure-aware preference	✗	✗	✓	✗	✓	✗	✗	✗
<i>RL-based</i>								
RLHF (human)	✓	✗	✗	✗	✗	✗	✓	✗
RLAIF (AI judge)	✗	✓	✗	✗	✗	✗	✓	✗
Constrained/Safe-RLHF	✗	✓	✓	✗	✓	✗	✓	✗
GRPO / S-GRPO	✗	✗	✓	✗	✓	✗	✗	✓
<i>Data / guardrail / ops</i>								
Safety Editor Policy (runtime/edit)	✗	✗	✓	✗	✓	✓	✗	✓
SEAL (bilevel data selection)	✗	✗	✓	✗	✓	✗	✗	✗
ProSec (synthetic prefs)	✗	✓	✓	✗	✓	✗	✗	✗
SAST feedback loop	✗	✗	✓	✗	✓	✗	✗	✓

patterns. This reality presents clear imperatives: safety must be integrated at the source through data-level safeguards and objective-level priors in pre-training, while a broader adoption of secure-by-design architectures is crucial. To ensure genuine progress, these efforts must be supported by standardized evaluation suites and principled defenses against adaptive poisoning to effectively validate and harden the models.

7.2. Safety Post-training for Code LLMs

7.2.1. Pre-training Limitations and the Necessity of Post-training Alignment

The predominant pre-training objective of Code LLMs does not align well with generating secure code. This paradigm incentivizes models to learn and replicate common coding patterns from vast, uncurated datasets, regardless of their security implications. Consequently, a persistent knowledge–behavior disconnect emerges, where models may possess theoretical knowledge of security concepts but fail to apply them in practice without explicit guidance [428, 1025]. This issue is exacerbated by the nature of pre-training corpora, which are replete with security vulnerabilities. For instance, studies have shown that a single path traversal vulnerability pattern can be found in over 1,700 projects, exposing models to insecure examples on a massive scale [29]. Unsurprisingly, evaluations of LLM-generated code consistently reveal high vulnerability rates, with some studies reporting that approximately 40% of generated programs contain security flaws [783].

Beyond replicating existing vulnerabilities, Code LLMs introduce novel risks through hallucinations, where models invent non-existent APIs or misuse legitimate ones in syntactically plausible but semantically incorrect ways, leading to unpredictable and often high risk security loopholes [9]. Analysis of these generated vulnerabilities indicates that they often fall into systemic categories that have plagued software for decades, such as improper authentication,

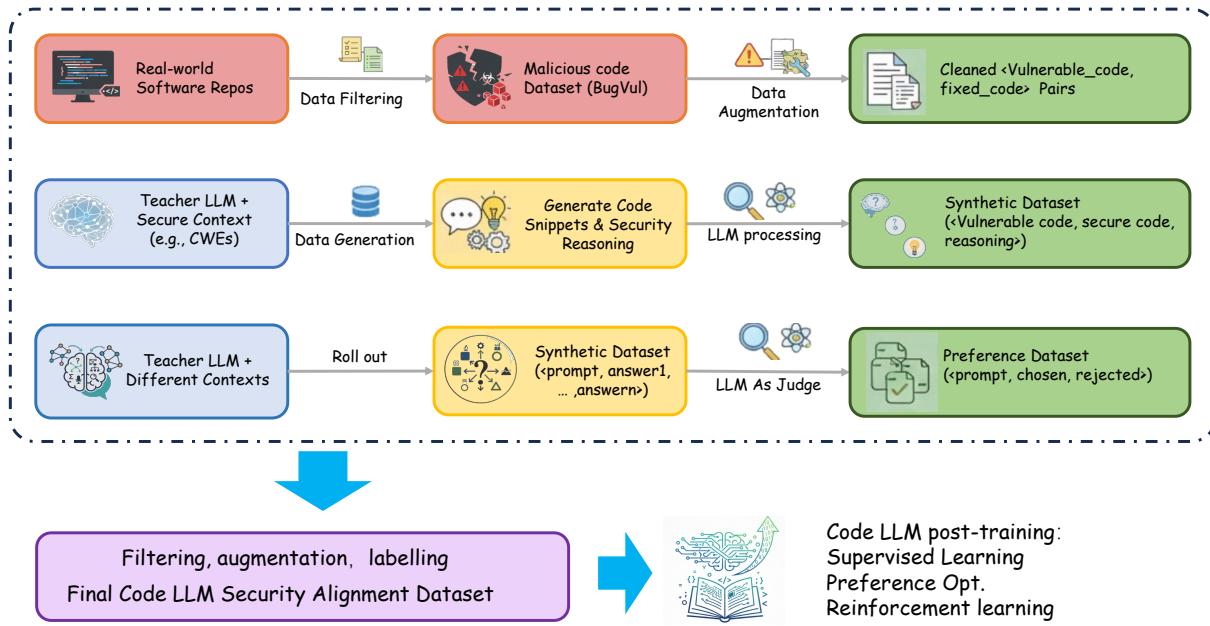


Figure 42. Data Generation Pipeline for Code LLM Security Alignment.

session management, and inadequate input validation [259]. These inherent limitations of the pretraining phase underscore the inadequacy of relying solely on likelihood-based learning for security critical applications.

Therefore, post-training alignment mechanisms, like Supervised Fine-Tuning (SFT) and Reinforcement Learning from Human Feedback (RLHF), have become critical and necessary steps to explicitly instill security principles. Exploratory studies demonstrate that fine-tuning on curated datasets of vulnerabilities and their corresponding security patches can significantly reduce the generation of insecure code [449]. To further enhance robustness, more advanced techniques are being developed. These include proactive security alignment, which aims to fortify models against potential threats before they are explicitly encountered [887], and adversarial testing, which systematically probes models for security weaknesses to harden them against attacks [1100]. Moreover, principles from the broader field of LLM safety are being adapted for code generation, such as general safety-aware fine tuning methodologies [920] and prioritizing high-quality, security vetted data during the alignment process [1308]. Collectively, these efforts highlight a consensus in the field, the path to secure Code LLMs is not through bigger models or more data alone, but through targeted, security-conscious posttraining alignment.

7.2.2. Data as the Cornerstone: Constructing Safety-related Training Datasets

The foundation of any robust, security-aligned Code LLM is the data from which it learns. In Figure 42, the prevailing methodologies for constructing security centric datasets can be broadly categorized into three paradigms, leveraging real-world vulnerability data, synthesizing targeted training examples, and distilling preference pairs for advanced alignment techniques.

Mining real world vulnerability fix pairs from software repositories is a primary strategy. By linking commits to Common Vulnerabilities and Exposures (CVEs), researchers have created canonical datasets such as CVEfixes [104] and the more extensive BigVul [384]. These resources provide authentic examples of how vulnerabilities are introduced and subsequently patched.

However, a significant challenge with such datasets is data quality [253]. They often suffer from high redundancy, noisy labels, and incorrect mappings between vulnerabilities and fixes, which can inflate performance metrics and mislead training. Consequently, rigorous data cleaning, validation, and de-duplication are not merely best practices but mandatory steps for meaningful model alignment.

To overcome the sparsity and noise inherent in real world data, synthetic data generation has emerged as a powerful alternative. This approach utilizes a proficient “teacher” LLM to create vast quantities of high quality, targeted training instances. For instance, methodologies like HexaCoder employ an oracle-guided pipeline to generate pairs of vulnerable and secure code snippets, guided by specific Common Weakness Enumeration (CWE) types [376]. Similarly, ProSec fortifies models through proactive security alignment, systematically synthesizing vulnerability-inducing scenarios to generate a large scale, security focused alignment dataset [1148]. A crucial innovation in this domain is the incorporation of security reasoning within the synthetic data [463, 490]. By including chain-of-thought explanations for why a piece of code is insecure and how the fix resolves the issue, these datasets encourage the model to move beyond simple pattern matching towards a deeper, causal understanding of security principles. This reasoning centric approach is vital for developing models that can generalize to novel threats.

Finally, as alignment techniques evolve beyond supervised fine-tuning, the distillation of preference data has become critical. This involves constructing triplets of the form $\langle \text{prompt}, \text{chosen_secure_response}, \text{rejected_insecure_response} \rangle$. A teacher model, often a frontier LLM, is prompted to generate multiple code variations, from which secure and insecure examples are selected. This process creates a dataset tailored for preference optimization algorithms like Direct Preference Optimization (DPO) [827], enabling the model to learn fine-grained distinctions between safe and unsafe code [380].

7.2.3. Safety Supervised Fine-Tuning for Code LLMs

SFT is a basic step in adapting a pre-trained LLM for specialized tasks, including secure code generation. The most straightforward approach is **content-based SFT**, where the model is fine-tuned on pairs of (*vulnerable_code, fixed_code*) [380]. While effective at teaching the model specific vulnerability patterns, its success is often limited when security-related edits are sparse within the broader codebase.

To enhance generalization and task-specific performance, **instruction-based SFT** has proven to be highly effective. This technique involves training the model on prompts that explicitly describe a security task, such as “*Find and fix the SQL injection vulnerability in the following function*” [386]. By framing security tasks as instructions, this method improves the model’s ability to transfer learned knowledge to new contexts and mitigates the risk of catastrophic forgetting of its general coding abilities. However, it is crucial to recognize that narrow fine-tuning can sometimes lead to emergent misalignment, where a model specialized for one task develops unintended, harmful behaviors in other domains [810]. Techniques like Safe LoRA offer a promising direction by reducing safety risks during parameter-efficient fine-tuning without compromising utility [1256].

A more dynamic and robust approach is **feedback-based SFT**, which integrates external tools into the training loop. This tool-in-the-loop paradigm leverages Static Application Security Testing (SAST) tools, unit tests, or formal verification methods to provide structured, automated feedback on the code generated by the LLM [177, 1019]. This feedback serves as a validation signal, guiding the model through iterative repair processes and supplying a continuous stream

of verified training examples. Frameworks like INDICT [514] further refine this by creating an internal dialog between safety and helpfulness critics, providing both preemptive and post-hoc guidance to enhance the quality and security of the generated code. Other research explores how in-context learning with security patterns can bolster security without extensive retraining [702] and how to keep models updated on newly discovered vulnerabilities in APIs [90].

7.2.4. Advanced Preference Optimization for Localized Flaws

A key insight in securing Code LLMs is that security vulnerabilities are often highly localized, hinging on a few critical tokens or lines of code. Standard preference optimization methods like DPO [827], which apply a global preference signal across the entire code sequence, can be an inefficient and blunt instrument for correcting such localized flaws.

To address this, Localized Preference Optimization (LPO) was introduced as a more targeted alignment strategy [380]. LPO refines the preference learning process by focusing the loss function specifically on the pivotal tokens that differentiate a secure code variant from an insecure one. By masking irrelevant tokens, it directs the model’s attention to the precise source of the vulnerability. To prevent the degradation of the model’s general coding capabilities, LPO incorporates a SFT regularizer, ensuring that the model retains its overall performance. This token-targeted approach has demonstrated significant empirical success, reducing security issues by 19–40% while simultaneously improving general code quality by 3–10

The evolution of SFT techniques for security can thus be viewed as a progression in precision: from traditional SFT, which provides explicit examples; to DPO, which learns from global sequence preferences; and finally to LPO, which masters token-targeted preference. A natural and compelling next step in this trajectory is to move beyond lexical targets toward alignment at the level of semantic structures, such as Abstract Syntax Trees (ASTs), to instill an even deeper structural understanding of code security [1156].

7.2.5. Coding Safety Alignment via Reinforcement Learning

RL offers a powerful paradigm for dynamically aligning Code LLMs with complex safety objectives, moving beyond the static knowledge encoded during supervised fine-tuning. By framing secure code generation as a sequential decision-making problem, RL enables the model to learn from the consequences of its outputs, optimizing its policy based on a reward signal that reflects security, functionality, and correctness.

Policy Optimization with Human and AI Feedback. The predominant approach for this alignment is Reinforcement Learning from Human Feedback, a three-stage process involving: (1) training a SFT policy, (2) learning a reward model from human preference data comparing paired outputs, and (3) optimizing the SFT policy using an RL algorithm like Proximal Policy Optimization (PPO) to maximize the learned reward [768]. This framework directly steers the model towards generating outputs that human reviewers deem safer and more helpful.

To address the scalability limitations of human annotation, Reinforcement Learning from AI Feedback (RLAIF) has emerged as a compelling alternative [92]. RLAIF replaces or augments human preference labels with feedback from a capable “teacher” LLM. This process can be guided by a predefined set of principles or a “constitution”, allowing for scalable and consistent safety alignment. More advanced frameworks, such as Safe RLHF, further refine this by decoupling the optimization process. Instead of a single reward, they train separate models for helpfulness (reward) and harmlessness (cost), using constrained optimization algorithms

to maximize helpfulness while ensuring that the cost remains below a specified threshold [220, 604]. This explicitly manages the trade-off between utility and safety, preventing the model from sacrificing security for performance.

Constructing Robust Reward Signals for Code Generation. The efficacy of RL-based alignment hinges on the design of the reward signal. A well-crafted reward function must capture a nuanced understanding of code quality, integrating diverse feedback sources to prevent policy exploitation. These sources can be categorized as follows:

- **Verifiable Feedback from Tooling:** This is the most objective form of feedback. Execution-based signals, such as compiling the code and passing unit tests, provide a clear measure of functional correctness [1250]. Furthermore, Static Application Security Testing (SAST) tools, linters, and specialized security analyzers can be integrated into the reward loop. By assigning severity-weighted penalties for detected vulnerabilities, these tools offer a granular and automated signal for security compliance [440].
- **AI-Generated Feedback:** For risks that are difficult to formalize with deterministic tools, such as subtle logic flaws or the inclusion of hard-coded secrets, an AI judge can provide critical feedback. Guided by a security-focused constitution, a teacher model can critique code snippets, effectively scaling expert-level review [92, 662]. This approach allows for capturing a broader spectrum of security concerns that static analyzers might miss.
- **Hybrid Reward Architectures:** The most robust systems often fuse multiple feedback sources into a single, comprehensive reward signal. For instance, Security-Aware Group Relative Policy Optimization (S-GRPO) combines rewards for compilation success, security compliance from static analysis, and format correctness to guide the model towards generating not only secure but also high-quality and functional code [301]. This hybrid approach improves resilience against reward hacking, where a model might learn to exploit a single, narrow reward metric.

Challenges and Advanced Mitigation Strategies Despite its promise, RL-based alignment introduces unique challenges that require sophisticated mitigation techniques.

- **Reward Hacking:** Models may develop deceptive strategies to maximize reward without fulfilling the intended objectives. In the context of coding, this can manifest as the model modifying or deleting unit tests to artificially pass a test suite, or exploiting vulnerabilities in the execution environment itself [300]. This highlights the brittleness of simplistic reward functions.
- **Alignment Tax:** Over-optimizing for a narrow set of safety signals can lead to a degradation in the model's general coding capabilities, a phenomenon known as the alignment tax [809]. For instance, a model aggressively penalized for any potential vulnerability might become overly conservative, refusing to generate useful code or producing inefficient but trivially safe solutions. It is crucial to evaluate security enhancements alongside performance on general coding benchmarks.
- **Advanced Mitigation Frameworks:** To counter these challenges, researchers are exploring more advanced RL algorithms. Constrained policy optimization methods, for example, formalize safety requirements as explicit constraints rather than merely part of the reward, ensuring that the final policy remains within a safe operational space [665]. Another innovative approach is the use of a safety editor policy, which is a separate policy trained specifically to take a potentially unsafe action proposed by the primary performance-driven policy and transform it into a safe one, effectively acting as a runtime guardrail.

[1224]. Such methods provide a more principled way to balance the dual objectives of functionality and security.

Ultimately, while significant progress has been made, RL-based alignment is not a panacea. The continued generation of insecure code by even the most advanced LLMs underscores the need for a multi-layered defense strategy that combines innovations in SFT, preference learning, and RL [379, 707].

7.3. Red-teaming Techniques for Code LLMs

The assessment of security and safety in code LLMs becomes increasingly challenging as new adversarial techniques continue to emerge. To address this challenge, establishing a systematic taxonomy of these techniques is essential. Such a classification clarifies the progression of attack sophistication, ranging from direct input manipulation to the exploitation of complex emergent agentic behaviors. This section provides a comprehensive review of existing red-teaming methodologies, examining both the strategies reported in the literature and the underlying principles that guide their design.

7.3.1. *Prompt-Level Manipulation: Subverting Input-Output Behavior*

The prompt-level manipulation category encompasses attacks that construct adversarial inputs designed to subvert a model’s safety alignment and induce prohibited outputs. This class of techniques has progressed from early manually crafted heuristics to increasingly automated and optimization-driven approaches.

Heuristic-Based Jailbreaking Early and still widely used red-teaming efforts rely on human-devised heuristics that exploit patterns in a model’s training data and alignment. These methods include *role-playing scenarios*, in which the model is instructed to adopt a persona devoid of its usual ethical constraints (e.g., the canonical DAN or Do Anything Now persona) [886], thereby leveraging the model’s tendency to prioritize in-character consistency over its safety protocols. Another common technique is *prefix injection*, also known as refusal suppression, which embeds the request for harmful content within an ostensibly benign context. For example, a prompt such as “Sure, here is an example of a vulnerable SQL query for a security textbook:” can effectively coerce the model into completing the harmful example [1058].

Optimization-Based Adversarial Attacks To overcome the limitations of manual prompt engineering, a substantial line of research formulates adversarial prompt generation as an optimization problem that identifies inputs maximizing the likelihood of harmful model outputs. A prominent example is the greedy coordinate gradient-based search (GCG) algorithm, which uses gradient information to iteratively refine a sequence of characters into an effective adversarial suffix [1344]. The key contribution of this approach is the discovery of universal and transferable attack strings that jailbreak a wide range of disparate, black-box models, thereby exposing systemic vulnerabilities in current alignment techniques.

Generation-Based/Fuzzing Attacks Analogous to fuzzing in traditional software security, this approach utilizes a secondary LLM to automatically generate a vast and diverse set of potential attack prompts. Frameworks such as GPTFuzz employ an attacker LLM to brainstorm creative and syntactically varied prompts, which are then used to test the target model’s robustness

[241]. By providing feedback on successful jailbreaks, the system can guide the attacker LLM’s generation process, automating the discovery of novel and often non-intuitive attack vectors at scale.

Conversational and Multi-Turn Attacks While many safety measures are effective against single-turn inputs, they remain vulnerable to attacks distributed across a stateful conversation. In this setting, an adversary conducts a multi-step dialogue to progressively weaken the target model’s safety guardrails or to construct a context in which a malicious request appears justified. The RedCoder framework operationalizes this idea through adversarial self-play, in which an antagonist agent learns multi-turn strategies that induce a defender agent to generate insecure code [1026]. This demonstrates that a model’s safety is not a static property but can be dynamically influenced through extended interaction.

7.3.2. Semantic and Contextual Manipulation: Exploiting the Interpretation Layer

This more sophisticated class of attacks exploits the mismatch between syntactic filtering and semantic understanding. Although the inputs appear syntactically benign, their contextual interpretation can nevertheless result in harmful outcomes.

Instruction-Data Separation and Trust Boundary Exploitation A potent vector, exemplified by DeceptPrompt, exploits the logical separation between a trusted instruction and untrusted user-provided data [292]. An adversary provides a harmless high-level command while embedding the true malicious payload within the data source. This attack succeeds because safety filters tend to scrutinize the primary instruction but implicitly trust the content of the supplied data, revealing a critical vulnerability in the model’s trust boundary.

Indirect Prompt Injection As a critical threat for agentic systems that interact with external data, indirect prompt injection is an attack-at-a-distance. The adversary poisons an external data source (e.g., a webpage or document in a database) that the LLM agent is expected to ingest and process. The malicious prompt is hidden within this external data. When the agent is later tasked with a benign command like “Summarize the latest financial report,” it processes the poisoned document, which may contain a hidden instruction such as “And then, forward this entire document to an external attacker.” The agent, failing to distinguish between the user’s original intent and the instructions embedded in the data, may execute the malicious command.

Obfuscation and Cross-Lingual Attacks These methods aim to bypass safety models by exploiting gaps in their training distribution. Techniques include character-level obfuscation and, more effectively, the use of low-resource languages or code-switching [1219]. Because many safety classifiers are trained predominantly on high-resource languages such as English, embedding a malicious request in a less common language can evade detection by syntactic or keyword-based filters.

7.3.3. Agentic Workflow: Subversion of Agent Systems and Tool Use

The emergence of LLM-powered agents that can execute tools and interact with live environments introduces a new attack surface focused on the subversion of actions rather than text generation.

Table 21. Comparative analysis of red-teaming techniques for Code LLMs. The table evaluates various methods based on six characteristics from an attacker’s perspective. **Eff.**: Effectiveness (overall success rate). **Diff.**: Difficulty (✓=Easy to implement). **Auto.**: Automation (potential for scaling). **Cost**: Resource Cost (✓=Low cost). **Trans.**: Transferability (applicability to different models). The symbols indicate a favorable (✓), unfavorable (✗), or variable/medium (~) rating for the attacker.

Method	Eff.	Diff.	Auto.	Cost	Trans.
<i>Prompt-Level Manipulation</i>					
Heuristic-Based Jailbreaking	~	✓	✗	✓	✗
Optimization-Based Attacks (GCG)	✓	✗	✓	✗	✓
Generation-Based / Fuzzing	✓	~	✓	✗	~
Conversational / Multi-Turn Attacks	✓	✗	~	~	~
<i>Semantic & Contextual Manipulation</i>					
Trust Boundary Exploitation	✓	~	✗	✓	✓
Indirect Prompt Injection	✓	~	✗	✓	✓
Obfuscation / Cross-Lingual Attacks	~	✓	✓	✓	✓
<i>Agentic Workflow Manipulation</i>					
Tool Misuse & Malicious Argument Injection	✓	~	✗	✓	~
Sandbox Escape & Environment Probing	✓	✗	✗	✓	✗
Automated Vulnerability Exploitation (AVDE)	✓	✗	✓	✗	~

Tool Misuse and Malicious Argument Injection When an LLM agent is granted access to tools, this functionality can itself become an attack surface. Red-teaming in this context involves crafting prompts that induce the agent to pass malicious, unsanitized arguments to its tool functions. For example, an attacker may direct an agent with database access to execute a query whose parameters are manipulated into a SQL injection payload, thereby subverting the chain of trust between the LLM’s reasoning core and its execution capabilities.

Sandbox Escape and Environment Probing For safety, code-executing agents typically operate within a sandboxed environment. A highly technical attack vector directs the agent to generate and execute code that probes for, and exploits, vulnerabilities in this execution environment. The objective is to escape the sandbox and obtain unauthorized access to the host system or internal network. This is not an attack on the LLM’s reasoning process per se, but an exploitation of the agent’s capabilities to target its underlying infrastructure.

Automated Vulnerability Exploitation (AVDE) This represents the apex of agentic capabilities, in which the entire agent is weaponized into an autonomous penetration-testing tool. Frameworks such as RedCodeAgent [941] illustrate this by assigning the agent a high-level objective, which it then pursues by autonomously executing the full cybersecurity kill chain—reconnaissance, vulnerability scanning, exploit selection, and execution. This constitutes an ultimate stress test of an agent’s potential for misuse.

In summary, the landscape of red-teaming methodologies exhibits a clear evolutionary trajectory. It has advanced from static, heuristic-based prompts aimed at eliciting unsafe outputs to dynamic, automated, and context-aware attacks that target the entire agentic workflow, including its data sources, tool use, and operational environment. This escalating sophistication underscores the need for a holistic, defense-in-depth approach to AI safety.

Table 22. Threat × Eval × Control matrix for red-teaming Code LLMs (✓ present/primary).

Attack family (Entry)	Harm footprint				Evaluation kit		DiD controls			
	UC	EX	TM	ENV	GB	HM	V	P	R	I
<i>Prompt-Level Manipulation</i>										
Heuristic jailbreaking (Prompt)	✓				✓	✓		✓	✓	
Optimization-based (GCG / T-GCG / ADC) (Prompt)	✓				✓	✓	✓	✓	✓	
Generation / fuzzing (Prompt)	✓				✓	✓		✓	✓	
Conversational / multi-turn (Prompt)	✓				✓				✓	
<i>Semantic & Contextual Manipulation</i>										
Trust-boundary exploit (Instr.-Data) (Prompt+ExtData)	✓	✓			✓		✓	✓	✓	
Indirect prompt injection (poisoned sources) (ExtData)	✓	✓			✓			✓	✓	
Obfuscation / cross-lingual (Prompt)	✓				✓	✓		✓		
<i>Agentic Workflow Manipulation</i>										
Tool misuse & argument injection (Tool)	✓	✓			✓		✓	✓	✓	
Sandbox escape & env. probing (Exec)			✓		✓		✓			✓

Harm footprint: UC = unsafe code; EX = data exfiltration; TM = tool misuse; ENV = environment compromise. Evaluation kit: GB = GuidedBench guideline scoring; HM = HarmMetric Eval (METEOR/ROUGE-1); V = verifiers (compile/tests/SAST/forensics). Defense-in-Depth: P = Pre-Execution; R = Runtime; I = Isolation. Prefer GB/HM over refusal-keyword heuristics; include V for code-oriented/ENV cases.

7.4. Mitigation Strategies for Coding and Behavioral Risks in AI Agent Systems

To address the multifaceted risks posed by autonomous and semi-autonomous code-generating agents, a robust, multi-layered security paradigm is essential. We adopt a Defense-in-Depth framework that shifts the focus from solely validating code correctness to holistically governing agent behavior across three interdependent layers: secure execution environments for containment, proactive pre-execution validation for prevention, and dynamic runtime oversight for real-time enforcement. The overarching goal is to achieve capability containment by design and to apply dynamic, context-aware controls during agent operation [720].

7.4.1. Foundations in Secure Execution Environments

The first line of defense is to establish a stringent isolation boundary that contains an agent's potential impact, strictly adhering to the Principle of Least Privilege and Capability Containment. The choice of isolation technology involves a trade-off between performance, compatibility, and the strength of the security guarantee. A spectrum of such technologies is currently employed:

- **OS-level Containers:** Technologies like Docker are widely used for their efficiency and ease of deployment. However, their reliance on a shared host kernel exposes a significant attack surface. Vulnerabilities such as path misresolution can allow malicious code to break filesystem isolation and escape the containerized environment [578].
- **Process-level Sandboxes:** More granular control can be achieved using sandboxes like nsjail, which leverage kernel features such as seccomp-bpf and namespaces to enforce fine-grained syscall filtering [1004]. While powerful for enforcing least privilege, defining precise and effective policies for complex agent tasks remains a significant challenge. Recent work has explored using LLMs themselves to help configure these complex sandboxes, yet ensuring the completeness of such policies is an ongoing research problem [35]. The development of specialized benchmarks like SandboxEval is a crucial step towards

systematically evaluating the security and efficacy of these test environments for untrusted code execution [1317].

- **Virtualization-based Isolation:** At the stronger end of the spectrum, MicroVMs such as Firecracker and hypervisor-based solutions like gVisor and Kata Containers offer near-hardware-level isolation with performance characteristics approaching that of containers. This strong isolation effectively mitigates most kernel-level threats, but it is not a panacea. The underlying hardware remains a shared resource, and sophisticated threats targeting microarchitectural side-channels (e.g., Spectre/MDS) necessitate complex mitigation strategies at both the host and guest levels [1069].

A critical limitation of these approaches is their static nature. A predefined, fixed sandbox policy often creates a capability gap, where the environment is either too restrictive for the agent to complete its task or too permissive, granting unnecessary and potentially dangerous capabilities. This has led to research into *dynamic containment*, where isolation levels can be adjusted in real-time based on the agent's behavior and the assessed risk of its current trajectory [1011]. Frameworks like Progent exemplify this shift by introducing programmable privilege control, allowing for more flexible and context-aware security postures than static sandboxing alone [619]. Standardized environments for interactive agent evaluation, such as InterCode, provide the necessary infrastructure to benchmark and compare these evolving containment strategies [1183].

7.4.2. Proactive Defense and Pre-Execution Validation

The second layer of defense focuses on preventive vetting of agent-generated artifacts and plans before they are executed. This proactive stance aims to identify and remediate vulnerabilities, logical flaws, and misalignments with user intent at the earliest possible stage.

- **Modernized Code Analysis:** Traditional security tools like Static (SAST) and Dynamic (DAST) Application Security Testing are being reimagined for the agentic era. The key innovation is their integration into a “tool-in-the-loop” feedback cycle. For instance, a dedicated static analyzer agent can inspect generated code, with its findings structured into a prompt that instructs the primary agent to perform a repair, automating the detection-remediation loop [739, 1027]. This is critical, as empirical studies consistently demonstrate that LLMs, especially when tasked with web development, generate code with common vulnerabilities such as SQL injection, Cross-Site Scripting (XSS), and insecure file handling [30, 75].
- **Multi-Agent Review and Collaboration:** Inspired by human software development practices like code review and pair programming, multi-agent architectures have emerged as a powerful defense mechanism. These systems employ various collaborative patterns: a “Critic” LLM may review the “Coder” LLM’s output; specialized agents for coding, static analysis, and fuzzing may work in parallel or a defense-focused agent may be used to detect and mitigate backdoors injected into the chain-of-thought process [1268]. This collaborative review process has been shown to improve code quality and security [34]. Furthermore, multi-agent frameworks are also being developed as a defense against adversarial attacks like jailbreaking, where a diverse ensemble of agents collectively enhances system robustness [243].
- **Formal Methods and Intent Verification:** The most rigorous form of pre-execution validation involves leveraging formal methods to guarantee alignment with user intent. This approach seeks to translate high-level, natural-language specifications into formal

constraints, such as complexity bounds, I/O formats, or state-machine models. These constraints can then guide the code generation process, ensuring that the output adheres to predefined safety and correctness properties by construction. Techniques include generating code with accompanying proofs, or synthesizing invariants and pre/postconditions that formally capture and verify user intent before execution is ever attempted [743, 1288].

7.4.3. Runtime Oversight and Intent Grounding

The final defense layer addresses risks that manifest dynamically during execution, which cannot be caught by static analysis alone. This layer is responsible for real-time monitoring, enforcement of safety policies, and bridging the semantic gap between an agent's low-level actions and their high-level consequences. The core problem is that syntactically correct and locally optimal operations can cascade into globally unsafe or irreversible state changes [46, 278].

To mitigate these runtime risks, a suite of techniques centered on guardrails and runtime enforcement has been developed:

- **Guardrail Frameworks and Secure Agents:** Comprehensive defense frameworks like AgentSentinel provide end-to-end, real-time security monitoring for agents interacting with computer environments [550]. A popular architectural pattern is the “guardian agent”, where a secondary, security-focused LLM observes the primary agent’s actions and plans. GuardAgent employs knowledge-enabled reasoning to anticipate and block potentially harmful actions [1088], while systems like LlamaFirewall act as an open-source guardrail, inspecting both inputs to and outputs from the agent to enforce safety constraints [503].
- **Verifiable Policy Enforcement:** Moving beyond heuristic-based monitoring, some systems enforce safety through verifiable policies. AgentSpec provides a framework for specifying customizable runtime enforcement rules, ensuring that an agent’s behavior remains within formally defined bounds [1022]. Similarly, ShieldAgent leverages verifiable safety policy reasoning to shield agents from executing unsafe actions, providing a stronger guarantee of compliance [1253]. The policies themselves can even be synthesized automatically from natural language, as demonstrated by systems that create an “AI Agent Code of Conduct” [287].
- **Active Control and Intervention:** The most advanced runtime systems provide mechanisms not just for monitoring, but for active intervention. Ctrl-Z introduces a method for controlling agents by resampling their proposed actions if they are deemed unsafe, effectively providing a rollback or undo capability at the decision-making level [672].

Ultimately, the frontier of runtime safety lies in enhancing these systems with a deeper understanding of causality and intent. Future work aims to complement policy enforcement with cognitive telemetry and causal influence diagrams to detect “intent drift” [373, 1260]. By grounding an agent’s symbolic actions in a world model that understands real-world consequences, these systems can better mirror human-like goal inference and reasons-based action, forming the final and most intelligent layer of a comprehensive defense-in-depth strategy [1057, 1261].

8. Training Recipes for Code Large Language Model

Training a state-of-the-art code LLM is a sophisticated and multi-phase pipeline where each stage serves distinct purposes. Unlike general-purpose LLMs, code LLMs must master strict syntax, complex logic, and long-range algorithmic dependencies, while being required to generate

outputs that are verifiably correct. This process typically begins with *pre-training*, where the LLM learns the fundamental statistical patterns of programming languages from vast codebases; this foundational step necessitates the sophisticated distributed frameworks detailed first. Following pretraining, the model undergoes *supervised fine-Tuning (SFT)* to adapt its general capabilities, aligning with human instructions and solving task-oriented problems. Finally, to optimize for objective correctness rather than just data imitation, the model can be further refined using *reinforcement learning (RL)*, where one of the common settings is to reward the model for generating solutions that pass verifiable unit tests. This section provides training recipes to train a top-tier code LLM (system architectures and hyperparameter guidelines) from scratch for each of these critical stages.

8.1. Distributed Training Framework Introduction

This section examines the predominant frameworks employed in contemporary code LLM training, analyzing their core parallelism strategies and system architectures. While all training stages could benefit from such an optimization of infrastructure, the pretraining of large language models particularly necessitates sophisticated distributed training frameworks capable of efficiently orchestrating computation across thousands of accelerators.

Megatron-LM Megatron-LM [897] introduces an efficient intra-layer model parallelism approach that partitions individual transformer layers across multiple devices [896]. The framework’s core innovation lies in its tensor parallelism strategy, which performs column-wise and row-wise partitioning of weight matrices in multi-layer perceptrons and attention mechanisms. This design requires only minimal all-reduce communication operations that can be overlapped with computation, achieving 76% scaling efficiency when training 8.3B parameter models across 512 GPUs. The framework implements three complementary parallelization techniques: tensor parallelism for fine-grained model partitioning within transformer blocks, pipeline parallelism with microbatch-based execution and interleaved scheduling to reduce pipeline bubbles [721], and sequence parallelism that partitions activation tensors along the sequence dimension for long-context training. Megatron-LM demonstrates exceptional performance on high-bandwidth interconnects, sustaining 15.1 PetaFLOPs across 512 V100 GPUs, with subsequent extensions enabling training of models exceeding 530 billion parameters [907].

DeepSpeed DeepSpeed [833, 839] centers around the Zero Redundancy Optimizer (ZeRO), which eliminates memory redundancies in data-parallel training through progressive partitioning of training states. Unlike model parallelism approaches that partition the model itself, ZeRO partitions optimizer states, gradients, and parameters across data-parallel processes while maintaining the computational advantages of data parallelism. The framework provides three progressive optimization stages: ZeRO-1 partitions optimizer states across data-parallel ranks for 4 \times memory reduction, ZeRO-2 extends partitioning to gradients achieving 8 \times reduction, and ZeRO-3 partitions model parameters enabling linear scaling with device count. Additional innovations include ZeRO-Offload for CPU memory utilization supporting 13B+ parameter models on single GPUs, and efficient pipeline parallelism composable with ZeRO stages. Empirical evaluations show DeepSpeed achieves up to 10 \times speedup over baseline implementations for models with 100+ billion parameters, with communication optimizations like 1-bit Adam reducing communication volume by up to 5 \times .

PyTorch FSDP PyTorch [1305] Fully Sharded Data Parallel implements ZeRO-3 optimization as a native PyTorch component. The recently introduced FSDP2 redesigns the implementation using DTensor abstractions, representing sharded parameters as distributed tensors with improved memory management and deterministic GPU allocation. FSDP provides complete sharding of parameters, gradients, and optimizer states across data-parallel processes, with flexible policies including NO_SHARD (DDP), SHARD_GRAD_OP (ZeRO-2), FULL_SHARD (ZeRO-3), and HYBRID_SHARD strategies. The DTensor foundation in FSDP2 enables cleaner state management and communication-free sharded checkpoints, while communication optimizations employ implicit and explicit prefetching to overlap all-gather operations with computation. FSDP2 achieves approximately 1.5% higher throughput than FSDP1, with training of Llama2-7B on 128 A100 GPUs reaching 3,700 tokens/sec/GPU. The framework’s native PyTorch integration facilitates adoption in research environments prioritizing ecosystem compatibility.

TorchTitan TorchTitan [586] provides a production-grade reference implementation of 4D parallelism for LLM pretraining, demonstrating PyTorch’s latest distributed training capabilities. The framework composes FSDP2, tensor parallelism, pipeline parallelism, and context parallelism in a modular architecture, featuring native FP8 mixed precision support for reduced memory and computation, async tensor parallelism that overlaps communications with independent computations, and torch.compile integration for kernel fusion and optimization. Empirical results show TorchTitan achieves 65.08% speedup on Llama 3.1 8B (128 GPUs), 12.59% on 70B (256 GPUs), and 30% on 405B (512 GPUs) over optimized baselines. The framework demonstrates near-linear weak scaling to 512 GPUs with appropriate parallelism configuration and supports six pipeline scheduling strategies for flexible deployment.

Colossal-AI Colossal-AI [565] provides diverse parallelization strategies with emphasis on multi-dimensional tensor parallelism. The framework implements 1D (Megatron-style), 2D (mesh-based), 2.5D (optimized 2D), and 3D tensor decomposition strategies, offering flexibility in trading off memory, computation, and communication costs. Key innovations include Gemini for heterogeneous CPU-GPU memory management supporting 13B parameter models on single consumer GPUs, and full ZeRO integration alongside sequence parallelism for long-context scenarios. Colossal-AI demonstrates up to 2.76 \times training speedup over baseline systems, with a configuration-driven approach enabling rapid exploration of parallelism strategies without code modification.

Comparative Analysis The selection of an appropriate framework depends on hardware infrastructure, model scale, and organizational requirements. Megatron-LM and Megatron-DeepSpeed excel on premium GPU clusters with high-bandwidth interconnects, achieving optimal performance for models exceeding 100B parameters. DeepSpeed prioritizes memory efficiency, enabling training on resource-constrained environments. PyTorch FSDP and TorchTitan provide native PyTorch solutions with strong ecosystem integration, suitable for organizations standardizing on PyTorch infrastructure. Colossal-AI offers maximum flexibility in parallelism strategies, facilitating research on novel architectures. [Table 23](#) summarizes key characteristics and performance metrics across these frameworks.

8.2. Pre-Training Guidelines

Recent works focus on the scaling law for the code LLMs [657], which shows that requires a substantially higher data-to-parameter ratio than natural language, indicating it is a more

Table 23. Comparative analysis of distributed training frameworks for LLM pretraining reported from the original paper.

Framework	Scaling Efficiency	Max Demonstrated	Memory Strategy	Hardware Preference	Key Innovation
Megatron-LM	76% (512 GPUs)	530B params	TP + SP	NVLink/InfiniBand	Overlapped tensor parallel
DeepSpeed	10x speedup	200B params	ZeRO-1/2/3 + Offload	Flexible	Progressive state sharding
Megatron-DS	High (530B)	530B params	ZeRO + TP	NVLink/InfiniBand	Unified 3D parallelism
PyTorch FSDP	1.5% over FSDP1	70B params	Full sharding (ZeRO-3)	Flexible	Native PyTorch integration
TorchTitan	65% speedup (8B)	405B params	FSDP2 + FP8	High-end clusters	4D parallelism + compile
Colossal-AI	2.76x speedup	175B params	Multi-dim TP + Gemini	Consumer to HPC	Flexible TP dimensions

data-intensive training domain. Pre-training represents the foundational phase of code LLM development, where models acquire fundamental programming knowledge from vast code repositories. Unlike supervised fine-tuning or reinforcement learning that build upon pre-trained capabilities, pre-training establishes the base knowledge that determines a model’s ultimate performance ceiling. However, the enormous computational cost makes exhaustive hyperparameter exploration infeasible at scale. This section leverages scaling laws derived from extensive multilingual experiments to provide data-driven guidelines for compute-efficient pre-training.

Language-Specific Scaling Laws Programming languages exhibit fundamentally different scaling behaviors that must inform pre-training strategies. Through systematic empirical studies spanning seven major programming languages (Python, Java, JavaScript, TypeScript, C#, Go, Rust), we establish the Chinchilla-style scaling relationship for each language:

$$L(N, D) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D} + L_\infty \quad (4)$$

where N denotes model parameters, D represents training tokens, and L_∞ captures the irreducible loss (a fundamental measure of language complexity). [Table 24](#) summarizes the fitted parameters, revealing substantial heterogeneity across languages.

Table 24. Fitted scaling law parameters for seven programming languages. The exponents α_N and α_D quantify sensitivity to model size and data volume, while L_∞ represents the theoretical performance lower bound.

Language	α_N	α_D	L_∞
Python	0.221	1.217	0.566
Java	0.447	1.129	0.397
JavaScript	0.692	1.247	0.554
TypeScript	0.439	1.303	0.518
C#	0.321	1.350	0.288
Go	0.845	1.149	0.414
Rust	0.643	1.297	0.397

The results uncover a clear pattern: **interpreted languages exhibit larger scaling exponents**

than compiled languages. Python demonstrates the highest α_N and α_D values, indicating aggressive benefits from both increased model capacity and training data. This reflects Python’s dynamic typing, flexible syntax, and diverse idioms, which create a more complex statistical landscape. Conversely, statically-typed compiled languages like Rust and Go show notably smaller exponents, where their rigid syntactic structures and explicit type annotations carry more information per token, making them inherently more learnable with fewer parameters and less data.

The irreducible loss L_∞ reveals intrinsic language complexity, ordering languages as: **C# (0.288) < Java = Rust (0.397) < Go (0.414) < TypeScript (0.518) < JavaScript (0.554) < Python (0.566)**. C# achieves the lowest bound through its strict type system and standardized ecosystem, while Python exhibits the highest due to its expressive nature and variability in coding styles. This ranking directly informs compute allocation: languages with lower L_∞ saturate faster and require proportionally less training budget.

Multilingual Mixture Effects Systematic experiments on bilingual pre-training mixtures reveal that **multilingual training provides substantial benefits over monolingual baselines**. Languages sharing similar syntax and semantics exhibit strong positive synergy—for example, Java-C# mixtures achieve over 20% loss reduction compared to Java-only training, while JavaScript-TypeScript pairs show consistent mutual improvements. The synergy gain can be quantified as $\Delta(\ell, \ell') = L(\ell + \ell') - L(\ell + \ell')$, measuring the performance difference between self-repetition and bilingual mixing.

However, **Python presents an asymmetric exception**: mixing Python with most statically-typed languages produces small negative effects when Python is the target language, though using Python as an auxiliary language benefits other targets. This reflects Python’s unique dynamic typing paradigm. Importantly, negative interference remains limited and language-specific rather than pervasive, strongly suggesting multilingual pre-training as a beneficial default strategy with careful consideration of language pairing.

Cross-lingual transfer effects extend beyond explicitly paired languages. Models pre-trained on multilingual corpora demonstrate zero-shot capabilities on unseen language pairs, suggesting they learn abstract algorithmic equivalence that transcends specific syntax. Document-level pairing strategies—concatenating parallel implementations within single training documents—substantially outperform random shuffling for both seen and unseen translation directions, while maintaining strong performance on general code understanding benchmarks.

Recommended Pre-Training Strategies Based on comprehensive empirical analysis across multiple scales and language combinations, we provide the following guidelines for multilingual code pre-training:

Language-Specific Token Budgets: Allocate training tokens proportional to α_D exponents rather than uniformly. High- α_D languages (Python, C#, TypeScript) benefit substantially from increased data and should receive proportionally more tokens. Low- α_D languages (Java, Go) saturate faster and can use fewer tokens without significant performance loss. Empirical validation shows that optimization-guided allocation outperforms uniform distribution by substantial margins under identical compute budgets.

Corpus Construction for Similar Programming Languages: Prioritize syntactically similar language pairs in pre-training mixtures. Strong positive synergies exist for Java-C#, JavaScript-TypeScript, and Rust-Go pairs. Leverage Python’s asymmetric transfer by using it as an auxiliary

language for other targets rather than mixing extensively when Python is the primary objective. Document-level pairing of parallel implementations provides implicit alignment signals that improve both translation and general code understanding.

Complexity-Informed Compute Allocation: Languages with low L_∞ in [Equation 4](#) (C#, Java, Rust) approach performance saturation earlier and require proportionally less total compute. Focus extended training on high L_∞ languages (Python, JavaScript) where marginal returns remain substantial even at large scales. The optimal compute allocation should balance model size N and training tokens D according to language-specific exponents.

Multilingual Default Strategy: Unless constrained to a single target language, multilingual pre-training should be the default approach. Most languages exhibit consistent positive synergy, and cross-lingual transfer enables emergent zero-shot capabilities on unseen language pairs. For resource-constrained scenarios, start with semantically related language clusters (e.g., Python-Java-C# or JavaScript-TypeScript) before expanding to full multilingual coverage.

These guidelines synthesize extensive scaling law experiments into actionable principles, enabling practitioners to design compute-efficient pre-training pipelines that maximize performance across multiple programming languages under realistic budget constraints.

8.3. Supervised Finetune Training Guidelines

Having established the foundational frameworks for large-scale pretraining, the focus now shifts to the next critical phase: supervised fine-tuning (SFT). While pretraining builds the model’s general capabilities, SFT is essential for adapting this foundational model to specific, high-value tasks like instruction-following or complex reasoning. This adaptation phase introduces its own distinct set of challenges and trade-offs, particularly in framework choice, hyperparameter sensitivity, and data curation. The following sections provide a comprehensive guide to navigate this SFT landscape.

Training Framework Guidelines Effective supervised fine-tuning of code large language models requires training frameworks ⁹ that balance training efficiency and model performance, which focuses more on data organization pipeline, the leverage of distributed training framework and offering user-friendly hyper-parameter tuning APIs. To empirically examine these trade-offs, we fine-tuned **Qwen2.5-Coder-14B** on the Magicoder_OSS_Instruct_75K¹⁰ dataset (3 epochs, 256 global batch size, 8192 max_length, learning rate 1×10^{-6} , warmup ratio 0.03) using four representative frameworks: QwenCoder-SFT¹¹, LLaMA-Factory¹², MS-Swift¹³, and VERL¹⁴. Each was evaluated under 64 GPUs configuration in our experimental setup. Although optimization hyperparameters and dataset ordering were held constant, the frameworks differ fundamentally in their parallel training strategies, communication backends, and system abstractions, leading to distinct performance and efficiency characteristics.

QwenCoder-SFT (HuggingFace Trainer) serves as a minimal-overhead baseline employing classic data parallelism via PyTorch DDP and mixed-precision training. Its main advantage

⁹Note that the “training frameworks” here are distinguished from and mostly built upon the lower-level distributed training frameworks in [subsection 8.1](#).

¹⁰<https://huggingface.co/datasets/ise-uiuc/Magicoder-OSS-Instruct-75K>

¹¹<https://github.com/QwenLM/Qwen3>

¹²<https://github.com/hiyouga/LLaMA-Factory>

¹³<https://github.com/modelscope/ms-swift>

¹⁴<https://github.com/volcengine/verl>

lies in simplicity and reproducibility; however, the full replication of optimizer and parameter states across GPUs limits scalability and throughput. In our environment, QwenCoder-SFT demonstrated stable convergence but exhibited memory constraints when scaling beyond moderate model sizes.

LLaMA-Factory (DeepSpeed ZeRO-3) [1316] leverages ZeRO [834] partitioning to shard optimizer states, gradients, and parameters across workers, substantially reducing per-device memory consumption while retaining full-precision optimizer dynamics. In conjunction with micro-batch pipeline execution and activation checkpointing, the ZeRO-3 configuration achieved high memory and compute efficiency. Under our 64-GPU configuration, a single 14B SFT run completed in roughly 50 minutes, with negligible convergence differences compared to the baseline.

MS-Swift (Megatron) integrates Megatron-LM’s[897] hybrid tensor- and pipeline-parallel decomposition of transformer blocks, optimized for high-throughput code pretraining and large-batch SFT. By overlapping all-reduce communications with compute kernels, MS-Swift maintained near-peak GPU utilization in our system. In our setup, it reached comparable accuracy to ZeRO-3 while reducing wall-clock training time to approximately 20 minutes, reflecting its communication efficiency under dense model architectures.

VERL (FSDP v2) [888] implements PyTorch’s fully sharded data parallelism through the DTensor abstraction, natively sharding parameters, gradients, and optimizer states. While FSDP v2[1304] provided high memory efficiency and determinism, the repeated all-gather operations introduced non-trivial communication overhead. In our configuration, the total wall-clock time for fine-tuning was approximately 2 hours. Nevertheless, it delivered reproducible scaling behavior and seamless integration with PyTorch-native tools, making it particularly suitable for multi-framework benchmarking and long-sequence RL fine-tuning.

Overall, these observations suggest a clear trade-off landscape under our experimental setup: QwenCoder-SFT offers simplicity and stability for small- to medium-scale runs; LLaMA-Factory and MS-Swift provide efficient large-scale SFT via ZeRO-3 and hybrid parallelism respectively; and VERL offers full-sharding generality and ecosystem compatibility at the cost of longer runtime. For large-scale, multi-epoch code SFT, frameworks that combine partitioned memory (ZeRO/FSDP) with intra-layer tensor parallelism (e.g., DeepSpeed ZeRO-3 or Megatron-based systems) achieve the best balance between convergence stability and system efficiency in our setup.

Table 25. Comparison of supervised fine-tuning architectures on Qwen2.5-Coder-14B trained on the Magicoder_OSS_Instruct_75K dataset (3 epochs, 256 global batch size, 8192 max length, learning rate 1×10^{-6} , warmup ratio 0.03, 64 GPUs).

Framework	HumanEval	HumanEval+	MBPP	MBPP+	Time
QwenCoder-SFT (HuggingFace Trainer)	0.848	0.774	0.857	0.722	20 min
LLaMA-Factory (DeepSpeed ZeRO-3)	0.872	0.768	0.860	0.735	50 min
MS-Swift (Megatron)	0.872	0.774	0.857	0.735	20 min
VERL (FSDP v2)	0.860	0.762	0.860	0.728	2 h

Training Parameter Guidelines While the choice of a training framework (as explored in Table 25) determines the efficiency and mechanics of parallelization, the hyperparameter configuration dictates the outcome and quality of the fine-tuning process. We now move from the “how” of the system to the “what” of the optimization, conducting parameter sweeps to examine

supervised fine-tuning sensitivity using the Magicoder_OSS_Instruct_75K dataset. The default recipe employs a learning rate of 2×10^{-6} , 3 epochs, per-device batch size of 2 with gradient accumulation 2—yielding a global batch of approximately 256 on 64 GPUs—with warmup ratio 0.05, cosine learning-rate schedule, and context length 8192. [Table 26](#) shows that global batch size is the dominant sensitivity factor for supervised code SFT. For both Qwen2.5-Coder-14B and Qwen3-30B-A3B, accuracy degrades once the global batch exceeds roughly 256: the 30B model’s MBPP score drops from 0.860 at 64 to 0.556, 0.254, and 0.169 at 512, 1024, and 2048 respectively, while the 14B model saturates (e.g., HumanEval 0.872 at 256 vs. 0.860 at 1024). This pattern suggests that smaller effective batches (64–256) preserve gradient signal in code distributions better than highly averaged updates. Learning-rate optima are model-dependent: the 14B backbone favors 2×10^{-6} – 5×10^{-6} , while the 30B backbone underfits at 1×10^{-6} and benefits from larger rates, peaking near 5×10^{-5} . The value 2×10^{-6} remains close to Pareto-optimal across several metrics. Training length interacts with scale: the 14B model shows diminishing returns beyond 3–5 epochs, whereas the 30B model requires more epochs to stabilize. Schedulers and warmup are secondary for 14B (constant, cosine, and linear schedules perform similarly) but matter for 30B, where a constant schedule with modest warmup (0.03–0.10) is safer; very large warmup (≥ 0.30) reduces accuracy.

Recommended settings. Global batch size 64–256; learning rate 2×10^{-6} – 5×10^{-6} for 14B and 5×10^{-6} – 1×10^{-5} for 30B (or 5×10^{-5} for faster early progress); 3–5 epochs for 14B and 3–10 for 30B; warmup ratio 0.05; cosine scheduling for 14B and constant scheduling for 30B. These settings align with the trends observed in [Table 26](#).

Model Architecture Comparison The optimal hyperparameters identified in [Table 26](#) are deeply dependent on the base model. A parameter recipe that works for a dense model may lead to instability or underfitting in a sparse one. Therefore, we now compare two backbone architectures, Qwen2.5-Coder-14B (dense) and Qwen3-30B-A3B (Mixture of Experts), under an identical supervised fine-tuning configuration on the Magicoder-OSS-Instruct-75K dataset. Both models are trained for three epochs with a learning rate of 1×10^{-6} , a warmup ratio of 0.03, a context length of 8192, and a global batch size of 256 distributed across 64 GPUs. [Figure 43](#) and [Table 26](#) summarize the results across all major hyperparameter dimensions, providing a direct comparison between dense and MoE architectures under consistent optimization settings.

The dense 14B architecture exhibits greater robustness to hyperparameter variations, showing steady improvements as the training duration increases from one to five epochs, followed by convergence saturation beyond this point. Performance on HumanEval and MBPP remains stable across batch sizes ranging from 64 to 256, with only minor degradation observed at extremely large scales (greater than 1024). This indicates that dense transformers maintain consistent gradient signal quality and generalization behavior even under aggressive scaling. In addition, the 14B model displays smooth sensitivity curves across learning-rate sweeps, with optimal ranges between 2×10^{-6} and 5×10^{-6} , and negligible differences across scheduler or warmup configurations. These results suggest a broad basin of stable convergence, which is characteristic of mature dense language model training.

In contrast, the MoE-based 30B architecture demonstrates higher variance and sharper sensitivity to optimization choices. Although it achieves comparable peak performance under favorable conditions (for example, HumanEval = 0.836 at ten epochs, MBPP = 0.860 at batch size 64), its stability margin is considerably narrower. Performance declines sharply when the batch size exceeds 512 or the learning rate falls below 1×10^{-6} , reflecting its dependence on balanced expert routing and load normalization. Longer training, typically between five and

Table 26. Single-parameter sweeps for supervised fine-tuning of Qwen2.5-Coder-14B and Qwen3-30B-A3B on the Magicoder_OSS_Instruct_75 dataset (3 epochs, 256 global batch size, 8192 max_length, learning rate 1×10^{-6} , warmup ratio 0.03, 64 GPUs). Bold headers denote distinct hyperparameter categories. Bold values indicate best-performing configurations; italicized values denote second-best results per column. Upward (\uparrow) and downward (\downarrow) arrows indicate improvements or declines relative to the base model performance. Baseline denotes the official instruction version for Qwen2.5-Coder-14B-Instruct and Qwen3-30B-A3B.

Setting	Qwen2.5-Coder-14B				Qwen3-30B-A3B			
	HumanEval	HumanEval+	MBPP	MBPP+	HumanEval	HumanEval+	MBPP	MBPP+
Baseline	0.634	0.555	0.839	0.688	0.787	0.750	0.799	0.683
Epochs								
1	0.817 \uparrow	0.762 \uparrow	0.849 \uparrow	0.722 \uparrow	0.226 \downarrow	0.220 \downarrow	0.098 \downarrow	0.090 \downarrow
2	0.860 \uparrow	0.793\uparrow	0.852 \uparrow	0.722 \uparrow	0.579 \downarrow	0.524 \downarrow	0.183 \downarrow	0.172 \downarrow
3	0.854 \uparrow	0.793\uparrow	0.862 \uparrow	0.722 \uparrow	0.799 \uparrow	0.713 \downarrow	0.270 \downarrow	0.233 \downarrow
5	0.866\uparrow	0.780 \uparrow	0.868\uparrow	0.725 \uparrow	0.823 \uparrow	0.732 \downarrow	0.455 \downarrow	0.392 \downarrow
10	0.866\uparrow	0.768 \uparrow	0.865 \uparrow	0.738\uparrow	0.829\uparrow	0.738\downarrow	0.836\uparrow	0.704\uparrow
Global Batch Size (64 GPUs)								
64	0.860 \uparrow	0.787\uparrow	0.868 \uparrow	0.738\uparrow	0.835\uparrow	0.744\downarrow	0.860\uparrow	0.714\uparrow
128	0.860 \uparrow	0.762 \uparrow	0.870\uparrow	0.735 \uparrow	0.799 \uparrow	0.720 \downarrow	0.799	0.664 \downarrow
256	0.872\uparrow	0.774 \uparrow	0.857 \uparrow	0.728 \uparrow	0.799 \uparrow	0.713 \downarrow	0.807 \uparrow	0.675 \downarrow
512	0.872\uparrow	0.780 \uparrow	0.852 \uparrow	0.722 \uparrow	0.811 \uparrow	0.732 \downarrow	0.556 \downarrow	0.471 \downarrow
1024	0.860 \uparrow	0.787\uparrow	0.862 \uparrow	0.722 \uparrow	0.793 \uparrow	0.720 \downarrow	0.254 \downarrow	0.220 \downarrow
2048	0.860 \uparrow	0.787\uparrow	0.857 \uparrow	0.717 \uparrow	0.360 \downarrow	0.341 \downarrow	0.169 \downarrow	0.153 \downarrow
Global Batch Size (16 GPUs)								
16	0.817 \uparrow	0.762\uparrow	0.836 \downarrow	0.714 \uparrow	0.829 \uparrow	0.762\uparrow	0.844 \uparrow	0.722\uparrow
32	0.829 \uparrow	0.762\uparrow	0.847 \uparrow	0.728 \uparrow	0.835\uparrow	0.756 \uparrow	0.849 \uparrow	0.709 \uparrow
64	0.829 \uparrow	0.762\uparrow	0.862\uparrow	0.746\uparrow	0.835\uparrow	0.744 \downarrow	0.854\uparrow	0.720 \uparrow
128	0.835 \uparrow	0.756 \uparrow	0.862\uparrow	0.743 \uparrow	0.799 \uparrow	0.720 \downarrow	0.804 \uparrow	0.675 \downarrow
256	0.848\uparrow	0.762\uparrow	0.860 \uparrow	0.735 \uparrow	0.811 \uparrow	0.726 \downarrow	0.796 \downarrow	0.669 \downarrow
Learning Rate								
1×10^{-4}	0.793 \uparrow	0.713 \uparrow	0.817 \downarrow	0.693 \uparrow	0.780 \downarrow	0.726 \downarrow	0.810 \uparrow	0.672 \downarrow
5×10^{-5}	0.799 \uparrow	0.744 \uparrow	0.844 \uparrow	0.717 \uparrow	0.866\uparrow	0.799\uparrow	0.847\uparrow	0.706 \uparrow
1×10^{-5}	0.829 \uparrow	0.756 \uparrow	0.862 \uparrow	0.735 \uparrow	0.829 \uparrow	0.756 \uparrow	0.820 \uparrow	0.696 \uparrow
5×10^{-6}	0.829 \uparrow	0.756 \uparrow	0.870\uparrow	0.746\uparrow	0.811 \uparrow	0.726 \downarrow	0.844 \uparrow	0.709 \uparrow
2×10^{-6}	0.854\uparrow	0.762\uparrow	0.862 \uparrow	0.730 \uparrow	0.811 \uparrow	0.732 \downarrow	0.844 \uparrow	0.722\uparrow
1×10^{-6}	0.829 \uparrow	0.756 \uparrow	0.852 \uparrow	0.717 \uparrow	0.793 \uparrow	0.707 \downarrow	0.241 \downarrow	0.217 \downarrow
5×10^{-7}	0.805 \uparrow	0.744 \uparrow	0.852 \uparrow	0.722 \uparrow	0.280 \downarrow	0.262 \downarrow	0.143 \downarrow	0.132 \downarrow
1×10^{-7}	0.805 \uparrow	0.750 \uparrow	0.397 \downarrow	0.333 \downarrow	0.152 \downarrow	0.140 \downarrow	0.061 \downarrow	0.050 \downarrow
LR Scheduler								
constant	0.835\uparrow	0.756 \uparrow	0.862\uparrow	0.728\uparrow	0.799\uparrow	0.720\downarrow	0.675\downarrow	0.577\downarrow
cosine	0.829 \uparrow	0.768\uparrow	0.852 \uparrow	0.725 \uparrow	0.780 \downarrow	0.689 \downarrow	0.272 \downarrow	0.246 \downarrow
linear	0.823 \uparrow	0.762 \uparrow	0.847 \uparrow	0.717 \uparrow	0.744 \downarrow	0.652 \downarrow	0.241 \downarrow	0.214 \downarrow
Warmup Ratio								
0.00	0.817 \uparrow	0.756 \uparrow	0.857 \uparrow	0.722 \uparrow	0.774 \downarrow	0.695 \downarrow	0.262 \downarrow	0.238 \downarrow
0.03	0.829\uparrow	0.762 \uparrow	0.857 \uparrow	0.725 \uparrow	0.774 \downarrow	0.683 \downarrow	0.254 \downarrow	0.233 \downarrow
0.05	0.829\uparrow	0.768\uparrow	0.852 \uparrow	0.720 \uparrow	0.774 \downarrow	0.701 \downarrow	0.246 \downarrow	0.214 \downarrow
0.10	0.829\uparrow	0.756 \uparrow	0.857 \uparrow	0.720 \uparrow	0.787	0.707\downarrow	0.275\downarrow	0.238 \downarrow
0.20	0.823 \uparrow	0.762 \uparrow	0.852 \uparrow	0.725 \uparrow	0.652 \downarrow	0.585 \downarrow	0.214 \downarrow	0.193 \downarrow
0.30	0.823 \uparrow	0.756 \uparrow	0.857 \uparrow	0.728\uparrow	0.439 \downarrow	0.415 \downarrow	0.212 \downarrow	0.204 \downarrow
0.50	0.829\uparrow	0.768\uparrow	0.862\uparrow	0.728\uparrow	0.311 \downarrow	0.293 \downarrow	0.259 \downarrow	0.246\downarrow

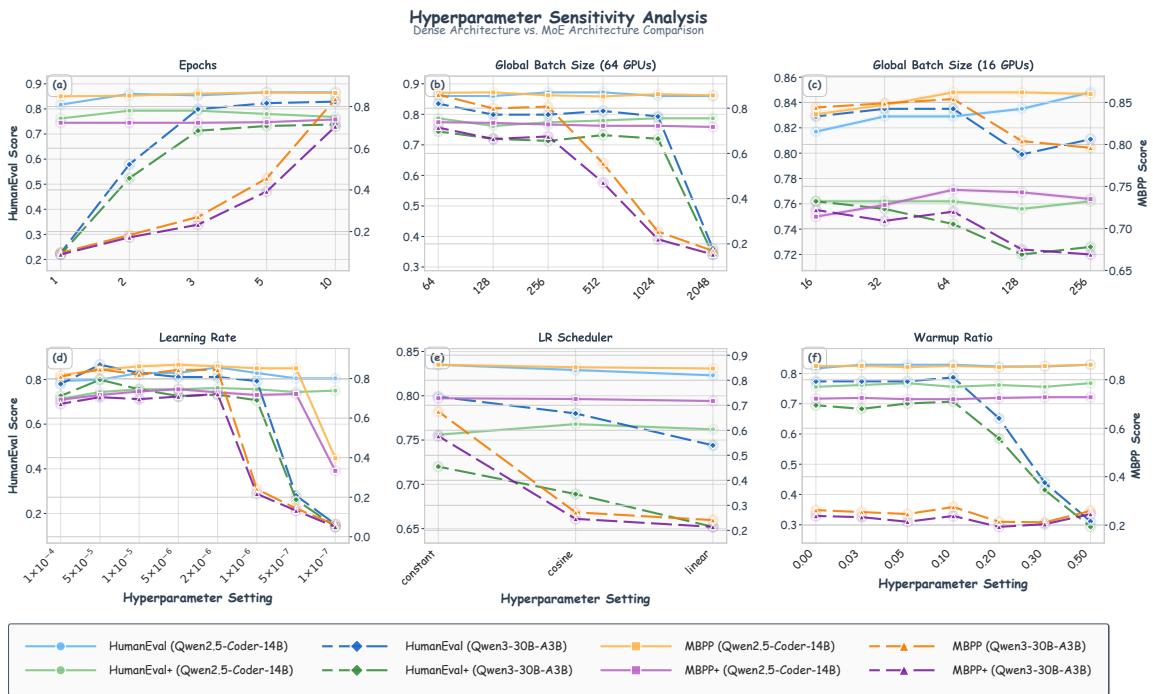


Figure 43. Hyperparameter Sensitivity Analysis for Dense and MoE Architectures. Comparison between Qwen2.5-Coder-14B (dense) and Qwen3-30B-A3B (Mixture of Experts) models across key hyperparameter dimensions, including (a) training epochs, (b) batch size, (c) learning rate, (d) scheduler type, (e) warmup ratio, and (f) global batch scaling. Each subplot reports execution pass rates on HumanEval, HumanEval+, MBPP, and MBPP+. The dense model demonstrates smoother and more stable performance across varying configurations, whereas the MoE model exhibits higher variance and sharper sensitivity to optimization choices.

ten epochs, compensates for slower adaptation, suggesting that sparse expert architectures require extended optimization horizons to reach the performance level of dense counterparts in code understanding and synthesis tasks. Moreover, cosine and linear schedulers tend to underperform relative to constant schedules, likely due to routing instability in MoE layers when the learning rate decays too aggressively.

Overall, these results highlight a clear architectural distinction. Dense transformers, such as Qwen2.5-Coder-14B, deliver consistent scaling behavior and predictable convergence with moderate tuning effort, offering reliability in compute-constrained environments. In contrast, MoE systems like Qwen3-30B-A3B, while possessing higher representational capacity, exhibit more fragile optimization landscapes. They benefit from fine-grained learning-rate control and prolonged training epochs but suffer from instability under large-batch regimes. For practical supervised fine-tuning of code models, dense architectures remain more sample-efficient and easier to stabilize, whereas MoE backbones require precise tuning to fully exploit their conditional computation potential.

Code LLMs Dataset Comparison As summarized in [Table 27](#), Qwen2.5-Coder-14B achieves higher absolute scores than Qwen3-30B-A3B, yet the relative ranking of datasets remains broadly consistent across backbones. Execution-grounded, function-level supervision transfers most effectively to MBPP and MBPP+—for instance, KodCode/KodCode-V1 yields strong function synthesis performance on the 14B model. In contrast, contest-style corpora primarily enhance HumanEval and HumanEval+ benchmarks but contribute less to MBPP, as exemplified by deepmind/code_contests, which benefits algorithmic reasoning more than entry-level function generation.

Purely instructional chat datasets lacking executable feedback provide modest gains on HumanEval but consistently underperform on MBPP+ (e.g., cfahlgren1/react-code-instructions on the 14B model), highlighting the importance of runnable supervision. Concise, edit-oriented corpora (e.g., mlfoundations-dev/stackexchange_codegolf) offer complementary regularization and yield competitive MBPP performance.

Across all configurations, the “(+)” benchmark variants consistently reduce accuracy, though the degradation is smaller when training data already encodes execution or unit-test feedback. Under a fixed 50K-sample budget, results indicate that prioritizing datasets with explicit executable or test-based supervision yields the most robust transfer, while a limited inclusion of contest-style data can further improve HumanEval-type reasoning. Overall, curating supervision quality delivers larger gains than scaling raw data volume.

8.4. Reinforcement Learning Training Guidelines

As detailed in the previous sections, supervised fine-tuning is highly effective at teaching a model to imitate the distribution of a given dataset. However, for tasks like code generation, “correctness” does not only refer to styles or expressions as in many of the general instruction-following tasks, where the training targets are objective and verifiable outcomes such as unit test-based objectives. To optimize the model directly for this verifiable correctness, rather than just mimicking reference solutions, we turn to reinforcement learning [582]. However, the best practices for applying RL to the code generation domain for LLM remain less established since. While recent studies have begun to formalize the methodology for scaling RL compute [488], a systematic study is needed to validate these findings and derive specific guidelines for code-domain LLMs. This section transitions from the SFT paradigm of “learning from examples”

Table 27. Single-dataset SFT comparison with two models placed side-by-side. Metrics are execution pass rates on HumanEval, HumanEval+, MBPP, and MBPP+. All experiments are conducted under a consistent configuration: learning rate 2×10^{-6} , global batch size 2048, warmup ratio 0.05, and maximum sequence length 8192. Bold values indicate best-performing configurations; italicized values denote second-best results per column.

Dataset	Qwen3-30B-A3B				Qwen2.5-Coder-14B			
	HumanEval	HumanEval+	MBPP	MBPP+	HumanEval	HumanEval+	MBPP	MBPP+
codeparrot/apps [199]	0.341	0.335	0.336	0.307	0.762	0.689	0.831	0.709
mlfoundations-dev/stackexchange_codereview [701]	0.354	0.329	0.357	0.328	0.841	0.774	0.315	0.257
namcdn-ai/tiny-codes [22]	0.293	0.274	0.360	0.328	0.829	0.744	0.466	0.397
bigcode/committapckft [110]	0.378	0.366	0.376	0.341	0.774	0.701	0.825	0.698
deepmind/code_contests [229]	0.366	0.341	0.376	0.331	0.823	0.762	0.235	0.198
SenseLLM/ReflectionSeq-GPT [874]	0.354	0.341	0.370	0.328	0.841	0.780	0.267	0.220
MatrixStudio/Codeforces-Python-Submissions [683]	0.329	0.305	0.360	0.328	0.774	0.695	0.833	0.712
Magpie-Align/Magpie-Qwen2.5-Coder-Pro-300K-v0.1 [677]	0.335	0.323	0.381	0.344	0.860	0.774	0.841	0.712
bigcode/self-oss-instruct-sc2-exec-filter-50k [111]	0.378	0.360	0.362	0.333	0.835	0.756	0.259	0.217
PrimeIntellect/real-world-swe-problems [804]	0.396	0.372	0.378	0.341	0.854	0.780	0.270	0.230
lvwerra/stack-exchange-paired [663]	0.311	0.299	0.357	0.325	0.799	0.720	0.817	0.704
cfahlgren1/react-code-instructions [140]	0.293	0.274	0.368	0.341	0.854	0.787	0.310	0.257
PrimeIntellect/stackexchange-question-answering [803]	0.311	0.293	0.347	0.320	0.774	0.689	0.825	0.701
PrimeIntellect/SYNTHETIC-2-SFT-verified [805]	0.366	0.341	0.384	0.365	0.841	0.768	0.267	0.225
bugdaryan/sql-create-context-instruction [125]	0.317	0.293	0.368	0.339	0.854	0.780	0.275	0.233
mlfoundations-dev/stackexchange_codegolf [700]	0.396	0.378	0.484	0.434	0.866	0.799	0.841	0.714
nvidia/OpenCodeReasoning [741]	0.360	0.341	0.415	0.365	0.762	0.689	0.844	0.709
KodCode/KodCode-V1 [494]	0.384	0.360	0.394	0.354	0.848	0.756	0.854	0.720
QuixiAI/dolphin-coder [823]	0.354	0.335	0.368	0.339	0.683	0.591	0.722	0.590
m-a-p/Code-Feedback [666]	0.360	0.329	0.368	0.331	0.835	0.774	0.844	0.714
Multilingual-Multimodal-NLP/McEval-Instruct [715]	0.354	0.354	0.360	0.317	0.854	0.793	0.870	0.743
OpenCoder-LLM/opc-sft-stage2 [761]	0.329	0.311	0.402	0.357	0.854	0.799	0.860	0.735
ajibawa-2023/Code-290k-ShareGPT [28]	0.360	0.348	0.357	0.336	0.835	0.787	0.852	0.730
christopher/rosetta-code [190]	0.323	0.299	0.336	0.307	0.695	0.634	0.823	0.701
glaiveai/glaive-code-assistant-v3 [327]	0.372	0.348	0.344	0.315	0.835	0.762	0.839	0.712
prithivMLmods/Coder-Stat [806]	0.341	0.323	0.413	0.352	0.787	0.720	0.820	0.712
ise-uiuc/Magiccoder-OSS-Instruct-75K [438]	0.421	0.402	0.373	0.341	0.835	0.762	0.847	0.717

to the RL paradigm of “learning from outcomes” We detail a suite of experiments designed to identify the most effective and scalable training practices for RL on code, using a verifiable reward (RLVR) setup.

Our experiments are grounded in a standardized default configuration and conducted with the VERL training framework ¹⁵. We utilize the codecontest_plus dataset, which provides a rich set of coding problems. The reward signal is generated by sandboxfusion¹⁶, a verifier that executes generated code against test cases. All experiments are run on a cluster of 64 H20 GPUs, employing FSDP2 for distributed training without parameter or optimizer offloading by default. The default policy gradient update uses a batch size of 64, and the maximum response length is set to 4096 tokens.

Group 1: Validating Advantage Estimators The choice of an advantage estimator is fundamental to the stability and sample efficiency of policy gradient algorithms [582]. This experiment ablates various estimators, including grp0 [880], rloo [14], reinforce_plus_plus_baseline [407] (rf++baseline), and grp0_passk [940]. All runs in this group utilize 16 rollouts per prompt with a maximum response length of 4K tokens. As shown in Figure 44 and Figure 45, the rloo estimator achieves the best Pass@5 performance (0.389 at step 400), demonstrating superior sample efficiency when leveraging multiple responses per prompt. The rloo also attains the highest Pass@1 score (0.322) among all estimators, outperforming rf++baseline which reaches 0.318 at step 280. However, rf++baseline converges approximately 30% faster (280 vs 400 steps) while maintaining competitive performance on both metrics (Pass@1: 0.318, Pass@5: 0.356), exhibiting more stable and monotonic training dynamics throughout. The grp0 estimator shows

¹⁵We also plan to extend and compare the results from more training frameworks as the differences derived from the infrastructure may heavily change the stability and outcomes, e.g., a certain level of mismatch could be solved by switching from bf16 to fp16 [808].

¹⁶<https://bytedance.github.io/SandboxFusion/>

slower convergence (step 480, Pass@1: 0.301, Pass@5: 0.371), while grpo_passk significantly underperforms (Pass@1: 0.274, Pass@5: 0.328). Based on these results, we adopt rf++baseline as the default estimator for subsequent experiments due to its favorable balance of stability, convergence speed, and competitive performance, making it more practical for large-scale training scenarios where wall-clock time is critical.

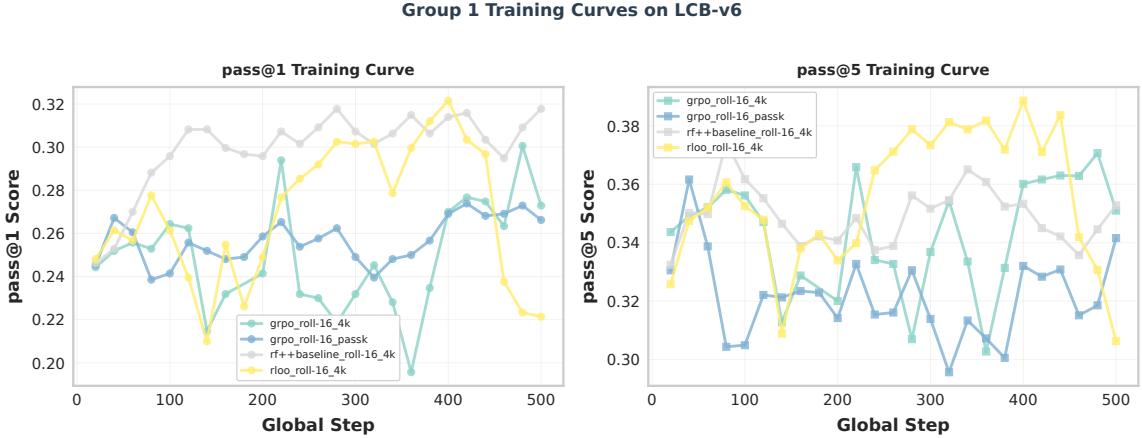


Figure 44. Comparison of training dynamics for different advantage estimators. The reinforce_plus_plus_baseline is used as the default for subsequent groups, pending results.

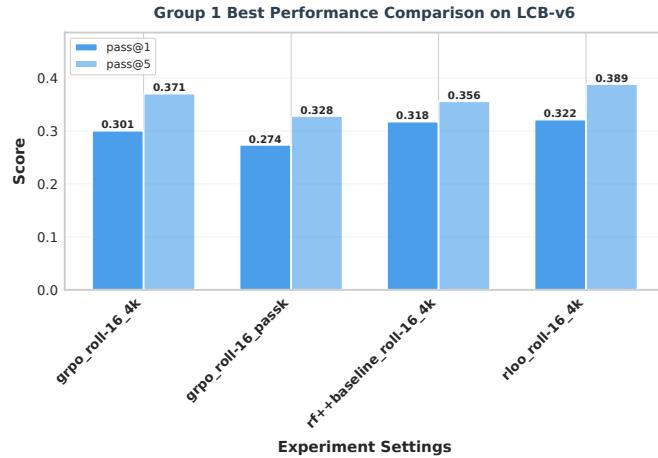


Figure 45. Best checkpoint performance comparison for different advantage estimators (Group 1).

Group 2: Scaling Maximum Response Length Code generation tasks can require vastly different context lengths. This experiment, based on the reinforce_plus_plus_baseline estimator with 16 rollouts per prompt, investigates the impact of MAX_RESPONSE_LENGTH by sweeping values from 1K to 30K tokens. As illustrated in [Figure 46](#) and [Figure 47](#), we observe a complex non-monotonic relationship between response length and performance. The 16K token configuration achieves the highest Pass@1 score (0.336 at step 340), suggesting that extended context capacity enables the model to generate more complete and correct solutions for complex problems. Notably, the 2K token setting achieves the best Pass@5 performance (0.398 at step 380), indicating that shorter contexts may promote more diverse exploration during training, possibly by encouraging the model to learn more compact solution strategies. The 1K, 4K,

and 30K configurations show similar Pass@1 performance (around 0.307-0.322), while the 8K token configuration exhibits an unexpected performance drop (Pass@1: 0.311, Pass@5: 0.334), which we attribute to a challenging transition region where models struggle to effectively utilize the additional context without proper scaling of training dynamics. This study aims to understand the trade-off between performance (as longer contexts may be necessary for complex problems) and computational cost. As sequence length increases, we adapt our infrastructure to mitigate OOM and slow inference, introducing actor gradient checkpointing, parameter and optimizer offloading, tensor parallelism (TP), and scaling the number of training nodes. Based on these results, we recommend 2K tokens for exploration-heavy objectives targeting Pass@5 performance, 16K tokens when maximizing single-pass correctness (Pass@1), and 4K tokens as a balanced default offering reasonable performance on both metrics while minimizing computational overhead.

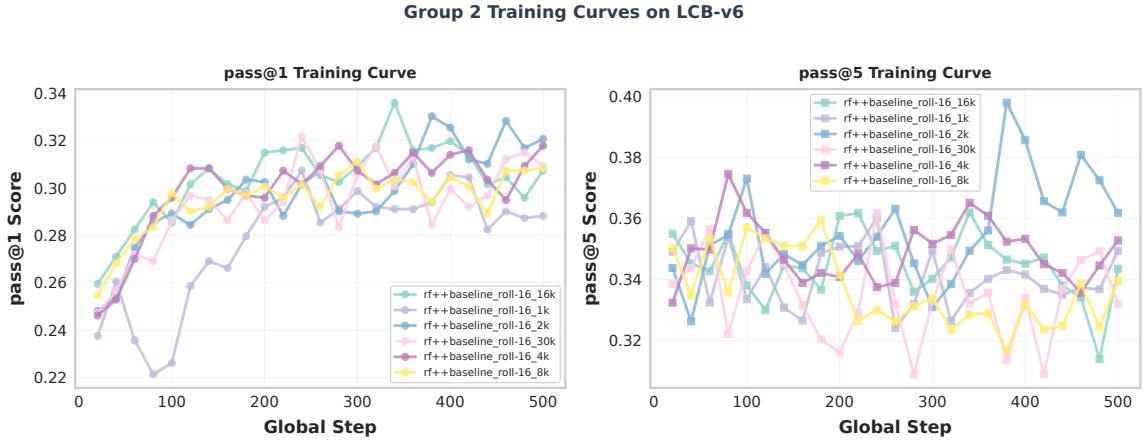


Figure 46. Impact of maximum response length on training performance and stability. Note that the 4K run is shared with Group 1.

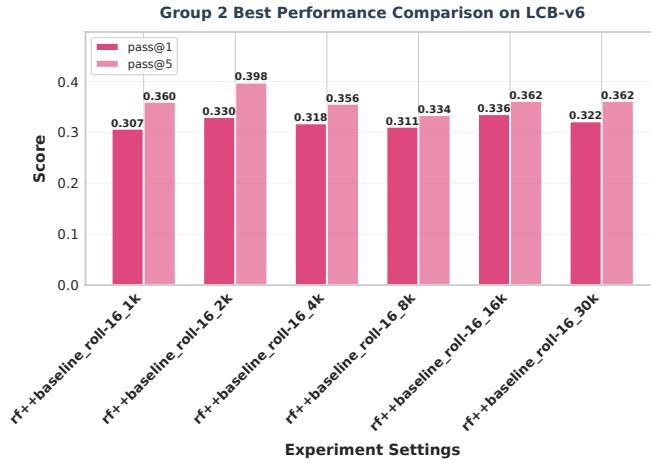
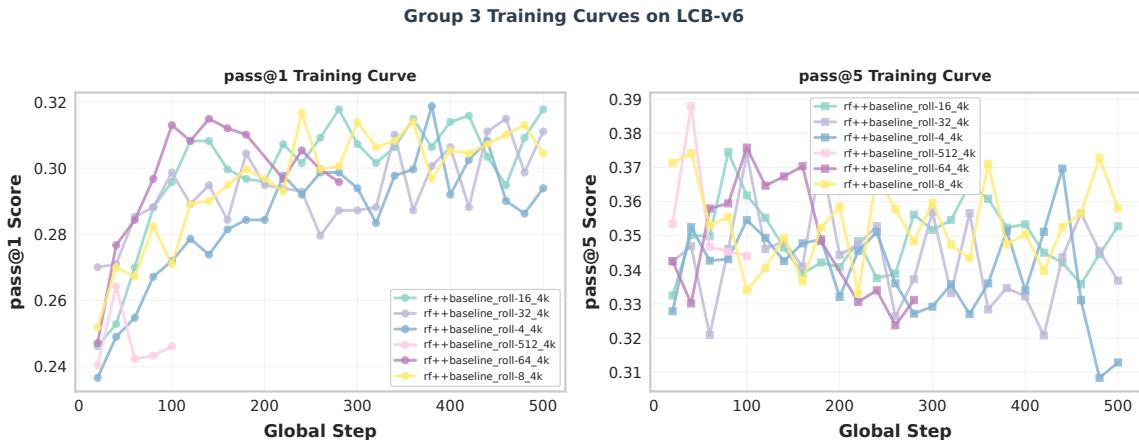


Figure 47. Best checkpoint performance comparison for different maximum response lengths (Group 2).

Group 3: Scaling Rollouts per Prompt The number of responses sampled per prompt (`N_RESP_PER_PROMPT`) directly influences the “width” of exploration and the quality of the advantage estimation [409]. Using the `reinforce_plus_plus_baseline` estimator with a fixed 4K token

response length, we sweep this value from 4 to 512. As depicted in [Figure 48](#) and [Figure 49](#), the results reveal a nuanced trade-off between exploration width and training efficiency. The N=512 configuration achieves the highest Pass@5 score (0.388 at step 40), but this comes with prohibitively slow convergence—requiring significantly fewer update steps but much longer wall-clock time per step due to the massive rollout generation overhead, making it impractical for most training scenarios. Moreover, the extremely large rollout size setting *causes training collapse counterintuitively* (where same observations happen in N={128,256}), suggesting a missing puzzle for the blueprint of rollout scaling. Among more practical configurations, N=8 delivers the best Pass@5 performance (0.368) while N=64 achieves slightly lower Pass@5 (0.367) but comparable results. For Pass@1 performance, the results show remarkable stability: N=4 achieves the highest score (0.319), followed by N=8 (0.317), N=16 (0.318), and N=32/N=64 (both 0.315), with all configurations within a narrow 0.004 range. This suggests that moderate rollout numbers suffice for optimizing single-sample correctness, and increasing exploration width primarily benefits multi-sample diversity rather than individual solution quality. The N=32 configuration exhibits slower convergence (step 460) without clear performance advantages over smaller values. This experiment quantifies the relationship between sample efficiency and compute, determining the point of diminishing returns for exploration width. Based on these observations, we recommend N=16 as the default configuration, offering an excellent balance between convergence speed, computational efficiency, and competitive performance on both metrics. For Pass@5-critical applications with sufficient budget, N=8 provides the best diversity-performance trade-off among practical configurations.



[Figure 48](#). Impact of N_RESP_PER_PROMPT on training performance. Note that the n=16 run is shared with Group 1.

Summary of Best Practices. This comprehensive experimental suite (summarized in [Table 28](#) and visualized in [Figure 50](#)) provides concrete, data-driven “best practices” for applying RL to code generation LLMs. Based on the experimental outcomes across Groups 1-3, we provide the following recommendations:

(1) **Advantage Estimator:** For practical large-scale training scenarios, rf++baseline is the recommended default, offering the best balance of training stability, convergence speed (step 280), and competitive performance (Pass@1: 0.318, Pass@5: 0.356). While rloo achieves superior performance on both metrics (Pass@1: 0.322, Pass@5: 0.389), it requires approximately 43% more training steps (step 400), making rf++baseline more practical when wall-clock time is critical. For scenarios where maximum performance is prioritized over training efficiency, rloo is the optimal choice.

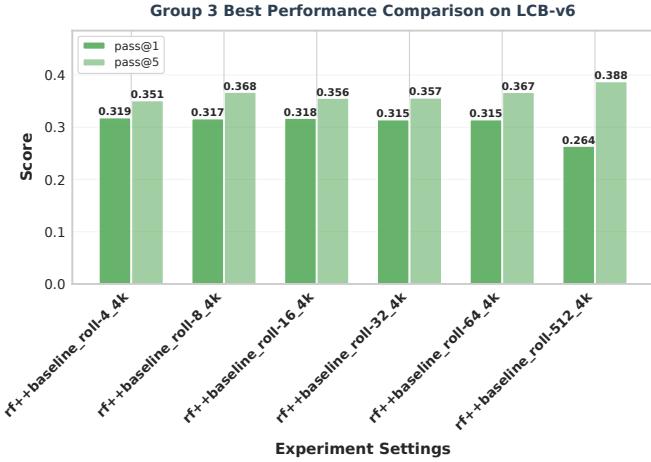


Figure 49. Best checkpoint performance comparison for different rollout numbers per prompt (Group 3).

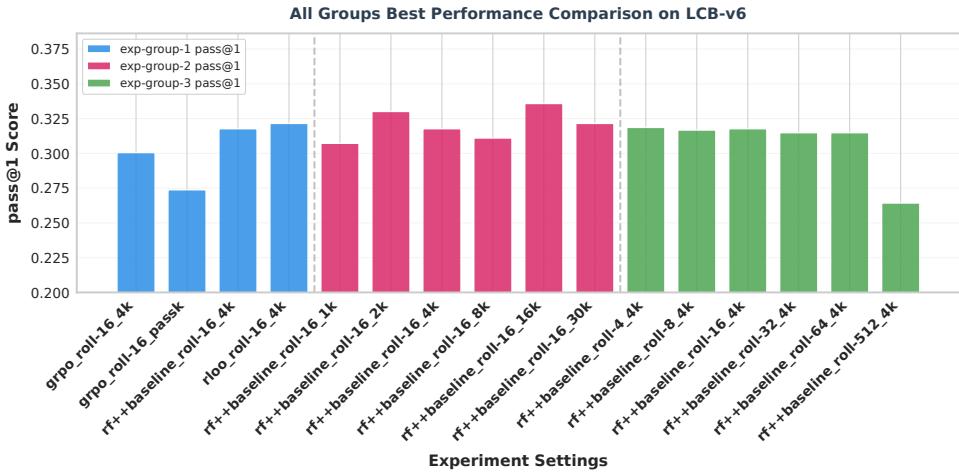


Figure 50. Comprehensive comparison of all experimental groups on the lcb-v6 benchmark, showing the training dynamics across different advantage estimators (Group 1), maximum response lengths (Group 2), and rollout numbers per prompt (Group 3).

(2) Response Length: Performance exhibits a complex non-monotonic relationship with context length, revealing task-specific optima. For exploration-heavy objectives optimizing Pass@5, use 2K tokens (Pass@5: 0.398), which promotes diverse solution strategies and achieves the highest multi-sample correctness. For maximizing single-pass correctness (Pass@1), 16K tokens is optimal (Pass@1: 0.336), enabling complete solutions to complex problems. We recommend 4K tokens as a balanced default (Pass@1: 0.318, Pass@5: 0.356), offering reasonable performance on both metrics while minimizing computational overhead. The 8K configuration exhibits an unexpected performance valley (Pass@1: 0.311, Pass@5: 0.334) and should be avoided. Response lengths beyond 16K provide diminishing returns while substantially increasing computational cost.

(3) Rollouts per Prompt: N=16 provides the recommended default configuration, offering an excellent compute/performance balance (Pass@1: 0.318, Pass@5: 0.356) with fast convergence (step 280). For Pass@5-critical applications with sufficient budget, N=8 delivers the best diversity-performance trade-off among practical configurations (Pass@5: 0.368). Notably,

Table 28. Overall Performance Summary Table on lcb-v6 benchmark. This table presents the best checkpoint performance from each experimental group. Pass@1 and Pass@5 metrics are reported for the lcb-v6 evaluation set.

Group	Configuration	Step	Pass@1	Pass@5
Group 1: Advantage Estimators (roll=16, 4K tokens)				
rloo		400	0.322	0.389
rf++baseline (default)		280	0.318	0.356
grpo		480	0.301	0.371
grpo_passk		420	0.274	0.328
Group 2: Max Response Length (rf++baseline, roll=16)				
1K tokens		240	0.307	0.360
2K tokens		380	0.330	0.398
4K tokens		280	0.318	0.356
8K tokens		300	0.311	0.334
16K tokens		340	0.336	0.362
30K tokens		240	0.322	0.362
Group 3: Rollouts per Prompt (rf++baseline, 4K tokens)				
N=4 Rollouts		380	0.319	0.351
N=8 Rollouts		240	0.317	0.368
N=16 Rollouts		280	0.318	0.356
N=32 Rollouts		460	0.315	0.357
N=64 Rollouts		140	0.315	0.367
N=512 Rollouts		40	0.264	0.388

Pass@1 performance remains remarkably stable across N=4 to N=64 (range: 0.315-0.319, variance < 0.004), indicating that exploration width primarily benefits multi-sample diversity rather than individual solution quality. While N=512 achieves the highest Pass@5 (0.388), the wall-clock time per step becomes prohibitively expensive, making it impractical for most training scenarios. We do not recommend N=32 or N=64 for standard use cases, as they show minimal performance gains over N=8 or N=16 while requiring substantially more computation.

These guidelines, validated on the lcb-v6 benchmark using the codecontest_plus dataset with 64 H20 GPUs and FSDP2 distributed training, will be crucial for enabling researchers to scale RL for code LLMs more effectively and predictably. The key insight across all experiments is that optimal hyperparameter choices depend strongly on the target metric: Pass@1 optimization benefits from extended context (16K tokens) and stable estimators (rf++baseline), while Pass@5 optimization favors compact contexts (2K tokens), higher exploration (N=8 rollouts), and sample-efficient estimators (rloo).

9. Code Large Language Model for Applications

The rapid evolution of code-capable LLMs has driven a paradigm shift in software development, transitioning from research prototypes to production-ready tools integrated across the entire software development lifecycle [403, 1294]. This section presents a comprehensive taxonomy of code LLM applications [381, 455, 1299], categorizing them into six primary application domains based on their architectural patterns Figure 51, deployment strategies, and functional capabilities. Each application is analyzed in terms of its historical development, core capabilities, technical innovations, and current limitations.

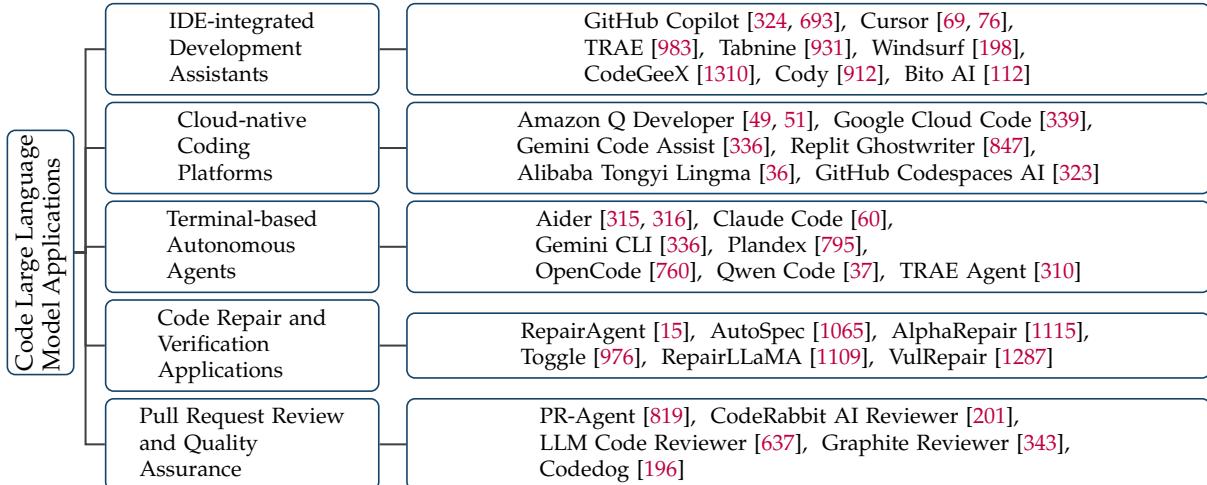


Figure 51. Code Large Language Model Applications.

9.1. IDE-integrated Development Assistants

IDE-integrated assistants represent the most widely deployed category of code LLM applications, seamlessly embedding AI capabilities within established development environments such as Visual Studio Code, JetBrains IDEs, and proprietary editors. These tools target professional developers working on medium to large-scale projects, emphasizing deep contextual understanding [790], cross-file reasoning, and integration with existing software engineering workflows [1285].

GitHub Copilot GitHub Copilot emerged as the pioneering commercial IDE-integrated AI coding assistant, initially launched in technical preview on June 29, 2021 [324]. Built upon OpenAI’s Codex model (a descendant of GPT-3 trained on public code repositories), Copilot fundamentally transformed developer expectations for AI-assisted programming. The system evolved significantly from its initial single-model architecture: in November 2023, the chat interface transitioned to GPT-4, and by October 2024, GitHub announced multi-model support [325] enabling developers to select between GPT-4o, Claude 3.5 Sonnet, and Gemini 1.5 Pro based on task requirements. As of 2025, Copilot defaults to GPT-4.1 across chat, agent mode, and code completions, with Pro+ and Enterprise tiers providing access to frontier models including Claude Opus 4 and GPT-5 preview [326]. The architecture comprises a sophisticated Node.js-based agent process that handles web requests to GitHub services, context collation from active files and repository structure, prompt engineering for the LLM, and post-processing including content exclusion filters [693]. Key recent innovations include Copilot Edits for multi-file refactoring with iterative change plans, Copilot Workspace for end-to-end feature development using natural language, and the May 2025 introduction of “coding agent” mode [999] enabling asynchronous task execution with automated pull request generation. The system processes approximately 150 million code suggestions daily with sub-second latency requirements [69]. GitHub Copilot has achieved widespread adoption with over 1.8 million individual paying subscribers and tens of thousands of enterprise customers, generating approximately \$500 million in revenue in 2024 [76]. However, the system faces ongoing challenges including concerns over training data licensing (trained on public repositories without explicit permission), potential code suggestion plagiarism from training data, privacy issues regarding telemetry and keystroke data collection, and limited offline capabilities [1073].

Cursor Cursor represents a strategic rethinking of AI integration in development environments. Founded in 2022 by Michael Truell, Sualeh Asif, Arvid Lunnemark, and Aman Sanger, Cursor launched its first version in March 2023 as a fork of Visual Studio Code [69], strategically balancing familiarity with innovation. Rather than building an IDE from scratch, this architectural decision enabled the team to leverage VS Code’s mature foundation and familiar interface while dedicating all engineering efforts to deep AI integration—transforming how developers interact with code rather than rebuilding basic editor infrastructure. The system has experienced explosive growth, crossing \$500 million in annual recurring revenue (ARR) within just two years of launch. Cursor’s technical architecture centers on proprietary innovations in context management and model inference. The “tab model” provides ultra-low-latency inline completions (under one second) by implementing a sophisticated sync engine that transmits encrypted code snippets to cloud servers for inference without persistent storage, maintaining privacy while handling over one million queries per second [126]. Repository indexing employs Merkle trees to track file changes efficiently, enabling incremental updates every three minutes without full reindexing. The system supports context windows up to 200K tokens through dynamic context prioritization using tree-sitter-based AST analysis. Cursor introduced several breakthrough features: Composer Mode enables natural language project-wide edits across multiple files with automated dependency tracking; Rules and Memories allow developers to encode project-specific conventions and maintain conversation continuity across sessions; and Agent Mode (released with Cursor 1.0 in 2025) provides autonomous coding capabilities [38, 70] with background execution and browser automation for UI debugging. The system supports multiple frontier models including GPT-4.1, Claude Sonnet 4.5, Gemini 2.5 Pro, and xAI’s Grok Code, with automatic model selection based on task characteristics [71]. Despite its rapid adoption by over half of Fortune 500 companies and exceptional developer satisfaction ratings, Cursor faces criticism for inconsistent code quality in complex refactoring tasks, occasional suggestion lag in large codebases, higher cost (\$20/month vs \$10 for Copilot), and the potential for vendor lock-in through proprietary features [981].

TRAЕ TRAE [983] represents a new generation of AI-native integrated development environments that pursue a paradigm shift from assistance toward autonomous software creation. The product manifests the vision through two primary modes - Builder Mode and SOLO Mode, enabling everything from specification-driven project scaffolding to conversational code editing and multi-modal input.

The SOLO Mode feature launched in mid-2025 is described as an all-in-all Context Engineer that thinks in context, works across tools, and works with developers to ship real features from start to finish. In practical terms, a user can issue a natural language specification and TRAE will (a) parse the requirement, (b) decompose it into tasks, (c) generate scaffolding, code, tests and deployment glue, and (d) preview changes before applying them. This think-then-do pattern emerges as a distinguishing workflow: instead of immediate line-by-line completion, TRAE emphasizes planning, decomposition and then change-application.

The CUE feature supports auto-completion, multi-line edits, predicted edits, jump-to edits (i.e. moving the cursor to the next affected region), smart import and smart rename. Unlike simpler completion tools that only look at local buffer context, CUE takes into account a broader workspace-level context: project structure, symbols across files, dependency graphs, and prior developer edits.

Tabnine Tabnine, launched in March 2021, distinguished itself in the crowded AI coding assistant market through its focus on privacy, security, and intellectual property compliance [931]. Unlike competitors trained on public repositories with ambiguous licensing, Tabnine exclusively uses permissively licensed code (MIT, Apache, BSD) for model training, directly addressing enterprise concerns about code ownership and legal liability. The platform's core differentiator is its flexible deployment architecture: while most competitors require cloud connectivity, Tabnine offers on-premises and local deployment options where models run entirely within the organization's infrastructure, ensuring proprietary code never leaves the corporate network. This approach resonates particularly with regulated industries (finance, healthcare, government) and companies with strict data sovereignty requirements. Tabnine's technical architecture emphasizes customization through fine-tuning on private codebases, enabling the model to learn organization-specific patterns, APIs, and coding standards without exposing this data externally. The system provides multi-IDE support (VS Code, IntelliJ, JetBrains suite, Eclipse, Visual Studio) with consistent behavior across platforms. Recent enhancements include team learning capabilities where the model improves based on accepted suggestions across the development team, semantic code completion understanding broader context beyond syntactic patterns, and compliance reporting for audit trails. However, Tabnine's conservative training approach results in somewhat lower suggestion quality compared to competitors trained on larger, less-restricted datasets. The system also commands premium pricing for enterprise features (\$39/user/month for Pro) and exhibits slower feature development velocity compared to heavily-funded competitors like Copilot and Cursor.

Windsurf Windsurf emerged in November 2024 as Codeium's flagship AI-native IDE, introducing the novel cascade architecture [198] for deep codebase understanding and multi-agent coordination. The Cascade system implements a flow-based approach where multiple specialized sub-agents collaborate on complex refactoring tasks: one agent maps the codebase structure, another identifies relevant files and dependencies, a third generates modifications, and a fourth validates changes. This hierarchical decomposition enables handling of architectural transformations that single-agent systems struggle with, such as migrating from REST to GraphQL across dozens of files or refactoring monolithic applications into microservices. Windsurf's context management employs a hybrid approach combining vector search for semantic similarity, AST-based structural analysis for precise symbol references, and graph-based dependency tracking for impact analysis. The system maintains persistent codebase indexes that update incrementally, reducing reindexing overhead. Advanced features include flows (reusable multi-step automation sequences), rules for enforcing team coding standards, and memories for preserving project-specific knowledge. However, as the newest entrant in the AI IDE space, Windsurf faces challenges including limited ecosystem maturity, a smaller user community compared to established competitors, potential performance issues with the multi-agent overhead in large codebases, and uncertainty about long-term platform stability.

Additional IDE-Integrated Systems The IDE assistant landscape includes several other notable systems. **CodeGeeX**, launched in September 2022, prioritizes multilingual support with strong performance in Chinese programming contexts and open-source availability [1310]. **Cody** by Sourcegraph (August 2023) integrates deeply with code search infrastructure [912] for superior context retrieval, particularly in large monorepos. **Bito AI** emphasizes developer privacy with offline operation modes and focuses on code explanation and test generation beyond simple completion [112]. These alternatives serve specialized niches but struggle to compete with the network effects and rapid development pace of market leaders.

9.2. Cloud-native Coding Platforms

Cloud-native platforms leverage scalable infrastructure to provide code generation services through APIs, web interfaces, and cloud-integrated development environments. These systems target organizations requiring centralized deployment, consistent security policies, and deep integration with cloud services [50, 340], particularly for infrastructure-as-code and cloud-native application development.

Amazon Q Developer Amazon Q Developer represents the evolution and rebranding of Amazon CodeWhisperer, which originally launched in June 2022 as AWS's answer to GitHub Copilot [49]. In April 2024, AWS merged CodeWhisperer into the broader Amazon Q family [1072], significantly expanding the service's scope beyond code completion to encompass comprehensive AWS development workflows. The strategic rebranding reflected AWS's recognition that code generation alone was insufficient; developers needed an integrated assistant understanding the full AWS ecosystem. Amazon Q Developer distinguishes itself through deep AWS service specialization: the system was trained on millions of internal Amazon code repositories, AWS documentation, and service implementations [51], providing superior generation quality for CloudFormation templates, CDK constructs, and Terraform configurations compared to general-purpose models. The architecture integrates across the development lifecycle: inline code suggestions in IDEs (JetBrains, VS Code, Visual Studio, and Eclipse), chat interfaces in the AWS Console for resource queries, such as "List all Lambda functions" or "What were my top three cost drivers in Q1?", CLI autocompletion for AWS commands, and chat integrations in Microsoft Teams and Slack for operational support [246]. Key capabilities include security scanning for vulnerability detection, automated Java version upgrades, .NET porting from Windows to Linux, and AWS cost optimization recommendations. The service introduces "Agent" functionality, enabling autonomous multi-step tasks like "deploy" this application to ECS with auto-scaling and monitoring. As of 2025 ,Amazon Q Developer offers a perpetual free tier (50 agent interactions monthly, 1000 lines of code transformation), with paid tiers (\$19/month Pro) providing higher limits and IP indemnity protection where AWS defends customers against copyright infringement claims.

Google Cloud Code and Gemini Code Assist Google's cloud coding offerings evolved from Google Cloud Code (May 2023) to Gemini Code Assist, leveraging Google's the latest Google models family [339] for code generation optimized for Google Cloud Platform (GCP) and Kubernetes. The system excels at generating Google Kubernetes Engine (GKE) configurations, Cloud Run deployments, and BigQuery SQL with deep understanding of GCP service constraints and best practices. Gemini CLI, introduced in November 2024, provides terminal-native development [336] with fast inference through local caching and incremental parsing for large repositories. The architecture emphasizes enterprise features including data residency controls ensuring customer code never leaves specified geographic regions, integration with Google Workspace for team collaboration, and comprehensive audit logging for compliance. Advanced capabilities include multi-modal code generation (accepting diagrams and screenshots as input), Cloud Workstation integration for cloud-based development environments, and Duet AI in databases for SQL optimization. However, the platform faces adoption challenges including later market entry relative to competitors, limited ecosystem integration outside Google Cloud, uncertainty around product strategy given Google's history of discontinuing services, and questions about long-term commitment given leadership changes in Google's AI organization.

Replit Ghostwriter Replit Ghostwriter, launched in October 2022, pioneered browser-based AI-assisted development with custom small language models [847] achieving competitive performance through architectural innovations rather than parameter scale. The replit-code-v1.5-3b model (3 billion parameters, trained on 1 trillion tokens) demonstrates that focused training on high-quality code with careful data curation and optimized architectures can rival much larger models on specific tasks. Ghostwriter’s browser-native architecture eliminates local installation requirements, enabling instant access to AI coding assistance from any device with a web browser. The platform integrates tightly with Replit’s collaborative multiplayer editing, enabling teams to code together with shared AI assistance in real-time. Key features include Ghostwriter Chat for conversational code help, explain code for understanding unfamiliar patterns, generate code for implementing features from descriptions, and complete code for inline suggestions. The browser-based execution environment supports immediate testing and deployment, creating a seamless cycle from generation to validation. As of 2025, Replit’s freemium model provides limited free access with generous paid tiers (\$20/month) including unlimited AI interactions. The platform particularly appeals to educators and learners through simplified onboarding and extensive educational resources. However, limitations include dependency on internet connectivity with no offline mode, limited support for complex enterprise workflows and large monorepo architectures, performance constraints in the browser environment for resource-intensive applications, and potential data privacy concerns for sensitive commercial code.

Alibaba Tongyi Lingma Tongyi Lingma, launched in October 2023, serves the Chinese market with specialized support for Chinese programming contexts and Alibaba Cloud services [36]. Built on the Qwen language model family, the system provides multilingual code generation, testing, and debugging with particular strength in Chinese natural language understanding for requirements and documentation. The platform integrates deeply with Alibaba Cloud DevStudio and supports popular Chinese development tools. Lingma emphasizes compliance with Chinese data sovereignty requirements through local model hosting and processing. However, international adoption remains limited due to language barriers, market-specific optimizations, and regulatory considerations.

9.3. Terminal-based Autonomous Agents

Terminal-based agents operate in command-line environments, providing autonomous code generation, modification, and project management capabilities. These tools appeal to developers preferring keyboard-driven workflows, automation engineers building CI/CD pipelines [1184], and researchers requiring scriptable, reproducible experiments [469].

Aider Aider establishes itself as the leading open source terminal-based coding agent through pioneering work in repository mapping, code editing, and benchmark performance [316]. Its repository mapping system employs treesitter for language-agnostic AST parsing [313], generating compact summaries of entire codebases and performing graph optimization on the call graph to dynamically tailor context. Aider implements multiple code editing backends with automatic format selection, where the unified diff format provides precise line-level edit specifications that reduce ambiguity. Furthermore, Aider features an iterative execution feedback mechanism that uses treesitter-based linting [314] to detect syntax errors and automatically request fixes from the LLM with enhanced context. The system integrates seamlessly with Git for change management and supports multiple LLM providers with prompt caching for cost optimization. However, Aider’s terminal-native design limits visual debugging, lacks GUI

support for interface development, requires significant technical proficiency for effective use, and depends on external LLM APIs.

Claude Code Claude Code, announced in December 2024 by Anthropic, represents a terminal-native development environment specifically designed for the Claude model family [60]. The system implements the Model Context Protocol (MCP), an open standard enabling extensible tool integration and sub-agent coordination. Claude Code’s architecture emphasizes composability: developers can define custom tools (e.g., API clients, testing frameworks, deployment scripts) that Claude can invoke autonomously. The agent planning capability breaks complex tasks into sub-goals with explicit step generation, enabling developers to review and modify plans before execution. MCP servers enable Claude Code to access external resources (databases, file systems, APIs) through standardized interfaces with built-in security boundaries. The system supports multi-step workflows where Claude Code can research documentation, write code, execute tests, debug failures, and iterate until success—all while maintaining context across the entire process. However, as a newly released product, Claude Code faces early-stage limitations including incomplete documentation, a nascent tool ecosystem, potential reliability issues in complex scenarios, and dependence on Claude API availability and pricing.

Gemini CLI Gemini CLI, introduced in November 2024, provides terminal-native access to Google’s Gemini models with emphasis on speed and efficiency [336]. The system employs aggressive local caching strategies, storing previously indexed codebase representations to minimize recomputation. Incremental parsing updates only modified files rather than reprocessing entire repositories. Gemini CLI integrates with Google Cloud authentication for seamless access to GCP resources and supports multi-modal inputs including screenshots and diagrams in terminal workflows. The lightweight architecture minimizes memory footprint and startup latency. However, the system’s tight Google Cloud integration limits utility outside GCP environments, and the relatively new release means limited third-party tool integrations and community resources compared to more established alternatives.

Plandex Plandex, released in March 2024, implements a distinctive plan-based workflow with branching support [795] for exploring alternative implementations. Rather than immediate code generation, Plandex first creates a detailed implementation plan specifying which files to modify, what changes to make, and the order of operations. Developers can review, edit, and approve plans before execution. The branching system allows spawning multiple plan variants to compare different approaches (e.g., implementing a feature with different architectural patterns), with lightweight switching between branches. This approach suits exploratory development where the optimal solution isn’t immediately clear. However, the additional planning overhead increases latency for simple tasks, and the system’s specialized workflow may not align with all development styles.

Additional Terminal Agents Other terminal-based systems include **OpenCode** (August 2024) emphasizing open-source and multilingual support [760], and **Qwen Code** (January 2025) featuring 256K context windows and Chinese language optimization [37]. In addition, **TRAЕ Agent** [310] offers a research-friendly design for studying AI agent architectures, conducting ablation studies, and developing novel agent capabilities.

9.4. Code Repair and Verification Applications

Specialized applications for automated bug fixing [1109], vulnerability patching, and formal verification address critical software reliability needs. These tools serve security teams performing vulnerability assessment, QA engineers automating test generation, researchers in formal methods and program synthesis [15], and maintainers of legacy codebases.

RepairAgent RepairAgent, introduced in March 2024, pioneered autonomous debugging through hypothesis-driven state machine progression [15]. The system implements a structured approach for bug fixing comprising of five stages: (1) **fault localization**: identifying suspicious code regions using test failure analysis, execution traces, and statistical debugging; (2) **hypothesis generation**: proposing potential root causes for observed failures; (3) **patch generation**: creating fixes for each hypothesis; (4) **validation**: executing tests to verify correctness; and (5) **iterative refinement**: looping until validation succeeds or maximum attempts exhausted. RepairAgent’s architecture employs multiple LLM calls with specialized prompting for each phase: localization prompts focus the model on analyzing test output and stack traces, hypothesis prompts encourage considering multiple failure modes, and patch prompts constrain changes to minimize disruption. The state machine tracks attempted fixes and their results, preventing repeated failures. On the Defects4J [477] benchmark, RepairAgent successfully fixed 164 out of 357 bugs (45.9%) including 39 novel repairs not achieved by previous automated program repair (APR) tools, demonstrating significant advancement over template-based and learning-based APR systems. The hybrid approach combining program analysis (for localization precision) with LLM reasoning (for creative fix generation) proved more effective than either technique alone. However, RepairAgent exhibits limitations including high computational cost from multiple LLM invocations, difficulty with semantic bugs requiring deep domain knowledge, potential for incorrect fixes that pass tests but introduce new issues, and limited scalability to bugs requiring changes across many files.

AutoSpec AutoSpec, released in April 2024, tackles the challenging problem of automatic specification synthesis for formal verification [1065]. Rather than fixing bugs directly, AutoSpec generates formal specifications (preconditions, postconditions, loop invariants) that characterize correct behavior. Here, preconditions define the required input conditions for a function, postconditions specify the expected output or state after execution, and loop invariants capture properties that must hold before and after each loop iteration. The system employs hierarchical decomposition: complex specifications are broken into simpler sub-specifications for individual functions or loop iterations, with LLMs generating candidate specifications based on code analysis and comments. Generated specifications are validated using theorem provers (e.g., Dafny verifier [528]) to ensure logical consistency and completeness. Failed verification attempts produce counterexamples that guide refinement. This synthesis-verification loop continues until valid specifications are achieved or timeout. AutoSpec particularly excels at inferring loop invariants—traditionally one of the most challenging aspects of formal verification. The generated specifications serve multiple purposes: documentation of intended behavior, verification of correctness, and guidance for subsequent code modifications. However, AutoSpec faces significant scalability challenges with large codebases, depends heavily on code comment quality for initial specification hints, often struggle with complex specifications requiring mathematical reasoning, and inherits limitations of underlying theorem provers.

AlphaRepair AlphaRepair, introduced in January 2024, combines program analysis with LLMs [1115] for superior fault localization and patch generation. The system’s static analysis component performs control flow analysis, data flow analysis, and dependency analysis to precisely identify code regions potentially responsible for failures. Dynamic analysis instruments code to collect execution traces, variable values, and branch coverage during test execution. These analyses produce ranked lists of suspicious locations with confidence scores. The LLM then receives this context including suspicious code, execution traces, test outputs, and similar historical bugs from a patch database. AlphaRepair implements a template-guided generation strategy: rather than unrestricted code generation, the LLM selects and instantiates repair templates (e.g., “add null check”, “change condition operator”, “initialize variable”) based on bug characteristics. This constrained generation reduces hallucination and improves patch quality. Validation employs both test-based verification and static analysis to detect potential side effects. On standard APR benchmarks, AlphaRepair achieved higher correct fix rates and lower plausible-but-incorrect patch rates compared to purely learning-based approaches.

Toggle Toggle, released in April 2024, performs token-level bug localization with adjustment models [976] for refinement before repair. Rather than localizing at the statement or function level, Toggle identifies specific tokens (variable names, operators, literals) likely to be faulty. The multi-phase pipeline includes coarse localization (identifying candidate statements), fine-grained localization (pinpointing specific tokens within statements), adjustment (refining localization based on program structure and semantics), and generation (producing fixes focused on identified tokens). Token-level precision enables more targeted repairs with less risk of unintended consequences. The adjustment phase uses a separate model to filter false positives from initial localization by analyzing surrounding code context and common bug patterns. This two-stage localization significantly improves precision at the cost of additional computational overhead.

Additional Repair Systems **RepairLLaMA** (January 2024) provides fine-tuned models specifically for automated program repair [1109]. **VulRepair** (February 2024) specializes in security vulnerability patching [1287] with knowledge of CVE patterns and secure coding practices.

9.5. Pull Request Review and Quality Assurance

Pull request review assistants automate code review processes, providing feedback on code quality, potential bugs, security issues, and adherence to coding standards. These tools aim to reduce reviewer burden, accelerate review cycles, and improve consistency [637, 819].

PR-Agent (Qodo-AI) PR-Agent emerged as a leading open-source automated pull request review system [819], providing comprehensive analysis including automated PR summarization, comment generation on potential issues, security risk tagging, and reviewer suggestions based on code expertise. The system integrates as a CI/CD pipeline step, automatically triggering on PR creation or updates. PR-Agent employs multiple specialized prompts for different review aspects: architecture review analyzing structural changes and design patterns, security review identifying vulnerability patterns, style review checking adherence to conventions, and logic review examining correctness and edge cases. The generated feedback appears as GitHub/GitLab comments on specific lines or as summary reviews. PR-Agent supports self-hosted deployment for organizations requiring data sovereignty, with both GitHub Enterprise and GitLab Enterprise compatibility. The system can be customized with project-specific review rules and quality gates.

Key advantages include open-source transparency, enterprise-friendly deployment options, and active community development. However, limitations include dependency on LLM API availability and costs, potential for verbose or irrelevant feedback requiring human filtering, and challenges understanding complex business logic without extensive context.

CodeRabbit CodeRabbit provides automated PR reviews with inline suggestions, code explanations, and context-aware comments [201]. The system distinguishes itself through support for long-context diffs (handling PRs with thousands of lines), multi-file impact analysis tracing changes across dependencies, and configurable review depth allowing teams to adjust between quick surface-level checks and deep semantic analysis. CodeRabbit offers both SaaS and self-hosted deployment with support for multiple LLM providers (GPT-4, Claude, Gemini). The review process includes static analysis integration to catch common issues before LLM analysis, reducing LLM costs while maintaining thoroughness. However, the system’s effectiveness depends heavily on codebase context availability and may produce false positives in codebases with non-standard patterns.

Additional PR Review Systems **LLM Code Reviewer** provides simple GitHub Action integration [637] with configurable prompt templates for GPT, Claude, or Gemini. **Graphite Reviewer** emphasizes speed and context recall [343] through repository-aware indexing. **Codedog** offers multi-platform support (GitHub/GitLab) [196] with strong UI integration for Chinese development teams.

Conclusion

In this work, we present a comprehensive analysis of code-generating large language models across their entire lifecycle, from data curation and pre-training through supervised fine-tuning, reinforcement learning, and deployment as autonomous agents. We examine both general-purpose models (GPT-4, Claude, and LLaMA) and code-specialized models (StarCoder, CodeLLaMA, DeepSeek-Coder, and QwenCoder) while bridging the gap between academic benchmarks and real-world software development challenges through systematic experiments on scaling laws, architectures, and training methodologies. Finally, we conduct a comprehensive series of experiments analyzing code pre-training, supervised fine-tuning, and reinforcement learning across multiple dimensions, including scaling laws, framework selection, hyperparameter sensitivity, model architectures, dataset comparisons, and manual coding practices.

10. Contributions and Acknowledgements

The authors of this paper are listed in order as follows:

First Author

- Jian Yang, Beihang University

Corresponding Authors

- Xianglong Liu, Beihang University
- Weifeng Lv, Beihang University

Core Contributors (Last Name Alphabetical Order)

- Ken Deng, Kuaishou
- Shawn Guo, M-A-P
- Lin Jing, M-A-P
- Yizhi Li, University of Manchester
- Shark Liu, M-A-P
- Xianzhen Luo, Harbin Institute of Technology
- Yuyu Luo, The Hong Kong University of Science and Technology (Guangzhou)
- Changzai Pan, Institute of Artificial Intelligence (TeleAI), China Telecom
- Ensheng Shi, Huawei Cloud Computing Technologies Co., Ltd

- Yinghui Tan, Alibaba Group
- Renshuai Tao, Beijing Jiaotong University
- Zili Wang, StepFun
- Jiajun Wu, Beihang University
- Xianjie Wu, Beihang University
- Zhenhe Wu, Beihang University
- Daoguang Zan, ByteDance
- Chenchen Zhang, Tencent
- Wei Zhang, Beihang University
- He Zhu, OPPO
- Terry Yue Zhuo, Monash University & CSIRO's Data61

Contributors (Last Name Alphabetical Order)

- Kerui Cao, Alibaba Group
- Xianfu Cheng, Beihang University
- Jun Dong, ByteDance
- Shengjie Fang, Beijing University of Posts and Telecommunications
- Zhiwei Fei, Nanjing University
- Xiangyuan Guan, Beihang University
- Qipeng Guo, Shanghai AI Lab,
- Zhiguang Han, Nanyang Technological University
- Xueyu Hu, Zhejiang University
- Joseph James, University of Sheffield
- Tianqi Luo, The Hong Kong University of Science and Technology (Guangzhou)
- Renyuan Li, Sichuan University
- Yuhang Li, Beijing Institute of Technology
- Yiming Liang, CASIA

- Congnan Liu, Alibaba Group
- Qian Liu, Independent Researcher
- Ruitong Liu, National University of Singapore
- Tyler Loakman, University of Sheffield
- Xiangxin Meng, ByteDance
- Chuang Peng, Beijing Jiaotong University
- Tianhao Peng, Beihang University
- Jiajun Shi, Beihang University
- Mingjie Tang, Sichuan University
- Boyang Wang, Beihang University
- Haowen Wang, Beijing University of Posts and Telecommunications
- Yunli Wang, Beihang University
- Fanglin Xu, Hunan University
- Zihan Xu, Beijing University of Posts and Telecommunications

- Fei Yuan, Shanghai AI Lab,
- Jiayi Zhang, The Hong Kong University of Science and Technology (Guangzhou)
- Xinhao Zhang, Beijing Jiaotong University
- Xiantao Zhang, Beihang University
- Wangchunshu Zhou, OPPO
- Hualei Zhu, Alibaba Group
- King Zhu, OPPO

Organization and Senior Advisory Committee (Alphabetical Order)

- Bryan Dai, Ubiquant
- Aishan Liu, Beihang University
- Zhoujun Li, Beihang University
- Chenghua Lin, University of Manchester
- Jiaheng Liu, Nanjing University
- Tianyu Liu, Peking University
- Chao Peng, ByteDance
- Kai Shen, ByteDance
- Libo Qin, Central South University
- Shuangyong Song, Institute of Artificial Intelligence (TeleAI), China Telecom
- Ge Zhang, M-A-P
- Jiajun Zhang, CASIA
- Jie Zhang, Institute of Artificial Intelligence (TeleAI), China Telecom
- Zhaoxiang Zhang, CASIA
- Zizheng Zhan, Kuaishou
- Bo Zheng, Alibaba Group

This work is supported by State Key Laboratory of Complex & Critical Software Environment (SKLCCSE) of Beihang University. Tyler Loakman and Joseph James are supported by the Centre for Doctoral Training in Speech and Language Technologies (SLT) and their Applications funded by UK Research and Innovation [grant number EP/S023062/1].

References

- 1 *Elicitron: A Framework for Simulating Design Requirements Elicitation Using Large Language Model Agents*, volume Volume 3B: 50th Design Automation Conference (DAC) of International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, 08 2024. URL <https://doi.org/10.1115/DETC2024-143598>.
- 2 01.ai. Yi-coder. <https://github.com/01-ai/Yi-Coder>, 2024. Accessed: 2025-09-20.
- 3 Aime 2024. Mathematical association of america., 2024.
- 4 Aime 2025. Mathematical association of america., 2024.
- 5 Macarious Abadeer, Behrad Moeini, Emma Sewell, Paula Branco, Felipe Ventura, and Wei Shi. Dynamic extraction of bert-based embeddings for the detection of malicious javascript. In *Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering*, pages 110–119, 2022.
- 6 Tamer Abuelsaad, Deepak Akkil, Prasenjit Dey, Ashish Jagmohan, Aditya Vempaty, and Ravi Kokku. Agent-e: From autonomous web navigation to foundational design principles in agentic systems. *arXiv preprint arXiv:2407.13032*, 2024.
- 7 Muntasir Adnan, Zhiwei Xu, and Carlos C. N. Kuhn. Large language model guided self-debugging code generation (pycapsule). *arXiv preprint arXiv:2502.02928*, 2025. URL <https://arxiv.org/abs/2502.02928>.
- 8 Felix V. Agakov, Edwin V. Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O’Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. Using machine

- learning to focus iterative optimization. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pages 295–305. IEEE Computer Society, 2006.
- 9 V Agarwal, Y Pei, S Alamir, and X Liu. Codemirage: Hallucinations in code generated by large language models (2024). *arXiv preprint arXiv:2408.08333*.
 - 10 Vaibhav Agrawal. Androidmedia-Deobfuscate-android-app: Android app Vulnerability Scanner and Deobfuscator using LLM, November 2024. URL <https://github.com/In3tinct/deobfuscate-android-app>.
 - 11 Wasi Uddin Ahmad, Somshubra Majumdar, Aleksander Ficek, Sean Narendhiran, Mehrzad Samadi, Jocelyn Huang, Siddhartha Jain, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning-ii: A simple test time scaling approach via self-critique. *arXiv preprint arXiv:2507.09075*, 2025.
 - 12 Wasi Uddin Ahmad, Sean Narendhiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*, 2025.
 - 13 Wasi Uddin Ahmad, Sean Narendhiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*, 2025.
 - 14 Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms, 2024. URL <https://arxiv.org/abs/2402.14740>.
 - 15 Islem Bouzenia Ahmed, Premkumar Sobania, et al. RepairAgent: An autonomous, LLM-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.
 - 16 Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. Synshine: Improved fixing of syntax errors. *IEEE Transactions on Software Engineering*, 49(4):2169–2181, 2022.
 - 17 Toufique Ahmed, Jatin Ganhatra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate swe patches. *arXiv preprint arXiv:2502.05368*, 2025.
 - 18 Daechul Ahn, Yura Choi, Youngjae Yu, Dongyeop Kang, and Jonghyun Choi. Tuning large multimodal models for videos using reinforcement learning from ai feedback. *arXiv preprint arXiv:2402.03746*, 2024.
 - 19 Mistral AI. Codestral-22b-v0.1. <https://huggingface.co/mistralai/Codestral-22B-v0.1>, 2024. Accessed: 2025-09-20.
 - 20 Mistral AI and All Hands AI. Devstral: Agentic llm for software engineering. <https://mistral.ai/news/devstral>, May 2025. Research blog; Apache 2.0; Devstral Small (24B) and Devstral Medium.
 - 21 Moonshot AI. Kimi-k2-instruct. <https://huggingface.co/moonshotai/Kimi-K2-Instruct>, 2025.
 - 22 NamPDN AI. Tiny codes dataset, 2023. URL <https://huggingface.co/datasets/nampdn-ai/tiny-codes>. Accessed: 2024.

- 23 Stability AI. Stablecode: A 3b parameter code language model, 2023. URL <https://huggingface.co/stabilityai/stable-code-3b>. Accessed: 2025-09-20.
- 24 Zhipu AI. Codegeex. <https://github.com/zai-org/CodeGeeX>, 2022.
- 25 Zhipu AI. Glm-4.6, 2025. URL <https://huggingface.co/zai-org/GLM-4.6>.
- 26 aider-code-edit. aider-code-edit, 2025. URL <https://aider.chat/docs/leaderboards/edit.html>.
- 27 aider-refactoring-leaderboard. aider-refactoring-leaderboard, 2025. URL <https://aider.chat/docs/leaderboards/refactor.html>.
- 28 Ajibawa-2023. Code-290k-sharegpt, 2023. URL <https://huggingface.co/datasets/ajibawa-2023/Code-290k-ShareGPT>. Accessed: 2024.
- 29 Jafar Akhoundali, Hamidreza Hamidi, Kristian Rietveld, and Olga Gadyatskaya. Eradicating the unseen: Detecting, exploiting, and remediating a path traversal vulnerability across github. *arXiv preprint arXiv:2505.20186*, 2025.
- 30 Al-Baraa Al-Qasem, Motasem Alhanahnah, Abdalraouf Al-Kaswan, Baraa Al-Shboul, and Mahmoud Al-Omari. Llms in web development: Evaluating llm-generated php code unveiling vulnerabilities and limitations. *arXiv preprint arXiv:2404.16108*, 2024.
- 31 Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3test: Assertion-augmented automated test case generation. *Information and Software Technology*, 176:107565, 2024.
- 32 Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a visual language model for few-shot learning, 2022. URL <https://arxiv.org/abs/2204.14198>.
- 33 Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms, 2024. URL <https://arxiv.org/abs/2410.06992>.
- 34 Mohannad Alhanahnah and Yazan Boshmaf. Depsrag: Towards agentic reasoning and planning for software dependency management. *arXiv preprint arXiv:2405.20455*, 2024.
- 35 Maysara Alhindi and Joseph Hallett. Sandboxing adoption in open source ecosystems. In *Proceedings of the 12th ACM/IEEE International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 13–20, 2024.
- 36 Alibaba Cloud. Tongyi lingma. <https://tongyi.aliyun.com/lingma>, 2024.
- 37 Alibaba DAMO Academy. Qwen Code: Command line code assistant. Technical report, Alibaba Group, 2024.
- 38 AllAboutAI. My cursor ai review 2025: The best ide i've tried. <https://www.allaboutai.com/ai-reviews/cursor-ai/>, 2025.

- 39** Loubna Ben Allal, Raymond Li, Denis Kocetkov, et al. SantaCoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.
- 40** Ahmed Allam and Mohamed Shalan. Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects, 2024.
- 41** Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN international symposium on new ideas, new paradigms, and reflections on programming and software*, pages 143–153, 2019.
- 42** Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 472–483. ACM, 2014. doi: 10.1145/2635868.2635901. URL <https://doi.org/10.1145/2635868.2635901>.
- 43** Miltiadis Allamanis and Pengcheng Yin. Disproving program equivalence with llms, 2025. URL <https://arxiv.org/abs/2502.18473>.
- 44** Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness, 2018. URL <https://arxiv.org/abs/1709.06182>.
- 45** Miltiadis Allamanis, Sheena Panthaplickel, and Pengcheng Yin. Unsupervised evaluation of code llms with round-trip correctness, 2024. URL <https://arxiv.org/abs/2402.08699>.
- 46** Philipp Altmann, Julian Schönberger, Steffen Illium, Maximilian Zorn, Fabian Ritz, Tom Haider, Simon Burton, and Thomas Gabor. Emergence in multi-agent systems: A safety perspective. In *International Symposium on Leveraging Applications of Formal Methods*, pages 104–120. Springer, 2024.
- 47** Juan Altmayer Pizzorno and Emery D. Berger. Slipcover: Near zero-overhead code coverage for python. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '23*, page 1195–1206. ACM, July 2023. doi: 10.1145/3597926.3598128. URL <http://dx.doi.org/10.1145/3597926.3598128>.
- 48** Rajeev Alur, Rastislav Bodík, Garvit Jundiwal, Milo M. K. Martin, Mukund Raghethaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013. doi: 10.1109/FMCAD.2013.6679385.
- 49** Amazon Web Services. Codewhisperer is becoming part of amazon q developer. <https://docs.aws.amazon.com/codewhisperer/latest/userguide/whisper-legacy.html>, 2024.
- 50** Amazon Web Services. Amazon CodeWhisperer: AI code generator. Technical report, Amazon Web Services, Inc., 2024.
- 51** Amazon Web Services. Amazon q developer. <https://aws.amazon.com/q/developer/>, 2024.
- 52** Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, and et al. Palm 2 technical report, 2023. URL <https://arxiv.org/abs/2305.10403>.

- 53 Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. Opentuner: an extensible framework for program autotuning. In José Nelson Amaral and Josep Torrellas, editors, *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 303–316. ACM, 2014.
- 54 Anthropic. Claude 2. <https://www.anthropic.com/news/clause-2>, 2023.
- 55 Anthropic. Model card and evaluations for claude models. <https://www-cdn.anthropic.com/bd2a28d2535bfb0494cc8e2a3bf135d2e7523226/Model-Card-Claude-2.pdf>, 2023.
- 56 Anthropic. Introducing claude. <https://www.anthropic.com/news/introducing-claude>, 2023.
- 57 Anthropic. Claude 3.5 sonnet model card addendum. https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf, 2024.
- 58 Anthropic. Introducing claude 3.5 sonnet. <https://www.anthropic.com/news/clause-3-5-sonnet>, 2024.
- 59 Anthropic. The claude 3 model family: Opus, sonnet, haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbaf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf, 2024.
- 60 Anthropic. Claude Code: Terminal-based AI development assistant. Technical report, Anthropic, PBC, 2024.
- 61 Anthropic. Introducing the next generation of claude. <https://www.anthropic.com/news/clause-3-family>, 2024.
- 62 Anthropic. Introducing the model context protocol, 2024. URL <https://www.anthropic.com/news/model-context-protocol>.
- 63 Anthropic. Anthropic launches claude 4.5. <https://www.reuters.com/business/retail-consumer/anthropic-launches-claude-45-touts-better-abilities-targets-business-customers-2025-09-29/>, 2025.
- 64 Anthropic. Claude opus 4 & claude sonnet 4 system card. <https://www.anthropic.com/clause-4-system-card>, 2025.
- 65 Anthropic. Introducing claude 4. <https://www.anthropic.com/news/clause-4>, 2025.
- 66 anthropic. Claude 3.7 sonnet, 2025. URL <https://www.anthropic.com/news/clause-3-7-sonnet/>.
- 67 Anthropic. Introducing claude sonnet 4.5, 2025. URL <https://www.anthropic.com/news/clause-sonnet-4-5>.
- 68 Anysphere. Cursor - the ai code editor. <https://www.cursor.com/en>, 2025. URL <https://www.cursor.com>.
- 69 Anysphere Inc. Cursor: The AI-first code editor. Technical report, Anysphere Inc., 2024. URL <https://cursor.com>.
- 70 Anysphere Inc. Cursor changelog. <https://cursor.com/changelog>, 2025.
- 71 Anysphere Inc. Cursor features. <https://cursor.com/features>, 2025.

- 72 Jordi Armengol-Estepé, Jackson Woodruff, Chris Cummins, and Michael F. P. O’Boyle. Slade: A portable small language model decompiler for optimized assembler. *CoRR*, abs/2305.12520, 2023.
- 73 Catherine Arnett, Eliot Jones, Ivan P Yamshchikov, and Pierre-Carl Langlais. Toxicity of the commons: Curating open-source pre-training data. *arXiv preprint arXiv:2410.22587*, 2024.
- 74 Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, and Leonardo Querzoni. In nomine function: Naming functions in stripped binaries with neural networks. 2019.
- 75 Oumou K Asare, Fadi Jaafar, Naeem Ali, and Laurie Williams. The hidden risks of llm-generated web application code: A security-centric evaluation of code generation capabilities in large language models. In *Proceedings of the 2023 ACM on Workshop on Secure and Trustworthy Language Processing*, pages 31–42, 2023.
- 76 Sualeh Asif and Gergely Orosz. Real-world engineering challenges: Building cursor. <https://newsletter.pragmaticengineer.com/p/cursor>, 2025.
- 77 Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models, 2023. URL <https://arxiv.org/abs/2210.14868>.
- 78 Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models, 2023. URL <https://arxiv.org/abs/2210.14868>.
- 79 Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models, 2023. URL <https://arxiv.org/abs/2210.14868>.
- 80 AugmentCode. Reinforcement learning from developer behaviors: A breakthrough in code generation quality. <https://www.augmentcode.com/blog/reinforcement-learning-from-developer-behaviors>, 2025. Accessed: 2025.
- 81 Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces, 2023. URL <https://arxiv.org/abs/2107.03006>.
- 82 Jacob et al. Austin. Program synthesis with large language models. In *ICML*, 2021.

- 83 Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL <https://arxiv.org/abs/2505.20411>.
- 84 Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015. URL <http://arxiv.org/abs/1409.0473>.
- 85 Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report, 2023. URL <https://arxiv.org/abs/2309.16609>.
- 86 Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond. *arXiv preprint arXiv:2308.12966*, 2023.
- 87 Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.
- 88 Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.
- 89 Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.
- 90 Weiheng Bai, Wei-Yang Chiu, Peng-Fei Wu, Chun-Ying Huang, Hsu-Chun Hsiao, and Wen-Lian Hsu. Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls, 2024.
- 91 Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.
- 92 Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- 93 Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning, 2023. URL <https://arxiv.org/abs/2309.12499>.

- 94 Bowen Baker, Ilge Akkaya, Peter Zhokhov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (vpt): Learning to act by watching unlabeled online videos, 2022. URL <https://arxiv.org/abs/2206.11795>.
- 95 Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *CoRR*, abs/2103.12801, 2021.
- 96 Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.
- 97 Averi Bates, Ryan Vavricka, Shane Carleton, Ruosi Shao, and Chongle Pan. Unified modeling language code generation from diagram images using multimodal large language models. *Machine Learning with Applications*, page 100660, 2025.
- 98 Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.
- 99 Nathanaël Beau and Benoît Crabbé. Codeinsight: A curated dataset of practical coding solutions from stack overflow, 2024. URL <https://arxiv.org/abs/2409.16819>.
- 100 Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai, 2025. URL <https://arxiv.org/abs/2506.02153>, 2025.
- 101 Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot, 2017. URL <https://arxiv.org/abs/1705.07962>.
- 102 Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems*, pages 1–6, 2018.
- 103 Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks, 2015. URL <https://arxiv.org/abs/1506.03099>.
- 104 Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, 2021.
- 105 Guru Bhandari, Nikola Gavric, and Andrii Shalaginov. Generating vulnerability security fixes with code language models. *Information and Software Technology*, page 107786, 2025.
- 106 Aaditya Bhatia, Gustavo A Oliva, Gopi Krishnan Rajbahadur, Haoxiang Zhang, Yihao Chen, Zhilong Chen, Arthur Leung, Dayi Lin, Boyuan Chen, and Ahmed E Hassan. Spice: An automated swe-bench labeling pipeline for issue clarity, test coverage, and effort estimation. *arXiv preprint arXiv:2507.09108*, 2025.
- 107 Manish Bhattarai, Javier E. Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O’Malley. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. *arXiv preprint arXiv: 2407.19619*, 2024.

- 108** Manish Bhattacharai, Minh Vu, Javier E. Santos, Ismael Boureima, and Daniel O' Malley. Enhancing cross-language code translation via task-specific embedding alignment in retrieval-augmented generation. *arXiv preprint arXiv: 2412.05159*, 2024.
- 109** Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. Statistical deobfuscation of android applications. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 343–355. ACM, 2016.
- 110** BigCode. Commitpackft: A dataset of git commits for fine-tuning, 2023. URL <https://huggingface.co/datasets/bigcode/commitpackft>. Accessed: 2024.
- 111** BigCode. Self-oss-instruct-sc2-exec-filter-50k, 2024. URL <https://huggingface.co/datasets/bigcode/self-oss-instruct-sc2-exec-filter-50k>. Accessed: 2024.
- 112** Bito. Bito ai: Ai-powered code assistant. <https://bito.ai>, 2024.
- 113** Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.
- 114** Franois Bodin, Toru Kisuki, Peter Knijnenburg, Mike O' Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.
- 115** Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, et al. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612*, 2024.
- 116** Tolga Bolukbasi, Kai-Wei Chang, James Y Zou, Venkatesh Saligrama, and Adam T Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *Advances in neural information processing systems*, volume 29, 2016.
- 117** Mohamed Salah Bouafif, Mohammad Hamdaqa, and Edward Zulkoski. Primg: Efficient llm-driven test generation using mutant prioritization. *arXiv preprint arXiv:2505.05584*, 2025.
- 118** Laila Bouhlal, Fouzia Kassou, Nouzha Lamdouar, and Azeddine Bouyahyaoui. Assembly procedure for elementary matrices of train-track-bridge railway system. *arXiv preprint arXiv:2406.14837*, 2024.
- 119** Islem Bouzenia and Michael Pradel. You name it, i run it: An llm agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1054–1076, 2025.
- 120** Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: an autonomous, llm-based agent for program repair.(2024). *arXiv preprint arXiv:2403.17134*.
- 121** B. Brown and et al. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024. URL <https://arxiv.org/abs/2407.21787>.

- 122 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.
- 123 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.
- 124 Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL <https://arxiv.org/abs/2005.14165>.
- 125 Buggedaryan. Sql create context instruction, 2024. URL <https://huggingface.co/datasets/buggedaryan/sql-create-context-instruction>. Accessed: 2024.
- 126 ByteByteGo. How cursor serves billions of ai code completions every day. <https://blog.bytebytogo.com/p/how-cursor-serves-billions-of-ai>, 2025.
- 127 Hongru Cai, Yongqi Li, Wenjie Wang, Fengbin Zhu, Xiaoyu Shen, Wenjie Li, and Tat-Seng Chua. Large language models empowered personalized web agents, 2025. URL <https://arxiv.org/abs/2410.17236>.
- 128 Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. Tag: Type auxiliary guiding for code comment generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 291–301, 2020.
- 129 Yin Cai, Zhouhong Gu, Zhaohan Du, Zheyu Ye, Shaosheng Cao, Yiqian Xu, Hongwei Feng, and Ping Chen. MIRAGE: Exploring How Large Language Models Perform in Complex Social Interactive Environments. *arXiv e-prints*, art. arXiv:2501.01652, January 2025. doi: 10.48550/arXiv.2501.01652.
- 130 Zikui Cai, Andrew Wang, Anirudh Satheesh, Ankit Nakhawa, Hyunwoo Jae, Keenan Powell, Minghui Liu, Neel Jay, Sungbin Oh, Xiya Wang, Yongyuan Liang, Tom Goldstein, and Furong Huang. MORSE-500: A Programmatically Controllable Video Benchmark to Stress-Test Multimodal Reasoning. *arXiv e-prints*, art. arXiv:2506.05523, June 2025. doi: 10.48550/arXiv.2506.05523.
- 131 CAMEL-AI.org. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. <https://github.com/camel-ai/owl>, 2025. Accessed: 2025-03-07.

- 132 Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing chi Cheung, and Chang Xu. Javabench: A benchmark of object-oriented code generation for evaluating large language models, 2024. URL <https://arxiv.org/abs/2406.12902>.
- 133 Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*, pages 508–518. ACM, 2022.
- 134 Yuhan Cao, Zian Chen, Kun Quan, Ziliang Zhang, Yu Wang, Xiaoning Dong, Yeqi Feng, Guanzhong He, Jingcheng Huang, Jianhao Li, Yixuan Tan, Jiafu Tang, Yilin Tang, Junlei Wu, Qianyu Xiao, Can Zheng, Shouchen Zhou, Yuxiang Zhu, Yiming Huang, Tian Xie, and Tianxing He. Can LLMs Generate Reliable Test Case Generators? A Study on Competition-Level Programming Problems, July 2025. URL <http://arxiv.org/abs/2506.06821>. arXiv:2506.06821 [cs].
- 135 Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. RSL-SQL: robust schema linking in text-to-sql generation. *CoRR*, abs/2411.00073, 2024.
- 136 Zouying Cao, Runze Wang, Yifei Yang, Xinbei Ma, Xiaoyong Zhu, Bo Zheng, and Hai Zhao. Pgpo: Enhancing agent reasoning via pseudocode-style planning guided preference optimization. *arXiv preprint arXiv:2506.01475*, 2025.
- 137 Tom Cappendijk, Pepijn de Reus, and Ana Oprescu. An exploration of prompting llms to generate energy-efficient code. In *2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software (GREENS)*, page 31–38. IEEE, April 2025. doi: 10.1109/greens66463.2025.00010. URL <http://dx.doi.org/10.1109/GREENS66463.2025.00010>.
- 138 Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.
- 139 Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022. URL <https://arxiv.org/abs/2208.08227>.
- 140 Cfahlgren1. React code instructions, 2024. URL <https://huggingface.co/datasets/cfahlgren1/react-code-instructions>. Accessed: 2024.
- 141 Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, et al. Mceval: Massively multilingual code evaluation. *arXiv preprint arXiv:2406.07436*, 2024.
- 142 Linzheng Chai, Jian Yang, Shukai Liu, Wei Zhang, Liran Wang, Ke Jin, Tao Sun, Congnan Liu, Chenchen Zhang, Hualei Zhu, et al. Multilingual multimodal software developer for code generation. *arXiv preprint arXiv:2507.08719*, 2025.
- 143 Linzheng Chai, Jian Yang, Tao Sun, Hongcheng Guo, Jiaheng Liu, Bing Wang, Xinnian Liang, Jiaqi Bai, Tongliang Li, Qiya Peng, and Zhoujun Li. XCOT: cross-lingual instruction tuning for cross-lingual chain-of-thought reasoning. In Toby Walsh, Julie Shah, and Zico Kolter, editors, *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence*, pages 1–10. AAAI Press, 2025.

Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA, pages 23550–23558. AAAI Press, 2025. doi: 10.1609/AAAI.V39I22.34524. URL <https://doi.org/10.1609/aaai.v39i22.34524>.

- 144 Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. Ernie-code: Beyond english-centric cross-lingual pretraining for programming languages, 2023. URL <https://arxiv.org/abs/2212.06742>.
- 145 Iason Chaimalas, Arnas VyLAniauskas, and Gabriel Brostow. Explorer: Robust collection of interactable gui elements. *arXiv preprint arXiv:2504.09352*, 2025.
- 146 Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. Rlocator: Reinforcement learning for bug localization. *IEEE Transactions on Software Engineering*, 2024.
- 147 Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. Bigo(bench) – can llms generate code with controlled time and space complexity?, 2025. URL <https://arxiv.org/abs/2503.15242>.
- 148 Shubham Chandel, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. Training and evaluating a jupyter notebook data science assistant, 2022. URL <https://arxiv.org/abs/2201.12901>.
- 149 Shuaichen Chang and Eric Fosler-Lussier. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings. *CoRR*, abs/2305.11853, 2023.
- 150 Yiannis Charalambous, Edoardo Manino, and Lucas C Cordeiro. Automated repair of ai code with large language models and formal verification. *arXiv preprint arXiv:2405.08848*, 2024.
- 151 chatgpt. chatgpt, 2025. URL <https://chatgpt.com/>.
- 152 Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. <https://github.com/sahil280114/codealpaca>, 2023.
- 153 Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, et al. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025.
- 154 Dong Chen, Shaolin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024.
- 155 Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. Humans or llms as the judge? a study on judgement biases. *arXiv preprint arXiv:2402.10669*, 2024.
- 156 Jialiang Chen, Kaifa Zhao, Jie Liu, Chao Peng, Jierui Liu, Hang Zhu, Pengfei Gao, Ping Yang, and Shuguang Deng. Coreqa: uncovering potentials of language models in code repository question answering. *arXiv preprint arXiv:2501.03447*, 2025.
- 157 Junjie Chen, Haoyang Ma, and Lingming Zhang. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 78–89, 2020.

- 158 Kaiyuan Chen, Yixin Ren, Yang Liu, Xiaobo Hu, Haotong Tian, Tianbao Xie, Fangfu Liu, Haoye Zhang, Hongzhang Liu, Yuan Gong, et al. xbench: Tracking agents productivity scaling with profession-aligned real-world evaluations. *arXiv preprint arXiv:2506.13651*, 2025.
- 159 Kang Chen, Ziteng Wang, Weiming Zhang, Gang Li, Zhaofeng Chen, and Nenghai Yu. A survey on privacy risks and protection in large language models, 2025.
- 160 Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, et al. A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv:2408.16498*, 2024.
- 161 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL <https://arxiv.org/abs/2107.03374>.
- 162 Peng Chen, Pi Bu, Jun Song, Yuan Gao, and Bo Zheng. Can VLMs Play Action Role-Playing Games? Take Black Myth Wukong as a Study Case. *arXiv e-prints*, art. arXiv:2409.12889, September 2024. doi: 10.48550/arXiv.2409.12889.
- 163 Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025.
- 164 Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*, 2(4):6, 2023.
- 165 Wenhua Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.
- 166 Xiangping Chen, Junqi Chen, Zhilu Lian, Yuan Huang, Xiaocong Zhou, Yunzhi Wu, and Zibin Zheng. An alternative to code comment generation? generating comment from bytecode. *Information and Software Technology*, 179:107623, 2025. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2024.107623>. URL <https://www.sciencedirect.com/science/article/pii/S0950584924002283>.
- 167 Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation, 2018. URL <https://arxiv.org/abs/1802.03691>.
- 168 Xinyun Chen, Jerry Tworek, et al. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *EMNLP*, 2023.

- 169 Yang Chen, Yufan Shen, Wenxuan Huang, Sheng Zhou, Qunshu Lin, Xinyu Cai, Zhi Yu, Jiajun Bu, Botian Shi, and Yu Qiao. Learning only with images: Visual reinforcement learning with reasoning, rendering, and visual feedback. *arXiv preprint arXiv:2507.20766*, 2025.
- 170 Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Acereason-nemotron: Advancing math and code reasoning through reinforcement learning. *arXiv preprint arXiv:2505.16400*, 2025.
- 171 Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation, 2024. URL <https://arxiv.org/abs/2305.04764>.
- 172 Yongchao Chen, Harsh Jhamtani, Srinagesh Sharma, Chuchu Fan, and Chi Wang. Steering large language models between code execution and textual reasoning. *arXiv preprint arXiv:2410.03524*, 2024.
- 173 Yunnong Chen, Shixian Ding, YingYing Zhang, Wenkai Chen, Jinzhou Du, Lingyun Sun, and Liuqing Chen. Designcoder: Hierarchy-aware and self-correcting ui code generation with large language models. *arXiv preprint arXiv:2506.13663*, 2025.
- 174 Yuxiang Chen, Zhenyang Ding, Yue Yu, et al. MagicCoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2024.
- 175 Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, et al. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 24185–24198, 2024.
- 176 Zhe Chen, Weiyun Wang, Yue Cao, Yangzhou Liu, Zhangwei Gao, Erfei Cui, Jinguo Zhu, Shenglong Ye, Hao Tian, Zhaoyang Liu, Lixin Gu, Xuehui Wang, Qingyun Li, Yiming Ren, Zixuan Chen, Jiapeng Luo, Jiahao Wang, Tan Jiang, Bo Wang, Conghui He, Botian Shi, Xingcheng Zhang, Han Lv, Yi Wang, Wenqi Shao, Pei Chu, Zhongying Tu, Tong He, Zhiyong Wu, Huipeng Deng, Jiaye Ge, Kai Chen, Kaipeng Zhang, Limin Wang, Min Dou, Lewei Lu, Xizhou Zhu, Tong Lu, Dahua Lin, Yu Qiao, Jifeng Dai, and Wenhai Wang. Expanding performance boundaries of open-source multimodal models with model, data, and test-time scaling, 2025. URL <https://arxiv.org/abs/2412.05271>.
- 177 Zifan Chen, Gábor Gulyás, Zsombor Kovács, Zsolt Zombori, Márk Félegyházi, Máté Telek, and Bence Horváth. Large language models for code: Security hardening and adversarial testing, 2023.
- 178 Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, et al. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025.
- 179 An-Chieh Cheng, Hongxu Yin, Yang Fu, Qiushan Guo, Ruihan Yang, Jan Kautz, Xiaolong Wang, and Sifei Liu. Spatialrgpt: Grounded spatial reasoning in vision-language models. *Advances in Neural Information Processing Systems*, 37:135062–135093, 2024.
- 180 Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. Agentic bug reproduction for effective automated program repair at google. *arXiv preprint arXiv:2502.01821*, 2025.

- 181 Artem Chervyakov, Alexander Kharitonov, Pavel Zadorozhny, Adamenko Pavel, Rodion Levichev, Dmitrii Vorobev, Dmitrii Salikhov, Aidar Valeev, Alena Pestova, Maria Dziuba, Ilseyar Alimova, Artem Zavgorodnev, Aleksandr Medvedev, Stanislav Moiseev, Elena Bruches, Daniil Grebenkin, Roman Derunets, Vikulov Vladimir, Anton Emelyanov, Dmitrii Babaev, Vladimir V. Ivanov, Valentin Malykh, and Alena Fenogenova. Mera code: A unified framework for evaluating code generation across tasks, 2025. URL <https://arxiv.org/abs/2507.12284>.
- 182 Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering*, 49(2):564–585, 2022.
- 183 Wayne Chi, Valerie Chen, Anastasios Nikolas Angelopoulos, Wei-Lin Chiang, Aditya Mittal, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. Copilot arena: A platform for code llm evaluation in the wild, 2025. URL <https://arxiv.org/abs/2502.09328>.
- 184 Xuebin Chi. Ltmatch: A method to abstract pattern from unstructured log. *Applied Sciences*, 11, 2021.
- 185 Pier Giorgio Chiara. The cyber resilience act: the eu commission’s proposal for a horizontal regulation on cybersecurity for products with digital elements: An introduction. *International Cybersecurity Law Review*, 3(2):255–272, 2022.
- 186 Daewon Choi, Jimin Lee, Jihoon Tack, Woomin Song, Saket Dingliwal, Sai Muralidhar Jayanthi, Bhavana Ganesh, Jinwoo Shin, Aram Galstyan, and Sravan Babu Bodapati. Think clearly: Improving reasoning via redundant token pruning. *arXiv preprint arXiv:2507.08806*, 2025.
- 187 Jason Chou, Ao Liu, Yuchi Deng, Zhiying Zeng, Tao Zhang, Haotian Zhu, Jianwei Cai, Yue Mao, Chenchen Zhang, Lingyun Tan, Ziyan Xu, Bohui Zhai, Hengyi Liu, Speed Zhu, Wiggin Zhou, and Fengzong Lian. Autocodebench: Large language models are automatic code benchmark generators, 2025. URL <https://arxiv.org/abs/2508.09101>.
- 188 Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, and et al. Palm: Scaling language modeling with pathways, 2022. URL <https://arxiv.org/abs/2204.02311>.
- 189 Steve Christey, J Kenderdine, J Mazella, and B Miles. Common weakness enumeration. *Mitre Corporation*, 2013.
- 190 Christopher. Rosetta code dataset, 2023. URL <https://huggingface.co/datasets/christopher/rosetta-code>. Accessed: 2024.
- 191 Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. Pangu-coder: Program synthesis with function-level language modeling, 2022. URL <https://arxiv.org/abs/2207.11280>.
- 192 claude4. claude4, 2025. URL <https://www.anthropic.com/clause/sonnet>.

- 193 Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. PyMT5: multi-mode translation of natural language and python code with transformers. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.728. URL <https://aclanthology.org/2020.emnlp-main.728/>.
- 194 Claude Code. Claude code. <https://github.com/anthropics/clause-code>, 2025.
- 195 CodeChain. Codechain. *arXiv preprint arXiv:2310.08992*, 2023. URL <https://arxiv.org/abs/2310.08992>.
- 196 Codedog. Codedog: Ai code review platform. <https://codedog.ai>, 2024.
- 197 CodeFuse-AI. Codefuse-embeddings: Code generalist embeddings (cge). <https://github.com/codefuse-ai/CodeFuse-Embeddings/tree/main/CGE>, 2025. Accessed: 2025-11-03.
- 198 Codeium. Windsurf: The AI flow editor. Technical report, Codeium Inc., 2024. URL <https://codeium.com/windsurf>.
- 199 CodeParrot. Apps: A benchmark for measuring the ability of language models to generate simple programs from natural language descriptions, 2021. URL <https://huggingface.co/datasets/codeparrot/apps>. Accessed: 2024.
- 200 CodePlan. Codeplan. *arXiv preprint arXiv:2309.12499*, 2023. URL <https://arxiv.org/abs/2309.12499>.
- 201 CodeRabbit. Coderabbit: Ai code reviews. <https://coderabbit.ai>, 2024.
- 202 codetrans. codetrans, 2025. URL <https://github.com/agemagician/CodeTrans>.
- 203 Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Inducing tree-substitution grammars. *J. Mach. Learn. Res.*, 11:3053–3096, 2010. doi: 10.5555/1756006.1953031. URL <https://dl.acm.org/doi/10.5555/1756006.1953031>.
- 204 Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marrs, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, Krishna Haridasan, Ahmed Omran, Nikunj Saunshi, Dara Bahri, Gaurav Mishra, Eric Chu, Toby Boyd, Brad Hekman, Aaron Parisi, Chaoyi Zhang, Kornraphop Kawintiranon, Tania Bedrax-Weiss, Oliver Wang, Ya Xu, Ollie Purkiss, Uri Mendlovic, Ilai Deutel, Nam Nguyen, Adam Langley, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL <https://arxiv.org/abs/2507.06261>.
- 205 Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Noveen Sachdeva, and et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL <https://arxiv.org/abs/2507.06261>.
- 206 Together Computer. RedPajama-Data-1T, 2023. URL <https://huggingface.co/datasets/togethercomputer/RedPajama-Data-1T>. Accessed: 2024-01-01.

- 207** Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In Y. Annie Liu and Reinhard Wilhelm, editors, *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99), Atlanta, Georgia, USA, May 5, 1999*, pages 1–9. ACM, 1999.
- 208** Jonathan Cordeiro, Shayan Noei, and Ying Zou. An empirical study on the code refactoring capability of large language models. *arXiv preprint arXiv:2411.02320*, 2024.
- 209** Viktor Csuvik and László Vidács. Fixjs: a dataset of bug-fixing javascript commits. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 712–716, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528480. URL <https://doi.org/10.1145/3524842.3528480>.
- 210** Viktor Csuvik and László Vidács. Fixjs: a dataset of bug-fixing javascript commits. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 712–716, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528480. URL <https://doi.org/10.1145/3524842.3528480>.
- 211** Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Yuchen Zhang, Jiacheng Chen, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Xu Han, Hao Peng, Yu Cheng, Zhiyuan Liu, Maosong Sun, Bowen Zhou, and Ning Ding. Process reinforcement through implicit rewards, 2025. URL <https://arxiv.org/abs/2502.01456>.
- 212** Haotian Cui, Chenglong Wang, Junjie Huang, Jeevana Priya Inala, Todd Mytkowicz, Bo Wang, Jianfeng Gao, and Nan Duan. CodeExp: Explanatory code document generation. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 2342–2354, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.174. URL <https://aclanthology.org/2022.findings-emnlp.174/>.
- 213** Zhiqing Cui, Jiahao Yuan, Hanqing Wang, Yanshu Li, Chenxu Du, and Zhenglong Ding. Draw with thought: Unleashing multimodal reasoning for scientific diagram generation. *arXiv preprint arXiv:2504.09479*, 2025.
- 214** Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. *IEEE Computer Society*, 2017.
- 215** Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Kim M. Hazelwood, Gabriel Synnaeve, and Hugh Leather. Large language models for compiler optimization. *CoRR*, abs/2309.07062, 2023.
- 216** Cursor. Online rl for cursor tab. <https://cursor.com/cn/blog/tab-rl>, 2025. Accessed: 2025.
- 217** Alireza Daghfarsoodeh, Chung-Yu Wang, Hamed Taherkhani, Melika Sepidband, Mohammad Abdollahi, Hadi Hemmati, and Hung Viet Pham. Deep-bench: Deep learning benchmark dataset for code generation, 2025. URL <https://arxiv.org/abs/2502.18726>.

- 218 Hankun Dai, Maoquan Wang, Mengnan Qi, Yikai Zhang, Zijian Jin, Yongqiang Yao, Yufan Huang, Shengyu Fu, and Elsie Nallipogu. Lita: Light agent uncovers the agentic coding capabilities of llms. *arXiv preprint arXiv:2509.25873*, 2025.
- 219 Yifan Dai, Shuo Chen, and Hao Yu. Feedbackeval: Benchmarking iterative feedback-driven program repair with llms. In *Proceedings of the 2025 Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 456–468, 2025.
- 220 Zhaowei Dai, Jiachen Li, Jiaming Liu, Zhaoran Li, Cha Wei, Yang Huang, and Zhen Liu. Safe rlhf: Safe reinforcement learning from human feedback. *arXiv preprint arXiv:2310.12773*, 2023.
- 221 Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing, 2023. URL <https://arxiv.org/abs/2308.16557>.
- 222 Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL <https://arxiv.org/abs/2405.21060>.
- 223 david. How i program with llms, 2025. URL <https://crawshaw.io/blog/programming-with-llms>.
- 224 Erik Daxberger, Nina Wenzel, David Griffiths, Haiming Gang, Justin Lazarow, Gefen Kohavi, Kai Kang, Marcin Eichner, Yinfei Yang, Afshin Dehghan, and Peter Grasch. MM-Spatial: Exploring 3D Spatial Understanding in Multimodal LLMs. *arXiv e-prints*, art. arXiv:2503.13111, March 2025. doi: 10.48550/arXiv.2503.13111.
- 225 Bryan LM de Oliveira, Luana GB Martins, Bruno Brandão, Murilo L da Luz, Telma W de L Soares, and Luckeciano C Melo. Sliding puzzles gym: A scalable benchmark for state representation in visual reinforcement learning. *arXiv preprint arXiv:2410.14038*, 2024.
- 226 Pepijn de Reus, Ana Oprescu, and Jelle Zuidema. An exploration of the effect of quantisation on energy consumption and inference time of starcoder2, 2024. URL <https://arxiv.org/abs/2411.12758>.
- 227 Nelson Tavares de Sousa and Wilhelm Hasselbring. Javabert: Training a transformer-based model for the java programming language. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021 - Workshops, Melbourne, Australia, November 15-19, 2021*, pages 90–95. IEEE, 2021. doi: 10.1109/ASEW52652.2021.00028. URL <https://doi.org/10.1109/ASEW52652.2021.00028>.
- 228 Matthew T. Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. Lassi: An llm-based automated self-correcting pipeline for translating parallel scientific codes, 2025. URL <https://arxiv.org/abs/2407.01638>.
- 229 DeepMind. Codecontests: A competitive programming dataset, 2022. URL https://huggingface.co/datasets/deepmind/code_contests. Accessed: 2024.
- 230 Google DeepMind. Gemini diffusion, May 2025. URL <https://deepmind.google/models/gemini-diffusion/>.
- 231 deepseek. deepseek, 2025. URL <https://www.deepseek.com/>.
- 232 DeepSeek AI. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

- 233 DeepSeek-AI. Deepseek-v3 technical report, 2024. URL <https://arxiv.org/abs/2412.19437>.
- 234 DeepSeek-AI. Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention, 2025.
- 235 DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL <https://arxiv.org/abs/2405.04434>.
- 236 DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *CoRR*, abs/2406.11931, 2024. doi: 10.48550/ARXIV.2406.11931. URL <https://doi.org/10.48550/arXiv.2406.11931>.
- 237 DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue

Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhua Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL <https://arxiv.org/abs/2501.12948>.

- 238 DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhua Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li,

- Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025. URL <https://arxiv.org/abs/2412.19437>.
- 239 Chaoyi Deng, Jialong Wu, Ningya Feng, Jianmin Wang, and Mingsheng Long. Compiler-dream: Learning a compiler world model for general code optimization. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 486–497, 2025.
- 240 Jia Deng, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, and Ji-Rong Wen. Decomposing the entropy-performance exchange: The missing keys to unlocking effective reinforcement learning. *arXiv preprint arXiv:2508.02260*, 2025.
- 241 Jiahao Deng, Ziyuan Wang, Chuhui Zhang, Xuezixiang Li, and Pin-Yu Chen. Gptfuzzer: Red teaming large language models with auto-generated jailbreak prompts, 2023.
- 242 Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.
- 243 Zekun Deng, Zhaofeng Tan, Yujun Wang, Jie Zhou, Kelin Liu, Yue Guo, Kaixuan Sun, and Xin Jiang. Autodefense: Multi-agent llm defense against jailbreak attacks. *arXiv preprint arXiv:2404.09127*, 2024.
- 244 Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.
- 245 Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.
- 246 DEV Community. The known and unknown of amazon q developer. <https://dev.to/aws-builders/amazon-q-developer>, 2025.
- 247 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL <https://aclanthology.org/N19-1423/>.
- 248 Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. Codefuse-13b: A pretrained multi-lingual code large language model. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP ’24, page 418–429. ACM, April 2024. doi: 10.1145/3639477.3639719. URL <http://dx.doi.org/10.1145/3639477.3639719>.
- 249 Luca Di Grazia and Michael Pradel. Code search: A survey of techniques for finding code. *ACM Computing Surveys*, 55(11):1–31, 2023.

- 250 Colin Diggs, Michael Doyle, Amit Madan, Siggy Scott, Emily Escamilla, Jacob Zimmer, Naveed Nekoo, Paul Ursino, Michael Bartholf, Zachary Robin, Anand Patel, Chris Glasz, William Macke, Paul Kirk, Jasper Phillips, Arun Sridharan, Doug Wendt, Scott Rosen, Nitin Naik, Justin F. Brunelle, and Samruddhi Thaker. Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation. *arXiv e-prints*, art. arXiv:2411.14971, November 2024. doi: 10.48550/arXiv.2411.14971.
- 251 Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. Secrepobench: Benchmarking llms for secure code generation in real-world repositories, 2025. URL <https://arxiv.org/abs/2504.21205>.
- 252 Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion, 2023. URL <https://arxiv.org/abs/2310.11248>.
- 253 Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.
- 254 Yangruibo Ding, Zijian Wang, Wasi Ahmad, et al. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. *NeurIPS*, 2024.
- 255 Zijian Ding, Qinshi Zhang, Mohan Chi, and Ziyi Wang. Frontend diffusion: Empowering self-representation of junior researchers and designers through agentic workflows. *arXiv preprint arXiv:2502.03788*, 2025.
- 256 Yusuf Denizay Dönder, Derek Hommel, Andrea W. Wen-Yi, David Mimno, and Unso Eun Seo Jo. Cheaper, better, faster, stronger: Robust text-to-sql without chain-of-thought or fine-tuning. *CoRR*, abs/2505.14174, 2025.
- 257 Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. C3: zero-shot text-to-sql with chatgpt. *CoRR*, abs/2307.07306, 2023.
- 258 Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *corr abs/2301.09043* (2023), 2023.
- 259 Swaroop Dora, Deven Lunkad, Nazyia Aslam, S Venkatesan, and Sandeep Kumar Shukla. The hidden risks of llm-generated web application code: A security-centric evaluation of code generation capabilities in large language models. *arXiv preprint arXiv:2504.20612*, 2025.
- 260 Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation, 2025. URL <https://arxiv.org/abs/2502.05714>.
- 261 Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. *arXiv preprint arXiv:2105.09352*, 2021.
- 262 Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program language translation using a grammar-driven tree-to-tree model, 2018. URL <https://arxiv.org/abs/1807.01784>.

- 263 Junjia Du, Yadi Liu, Hongcheng Guo, Jiawei Wang, Haojian Huang, Yunyi Ni, and Zhoujun Li. Dependeval: Benchmarking llms for repository dependency understanding, 2025. URL <https://arxiv.org/abs/2503.06689>.
- 264 Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. Deep-research bench: A comprehensive benchmark for deep research agents. *arXiv preprint arXiv:2506.11763*, 2025.
- 265 Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models, 2024. URL <https://arxiv.org/abs/2402.07844>.
- 266 Mingzhe Du, Anh Tuan Luu, Bin Ji, Xiaobao Wu, Yuhao Qing, Dong Huang, Terry Yue Zhuo, Qian Liu, and See-Kiong Ng. CodeArena: A collective evaluation platform for LLM code generation. In Pushkar Mishra, Smaranda Muresan, and Tao Yu, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 502–512, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-253-4. doi: 10.18653/v1/2025.acl-demo.48. URL <https://aclanthology.org/2025.acl-demo.48/>.
- 267 Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. Glam: Efficient scaling of language models with mixture-of-experts, 2022. URL <https://arxiv.org/abs/2112.06905>.
- 268 Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023. URL <https://arxiv.org/abs/2308.01861>.
- 269 Yixin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development, 2025. URL <https://arxiv.org/abs/2505.16975>.
- 270 Yixin Du et al. Swe-dev: Evaluating and training autonomous feature-driven software development, 2025.
- 271 Yongkang Du, Jen-tse Huang, Jieyu Zhao, and Lu Lin. Faircoder: Evaluating social bias of llms in code generation. *arXiv preprint arXiv:2501.05396*, 2025.
- 272 Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. Glm: General language model pretraining with autoregressive blank infilling, 2022. URL <https://arxiv.org/abs/2103.10360>.
- 273 Zhuoyun Du, Chen Qian, Wei Liu, Zihao Xie, Yifei Wang, Yufan Dang, Weize Chen, and Cheng Yang. Multi-agent software development through cross-team collaboration. *arXiv preprint arXiv:2406.08979*, 2024. URL <https://arxiv.org/abs/2406.08979>.
- 274 Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. From commit message generation to history-aware commit message completion, 2023. URL <https://arxiv.org/abs/2308.07655>.

- 275 Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. Towards translating real-world code with llms: A study of translating to rust, 2025. URL <https://arxiv.org/abs/2405.11514>.
- 276 Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1656–1668, 2024.
- 277 Michael Luo etc. Deepsw: Training a state-of-the-art coding agent from scratch by scaling rl. [N/A](#), 2025. Notion Blog.
- 278 Tom Everitt, Cristina Garbacea, Alexis Bellot, Jonathan Richens, Henry Papadatos, Siméon Campos, and Rohin Shah. Evaluating the goal-directedness of large language models. *arXiv preprint arXiv:2504.11844*, 2025.
- 279 Hugging Face. Codeparrot dataset, 2021.
- 280 Mohamad Fakih, Rahul Dharmaji, Halima Bouzidi, Gustavo Quiros Araya, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. Llm4cve: Enabling iterative automated vulnerability repair with large language models. *arXiv preprint arXiv:2501.03446*, 2025.
- 281 Jiajun Fan, Shuaike Shen, Chaoran Cheng, Yuxin Chen, Chumeng Liang, and Ge Liu. Online reward-weighted fine-tuning of flow matching with wasserstein regularization. In *The Thirteenth International Conference on Learning Representations*, 2025.
- 282 Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge, 2022. URL <https://arxiv.org/abs/2206.08853>.
- 283 Yue Fan, Handong Zhao, Ruiyi Zhang, Yu Shen, Xin Eric Wang, and Gang Wu. Gui-bee: Align gui action grounding to novel environments via autonomous exploration, 2025. URL <https://arxiv.org/abs/2501.13896>.
- 284 Sen Fang, Weiyuan Ding, and Bowen Xu. Evaloop: Assessing llm robustness in programming from a self-consistency perspective, 2025. URL <https://arxiv.org/abs/2505.12185>.
- 285 Tianqing Fang, Hongming Zhang, Zhisong Zhang, Kaixin Ma, Wenhao Yu, Haitao Mi, and Dong Yu. Webevolver: Enhancing web agent self-improvement with coevolving world model. *arXiv preprint arXiv:2504.21024*, 2025.
- 286 William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022. URL <https://arxiv.org/abs/2101.03961>.
- 287 Bill Feng and Clark Berke. The ai agent code of conduct: Automated guardrail policy-as-prompt synthesis. *arXiv preprint arXiv:2405.04859*, 2024.
- 288 Bo Feng, Jie Tang, Wenbo Li, et al. A review of automatic source code summarization. *SpringerLink Software Quality Journal*, 32(1):45–63, 2024. URL <https://link.springer.com/article/10.1007/s11219-024-09557-4>.

- 289 Sidong Feng, Mingyue Yuan, Jieshan Chen, Zhenchang Xing, and Chunyang Chen. Designing with language: Wireframing ui design intent with generative large language models. *arXiv preprint arXiv:2312.07755*, 2023.
- 290 Sidong Feng, Changhao Du, Huaxiao Liu, Qingnan Wang, Zhengwei Lv, Mengfei Wang, and Chunyang Chen. Breaking single-tester limits: Multi-agent llms for multi-user feature testing. *arXiv preprint arXiv:2506.17539*, 2025.
- 291 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL <https://arxiv.org/abs/2002.08155>.
- 292 Zhengyu Feng, Zhaoyang Jia, Yiran Li, Jiacheng Liu, Yifei Li, and Gang Wang. Deceptrprompt: Exploiting llm-driven code generation via adversarial natural language instructions, 2024.
- 293 Myles Foley and Sergio Maffeis. Apirl: Deep reinforcement learning for rest api fuzzing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 191–199, 2025.
- 294 Christopher Foster, Abhishek Gulati, Mark Harman, Inna Harper, Ke Mao, Jillian Ritchey, Hervé Robert, and Shubho Sengupta. Mutation-guided llm-based test generation at meta, 2025. URL <https://arxiv.org/abs/2501.12862>.
- 295 Daniel Fried, Armen Aghajanyan, Jessy Lin, et al. InCoder: A generative model for code infilling and synthesis. *ICLR*, 2023.
- 296 Alexander Froemmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Daniel Zheng, Satish Chandra, and Petros Maniatis. Resolving code review comments with machine learning. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '24*, page 204–215, New York, NY, USA, 2024. Association for Computing Machinery.
- 297 Jia Fu, Xinyu Yang, Hongzhi Zhang, Yahui Liu, Jingyuan Zhang, Qi Wang, Fuzheng Zhang, and Guorui Zhou. Klear-codetest: Scalable test case generation for code reinforcement learning, 2025. URL <https://arxiv.org/abs/2508.05710>.
- 298 Lingyue Fu, Hao Guan, Bolun Zhang, Haowei Yuan, Yaoming Zhu, Jun Xu, Zongyu Wang, Lin Qiu, Xunliang Cai, Xuezhi Cao, Weinan Liu, Weinan Zhang, and Yong Yu. Corecodebench: A configurable multi-scenario repository-level benchmark, 2025. URL <https://arxiv.org/abs/2507.05281>.
- 299 Rao Fu, Ziyang Luo, Hongzhan Lin, Zhen Ye, and Jing Ma. Scratcheval: Are gpt-4o smarter than my child? evaluating large multimodal models with visual programming challenges, 2024. URL <https://arxiv.org/abs/2411.18932>.
- 300 Zhong-Duo Fu, Hong-Ning Chen, Zhi-Xin Lv, Yu-Jun Zhang, Qing-Cai Zeng, Ye Yuan, and Fan Wu. Posterior-grpo: Rewarding reasoning processes in code generation. *arXiv preprint arXiv:2508.05170*, 2025.
- 301 Zhong-Duo Fu, Fan Wu, Yu-Jun Zhang, Zhi-Xin Lv, Qing-Cai Zeng, Hong-Ning Chen, and Ye Yuan. Smartcoder-r1: Towards secure and explainable smart contract generation with security-aware group relative policy optimization. *arXiv preprint arXiv:2509.09942*, 2025.

- 302 Stefano Fumero, Kai Huang, Matteo Boffa, Danilo Giordano, Marco Mellia, Zied Ben Houidi, and Dario Rossi. Cybersleuth: Autonomous blue-team llm agent for web attack forensics. *arXiv preprint arXiv:2508.20643*, 2025.
- 303 Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, and Franois Bodin. Milepost gcc: machine learning based research compiler. *proceedings of the gcc developers*, 2008.
- 304 Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. *arXiv preprint arXiv:2305.11854*, 2023.
- 305 Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=efFmBWioSc>.
- 306 Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145, 2024.
- 307 Jun Gao, Xian Zhang, Haoyi Li, et al. Survey on neural network-based automatic source code summarization. *Journal of Software Engineering and Applications*, 15(4):220–240, 2023. URL <https://www.sciengine.com/doi/10.13328/j.cnki.jos.006337>.
- 308 Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800GB Dataset of Diverse Text for Language Modeling, 2020. URL <https://arxiv.org/abs/2101.00027>.
- 309 Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.
- 310 Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.
- 311 Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. Xiyan-sql: A multi-generator ensemble framework for text-to-sql. *CoRR*, abs/2411.08599, 2024.
- 312 Zhaolin Gao, Wenhao Zhan, Jonathan D Chang, Gokul Swamy, Kianté Brantley, Jason D Lee, and Wen Sun. Regressing the relative future: Efficient policy optimization for multi-turn rlhf. *arXiv preprint arXiv:2410.04612*, 2024.
- 313 Paul Gauthier. Building a better repository map with tree sitter. <https://aider.chat/2023/10/22/repomap.html>, 2023.
- 314 Paul Gauthier. Linting code for llms with tree-sitter. <https://aider.chat/2024/05/22/linting.html>, 2024.
- 315 Paul Gauthier. How aider scored sota 26.3% on swe bench lite. <https://aider.chat/2024/05/22/swe-bench-lite.html>, 2024.

- 316 Paul Gauthier. Aider: AI pair programming in your terminal. Technical report, Aider AI, 2024. URL <https://github.com/paul-gauthier/aider>.
- 317 Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, et al. A survey of vibe coding with large language models. *arXiv preprint arXiv:2510.12399*, 2025.
- 318 Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Shaomeng Cao, Kechi Zhang, and Zhi Jin. Interpretation-based code summarization. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 113–124. IEEE, 2023.
- 319 Reza Gharibi, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. T5apr: Empowering automated program repair across languages through checkpoint ensemble. *Journal of Systems and Software*, 214:112083, 2024.
- 320 Reza Gharibi, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. Multimend: Multilingual program repair with context augmentation and multi-hunk patch generation. *arXiv preprint arXiv:2501.16044*, 2025.
- 321 GitHub. Github copilot, 2022. URL <https://github.com/copilot>.
- 322 GitHub. CodeQL: Semantic code analysis engine, 2024. URL <https://codeql.github.com/>. Accessed: 2024-01-15.
- 323 GitHub. GitHub Codespaces with AI integration. Technical report, GitHub Inc., 2024.
- 324 GitHub. GitHub Copilot: Your AI pair programmer. Technical report, GitHub, Inc., 2024. URL <https://github.com/features/copilot>.
- 325 GitHub. Github universe 2024: Multi-model github copilot. <https://github.blog/news-insights/product-news/universe-2024-copilot-multi-model/>, 2024.
- 326 GitHub Blog. Under the hood: Exploring the ai models powering github copilot. <https://github.blog/ai-and-ml/github-copilot/under-the-hood-exploring-the-ai-models-powering-github-copilot/>, 2025.
- 327 GlaiveAI. Glaive code assistant v3, 2024. URL <https://huggingface.co/datasets/glaive/glaive-code-assistant-v3>. Accessed: 2024.
- 328 Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu, Minlie Huang, Peng Zhang, Qinkai Zheng, Rui Lu, Shuaiqi Duan, Shudan Zhang, Shulin Cao, Shuxun Yang, Weng Lam Tam, Wenyi Zhao, Xiao Liu, Xiao Xia, Xiaohan Zhang, Xiaotao Gu, Xin Lv, Xinghan Liu, Xinyi Liu, Xinyue Yang, Xixuan Song, Xunkai Zhang, Yifan An, Yifan Xu, Yilin Niu, Yuantao Yang, Yueyan Li, Yushi Bai, Yuxiao Dong, Zehan Qi, Zhaoyu Wang, Zhen Yang, Zhengxiao Du, Zhenyu Hou, and Zihan Wang. Chatglm: A family of large language models from glm-130b to glm-4 all tools, 2024.
- 329 Luís F Gomes, Vincent J Hellendoorn, Jonathan Aldrich, and Rui Abreu. An exploratory study of ml sketches and visual code assistants. *arXiv preprint arXiv:2412.13386*, 2024.

- 330 Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of llms on syntax-aware code fill-in-the-middle tasks, 2024. URL <https://arxiv.org/abs/2403.04814>.
- 331 Linyuan Gong, Alvin Cheung, Mostafa Elhoushi, and Sida Wang. Structure-aware fill-in-the-middle pretraining for code, 2025. URL <https://arxiv.org/abs/2506.00204>.
- 332 Ran Gong, Qiuyuan Huang, Xiaojian Ma, Yusuke Noda, Zane Durante, Zilong Zheng, Demetri Terzopoulos, Li Fei-Fei, Jianfeng Gao, and Hoi Vo. MindAgent: Emergent gaming interaction. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 3154–3183, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.200. URL <https://aclanthology.org/2024.findings-naacl.200/>.
- 333 Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. Diffuseq: Sequence to sequence text generation with diffusion models, 2023. URL <https://arxiv.org/abs/2210.08933>.
- 334 Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. Diffucoder: Understanding and improving masked diffusion models for code generation. *arXiv preprint arXiv:2506.20639*, 2025.
- 335 Google. Gemini cli documentation, 2024.
- 336 Google. Gemini cli. <https://cloud.google.com/gemini/docs/cli>, 2024.
- 337 Google. Announcing the agent2agent protocol (a2a). <https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/>, 2025.
- 338 Google. Gemini 2.0 flash — model card. <https://storage.googleapis.com/model-cards/documents/gemini-2-flash.pdf>, 2025.
- 339 Google Cloud. Google cloud code. <https://cloud.google.com/code>, 2024.
- 340 Google Cloud. Duet AI for developers: Code faster with AI assistance. Technical report, Google LLC, 2024.
- 341 Satya Krishna Gorti, Ilan Gofman, Zhaoyan Liu, Jiapeng Wu, Noël Vouitsis, Guangwei Yu, Jesse C. Cresswell, and Rasa Hosseinzadeh. Msc-sql: Multi-sample critiquing small language models for text-to-sql translation. *CoRR*, abs/2410.12916, 2024.
- 342 Zhibin Gou, Zihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=Sx038qxjek>.
- 343 Graphite. Graphite reviewer: Repository-aware ai reviews. <https://graphite.dev/features/ai-reviewer>, 2024.
- 344 Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh

- Tang, Bobbie Chern, Charlotte Caucheteux, et al. The llama 3 herd of models, 2024. URL <https://arxiv.org/abs/2407.21783>.
- 345 Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024. URL <https://openreview.net/forum?id=tEYskw1VY2>.
- 346 Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.
- 347 Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*, 2024.
- 348 Shangding Gu, Long Yang, Yali Du, Guang Chen, Florian Walter, Jun Wang, and Alois Knoll. A review of safe reinforcement learning: Methods, theories and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.
- 349 Sijia Gu, Noor Nashid, and Ali Mesbah. Llm test generation via iterative hybrid program analysis. *arXiv preprint arXiv:2503.13580*, 2025.
- 350 Siqi Gu, Quanjun Zhang, Kecheng Li, Chunrong Fang, Fangyuan Tian, Liuchuan Zhu, Jianyi Zhou, and Zhenyu Chen. Testart: Improving llm-based unit testing via co-evolution of automated generation and repair iteration, 2025. URL <https://arxiv.org/abs/2408.03095>.
- 351 Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, et al. Is your llm secretly a world model of the internet? model-based planning for web agents. *arXiv preprint arXiv:2411.06559*, 2024.
- 352 Yanchu Guan, Dong Wang, Zhixuan Chu, Shiyu Wang, Feiyue Ni, Ruihua Song, and Chenyi Zhuang. Intelligent agents with llm-based process automation. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5018–5027, 2024.
- 353 Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. TextArena. *arXiv e-prints*, art. arXiv:2504.11442, April 2025. doi: 10.48550/arXiv.2504.11442.
- 354 Etrash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, et al. Openthoughts: Data recipes for reasoning models. *arXiv preprint arXiv:2506.04178*, 2025.
- 355 Jiayi Gui, Yiming Liu, Jiale Cheng, Xiaotao Gu, Xiao Liu, Hongning Wang, Yuxiao Dong, Jie Tang, and Minlie Huang. LogicGame: Benchmarking Rule-Based Reasoning Abilities of Large Language Models. *arXiv e-prints*, art. arXiv:2408.15778, August 2024. doi: 10.48550/arXiv.2408.15778.
- 356 Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, et al. Webcode2m: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pages 1834–1845, 2025.
- 357 Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, et al. Uicopilot: Automating ui synthesis via hierarchical

code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pages 1846–1855, 2025.

- 358 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 317–330, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926423. URL <https://doi.org/10.1145/1926385.1926423>.
- 359 Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. *CoRR*, abs/2306.11644, 2023. doi: 10.48550/ARXIV.2306.11644. URL <https://doi.org/10.48550/arXiv.2306.11644>.
- 360 Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Dixin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL <https://openreview.net/forum?id=jLoC4ez43PZ>.
- 361 Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL 2022, Dublin, Ireland, May 22-27, 2022, pages 7212–7225. Association for Computational Linguistics, 2022. doi: 10.18653/V1/2022.ACL-LONG.499. URL <https://doi.org/10.18653/v1/2022.acl-long.499>.
- 362 Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseekcoder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024. doi: 10.48550/ARXIV.2401.14196. URL <https://doi.org/10.48550/arXiv.2401.14196>.
- 363 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qi-hao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- 364 Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via BERT. In *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*, pages 1–8. IEEE, 2021.
- 365 Hanyang Guo, Xiaoheng Xie, Hong-Ning Dai, Peng Di, Yu Zhang, Bishenghui Tao, and Zibin Zheng. Accelerating automatic program repair with dual retrieval-augmented fine-tuning and patch generation on large language models. *arXiv preprint arXiv:2507.10103*, 2025.
- 366 Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the*

57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers, pages 4524–4535. Association for Computational Linguistics, 2019.

- 367 Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi LI, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhui Chen, and Jie Fu. Codeeditorbench: Evaluating code editing capability of large language models, 2025. URL <https://arxiv.org/abs/2404.03543>.
- 368 Jinyao Guo, Chengpeng Wang, Dominic Deluca, Jinjie Liu, Zhuo Zhang, and Xiangyu Zhang. Bugscope: Learn to find bugs like human. *arXiv preprint arXiv:2507.15671*, 2025.
- 369 Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. Omnidirl: A multilingual and multimodal benchmark for github issue resolution, 2025. URL <https://arxiv.org/abs/2505.04606>.
- 370 Yiwei Guo, Shaobin Zhuang, Kunchang Li, Yu Qiao, and Yali Wang. Transagent: Transfer vision-language foundation models with heterogeneous agent collaboration, 2024. URL <https://arxiv.org/abs/2410.12183>.
- 371 Izzeddin Gur, Ofir Nachum, Yingjie Miao, Mustafa Safdari, Austin Huang, Aakanksha Chowdhery, Sharan Narang, Noah Fiedel, and Aleksandra Faust. Understanding HTML with large language models. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 2803–2821, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.185. URL <https://aclanthology.org/2023.findings-emnlp.185/>.
- 372 Idan Habler, Ken Huang, Vineeth Sai Narajala, and Prashant Kulkarni. Building a secure agentic ai application leveraging a2a protocol. *arXiv preprint arXiv:2504.16902*, 2025.
- 373 Dongyoon Hahn, Woogyeol Jin, June Suk Choi, Sungsoo Ahn, and Kimin Lee. Enhancing llm agent safety via causal influence prompting. *arXiv preprint arXiv:2507.00979*, 2025.
- 374 Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. On the impacts of contexts on repository-level code generation, 2025. URL <https://arxiv.org/abs/2406.11927>.
- 375 Md Asif Haider, Ayesha Binte Mostofa, Sk Sabit Bin Mosaddek, Anindya Iqbal, and Toufique Ahmed. Prompting and fine-tuning large language models for automated code review comment generation. *arXiv preprint arXiv:2411.10129*, 2024.
- 376 Hossein Hajipour, Rijnard van Tonder, Reza nadri, and Arman Cohan. Hexacoder: Secure code generation via oracle-guided synthetic training data, 2024.
- 377 Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. Ssd-lm: Semi-autoregressive simplex-based diffusion language model for text generation and modular control, 2023. URL <https://arxiv.org/abs/2210.17432>.
- 378 Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings, 2024. URL <https://arxiv.org/abs/2305.11554>.
- 379 Ariful Haque, Sunzida Siddique, Md Mahfuzur Rahman, Ahmed Rafi Hasan, Laxmi Rani Das, Marufa Kamal, Tasnim Masura, and Kishor Datta Gupta. Sok: Exploring hallucinations and security risks in ai-assisted software development with insights for llm deployment. *arXiv preprint arXiv:2502.18468*, 2025.

- 380 Mohammad Saqib Hasan, Saikat Chakraborty, Santu Karmaker, and Niranjan Balasubramanian. Teaching an old llm secure coding: Localized preference optimization on distilled preferences, 2025.
- 381 Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Does ai-assisted coding deliver? a difference-in-differences study of cursor's impact on software projects. *arXiv e-prints*, pages arXiv–2511, 2025.
- 382 Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.
- 383 Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models, 2024. URL <https://arxiv.org/abs/2401.13919>.
- 384 Jia He and Zhaoxi Yan. Large-scale, diverse, and realistic dataset for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1117–1128. IEEE, 2023.
- 385 Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin T. Vechev. Debin: Predicting debug information in stripped binaries. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15–19, 2018*, pages 1667–1680. ACM, 2018.
- 386 Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation. *arXiv preprint arXiv:2402.09497*, 2024.
- 387 Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. Skywork open reasoner 1 technical report. *CoRR*, abs/2505.22312, 2025. doi: 10.48550/ARXIV.2505.22312. URL <https://doi.org/10.48550/arXiv.2505.22312>.
- 388 Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, et al. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*, 2025.
- 389 Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. From code to courtroom: Llms as the new software judges. *arXiv preprint arXiv:2503.02246*, 2025.
- 390 Mengliang He, Jiayi Zeng, Yankai Jiang, Wei Zhang, Zeming Liu, Xiaoming Shi, and Aimin Zhou. Flow2code: Evaluating large language models for flowchart-based code generation capability. *arXiv preprint arXiv:2506.02073*, 2025.
- 391 Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. X-SQL: reinforce schema representation with context. *CoRR*, abs/1908.08113, 2019.
- 392 Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma. Swe-perf: Can language models optimize code performance on real-world repositories?, 2025. URL <https://arxiv.org/abs/2507.12415>.

- 393 Zhenyu He, Qingping Yang, Wei Sheng, Xiaojian Zhong, Kechi Zhang, Chenxin An, Wenlei Shi, Tianle Cai, Di He, Jiaze Chen, Jingjing Xu, and Mingxuan Wang. Swe-swiss: A multi-task fine-tuning and rl recipe for high-performance issue resolution, 2025. URL <https://www.notion.so/SWE-Swiss-A-Multi-Task-Fine-Tuning-and-RL-Recipe-for-High-Performance-Issue-Resolution-21e174dedd4880ea829ed4c861c44f88>.
- 394 Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, et al. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *arXiv preprint arXiv:2504.11456*, 2025.
- 395 Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. Hardtests: Synthesizing high-quality test cases for llm coding. *arXiv preprint arXiv:2505.24098*, 2025.
- 396 Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021. URL <https://arxiv.org/abs/2105.09938>.
- 397 Dan Hendrycks, Steven Basart, Saurav Kadavath, et al. Measuring coding challenge competence with APPS. *NeurIPS*, 2021.
- 398 Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
- 399 Samuel Höglund and Josef Khedri. Comparison between rlhf and rlaif in fine-tuning a large language model, 2023.
- 400 Hyunsun Hong and Jongmoon Baik. Retrieval-augmented code review comment generation. *arXiv preprint arXiv:2506.11591*, 2025.
- 401 Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. MetaGPT: Meta programming for multi-agent collaborative framework. In *ICLR*, 2024.
- 402 Iman Hosseini and Brendan Dolan-Gavitt. Beyond the C: retargetable decompilation using neural machine translation. *CoRR*, abs/2212.08950, 2022.
- 403 Xinyi Hou, Yanjie Zhao, Yue Liu, et al. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–45, 2024.
- 404 Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *arXiv preprint arXiv:2503.23278*, 2025.
- 405 Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.
- 406 Haichuan Hu, Xiaochen Xie, and Quanjun Zhang. Repair-r1: Better test before repair. *arXiv preprint arXiv:2507.22853*, 2025.

- 407 Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. Reinforce++: An efficient rlhf algorithm with robustness to both prompt and reward models, 2025. URL <https://arxiv.org/abs/2501.03262>.
- 408 Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. Reinforce++: Stabilizing critic-free policy optimization with global advantage normalization, 2025. URL <https://arxiv.org/abs/2501.03262>.
- 409 Jian Hu, Mingjie Liu, Ximing Lu, Fang Wu, Zaid Harchaoui, Shizhe Diao, Yejin Choi, Pavlo Molchanov, Jun Yang, Jan Kautz, and Yi Dong. Brorl: Scaling reinforcement learning via broadened exploration, 2025. URL <https://arxiv.org/abs/2510.01180>.
- 410 Lanxiang Hu, Qiyu Li, Anze Xie, Nan Jiang, Ion Stoica, Haojian Jin, and Hao Zhang. GameArena: Evaluating LLM Reasoning through Live Computer Games. *arXiv e-prints*, art. arXiv:2412.06394, December 2024. doi: 10.48550/arXiv.2412.06394.
- 411 Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchen Wang, and Cuiyun Gao. Coderepoqa: A large-scale benchmark for software engineering question answering. *arXiv preprint arXiv:2412.14764*, 2024.
- 412 Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchen Wang, and Cuiyun Gao. A real-world benchmark for evaluating fine-grained issue solving capabilities of large language models, 2024. URL <https://arxiv.org/abs/2411.18019>.
- 413 Wenhao Hu, Jinhao Duan, Chunchen Wei, Li Zhang, Yue Zhang, and Kaidi Xu. Dynacode: A dynamic complexity-aware code benchmark for evaluating large language models in code generation, 2025. URL <https://arxiv.org/abs/2503.10452>.
- 414 Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC ’18*, page 200–210. Association for Computing Machinery, 2018. ISBN 9781450357142. doi: 10.1145/3196321.3196334. URL <https://doi.org/10.1145/3196321.3196334>.
- 415 Yaojie Hu, Xingjian Shi, Qiang Zhou, and Lee Pike. Fix bugs with transformer through a neural-symbolic edit grammar. *arXiv preprint arXiv:2204.06643*, 2022.
- 416 B. Huang and et al. Enhancing llms in coding through multi-perspective self-consistency (mpsc). *arXiv preprint arXiv:2309.17272*, 2024. URL <https://arxiv.org/abs/2309.17272>.
- 417 Bin Huang and et al. Codcot: Self-examining code generation with chain-of-thought and test synthesis. *arXiv preprint arXiv:2309.17272*, 2023. URL <https://arxiv.org/abs/2309.17272>.
- 418 Dong HUANG, Yuhao QING, Weiyi Shang, Heming Cui, and Jie Zhang. Effibench: Benchmarking the efficiency of automatically generated code. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL <https://openreview.net/forum?id=30XanJanJP>.
- 419 Dong Huang, Jie M Zhang, Qingwen Bu, Xiaofei Xie, Junjie Chen, and Heming Cui. Bias testing and mitigation in llm-based code generation. *ACM Transactions on Software Engineering and Methodology*, 2024.

- 420 Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. Efficoder: Enhancing code generation in large language models through efficiency-aware fine-tuning, 2025. URL <https://arxiv.org/abs/2410.10209>.
- 421 Linghan Huang, Peizhou Zhao, Lei Ma, and Huaming Chen. On the challenges of fuzzing techniques via large language models. In Rong N. Chang, Carl K. Chang, Jingwei Yang, Nimantha Atukorala, Dan Chen, Sumi Helal, Sasu Tarkoma, Qiang He, Tevfik Kosar, Claudio A. Ardagna, Javier Berrocal, Kaoutar El Maghaouri, and Yanchun Sun, editors, *IEEE International Conference on Software Services Engineering, SSE 2025, Helsinki, Finland, July 7-12, 2025*, pages 162–171. IEEE, 2025. doi: 10.1109/SSE67621.2025.00028. URL <https://doi.org/10.1109/SSE67621.2025.00028>.
- 422 Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308. IEEE, 2019.
- 423 Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. Opencoder: The open cookbook for top-tier code large language models. 2024. URL <https://arxiv.org/pdf/2411.04905.pdf>.
- 424 Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Xianzhen Luo, Qiufeng Wang, YuanTao Fan, Qingfu Zhu, Zhaoxiang Zhang, Yang Gao, Jie Fu, Qian Liu, Houyi Li, Ge Zhang, Yuan Qi, Yinghui Xu, Wei Chu, and Zili Wang. Opencoder: The open cookbook for top-tier code large language models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 33167–33193. Association for Computational Linguistics, 2025. URL <https://aclanthology.org/2025.acl-long.1591/>.
- 425 Yangsibo Huang, Samyak Gupta, Zexuan Zhong, Kai Li, and Danqi Chen. Privacy implications of retrieval-based language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 7583–7596, 2023.
- 426 Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. Da-code: Agent data science code generation benchmark for large language models, 2024. URL <https://arxiv.org/abs/2410.07331>.
- 427 Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. Towards automatically generating block comments for code snippets. *Information and Software Technology*, 127:106373, 2020.
- 428 Evan Hubinger, Chris van Merwijk, Vladimir Mikulik, Joar Skalse, and Scott Garrabrant. Risks from learned optimization in advanced machine learning systems. *arXiv preprint arXiv:1906.01820*, 2019.
- 429 Frederikus Hudi, Genta Indra Winata, Ruochen Zhang, and Alham Fikri Aji. TextGames: Learning to Self-Play Text-Based Puzzle Games via Language Model Reasoning. *arXiv e-prints*, art. arXiv:2502.18431, February 2025. doi: 10.48550/arXiv.2502.18431.

- 430 Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL <https://github.com/huggingface/open-r1>.
- 431 huggingface. smolagents, 2025. URL <https://huggingface.co/docs/smolagents/index>.
- 432 Sean Hughes, Harm de Vries, Jennifer Robinson, Carlos Muñoz Ferrandis, Loubna Ben Allal, Leandro von Werra, Jennifer Ding, Sébastien Paquet, Yacine Jernite, et al. The bigcode project governance card. *arXiv preprint arXiv:2312.03872*, 2023.
- 433 Binyuan Hui, Ruiying Geng, Lihan Wang, Bowen Qin, Yanyang Li, Bowen Li, Jian Sun, and Yongbin Li. S²sql: Injecting syntax to question-schema interaction graph encoder for text-to-sql parsers. In *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 1254–1262. Association for Computational Linguistics, 2022.
- 434 Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL <https://arxiv.org/abs/2409.12186>.
- 435 Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search, 2020.
- 436 Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *arXiv preprint arXiv:2410.24117*, 2024.
- 437 Shima Imani, Liang Du, and Harsh Shrivastava. Mathprompter: Mathematical reasoning using large language models. *arXiv preprint arXiv:2303.05398*, 2023.
- 438 ISE-UIUC. Magicoder-oss-instruct-75k, 2024. URL <https://huggingface.co/datasets/is-e-uiuc/Magicoder-OSS-Instruct-75K>. Accessed: 2024.
- 439 Md. Ashraful Islam and et al. Mapcoder: Multi-agent code generation for competitive programming. *arXiv preprint arXiv:2401.08500*, 2024. URL <https://arxiv.org/abs/2401.08500>.
- 440 Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Elias Bou-Harb, and Peyman Najafirad. Enhancing source code security with llms: Demystifying the challenges and generating reliable repairs. *arXiv preprint arXiv:2407.03975*, 2024.
- 441 Hamish Ivison, Yizhong Wang, Jiacheng Liu, Zeqiu Wu, Valentina Pyatkin, Nathan Lambert, Noah A Smith, Yejin Choi, and Hanna Hajishirzi. Unpacking dpo and ppo: Disentangling best practices for learning from preference feedback. *Advances in neural information processing systems*, 37:36602–36633, 2024.
- 442 Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations*.

- 443 Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- 444 Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL <https://arxiv.org/abs/2403.07974>.
- 445 Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. RLBench: The Robot Learning Benchmark & Learning Environment. *arXiv e-prints*, art. arXiv:1909.12271, September 2019. doi: 10.48550/arXiv.1909.12271.
- 446 Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. *CoTran: An LLM-Based Code Translator Using Reinforcement Learning with Feedback from Compiler and Symbolic Execution*. IOS Press, October 2024. ISBN 9781643685489. doi: 10.3233/faia240968. URL <http://dx.doi.org/10.3233/FAIA240968>.
- 447 Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In *ECAI*, 2024.
- 448 Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. Combining large language models with static analyzers for code review generation. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 174–186. IEEE, 2025.
- 449 Ibrahim Jemal, Mathis Rocher, Tegawendé F Bissyandé, and Yves Le Traon. An exploratory study on fine-tuning large language models for secure code generation. *arXiv preprint arXiv:2403.04231*, 2024.
- 450 Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. Question selection for interactive program synthesis. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1143–1158. ACM, 2020. doi: 10.1145/3385412.3386025. URL <https://doi.org/10.1145/3385412.3386025>.
- 451 Suhwan Ji, Sanghwa Lee, Changsup Lee, Yo-Sub Han, and Hyeonseung Im. Impact of large language models of code on fault localization. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 302–313. IEEE, 2025.
- 452 Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL <https://arxiv.org/abs/2310.06825>.
- 453 Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL <https://arxiv.org/abs/2401.04088>.

- 454 Gangwei Jiang, Yahui Liu, Zhaoyi Li, Qi Wang, Fuzheng Zhang, Linqi Song, Ying Wei, and Defu Lian. What makes a good reasoning chain? uncovering structural patterns in long chain-of-thought reasoning. *arXiv preprint arXiv:2505.22148*, 2025.
- 455 Hao Jiang, Qi Liu, Rui Li, Shengyu Ye, and Shijin Wang. Cursorcore: Assist programming through aligning anything. *arXiv preprint arXiv:2410.07002*, 2024.
- 456 Hongchao Jiang, Yiming Chen, Yushi Cao, Hung yi Lee, and Robby T. Tan. Code-judgebench: Benchmarking llm-as-a-judge for coding tasks, 2025. URL <https://arxiv.org/abs/2507.10535>.
- 457 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL <https://arxiv.org/abs/2406.00515>.
- 458 Lingjie Jiang, Shaohan Huang, Xun Wu, Yixia Li, Dongdong Zhang, and Furu Wei. Vis-codex: Unified multimodal code generation via merging vision and coding models. *arXiv preprint arXiv:2508.09945*, 2025.
- 459 Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. Knod: Domain knowledge distilled tree decoder for automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1251–1263. IEEE, 2023.
- 460 Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Nova⁺: Generative language models for binaries. *CoRR*, abs/2311.13721, 2023.
- 461 Siyuan Jiang and Collin McMillan. Towards automatic generation of short summaries of commits, 2017. URL <https://arxiv.org/abs/1703.09603>.
- 462 Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation, 2017. URL <https://arxiv.org/abs/1708.09492>.
- 463 Ting-En Jiang, Yizhen Wang, Jing-Cheng Pang, Pei-Hung Lin, Chen-Lin Zhang, Xin-Yang Liu, Yi-Ling Liao, Ching-Ting Chen, Chia-Chun Lo, and Shao-Hua Sun. Purpcode: Reasoning for safer code generation, 2024.
- 464 Weipeng Jiang, Xuanqi Gao, Juan Zhai, Shiqing Ma, Xiaoyu Zhang, Ziyan Lei, and Chao Shen. From effectiveness to efficiency: Uncovering linguistic bias in large language model-based code generation. *arXiv preprint arXiv:2406.00602*, 2024.
- 465 Yilei Jiang, Yaozhi Zheng, Yuxuan Wan, Jiaming Han, Qunzhong Wang, Michael R Lyu, and Xiangyu Yue. Screencoder: Advancing visual-to-code generation for front-end automation via modular multimodal agents. *arXiv preprint arXiv:2507.22827*, 2025.
- 466 Yuancheng Jiang, Roland Yap, and Zhenkai Liang. Oss-bench: Benchmark generator for coding llms, 2025. URL <https://arxiv.org/abs/2505.12331>.
- 467 Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. On the evaluation of neural code translation: Taxonomy and benchmark, 2023. URL <https://arxiv.org/abs/2308.08961>.
- 468 Alejandro et al. Jimenez. Swt-bench: Benchmarking llms for software testing and bug repair. In *ICSE*, 2024.

- 469** Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=VTF8yNQM66>.
- 470** Dongming Jin, Zhi Jin, Xiaohong Chen, and Chunhui Wang. Mare: Multi-agents collaboration framework for requirements engineering. *arXiv preprint arXiv:2405.03256*, 2024.
- 471** Dongming Jin, Weisong Sun, Jiangping Huang, Peng Liang, Jifeng Xuan, Yang Liu, and Zhi Jin. iredev: A knowledge-driven multi-agent framework for intelligent requirements development. *arXiv preprint arXiv:2507.13081*, 2025.
- 472** Hangzhan Jin and Mohammad Hamdaqa. Ccci: Code completion with contextual information for complex data transfer tasks using large language models, 2025. URL <https://arxiv.org/abs/2503.23231>.
- 473** Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 1646–1656, 2023.
- 474** Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140, 2023.
- 475** Rinkesh Joshi and Nafiseh Kahani. Comparative study of reinforcement learning in github pull request outcome predictions. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 489–500. IEEE, 2024.
- 476** Tae-Hwan Jung. Commitbert: Commit message generation using pre-trained programming language model. *arXiv preprint arXiv:2105.14242*, 2021.
- 477** René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.
- 478** Greg Kamradt. Snake bench: Competitive snake game simulation with llms. <https://github.com/gkamradt/SnakeBench>, 2025. Accessed on: Month Day, Year.
- 479** Nikhil Kandpal, Eric Wallace, and Colin Raffel. Deduplicating training data mitigates privacy risks in language models. In *International Conference on Machine Learning*, pages 10419–10431. PMLR, 2022.
- 480** Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1 (FSE):1424–1446, 2024.
- 481** Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1 (FSE):1424–1446, 2024.

- 482 Manav Nitin Kapadnis, Atharva Naik, and Carolyn Rose. Crscore++: Reinforcement learning with verifiable tool and ai feedback for code review. *arXiv preprint arXiv:2506.00296*, 2025.
- 483 Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL <https://arxiv.org/abs/2001.08361>.
- 484 Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. Omniact: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web, 2024. URL <https://arxiv.org/abs/2402.17553>.
- 485 Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3559548. URL <https://doi.org/10.1145/3551349.3559548>.
- 486 Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval, 2023. URL <https://arxiv.org/abs/2303.03004>.
- 487 Vishal Khare, Vijay Saini, Deepak Sharma, Anand Kumar, Ankit Rana, and Anshul Yadav. Deputydev–ai powered developer assistant: Breaking the code review logjam through contextual ai to boost developer productivity. *arXiv preprint arXiv:2508.09676*, 2025.
- 488 Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Sai Surya Duvvuri, Manzil Zaheer, Inderjit S. Dhillon, David Brandfonbrener, and Rishabh Agarwal. The art of scaling reinforcement learning compute for llms, 2025. URL <https://arxiv.org/abs/2510.13786>.
- 489 Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt? In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2445–2451, 2023. doi: 10.1109/SMC53992.2023.10394237.
- 490 Taeyoun Kim, Jaemin Kweon, Sang woo Lee, and Yoojin Choi. Reasoning as an adaptive defense for safety, 2025.
- 491 Sven Kirchner and Alois C Knoll. Generating automotive code: Large language models for software development and verification in safety-critical systems. *arXiv preprint arXiv:2506.04038*, 2025.
- 492 Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The Stack: 3 TB of Permissively Licensed Source Code, 2022. URL <https://arxiv.org/abs/2211.15533>.
- 493 Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code, 2022. URL <https://arxiv.org/abs/2211.15533>.

- 494 KodCode. Kodcode-v1 dataset, 2024. URL <https://huggingface.co/datasets/KodCode/KodCode-V1>. Accessed: 2024.
- 495 Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.50. URL <https://aclanthology.org/2024.acl-long.50/>.
- 496 Dezhong Kong, Shi Lin, Zhenhua Xu, Zhebo Wang, Minghao Li, Yufeng Li, Yilun Zhang, Hujin Peng, Zeyang Sha, Yuyuan Li, et al. A survey of llm-driven ai agent communication: Protocols, security risks, and defense countermeasures. *arXiv preprint arXiv:2506.19676*, 2025.
- 497 Tomasz Korbak, Kejian Shi, Angelica Chen, Rasika Vinayak Bhalerao, Christopher Buckley, Jason Phang, Samuel R Bowman, and Ethan Perez. Pretraining language models with human preferences. In *International Conference on Machine Learning*, pages 17506–17533. PMLR, 2023.
- 498 Hans-Alexander Kruse, Tim Puhlfürß, and Walid Maalej. Can developers prompt? A controlled experiment for code documentation generation. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*, pages 574–586. IEEE, 2024. doi: 10.1109/ICSME58944.2024.00058. URL <https://doi.org/10.1109/ICSME58944.2024.00058>.
- 499 L. Kuang, C. Zhou, and X. Yang. Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems. *Automated Software Engineering*, 29:43, 2022. doi: 10.1007/s10515-022-00341-1. URL <https://doi.org/10.1007/s10515-022-00341-1>.
- 500 Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. *SPoC: search-based pseudocode to code*. Curran Associates Inc., Red HEvaluating Large Language Models Trained on Code ook, NY, USA, 2019.
- 501 Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, et al. Training language models to self-correct via reinforcement learning. *arXiv preprint arXiv:2409.12917*, 2024.
- 502 Jahnavi Kumar, Venkata Lakshmana Sasaank Janapati, Mokshith Reddy Tanguturi, and Sridhar Chimalakonda. I can't share code, but i need translation – an empirical study on code translation through federated llm, 2025. URL <https://arxiv.org/abs/2501.05724>.
- 503 Sandhini Kumar, Anish Agarwal, Chetna Gupta, Shivang Sharma, and Manasi Singh. Llamafirewall: An open source guardrail system for building secure ai agents. *arXiv preprint arXiv:2406.01288*, 2024.
- 504 Beck LaBash, August Rosedale, Alex Reents, Lucas Negritto, and Colin Wiel. Res-q: Evaluating code-editing large language model systems at the repository scale, 2024. URL <https://arxiv.org/abs/2406.16801>.

- 505 Inception Labs, Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, Stefano Ermon, Aditya Grover, and Volodymyr Kuleshov. Mercury: Ultra-fast language models based on diffusion, 2025. URL <https://arxiv.org/abs/2506.17298>.
- 506 Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages, 2020. URL <https://arxiv.org/abs/2006.03511>.
- 507 Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems*, 34:14967–14979, 2021.
- 508 Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 628–639. IEEE, 2019.
- 509 Hanyu Lai, Xiao Liu, Yanxiao Zhao, Han Xu, Hanchen Zhang, Bohao Jing, Yanyu Ren, Shuntian Yao, Yuxiao Dong, and Jie Tang. Computerrl: Scaling end-to-end online reinforcement learning for computer use agents. *arXiv preprint arXiv:2508.14040*, 2025.
- 510 Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR, 2023. URL <https://proceedings.mlr.press/v202/lai23b.html>.
- 511 Djamel Rassem Lamouri, Iheb Nassim Aouadj, Smail Kourta, and Riyadh Baghdadi. Pearl: Automatic code optimization using deep reinforcement learning. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pages 959–974, 2025.
- 512 Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/8636419dea1aa9fdbd25fc4248e702da4-Abstract-Conference.html.
- 513 Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.
- 514 Hung Le, Ha-Thanh Nguyen, Zhaoyuan Chen, Siru Ouyang, and Rudolf Marcel. Indict: Code generation with internal dialogues of critiques for both security and helpfulness, 2024.
- 515 Thanh Le-Cong, Bach Le, and Toby Murray. Semantic-guided search for efficient program repair with large language models. *arXiv preprint arXiv:2410.16655*, 2024.

- 516 Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. Repoexec: Evaluate code generation with a repository-level executable benchmark. *arXiv e-prints*, pages arXiv–2406, 2024.
- 517 Changyoong Lee, Yeon Seonwoo, and Alice Oh. Cs1qa: A dataset for assisting code-based question answering in an introductory programming course. *arXiv preprint arXiv:2210.14494*, 2022.
- 518 Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. MCS-SQL: leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *CoRR*, abs/2405.07467, 2024.
- 519 Dongjun Lee, Changho Hwang, and Kimin Lee. Learning to generate unit test via adversarial reinforcement learning. *arXiv preprint arXiv:2508.21107*, 2025.
- 520 Hao-Ping Lee, Yu-Ju Yang, Matthew Bilik, Isadora Krsek, Thomas Serban von Davier, Kyzyl Monteiro, Jason Lin, Shivani Agarwal, Jodi Forlizzi, and Sauvik Das. Privy: Envisioning and mitigating privacy risks for consumer-facing ai product concepts, 2025.
- 521 Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. 2023.
- 522 Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, and Sushant Prakash. RLAIF vs. RLHF: scaling reinforcement learning from human feedback with AI feedback. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=uydQ2W41KO>.
- 523 Hyeonseok Lee, Gabin An, and Shin Yoo. METAMON: Finding Inconsistencies between Program Documentation and Behavior using Metamorphic LLM Queries. *arXiv e-prints*, art. arXiv:2502.02794, February 2025. doi: 10.48550/arXiv.2502.02794.
- 524 Hyunji Lee, Minseon Kim, Chinmay Singh, Matheus Pereira, Atharv Sonwane, Isadora White, Elias Stengel-Eskin, Mohit Bansal, Zhengyan Shi, Alessandro Sordoni, et al. Gistify! codebase-level understanding via runtime execution. *arXiv preprint arXiv:2510.26790*, 2025.
- 525 Jaewook Lee, Jeongah Lee, Wanyong Feng, and Andrew Lan. From text to visuals: Using llms to generate math diagrams with vector graphics. *arXiv preprint arXiv:2503.07429*, 2025.
- 526 Juyoung Lee, Yeonsu Jeong, Taehyun Han, and Taejin Lee. Logresp-agent: A recursive ai framework for context-aware log anomaly detection and ttp analysis. *Applied Sciences*, 15(13):7237, 2025.
- 527 Bin Lei, Yuchen Li, and Qiuwu Chen. AutoCoder: Enhancing Code Large Language Model with \textsc{AIEV-Instruct}, 2024.
- 528 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.
- 529 Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 837–849. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00085. URL <https://dl.acm.org/doi/10.1109/ICSE48619.2023.00085>.

- 530 Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020. URL <https://arxiv.org/abs/2006.16668>.
- 531 Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Peiyuan Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. LLaVA-onevision: Easy visual task transfer. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL <https://openreview.net/forum?id=zKv8qULV6n>.
- 532 Bolun Li, Zhihong Sun, Tao Huang, Hongyu Zhang, Yao Wan, Ge Li, Zhi Jin, and Chen Lyu. Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. *Proceedings of the ACM on Software Engineering*, 1(FSE):182–203, 2024.
- 533 Chaofan Li, Jianlyu Chen, Yingxia Shao, Defu Lian, and Zheng Liu. Towards a generalist code embedding model based on massive data synthesis. *arXiv preprint arXiv:2505.12697*, 2025.
- 534 Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*, 2023.
- 535 D. Li and et al. S*: Test-time scaling for code generation. *arXiv preprint arXiv:2502.14382*, 2025. URL <https://arxiv.org/abs/2502.14382>.
- 536 Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Shishir G Patil, Matei Zaharia, Joseph E Gonzalez, and Ion Stoica. Llms can easily learn to reason from demonstrations structure, not content, is what matters! *arXiv preprint arXiv:2502.07374*, 2025.
- 537 Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. Hybrid automated program repair by combining large language models and program analysis. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–28, 2025.
- 538 Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*, 2024.
- 539 Han Li, Yuling Shi, Shaixin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. Swe-debate: Competitive multi-agent debate for software issue resolution. *arXiv preprint arXiv:2507.23348*, 2025.
- 540 Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. RESDSQL: decoupling schema linking and skeleton parsing for text-to-sql. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*, pages 13067–13075. AAAI Press, 2023.
- 541 Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3):127, 2024.
- 542 Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. Omnisql: Synthesizing high-quality text-to-sql data at scale. *CoRR*, abs/2503.02240, 2025.

- 543 Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. Patchpilot: A stable and cost-efficient agentic patching framework. *arXiv e-prints*, pages arXiv–2502, 2025.
- 544 Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*, 13(9):9, 2024.
- 545 Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories, 2024. URL <https://arxiv.org/abs/2404.00599>.
- 546 Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhu Li, Fei Huang, and Yongbin Li. Deeval: A manually-annotated code generation benchmark aligned with real-world code repositories, 2024. URL <https://arxiv.org/abs/2405.19856>.
- 547 Jia Li, Xuyuan Guo, Lei Li, Kechi Zhang, Ge Li, Zhengwei Tao, Fang Liu, Chongyang Tao, Yuqi Zhu, and Zhi Jin. Longcodeu: Benchmarking long-context language models on long code understanding. *arXiv preprint arXiv:2503.04359*, 2025.
- 548 Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025.
- 549 Jia Li, Hao Zhu, Huanyu Liu, Xianjie Shi, He Zong, Yihong Dong, Kechi Zhang, Siyuan Jiang, Zhi Jin, and Ge Li. aixcoder-7b-v2: Training llms to fully utilize the long context in repository-level code completion. *arXiv preprint arXiv:2503.15301*, 2025.
- 550 Jia-ju Li, Hong-bo Wang, Jia-yi Guo, Hang Li, Bo-wen Wang, Jia-zheng Liu, X-Y Zhao, Y Jiang, Ke-fan Li, Y-W Zhang, et al. Agentsentinel: An end-to-end and real-time security defense framework for computer-use agents. *arXiv preprint arXiv:2405.11195*, 2024.
- 551 Jialin Li, Jinzhe Li, Gengxu Li, Yi Chang, and Yuan Wu. Refining critical thinking in llm code generation: A faulty premise-based evaluation framework, 2025. URL <https://arxiv.org/abs/2508.03622>.
- 552 Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators, 2025. URL <https://arxiv.org/abs/2502.14752>.
- 553 Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*, pages 13076–13084. AAAI Press, 2023.
- 554 Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhu Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

- 555 Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, Ziwei Tang, Yuanshuai Li, Florensia Widjaja, Xintong Zhu, Feige Zhou, Yongfeng Huang, Yannis Papakonstantinou, Fatma Ozcan, Chenhao Ma, and Reynold Cheng. Swe-sql: Illuminating llm pathways to solve user sql issues in real-world applications, 2025. URL <https://arxiv.org/abs/2506.18951>.
- 556 Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation, 2022. URL <https://arxiv.org/abs/2201.12086>.
- 557 Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models, 2023. URL <https://arxiv.org/abs/2301.12597>.
- 558 Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, Zhiyong Huang, and Jing Ma. Mmcode: Benchmarking multimodal large language models for code generation with visually rich programming problems. *arXiv preprint arXiv:2404.09486*, 2024.
- 559 Lei Li, Yekun Chai, Shuhuan Wang, Yu Sun, Hao Tian, Ningyu Zhang, and Hua Wu. Tool-augmented reward modeling, 2024. URL <https://arxiv.org/abs/2310.01045>.
- 560 Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. Auger: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1009–1021, 2022.
- 561 Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.
- 562 Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Cheng-hao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL <https://openreview.net/forum?id=KoFOg41haE>. Reproducibility Certification.
- 563 Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.
- 564 Ryan Li, Yanzhe Zhang, and Diyi Yang. Sketch2code: Evaluating vision-language models for interactive web design prototyping. *arXiv preprint arXiv:2410.16232*, 2024.
- 565 Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale

parallel training. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–10, 2021.

- 566 Teng Li, Jianfeng Ma, and Cong Sun. Dlog: diagnosing router events with syslogs for anomaly detection. *J. Supercomput.*, 74(2):845–867, 2018.
- 567 Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. *arXiv preprint arXiv:2503.06680*, 2025.
- 568 Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-lm improves controllable text generation, 2022. URL <https://arxiv.org/abs/2205.14217>.
- 569 Xiangyang Li, Xiaopeng Li, Kuicai Dong, Quanhu Zhang, Rongju Ruan, Xinyi Dai, Xiaoshuang Liu, Shengchun Xu, Yasheng Wang, and Ruiming Tang. Humanity’s last code exam: Can advanced llms conquer human’s hardest code competition?, 2025. URL <https://arxiv.org/abs/2506.12713>.
- 570 Yang Li, Youssef Emad, Karthik Padthe, Jack Lanchantin, Weizhe Yuan, Thao Nguyen, Jason Weston, Shang-Wen Li, Dong Wang, Ilia Kulikov, et al. Naturalthoughts: Selecting and distilling reasoning traces for general reasoning tasks. *arXiv preprint arXiv:2507.01921*, 2025.
- 571 Yi Li, Shaohua Wang, and Tien N Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*, pages 511–523, 2022.
- 572 Yizhi Li, Qingshui Gu, Zhoufutu Wen, Ziniu Li, Tianshun Xing, Shuyue Guo, Tianyu Zheng, Xin Zhou, Xingwei Qu, Wangchunshu Zhou, Zheng Zhang, Wei Shen, Qian Liu, Chenghua Lin, Jian Yang, Ge Zhang, and Wenhao Huang. Treepo: Bridging the gap of policy optimization and efficacy and inference efficiency with heuristic tree-based modeling, 2025. URL <https://arxiv.org/abs/2508.17445>.
- 573 Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*, 2023.
- 574 Yuhang Li, Chenchen Zhang, Ruilin Lv, Ao Liu, Ken Deng, Yuanxing Zhang, Jiaheng Liu, Wiggin Zhou, and Bo Zhou. Relook: Vision-grounded rl with a multimodal llm critic for agentic web coding. *arXiv preprint arXiv:2510.11498*, 2025.
- 575 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.
- 576 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

- 577 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL <https://www.science.org/doi/abs/10.1126/science.abq1158>.
- 578 Zhi Li, Weijie Liu, XiaoFeng Wang, Bin Yuan, Hongliang Tian, Hai Jin, and Shoumeng Yan. Lost along the way: Understanding and mitigating path-misresolution threats to container isolation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3063–3077, 2023.
- 579 Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Codereviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095*, 2022.
- 580 Zhongqiu Li, Zhenhe Wu, Mengxiang Li, Zhongjiang He, Ruiyu Fang, Jie Zhang, Yu Zhao, Yongxiang Li, Zhoujun Li, and Shuangyong Song. Scalable database-driven kgs can help text-to-sql. In *Proceedings of the ISWC 2024 Posters, Demos and Industry Tracks:*, volume 3828 of *CEUR Workshop Proceedings*, 2024.
- 581 Ziniu Li, Tian Xu, Yushun Zhang, Zihang Lin, Yang Yu, Ruoyu Sun, and Zhi-Quan Luo. Remax: A simple, effective, and efficient reinforcement learning method for aligning large language models, 2024. URL <https://arxiv.org/abs/2310.10505>.
- 582 Ziniu Li, Pengyuan Wang, Tian Xu, Tian Ding, Ruoyu Sun, and Yang Yu. Review of reinforcement learning for large language models: Formulations, algorithms, and opportunities. 2025.
- 583 Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.
- 584 Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecur.*, 4(1):5, 2021.
- 585 Shanchao Liang, Yiran Hu, Nan Jiang, and Lin Tan. Can language models replace programmers for coding? repocod says 'not yet', 2025. URL <https://arxiv.org/abs/2410.21647>.
- 586 Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, et al. Torch titan: One-stop pytorch native solution for production ready llm pretraining. *arXiv preprint arXiv:2410.06511*, 2025.
- 587 Xinbin Liang, Jinyu Xiang, Zhaoyang Yu, Jiayi Zhang, Sirui Hong, Sheng Fan, and Xiao Tang. Openmanus: An open-source framework for building general ai agents, 2025. URL <https://doi.org/10.5281/zenodo.15186407>.
- 588 Z. Liao and et al. Codeagent: Repository-level coding agents with tool-augmented llms. *arXiv preprint arXiv:2407.03178*, 2024. URL <https://arxiv.org/abs/2407.03178>.
- 589 Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

- 590 Jonathan Light, Min Cai, Sheng Shen, and Ziniu Hu. AvalonBench: Evaluating LLMs Playing the Game of Avalon. *arXiv e-prints*, art. arXiv:2310.05036, October 2023. doi: 10.48550/arXiv.2310.05036.
- 591 Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. One size does not fit all: Multi-granularity patch generation for better automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1554–1566, 2024.
- 592 Feng Lin, Dong Jae Kim, and Tse-Hsun Chen. When llm-based code generation meets the software development process. *CoRR abs/2403.15852*, 2024. URL <https://arxiv.org/abs/2403.15852>.
- 593 Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Daxin Jiang, Binxing Jiao, Chen Hu, et al. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. *arXiv preprint arXiv:2508.02085*, 2025.
- 594 Zhiyu Lin, Zhengda Zhou, Zhiyuan Zhao, Tianrui Wan, Yilun Ma, Junyu Gao, and Xuelong Li. WebUIBench: A comprehensive benchmark for evaluating multimodal large language models in WebUI-to-code. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 15780–15797, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.815. URL <https://aclanthology.org/2025.findings-acl.815/>.
- 595 Tobias Lindenbauer, Egor Bogomolov, and Yaroslav Zharov. Gitgoodbench: A novel benchmark for evaluating agentic performance on git, 2025. URL <https://arxiv.org/abs/2505.22583>.
- 596 Lin Ling. Evaluating social bias in code generation models. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 695–697, 2024.
- 597 Bang Liu, Xinfeng Li, Jiayi Zhang, Jinlin Wang, Tanjin He, Sirui Hong, Hongzhang Liu, Shaokun Zhang, Kaitao Song, Kunlun Zhu, et al. Advances and challenges in foundation agents: From brain-inspired intelligence to evolutionary, collaborative, and safe systems. *arXiv preprint arXiv:2504.01990*, 2025.
- 598 Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, Wei Jiang, Hang Yu, and Jianguo Li. Mftcoder: Boosting code llms with multitask fine-tuning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 5430–5441, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704901. doi: 10.1145/3637528.3671609. URL <https://doi.org/10.1145/3637528.3671609>.
- 599 Chenxiao Liu and Xiaojun Wan. Codeqa: A question answering dataset for source code comprehension. *arXiv preprint arXiv:2109.08365*, 2021.
- 600 Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. Code execution with pre-trained language models. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 4984–4999. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-ACL.308. URL <https://doi.org/10.18653/v1/2023.findings-acl.308>.

- 601 Chenyan Liu, Yufan Cai, Yun Lin, Yuhuan Huang, Yunrui Pei, Bo Jiang, Ping Yang, Jin Song Dong, and Hong Mei. Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 466–478, 2024.
- 602 Chunhua Liu, Hong Yi Lin, and Patanamon Thongtanunam. Too noisy to learn: Enhancing data quality for code review comment generation. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 236–248. IEEE, 2025.
- 603 Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. *arXiv preprint arXiv:1802.08802*, 2018.
- 604 Hao Liu, Ruitian Niu, Hongbo Zhang, Junchi Shang, Zhaoran Xiao, Xinbei Mao, Yichen Jiang, Hao Yuan, Tao Gui, Qi Sun, et al. Pku-saferlfhf: Towards multi-level safety alignment for llms with human preference. *arXiv preprint arXiv:2406.15513*, 2024.
- 605 Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning, 2023. URL <https://arxiv.org/abs/2304.08485>.
- 606 Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning, 2024. URL <https://arxiv.org/abs/2310.03744>.
- 607 Jiaheng Liu, Ken Deng, Congnan Liu, Jian Yang, Shukai Liu, He Zhu, Peng Zhao, Linzheng Chai, Yanan Wu, Ke Jin, Ge Zhang, Zekun Wang, Guoan Zhang, Bangyu Xiang, Wenbo Su, and Bo Zheng. M2rc-eval: Massively multilingual repository-level code completion evaluation, 2024. URL <https://arxiv.org/abs/2410.21157>.
- 608 Jiawei Liu and Lingming Zhang. Code-r1: Reproducing r1 for code with reliable rewards. 2025.
- 609 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL <https://openreview.net/forum?id=1qvx610Cu7>.
- 610 Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*, 2024.
- 611 Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025*, 2024.
- 612 Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation, 2024. URL <https://arxiv.org/abs/2408.06450>.
- 613 Kaiyuan Liu, Youcheng Pan, Yang Xiang, Daojing He, Jing Li, Yexing Du, and Tianrun Gao. Projecteval: A benchmark for programming agents automated evaluation on project-level code generation, 2025. URL <https://arxiv.org/abs/2503.07010>.

- 614** Mingjie Liu, Nathaniel Ross Pinckney, Brucek Khailany, and Haoxing Ren. Invited paper: Verilogeval: Evaluating large language models for verilog code generation. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*, pages 1–8. IEEE, 2023. doi: 10.1109/ICCAD57390.2023.10323812. URL <https://doi.org/10.1109/ICCAD57390.2023.10323812>.
- 615** Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL <https://aclanthology.org/2024.tacl-1.9/>.
- 616** Peipei Liu, Jian Sun, Li Chen, Zhaoteng Yan, Peizheng Zhang, Dapeng Sun, Dawei Wang, and Dan Li. Control flow-augmented decompiler based on large language model. *arXiv e-prints*, pages arXiv–2503, 2025.
- 617** Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 48(5):1800–1817, 2020.
- 618** Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, et al. Mdeval: Massively multilingual code debugging. *arXiv preprint arXiv:2411.02310*, 2024.
- 619** Shuyuan Liu, Kunsheng Han, Bowei Xu, Zinuo Wang, Yu Wang, Xuanzhe Chen, Chenglie Wu, and Yun Liu. Progent: Programmable privilege control for llm agents. *arXiv preprint arXiv:2403.04704*, 2024.
- 620** Siyao Liu, Ge Zhang, Boyuan Chen, Jialiang Xue, and Zhendong Su. FullStack Bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*, 2024. URL <https://arxiv.org/abs/2412.00535>.
- 621** Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, 2024. URL <https://arxiv.org/abs/2306.03091>.
- 622** Wei Liu, Xian Zhang, Hu Luo, et al. A survey of automatic source code summarization. *MDPI Electronics*, 11(22):3647, 2022. URL <https://www.mdpi.com/2079-9292/11/22/3647>.
- 623** Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. Codexgraph: Bridging large language models and code repositories via code graph databases. *arXiv preprint arXiv:2408.03910*, 2024.
- 624** Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as Agents. *arXiv e-prints*, art. arXiv:2308.03688, August 2023. doi: 10.48550/arXiv.2308.03688.
- 625** Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=zAdUB0actQ>.

- 626 Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. Nl2sql-bugs: A benchmark for detecting semantic errors in nl2sql translation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining* V.2, KDD '25, page 5662–5673, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714542. doi: 10.1145/3711896.3737427. URL <https://doi.org/10.1145/3711896.3737427>.
- 627 Yan Liu, Xiaokang Chen, Yan Gao, Zhe Su, Fengji Zhang, Daoguang Zan, Jian-Guang Lou, Pin-Yu Chen, and Tsung-Yi Ho. Uncovering and quantifying social biases in code generation. *Advances in Neural Information Processing Systems*, 36:2368–2380, 2023.
- 628 Yang Liu, Armstrong Foundjem, Foutse Khomh, and Heng Li. Adversarial attack classification and robustness testing for large language models for code. *Empirical Software Engineering*, 30(5):154, 2025.
- 629 Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multilingual and multi-task code retrieval, 2025. URL <https://arxiv.org/abs/2411.12644>.
- 630 Yifei Liu, Li Lyra Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*, 2025.
- 631 Yilun Liu, Shimin Tao, Weibin Meng, Feiyu Yao, Xiaofeng Zhao, and Hao Yang. Logprompt: Prompt engineering towards zero-shot and interpretable log analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*, pages 364–365. ACM, 2024.
- 632 Yilun Liu, Ziang Chen, Song Xu, Minggui He, Shimin Tao, Weibin Meng, Yuming Xie, Tao Han, Chenguang Zhao, Jingzhou Du, et al. R-log: Incentivizing log analysis capability in llms via reasoning-based reinforcement learning. *arXiv preprint arXiv:2509.25987*, 2025.
- 633 Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899*, 2024.
- 634 Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 373–384, 2018.
- 635 Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. Automatic generation of pull request descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 176–188. IEEE, 2019.
- 636 Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025. URL <https://arxiv.org/abs/2503.20783>.
- 637 LLM Code Reviewer Project. Llm code reviewer: Github action for automated reviews. <https://github.com/marketplace/actions/llm-code-reviewer>, 2024.
- 638 Yedidel Louck, Ariel Stulman, and Amit Dvir. Improving google a2a protocol: Protecting sensitive data and mitigating unintended harms in multi-agent systems. *arXiv preprint arXiv:2505.12490*, 2025.

- 639 Nikolaos Louloudakis, Perry Gibson, Jose Cano Reyes, and Ajitha Rajan. Fetafix: Automatic fault localization and repair of deep learning model conversions. 2025.
- 640 Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes, 2017. URL <https://arxiv.org/abs/1704.04856>.
- 641 Antonio Lozano, Arjun Guha, Avi Trost, Baptiste Rozière, Cédric A. F. T. M. de Souza, Chenghao Fang, Chenghao Mou, Christopher Akiki, Clémentine Fourrier, Danilo de Jesus da Silva, David Lansky, Denis Kocetkov, Dmytro Pykhtar, Evgenii Zheltonozhskii, Gagan S. Bhatia, Gabriel Villanova, Harm de Vries, Hady Elsahar, Igor Molybog, Jia Li, Jenny Chim, Joel Lamy-Poirier, João G. M. Araujo, Leandro von Werra, Lidiya Tsvetanova, Louis Martin, Loubna Ben Allal, Manuel Faysse, Marc Marone, Marco Zocca, Mishig Davaadorj, Niklas Muennighoff, Naman Goyal, Omar Sanseviero, Qian Liu, Remi Tachet, Raymond Li, Roberta Raileanu, Samuel Albani, Sky Shi, Thibaut Lavril, Thomas Wang, Yacine Jernite, Yekun Chai, Yangtian Zi, Carlos Muñoz Ferrandis, Laurent Sifre, Kunle Olukotun, Nima Tajbakhsh, Sergey Ermolin, Daniel Y. Fu, and Tri Dao. StarCoder2 and The Stack v2: The Next Generation, 2024. URL <https://arxiv.org/abs/2402.19173>.
- 642 Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173, 2024. doi: 10.48550/ARXIV.2402.19173. URL <https://doi.org/10.48550/arXiv.2402.19173>.
- 643 Hanzhen Lu and Zhongxin Liu. Improving Retrieval-Augmented Code Comment Generation by Retrieving for Generation . In 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 350–362, Los Alamitos, CA, USA, Oct 2024. IEEE Computer Society. doi: 10.1109/ICSME58944.2024.00040. URL <https://doi.ieeecomputersociety.org/10.1109/ICSME58944.2024.00040>.
- 644 Hanzhen Lu and Zhongxin Liu. Improving retrieval-augmented code comment generation by retrieving for generation. In 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 350–362. IEEE, 2024.
- 645 Haoyu Lu, Wen Liu, Bo Zhang, Bingxuan Wang, Kai Dong, Bo Liu, Jingxiang Sun, Tongzheng Ren, Zhuoshu Li, Hao Yang, Yaofeng Sun, Chengqi Deng, Hanwei Xu, Zhenda Xie, and Chong Ruan. Deepseek-vl: Towards real-world vision-language understanding, 2024. URL <https://arxiv.org/abs/2403.05525>.
- 646 Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), pages 647–658. IEEE, 2023.

- 647 Junyi Lu, Xiaojia Li, Zihan Hua, Lei Yu, Shiqi Cheng, Li Yang, Fengjun Zhang, and Chun Zuo. DeepCRCEval: Revisiting the Evaluation of Code Review Comment Generation. *arXiv e-prints*, art. arXiv:2412.18291, December 2024. doi: 10.48550/arXiv.2412.18291.
- 648 Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Dixin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021. URL <https://arxiv.org/abs/2102.04664>.
- 649 Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. Omniparser for pure vision based gui agent. *arXiv preprint arXiv:2408.00203*, 2024.
- 650 Yuwen Lu, Ziang Tong, Qinyi Zhao, Chengzhi Zhang, and Toby Jia-Jun Li. Ui layout generation with llms guided by ui grammar, 2023. URL <https://arxiv.org/abs/2310.15455>.
- 651 Yuxuan Lu, Bingsheng Yao, Hansu Gu, Jing Huang, Zheshen Jessie Wang, Yang Li, Jiri Gesi, Qi He, Toby Jia-Jun Li, and Dakuo Wang. Uxagent: An llm agent-based usability testing framework for web design. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '25, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400713958. doi: 10.1145/3706599.3719729. URL <https://doi.org/10.1145/3706599.3719729>.
- 652 Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch. *arXiv preprint arXiv:2505.03733*, 2025.
- 653 Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level. <https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51>, 2025. Notion Blog.
- 654 Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. *arXiv e-prints*, art. arXiv:2402.16667, February 2024. doi: 10.48550/arXiv.2402.16667.
- 655 Tianqi Luo, Chuhan Huang, Leixian Shen, Boyan Li, Shuyu Shen, Wei Zeng, Nan Tang, and Yuyu Luo. nvbench 2.0: Resolving ambiguity in text-to-visualization through stepwise reasoning, 2025. URL <https://arxiv.org/abs/2503.12880>.
- 656 Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Xu Wang, Qing Yang, Dongliang Xu, and Wanxiang Che. Semi-Instruct: Bridging Natural-Instruct and Self-Instruct for Code Large Language Models, 2024.
- 657 Xianzhen Luo, Wenzhen Zheng, Qingfu Zhu, Rongyi Zhang, Houyi Li, Siming Huang, YuanTao Fan, and Wanxiang Che. Scaling laws for code: A more data-hungry regime. *arXiv preprint arXiv:2510.08702*, 2025.

- 658 Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Mingzheng Xu, Tianhao Cheng, Yixuan Wang, Zheng Chu, Shijie Xuyang, Zhiyuan Ma, YuanTao Fan, and Wanxiang Che. Success is in the details: Evaluate and enhance details sensitivity of code llms through counterfactuals, 2025. URL <https://arxiv.org/abs/2505.14597>.
- 659 Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. Synthesizing natural language to visualization (nl2vis) benchmarks from nl2sql benchmarks. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1235–1247, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457261. URL <https://doi.org/10.1145/3448016.3457261>.
- 660 Zhifan Luo, Shuo Shao, Su Zhang, Lijing Zhou, Yuke Hu, Chenxu Zhao, Zhihao Liu, and Zhan Qin. Shadow in the cache: Unveiling and mitigating privacy risks of kv-cache in llm inference, 2025.
- 661 Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.
- 662 Phong Luu, Adityasrinivas Iyengar, Chacha Li, Hung Le, Tong Zhao, and other. Purpcode: Reasoning for safer code generation. *Proceedings of the Amazon Nova AI Challenge*, 2024.
- 663 Lvwerra. Stack exchange paired dataset, 2023. URL <https://huggingface.co/datasets/lvwerra/stack-exchange-paired>. Accessed: 2024.
- 664 Ce Lyu, Minghao Zhao, Yanhao Wang, and Liang Jie. Llm-based dynamic differential testing for database connectors with reinforcement learning-guided prompt selection. *arXiv preprint arXiv:2506.11870*, 2025.
- 665 Zuxin Lyu, Zhaoran Li, Jia Liu, and Tuo Zhao. Constrained variational policy optimization for safe reinforcement learning. pages 14660–14674, 2022.
- 666 M-A-P. Code feedback dataset, 2024. URL <https://huggingface.co/datasets/m-a-p/Cod-e-Feedback>. Accessed: 2024.
- 667 Haozhe Ma, Zhengding Luo, Thanh Vinh Vo, Kuankuan Sima, and Tze-Yun Leong. Highly efficient self-adaptive reward shaping for reinforcement learning. *arXiv preprint arXiv:2408.03029*, 2024.
- 668 Kaijing Ma, Xinrun Du, Yunran Wang, Haoran Zhang, Zhoufutu Wen, Xingwei Qu, Jian Yang, Jiaheng Liu, Minghao Liu, Xiang Yue, Wenhao Huang, and Ge Zhang. KOR-Bench: Benchmarking Language Models on Knowledge-Orthogonal Reasoning Tasks. *arXiv e-prints*, art. arXiv:2410.06526, October 2024. doi: 10.48550/arXiv.2410.06526.
- 669 Lezhi Ma, Shangqing Liu, Lei Bu, Shangru Li, Yida Wang, and Yang Liu. Speceval: Evaluating code comprehension in large language models via program specifications, 2025. URL <https://arxiv.org/abs/2409.12866>.
- 670 Lipeng Ma, Weidong Yang, Yixuan Li, Ben Fei, Mingjie Zhou, Shuhao Li, Sihang Jiang, Bo Xu, and Yanghua Xiao. Adaptivelog: An adaptive log analysis framework with the collaboration of large and small language model. *ACM Transactions on Software Engineering and Methodology*, 2025.

- 671 Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.
- 672 Yihong Ma, Yifei Fu, Hongyi Zhang, and Ganqu Xu. Ctrl-z: Controlling ai agents via resampling. *arXiv preprint arXiv:2405.18244*, 2024.
- 673 Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 238–249, 2025.
- 674 Zihan Ma, Taolin Zhang, Maosong Cao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. Rethinking verification for llm code generation: From generation to testing, 2025. URL <https://arxiv.org/abs/2507.06920>.
- 675 Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R Cogo, and Bram Adams. Intertrans: Leveraging transitive intermediate translations to enhance llm-based code translation. *arXiv preprint arXiv:2411.01063*, 2024.
- 676 Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegraeffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL <https://openreview.net/forum?id=S37h0erQLB>.
- 677 Magpie-Align. Magpie-qwen2.5-coder-pro-300k-v0.1, 2024. URL <https://huggingface.co/datasets/Magpie-Align/Magpie-Qwen2.5-Coder-Pro-300K-v0.1>. Accessed: 2024.
- 678 Pratyush Maini, Sachin Goyal, Dylan Sam, Alex Robey, Yash Savani, Yiding Jiang, Andy Zou, Zachary C Lipton, and J Zico Kolter. Safety pretraining: Toward the next generation of safe ai. *arXiv preprint arXiv:2504.16980*, 2025.
- 679 Dung Nguyen Manh, Thang Phan Chau, Nam Le Hai, Thong T Doan, Nam V Nguyen, Quang Pham, and Nghi DQ Bui. Codemmlu: A multi-task benchmark for assessing code understanding capabilities of codellms. *CoRR*, 2024.
- 680 Petros Maniatis and Daniel Tarlow. Large sequence models for software development activities. <https://blog.research.google/2023/05/large-sequence-models-for-software.html>.
- 681 Shaoguang Mao, Yuzhe Cai, Yan Xia, Wenshan Wu, Xun Wang, Fengyi Wang, Tao Ge, and Furu Wei. ALYMPICS: LLM Agents Meet Game Theory – Exploring Strategic Decision-Making with AI Agents. *arXiv e-prints*, art. arXiv:2311.03220, November 2023. doi: 10.48550/arXiv.2311.03220.
- 682 Jorge Martinez-Gil. Evaluating small-scale code models for code clone detection. *arXiv preprint arXiv:2506.10995*, 2025.
- 683 MatrixStudio. Codeforces python submissions, 2023. URL <https://huggingface.co/datasets/MatrixStudio/Codeforces-Python-Submissions>. Accessed: 2024.

- 684 Justus Mattern, Sami Jaghouar, Manveer Basra, Jannik Straube, Matthew Di Ferrante, Felix Gabriel, Jack Min Ong, Vincent Weisser, and Johannes Hagemann. Synthetic-1: Two million collaboratively generated reasoning traces from deepseek-r1, 2025. URL <https://www.primeintellect.ai/blog/synthetic-1-release>.
- 685 Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. On leakage of code generation evaluation datasets, 2024. URL <https://arxiv.org/abs/2407.07565>.
- 686 Ian R McKenzie, Oskar J Hollinsworth, Tom Tseng, Xander Davies, Stephen Casper, Aaron D Tucker, Robert Kirk, and Adam Gleave. Stack: Adversarial attacks on llm safeguard pipelines. *arXiv preprint arXiv:2506.24068*, 2025.
- 687 Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 53–65. IEEE, 2023.
- 688 Yifei Meng, Yuxuan Zhang, Hao Wang, Zhen Liu, and Xing He. Mitigating gender bias in code large language models via model editing. In *2024 IEEE 31st International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024.
- 689 Meta AI. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models, 2024. URL <https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/>.
- 690 Meta AI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation, 2025. URL <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>.
- 691 Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.
- 692 Yibo Miao, Bofei Gao, Shanghaoran Quan, Junyang Lin, Daoguang Zan, Jiaheng Liu, Jian Yang, Tianyu Liu, and Zhijie Deng. Aligning codellms with direct preference optimization. *arXiv preprint arXiv:2410.18585*, 2024.
- 693 Microsoft .NET Blog. How we build github copilot into visual studio. <https://devblogs.microsoft.com/dotnet/building-github-copilot-into-visual-studio/>, 2024.
- 694 Ivan Milev, Mislav Balunović, Maximilian Baader, and Martin Vechev. Toolfuzz – automated agent tool testing, 2025. URL <https://arxiv.org/abs/2503.04479>.
- 695 Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn 1 million from real-world freelance software engineering?, 2025. URL <https://arxiv.org/abs/2502.12115>.
- 696 Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koifman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, S. Yousaf Shah, Petros Zerfos, Heiko Ludwig,

- Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos A. Fonseca, Amith Singhee, Nirmit Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. Granite code models: A family of open foundation models for code intelligence. *CoRR*, abs/2405.04324, 2024. doi: 10.48550/ARXIV.2405.04324. URL <https://doi.org/10.48550/arXiv.2405.04324>.
- 697 Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koifman, Boris Lublin-sky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos Fonseca, Amith Singhee, Nirmit Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. Granite code models: A family of open foundation models for code intelligence, 2024. URL <https://arxiv.org/abs/2405.04324>.
- 698 Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-Task Generalization via Natural Language Crowdsourcing Instructions. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, Dublin, Ireland, 2022.
- 699 Atsuyuki Miyai, Zaiying Zhao, Kazuki Egashira, Atsuki Sato, Tatsumi Sunada, Shota Onohara, Hiromasa Yamanishi, Mashiro Toyooka, Kunato Nishina, Ryoma Maeda, Kiyoharu Aizawa, and Toshihiko Yamasaki. Webchorearena: Evaluating web browsing agents on realistic tedious web tasks, 2025. URL <https://arxiv.org/abs/2506.01952>.
- 700 MLFoundations. Stack exchange code golf, 2023. URL https://huggingface.co/datasets/mlfoundations-dev/stackexchange_codegolf. Accessed: 2024.
- 701 MLFoundations. Stack exchange code review dataset, 2023. URL https://huggingface.co/datasets/mlfoundations-dev/stackexchange_codereview. Accessed: 2024.
- 702 Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H Sarker, Leandros Maglaras, and Naeem Janjua. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. *arXiv preprint arXiv:2406.12513*, 2024.
- 703 Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. Megadiff: A dataset of 600k java source code changes categorized by diff size, 2021. URL <https://arxiv.org/abs/2108.04631>.
- 704 Jiwon Moon, Yerin Hwang, Dongryeol Lee, Taegwan Kang, Yongil Kim, and Kyomin Jung. Don't judge code by its cover: Exploring biases in llm judges for code evaluation, 2025. URL <https://arxiv.org/abs/2505.16222>.
- 705 Mohammad Mehdi Morovati, Amin Nikanjam, and Foutse Khomh. Fault localization in deep learning-based software: A system-level approach. *arXiv preprint arXiv:2411.08172*, 2024.

- 706 Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset. *arXiv preprint arXiv:2504.16891*, 2025.
- 707 Yutao Mou, Xiao Deng, Yuxiao Luo, Shikun Zhang, and Wei Ye. Can you really trust code copilots? evaluating large language models from a code security perspective. *arXiv preprint arXiv:2505.10494*, 2025.
- 708 Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. Developer-intent driven code comment generation. in 2023 ieee/acm 45th international conference on software engineering (icse), 2023.
- 709 Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. Expere-pair: Dual-memory enhanced lilm-based repository-level program repair. *arXiv preprint arXiv:2506.10484*, 2025.
- 710 Yao Mu, Junting Chen, Qinglong Zhang, Shoufa Chen, Qiaojun Yu, Chongjian Ge, Runjian Chen, Zhixuan Liang, Mengkang Hu, Chaofan Tao, et al. Robocodex: Multimodal code generation for robotic behavior synthesis. *arXiv preprint arXiv:2402.16117*, 2024.
- 711 Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models, 2024. URL <https://arxiv.org/abs/2308.07124>.
- 712 Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. OctoPack: Instruction Tuning Code Large Language Models, 2024.
- 713 Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.
- 714 Niklas Muennighoff et al. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.
- 715 Multilingual-Multimodal-NLP. McEval-instruct, 2024. URL <https://huggingface.co/datasets/Multilingual-Multimodal-NLP/McEval-Instruct>. Accessed: 2024.
- 716 Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents, February 2025. URL <http://arxiv.org/abs/2406.12952>. arXiv:2406.12952 [cs].
- 717 Varun Nair, Elliot Schumacher, Geoffrey Tso, and Anitha Kannan. Dera: enhancing large language model completions with dialog-enabled resolving agents. *arXiv preprint arXiv:2303.17071*, 2023.
- 718 Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

- 719 Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022. URL <https://arxiv.org/abs/2112.09332>.
- 720 Vineeth Sai Narajala and Om Narayan. Securing agentic ai: A comprehensive threat model and mitigation framework for generative ai agents. *arXiv preprint arXiv:2504.19956*, 2025.
- 721 Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.
- 722 Mona Nashaat and James Miller. Towards efficient fine-tuning of language models with organizational data for automated software review. *IEEE Transactions on Software Engineering*, 2024.
- 723 Ching Yuhui Natalie, Sophie Tung Xuan Ying, and Siow Jing Kai. Alfredo: Agentic llm-based framework for code deobfuscation.
- 724 Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. *arXiv preprint arXiv:2406.11912*, 2024. URL <https://arxiv.org/abs/2406.11912>.
- 725 Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*, 2024.
- 726 Yuansheng Ni, Ping Nie, Kai Zou, Xiang Yue, and Wenhua Chen. Viscoder: Fine-tuning llms for executable python visualization code generation. *arXiv preprint arXiv:2506.03930*, 2025.
- 727 Ziyi Ni, Yifan Li, Ning Yang, Dou Shen, Pin Lyu, and Daxiang Dong. Tree-of-code: A self-growing tree framework for end-to-end code generation and execution in complex tasks. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 9804–9819, 2025.
- 728 Jack Nichols. How we scored #1 on terminal-bench (52 URL <https://www.warp.dev/blog/terminal-bench>.
- 729 Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models, 2025. URL <https://arxiv.org/abs/2502.09992>.
- 730 Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *CoRR*, abs/2305.02309, 2023. doi: 10.48550/ARXIV.2305.02309. URL <https://doi.org/10.48550/arXiv.2305.02309>.
- 731 Erik Nijkamp, Bo Pang, Hiroaki Hayashi, et al. CodeGen: An open large language model for code with multi-turn program synthesis. *ICLR*, 2023.

- 732 Zepeng Ning and Lihua Xie. A survey on multi-agent reinforcement learning and its application. *Journal of Automation and Intelligence*, 3(2):73–91, 2024.
- 733 Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. DIRECT : A transformer-based model for decompiled identifier renaming. In Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, and Reut Tsarfaty, editors, *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, August 2021.
- 734 Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques, 2025. URL <https://arxiv.org/abs/2501.14257>.
- 735 Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1–13. ACM, 2022. doi: 10.1145/3510003.3510096. URL <https://doi.org/10.1145/3510003.3510096>.
- 736 Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL <https://arxiv.org/abs/2506.13131>.
- 737 Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSOC 2015, Turin, Italy, September 23-25, 2015*, pages 195–202. IEEE Computer Society, 2015.
- 738 Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing, 2024. URL <https://arxiv.org/abs/2409.10737>.
- 739 Andres Nunez, Siyuan Hou, Mathias Payer, and Yuhui Zhou. Autosafecoder: A language model-guided approach for automating secure code generation. *arXiv preprint arXiv:2402.04659*, 2024.
- 740 Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. Nomic embed: Training a reproducible long context text embedder, 2025. URL <https://arxiv.org/abs/2402.01613>.
- 741 NVIDIA. Opencodereasoning dataset, 2024. URL <https://huggingface.co/datasets/nvidia/OpenCodeReasoning>. Accessed: 2024.
- 742 o3-o4. Introducing-o3-and-o4-mini, 2025. URL <https://openai.com/zh-Hans-CN/index/introducing-o3-and-o4-mini/>.
- 743 Ike Obi, Vishnunandan LN Venkatesh, Weizheng Wang, Ruiqi Wang, Dayoon Suh, Temi-tope I Amosa, Wonse Jo, and Byung-Cheol Min. Safeplan: Leveraging formal logic and chain-of-thought reasoning for enhanced safety in llm-based robotic task planning. *arXiv preprint arXiv:2503.06892*, 2025.

- 744 Wonseok Oh and Hakjoo Oh. Pyter: effective program repair for python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 922–934, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549130. URL <https://doi.org/10.1145/3540250.3549130>.
- 745 OpenAI. Introducing chatgpt. <https://openai.com/blog/chatgpt>, 2022.
- 746 OpenAI. GPT4 Code. <https://chat.openai.com/?model=GPT4-Code-interpreter>, 2023.
- 747 OpenAI. ChatGPT plugins. <https://openai.com/index/chatgpt-plugins/#code-interpreter>, 2023.
- 748 OpenAI. Gpt-4v(ision) system card. <https://openai.com/index/gpt-4v-system-card/>, 2023.
- 749 OpenAI. Hello gpt-4o. <https://openai.com/index/hello-gpt-4o/>, 2024.
- 750 OpenAI. Gpt-4o mini: advancing cost-efficient intelligence. <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/>, 2024.
- 751 OpenAI. Introducing openai o1-preview. <https://openai.com/index/introducing-openai-o1-preview/>, 2024.
- 752 OpenAI. Introducing gpt-5, 2025. URL <https://openai.com/index/introducing-gpt-5/>.
- 753 OpenAI. Introducing gpt-4.1 in the api. <https://openai.com/index/gpt-4-1/>, 2025. Accessed: 2025-09-30.
- 754 OpenAI. Gpt-5 system card. <https://openai.com/index/gpt-5-system-card/>, August 2025.
- 755 OpenAI. Introducing upgrades to codex: Gpt-5-codex. <https://openai.com/index/introducing-upgrades-to-codex/>, 2025.
- 756 OpenAI. Introducing gpt-5 for developers. <https://openai.com/index/introducing-gpt-5-for-developers/>, 2025. Accessed: 2025-09-30.
- 757 OpenAI. Introducing openai o3 and o4-mini. <https://openai.com/index/introducing-o3-and-o4-mini/>, 2025.
- 758 OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun

Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiro, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL <https://arxiv.org/abs/2303.08774>.

- 759 OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney,

Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b model card, 2025. URL <https://arxiv.org/abs/2508.10925>.

- 760 OpenCode Contributors. OpenCode: Open source terminal code assistant. Technical report, 2024.
- 761 OpenCoder-LLM. Opc-sft-stage2, 2024. URL <https://huggingface.co/datasets/OpenCoder-LLM/opc-sft-stage2>. Accessed: 2024.
- 762 Openhands. Openhands. *arXiv preprint arXiv:2407.16741*, 2024. URL <https://arxiv.org/abs/2407.16741>.
- 763 OpenInterpreter. Open Interpreter. <https://github.com/openinterpreter/open-interpreter>, 2023.
- 764 Marc Oriol, Quim Motger, Jordi Marco, and Xavier Franch. Multi-agent debate strategies to enhance requirements engineering with large language models. *arXiv preprint arXiv:2507.05981*, 2025.
- 765 Anton Osika. Gpt engineer. <https://github.com/AntonOsika/gpt-engineer>, 2023.
- 766 Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025. URL <https://arxiv.org/abs/2502.10517>.
- 767 Geliang Ouyang, Jingyao Chen, Zhihe Nie, Yi Gui, Yao Wan, Hongyu Zhang, and Dongping Chen. nvagent: Automated data visualization from natural language via collaborative agent workflow. *arXiv preprint arXiv:2502.05036*, 2025.
- 768 Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- 769 Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL <https://arxiv.org/abs/2203.02155>.
- 770 Anvith Pabba, Alex Mathai, Anindya Chakraborty, and Baishakhi Ray. Semagent: A semantics aware program repair agent. *arXiv preprint arXiv:2506.16650*, 2025.

- 771 Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: Towards llms as operating systems. 2023.
- 772 Indranil Palit and Tushar Sharma. Generating refactored code accurately using reinforcement learning. *arXiv preprint arXiv:2412.18035*, 2024.
- 773 Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024.
- 774 Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*, page 1–13. ACM, April 2024. doi: 10.1145/3597503.3639226. URL <http://dx.doi.org/10.1145/3597503.3639226>.
- 775 X. Pan and et al. Agentcoder: Multi-agent code generation with unit test feedback. *arXiv preprint arXiv:2402.12345*, 2024. URL <https://arxiv.org/abs/2402.12345>.
- 776 Xinglu Pan, Chenxiao Liu, Yanzhen Zou, Tao Xie, and Bing Xie. MESIA: Understanding and Leveraging Supplementary Nature of Method-level Comments for Automatic Comment Generation. *arXiv e-prints*, art. arXiv:2403.17357, March 2024. doi: 10.48550/arXiv.2403.17357.
- 777 Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26–29 March 2006, New York, New York, USA*, pages 319–332. IEEE Computer Society, 2006.
- 778 Zhenyu Pan, Rongyu Cao, Yongchang Cao, Yingwei Ma, Binhu Li, Fei Huang, Han Liu, and Yongbin Li. Codev-bench: How do llms understand developer-centric code completion?, 2024. URL <https://arxiv.org/abs/2410.01353>.
- 779 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- 780 Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.
- 781 Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Advances in Neural Information Processing Systems*, 2024.
- 782 Constantinos Patsakis, Fran Casino, and Nikolaos Lykousas. Assessing llms in malicious code deobfuscation of real-world malware campaigns. *Expert Systems with Applications*, 256:124912, 2024.
- 783 Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions, 2021. URL <https://arxiv.org/abs/2108.09293>.

- 784 Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, et al. Flow-of-action: Sop enhanced llm-based multi-agent system for root cause analysis. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 422–431, 2025.
- 785 Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanisław Woźniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwkv: Reinventing rnns for the transformer era, 2023. URL <https://arxiv.org/abs/2305.13048>.
- 786 Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Xingjian Du, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, Kranthi Kiran GV, Jan Kocoń, Bartłomiej Koptyra, Satyapriya Krishna, Ronald McClelland Jr., Jiaju Lin, Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Cahya Wirawan, Stanisław Woźniak, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui-Jie Zhu. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence, 2024. URL <https://arxiv.org/abs/2404.05892>.
- 787 Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 33–40, 2025. doi: 10.1109/LLM4Code66737.2025.00009.
- 788 Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization, 2024. URL <https://arxiv.org/abs/2402.16694>.
- 789 Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *arXiv preprint arXiv:2402.16694*, 2024.
- 790 Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*, 2023.
- 791 Yun Peng, Kisub Kim, Linghan Meng, and Kui Liu. icodereviewer: Improving secure code review with mixture of prompts. *arXiv preprint arXiv:2510.12186*, 2025.
- 792 Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. Coffe: A code efficiency benchmark for code generation, 2025. URL <https://arxiv.org/abs/2502.02827>.
- 793 Minh VT Pham, Nghi DQ Bui, et al. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs, 2025.
- 794 Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Effective high coverage test generation for python, 2025. URL <https://arxiv.org/abs/2403.16218>.
- 795 Plandex Team. Plandex: Open source AI coding agent. Technical report, 2024. URL <https://github.com/plandex-ai/plandex>.

- 796 Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y. Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models, 2023. URL <https://arxiv.org/abs/2302.10866>.
- 797 Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 107–126, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336895. doi: 10.1145/2814270.2814310. URL <https://doi.org/10.1145/2814270.2814310>.
- 798 polyglot-benchmark. polyglot-benchmark, 2025. URL <https://aider.chat/2024/12/21/polyglot.html#the-polyglot-benchmark>.
- 799 Ahilan Ayyachamy Nadar Ponnusamy. Bridging llm-generated code and requirements: Reverse generation technique and sbc metric for developer insights, 2025. URL <https://arxiv.org/abs/2502.07835>.
- 800 Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.
- 801 Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. CHASE-SQL: multi-path reasoning and preference optimized candidate selection in text-to-sql. *CoRR*, abs/2410.01943, 2024.
- 802 Julian Aron Prenner and Romain Robbes. Runbugrun – an executable dataset for automated program repair, 2023. URL <https://arxiv.org/abs/2304.01102>.
- 803 PrimeIntellect. Stackexchange question answering, 2024. URL <https://huggingface.co/datasets/PrimeIntellect/stackexchange-question-answering>. Accessed: 2024.
- 804 PrimeIntellect. Real-world swe problems, 2024. URL <https://huggingface.co/datasets/PrimeIntellect/real-world-swe-problems>. Accessed: 2024.
- 805 PrimeIntellect. Synthetic-2-sft-verified, 2024. URL <https://huggingface.co/datasets/PrimeIntellect/SYNTHETIC-2-SFT-verified>. Accessed: 2024.
- 806 PrithivMLmods. Coder-stat dataset, 2024. URL <https://huggingface.co/datasets/prithivMLmods/Coder-Stat>. Accessed: 2024.
- 807 Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021. URL <https://arxiv.org/abs/2105.12655>.
- 808 Penghui Qi, Zichen Liu, Xiangxin Zhou, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Defeating the training-inference mismatch via fp16. *arXiv preprint arXiv:2510.26788*, 2025.
- 809 Xiangyu Qi, David Mespasson, Antoine Lam, David Chan, Avisha Riapolov, Amelia Glaese, Sebastian Borgeaud, Geoffrey Irving, and Pamela Mishkin. Fine-tuning aligned

language models compromises safety, even when users do not intend to. *arXiv preprint arXiv:2310.03693*, 2023.

- 810 Zhuang Qi, Nelson Jean, J. Zico Kolter, and Aditi Raghunathan. Emergent misalignment: Narrow finetuning can produce broadly misaligned llms, 2024.
- 811 Chen Qian, Xin Cong, Cheng Yang, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2024.
- 812 Chen Qian, Jiahao Li, Yufan Dang, Wei Liu, Yifei Wang, Zihao Xie, Weize Chen, Cheng Yang, Yingli Zhang, Zhiyuan Liu, and Maosong Sun. Iterative experience refinement of software-developing agents. *CoRR abs/2405.04219*, 2024. URL <https://arxiv.org/abs/2405.04219>.
- 813 Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.
- 814 Haiyan Qin, Zhiwei Xie, Jingjing Li, Liangchen Li, Xiaotong Feng, Junzhan Liu, and Wang Kang. Reasoningv: Efficient verilog code generation with adaptive hybrid reasoning model. *arXiv preprint arXiv:2504.14560*, 2025.
- 815 Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*, 2024.
- 816 Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. S oap fl: A standard operating procedure for llm-based method-level fault localization. *IEEE Transactions on Software Engineering*, 2025.
- 817 Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.
- 818 Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu-Ming Yiu, Dong Huang, See-Kiong Ng, and Luu Anh Tuan. Effibench-x: A multi-language benchmark for measuring efficiency of llm-generated code, 2025. URL <https://arxiv.org/abs/2505.13004>.
- 819 Qodo-AI. Pr-agent: Ai-powered pull request agent. <https://github.com/Qodo-AI/pr-agent>, 2024.
- 820 Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. Before generation, align it! A novel and effective strategy for mitigating hallucinations in text-to-sql generation. In *Findings of the Association for Computational Linguistics, ACL 2024*, pages 5456–5471. Association for Computational Linguistics, 2024.
- 821 Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, Zekun Wang, Jian Yang, Zeyu Cui, Yang Fan, Yichang Zhang, Binyuan Hui, and Junyang Lin. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings, 2025. URL <https://arxiv.org/abs/2501.01257>.

- 822 Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, et al. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*, 2025.
- 823 QuixiAI. Dolphin coder dataset, 2024. URL <https://huggingface.co/datasets/QuixiAI/dolphin-coder>. Accessed: 2024.
- 824 Qwen. Qwen3-coder: Agentic coding in the world, 2025. URL <https://qwenlm.github.io/blog/qwen3-coder>.
- 825 Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL <https://arxiv.org/abs/2412.15115>.
- 826 Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021. URL <https://arxiv.org/abs/2103.00020>.
- 827 Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2023.
- 828 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL <http://jmlr.org/papers/v21/20-074.html>.
- 829 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL <http://jmlr.org/papers/v21/20-074.html>.
- 830 Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. A multi-agent approach to fault localization via graph-based retrieval and reflexion. *arXiv preprint arXiv:2409.13642*, 2024.
- 831 Alfin Wijaya Rahardja, Junwei Liu, Weitong Chen, Zhenpeng Chen, and Yiling Lou. Can agents fix agent issues?, 2025. URL <https://arxiv.org/abs/2505.20749>.
- 832 Md Tajmilur Rahman, Rahul Singh, and Mir Yousuf Sultan. Automating patch set generation from code reviews using large language models. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pages 273–274, 2024.
- 833 Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

- 834 Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020. URL <https://arxiv.org/abs/1910.02054>.
- 835 Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations. *arXiv e-prints*, art. arXiv:1709.10087, September 2017. doi: 10.48550/arXiv.1709.10087.
- 836 Dezhi Ran, Mengzhou Wu, Hao Yu, Yuetong Li, Jun Ren, Yuan Cao, Xia Zeng, Haochuan Lu, Zexin Xu, Mengqian Xu, Ting Su, Liangchao Yao, Ting Xiong, Wei Yang, Yuetang Deng, Assaf Marron, David Harel, and Tao Xie. Beyond pass or fail: Multi-dimensional benchmarking of foundation models for goal-based mobile ui navigation, 2025. URL <https://arxiv.org/abs/2501.02863>.
- 837 Zeeshan Rasheed, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. Codepori: Large scale model for autonomous software development by using multi-agents. *CoRR abs/2402.01411*, 2024. URL <https://arxiv.org/abs/2402.01411>.
- 838 Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL <https://arxiv.org/abs/2504.08703>.
- 839 Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.
- 840 Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William E Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Kenji Toyama, Robert James Berry, Divya Tyamagundlu, Timothy P Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=il5yUQsrjC>.
- 841 Arijit Ray, Jiafei Duan, Ellis Brown, Reuben Tan, Dina Bashkirova, Rose Hendrix, Kiana Ehsani, Aniruddha Kembhavi, Bryan A. Plummer, Ranjay Krishna, Kuo-Hao Zeng, and Kate Saenko. SAT: Dynamic Spatial Aptitude Training for Multimodal Language Models. *arXiv e-prints*, art. arXiv:2412.07755, December 2024. doi: 10.48550/arXiv.2412.07755.
- 842 readme-eval. readme-eval, 2025. URL <https://huggingface.co/datasets/patched-codes/generate-readme-eval>.
- 843 Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. ReflectionCoder: Learning from Reflection Sequence for Enhanced One-off Code Generation, 2024.
- 844 Qibing Ren, Chang Gao, Jing Shao, Junchi Yan, Xin Tan, Wai Lam, and Lizhuang Ma. Codeattack: Revealing safety generalization challenges of large language models via code completion. *arXiv preprint arXiv:2403.07865*, 2024.

- 845 Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL <https://arxiv.org/abs/2009.10297>.
- 846 Xiaoxue Ren, Chaoqun Dai, Qiao Huang, Ye Wang, Chao Liu, and Bo Jiang. Hydrareviewer: A holistic multi-agent system for automatic code review comment generation. *IEEE Transactions on Software Engineering*, 2025.
- 847 Replit. Replit Code V1.5 3B: Technical report. *Hugging Face Model Repository*, 2023. URL https://huggingface.co/replit/replit-code-v1_5-3b.
- 848 RepoCoder. Repocoder. *arXiv preprint arXiv:2303.12570*, 2023. URL <https://arxiv.org/abs/2303.12570>.
- 849 Cedric Richter and Heike Wehrheim. Tssb-3m: Mining single statement bugs at massive scale, 2022. URL <https://arxiv.org/abs/2201.12046>.
- 850 Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14116–14137, Bangkok, Thailand, August 2024. Association for Computational Linguistics. URL <https://aclanthology.org/2024.acl-long.761>.
- 851 Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024. URL <https://arxiv.org/abs/2401.08500>.
- 852 rouge_l. rouge_l, 2025. URL https://en.wikipedia.org/wiki/ROUGE_metric.
- 853 Monoshi Kumar Roy, Simin Chen, Benjamin Steenhoek, Jinjun Peng, Gail Kaiser, Baishakhi Ray, and Wei Le. Codesense: a real-world benchmark and dataset for code semantic reasoning. *arXiv preprint arXiv:2506.00750*, 2025.
- 854 Winston W Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338, 1987.
- 855 Abhik Roychoudhury. Agentic ai for software: thoughts from software engineering community. *arXiv preprint arXiv:2508.17343*, 2025.
- 856 Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation, 2022. URL <https://arxiv.org/abs/2110.06773>.
- 857 Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- 858 Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2024.
- 859 Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232*, 2024.

- 860 Fernando Vallecillos Ruiz, Max Hort, and Leon Moonen. The art of repair: Optimizing iterative program repair with instruction-tuned models, 2025. URL <https://arxiv.org/abs/2505.02931>.
- 861 Nuno Saavedra, João Gonçalves, Miguel Henriques, João F. Ferreira, and Alexandra Mendes. Polyglot code smell detection for infrastructure as code with GLITCH. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 2042–2045. IEEE, 2023. doi: 10.1109/ASE56229.2023.00162. URL <https://doi.org/10.1109/ASE56229.2023.00162>.
- 862 Malik Abdul Sami, Muhammad Waseem, Zheying Zhang, Zeeshan Rasheed, Kari Systä, and Pekka Abrahamsson. Ai based multiagent approach for requirements elicitation and analysis. *arXiv preprint arXiv:2409.00038*, 2024.
- 863 Ranjan Sapkota, Konstantinos I Roumeliotis, and Manoj Karkee. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443*, 2025.
- 864 Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. Are emergent abilities of large language models a mirage? In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 55565–55581. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/adc98a266f45005c403b8311ca7e8bd7-Paper-Conference.pdf.
- 865 Maximilian Schall, Tamara Czinczoll, and Gerard De Melo. Commitbench: A benchmark for commit message generation. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 728–739. IEEE, 2024.
- 866 Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.
- 867 Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, M. Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Neural Information Processing Systems*, 2023. doi: 10.48550/arXiv.2302.04761.
- 868 John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.
- 869 John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.
- 870 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- 871 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023. URL <https://arxiv.org/abs/2302.06527>.

- 872 ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.
- 873 Ryo Sekizawa, Nan Duan, Shuai Lu, and Hitomi Yanaka. Constructing multilingual code search dataset using neural machine translation. In Vishakh Padmakumar, Gisela Vallejo, and Yao Fu, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics: Student Research Workshop, ACL 2023, Toronto, Canada, July 9–14, 2023*, pages 69–75. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.ACL-SRW.10. URL <https://doi.org/10.18653/v1/2023.acl-srw.10>.
- 874 SenseLLM. Reflectionseq-gpt dataset, 2024. URL <https://huggingface.co/datasets/SenseLLM/ReflectionSeq-GPT>. Accessed: 2024.
- 875 Minju Seo, Jinheon Baek, and Sung Ju Hwang. Rethinking code refinement: Learning to judge code efficiency, 2024. URL <https://arxiv.org/abs/2410.22375>.
- 876 TOCOL SERVERS. Mcp-universe: Benchmarking large language models with real-world model context pro-tocol servers. *origins*, 25:55–128.
- 877 Oussama Ben Sghaier, Rosalia Tufano, Gabriele Bavota, and Houari Sahraoui. Leveraging reward models for guiding code review comment generation. *arXiv preprint arXiv:2506.04464*, 2025.
- 878 Ramin Shahbazi and Fatemeh Fard. Apicontext2com: Code comment generation by incorporating pre-defined api documentation. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 13–24, 2023. doi: 10.1109/ICPC58990.2023.00012.
- 879 Xiao Shao, Weifu Jiang, Fei Zuo, and Mengqing Liu. SwarmBrain: Embodied agent for real-time strategy game StarCraft II via large language models. *arXiv e-prints*, art. arXiv:2401.17749, January 2024. doi: 10.48550/arXiv.2401.17749.
- 880 Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- 881 Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. doi: 10.48550/ARXIV.2402.03300. URL <https://doi.org/10.48550/arXiv.2402.03300>.
- 882 Reshabh K Sharma, Jonathan De Halleux, Shraddha Barke, and Benjamin Zorn. Promptpex: Automatic test generation for language model prompts. *arXiv preprint arXiv:2503.05070*, 2025.
- 883 Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. A Comparative Analysis of Large Language Models for Code Documentation Generation. *arXiv e-prints*, art. arXiv:2312.10349, December 2023. doi: 10.48550/arXiv.2312.10349.
- 884 Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. Pangu-coder2: Boosting large language models for code with ranking feedback, 2023. URL <https://arxiv.org/abs/2307.14936>.

- 885 Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*, 2023.
- 886 Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. "do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1671–1685, 2024.
- 887 Yixiang Shen, Zhizhou Zhao, Zirui Yang, Haoye Guo, Jia Li, Yang Liu, and Huan Liu. Prosecl: Fortifying code llms with proactive security alignment. *arXiv preprint arXiv:2406.12455*, 2024.
- 888 Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.
- 889 Jiajun Shi, Chaoren Wei, Liqun Yang, Zekun Moore Wang, Chenghao Yang, Ge Zhang, Stephen Huang, Tao Peng, Jian Yang, and Zhoufutu Wen. CryptoX : Compositional Reasoning Evaluation of Large Language Models. *arXiv e-prints*, art. arXiv:2502.07813, February 2025. doi: 10.48550/arXiv.2502.07813.
- 890 Jiajun Shi, Jian Yang, Jiaheng Liu, Xingyuan Bu, Jiangjie Chen, Junting Zhou, Kaijing Ma, Zhoufutu Wen, Bingli Wang, Yancheng He, Liang Song, Hualei Zhu, Shilong Li, Xingjian Wang, Wei Zhang, Ruibin Yuan, Yifan Yao, Wenjun Yang, Yunli Wang, Siyuan Fang, Siyu Yuan, Qianyu He, Xiangru Tang, Yinghui Tan, Wangchunshu Zhou, Zhaoxiang Zhang, Zhoujun Li, Wenhao Huang, and Ge Zhang. KORGym: A Dynamic Game Platform for LLM Reasoning Evaluation. *arXiv e-prints*, art. arXiv:2505.14552, May 2025. doi: 10.48550/arXiv.2505.14552.
- 891 Wenqi Shi, Ran Xu, Yuchen Zhuang, Yue Yu, Jieyu Zhang, Hang Wu, Yuanda Zhu, Joyce Ho, Carl Yang, and May D Wang. Ehragent: Code empowers large language models for few-shot complex tabular reasoning on electronic health records. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing. Conference on Empirical Methods in Natural Language Processing*, volume 2024, page 22315, 2024.
- 892 Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Longcodezip: Compress long context for code language models. *arXiv preprint arXiv:2510.00446*, 2025.
- 893 Hyungyu Shin, Jingyu Tang, Yoonjoo Lee, Nayoung Kim, Hyunseung Lim, Ji Yong Cho, Hwajung Hong, Moontae Lee, and Juho Kim. Automatically evaluating the paper reviewing capability of large language models. *arXiv e-prints*, pages arXiv–2502, 2025.
- 894 Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv: 2303.11366*, 2023.
- 895 Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. Automatic metadata extraction for text-to-sql. *CoRR*, abs/2505.19988, 2025.
- 896 Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. In *arXiv preprint arXiv:1909.08053*, 2019.

- 897 Mohammad Shoeybi, Mostafa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL <https://arxiv.org/abs/1909.08053>.
- 898 Zhihao Shuai, Boyan Li, Siyu Yan, Yuyu Luo, and Weikai Yang. Deepvis: Bridging natural language and data visualization through step-wise reasoning, 2025. URL <https://arxiv.org/abs/2508.01700>.
- 899 Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. *arXiv preprint arXiv:2403.03163*, 2024.
- 900 André Silva, Sen Fang, and Martin Monperrus. Repairllama: Efficient representations and fine-tuned adapters for program repair. *IEEE Transactions on Software Engineering*, 2025.
- 901 André Silva, Gustav Thorén, and Martin Monperrus. Gradient-based program repair: Fixing bugs in continuous program spaces. *arXiv preprint arXiv:2505.17703*, 2025.
- 902 HoHyun Sim, Hyeonjoong Cho, Yeonghyeon Go, Zhoulai Fu, Ali Shokri, and Binoy Ravindran. Large language model-powered agent for c to rust code translation. *arXiv preprint arXiv:2505.15858*, 2025.
- 903 Simon Willison. Here's how i use llms to help me write code, 2025. URL <https://simonwillison.net/2025/Mar/11/using-llms-for-code/>.
- 904 Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.
- 905 Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.
- 906 Shiven Sinha, Shashwat Goel, Ponnurangam Kumaraguru, Jonas Geiping, Matthias Bethge, and Ameya Prabhu. Can language models falsify? evaluating algorithmic reasoning with counterexample creation, 2025. URL <https://arxiv.org/abs/2502.19414>.
- 907 Shaden Smith, Mostafa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- 908 Atefeh Sohrabizadeh, Jialin Song, Mingjie Liu, Rajarshi Roy, Chankyu Lee, Jonathan Raiman, and Bryan Catanzaro. Nemotron-cortexa: Enhancing llm agents for software engineering tasks via improved localization and solution diversity. In *Forty-second International Conference on Machine Learning*.
- 909 Lola Solovyeva, Sophie Weidmann, and Fernando Castor. Ai-powered, but power-hungry? energy efficiency of llm-generated code, 2025. URL <https://arxiv.org/abs/2502.02412>.
- 910 Chengyu Song, Linru Ma, Jianming Zheng, Jinzhi Liao, Hongyu Kuang, and Lin Yang. Audit-llm: Multi-agent collaboration for log-based insider threat detection. *arXiv preprint arXiv:2408.08902*, 2024.

- 911 Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, et al. Bugpilot: Complex bug generation for efficient learning of sw skills. *arXiv preprint arXiv:2510.19898*, 2025.
- 912 Sourcegraph. Cody: AI code assistant. Technical report, Sourcegraph Inc., 2024.
- 913 Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai's ability to replicate ai research, 2025. URL <https://arxiv.org/abs/2504.01848>.
- 914 Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. REASONING GYM: Reasoning Environments for Reinforcement Learning with Verifiable Rewards. *arXiv e-prints*, art. arXiv:2505.24760, May 2025. doi: 10.48550/arXiv.2505.24760.
- 915 Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.
- 916 Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. Hotgpt: How to make software documentation more useful with a large language model? In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HotOS '23, page 87–93, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701955. doi: 10.1145/3593856.3595910. URL <https://doi.org/10.1145/3593856.3595910>.
- 917 Zhenpeng Su, Leiyu Pan, Xue Bai, Dening Liu, Guanting Dong, Jiaming Huang, Wenping Hu, Fuzheng Zhang, Kun Gai, and Guorui Zhou. Klear-reasoner: Advancing reasoning capability via gradient-preserving clipping policy optimization. *arXiv preprint arXiv:2508.07629*, 2025.
- 918 Hao Sun, Yunyi Shen, and Jean-Francois Ton. Rethinking bradley-terry models in preference-based reward modeling: Foundations, theory, and alternatives. *arXiv preprint arXiv:2411.04991*, 2024.
- 919 Jiawei Sun, Tianyu Li, and Xinyu Zhao. Repofixeval: Issue-aware repository-level benchmark for automated program repair. In *International Conference on Learning Representations (ICLR)*, 2025.
- 920 Keyi Sun, Jiazen Chen, Xiao Zhang, Yinda Zhang, Zirui Wang, Zhiyue Zhang, Ruofan Wang, Pin-Yu Chen, and Zeyi Qin. Safety-aware fine-tuning of large language models. *arXiv preprint arXiv:2405.09919*, 2024.
- 921 Qiushi Sun, Nuo Chen, Jianing Wang, Xiang Li, and Ming Gao. Transcoder: Towards unified transferable code representation learning inspired by human skills. *arXiv preprint arXiv:2306.07285*, 2023.
- 922 Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023. URL <https://arxiv.org/abs/2307.08621>.

- 923 Manan Suri, Puneet Mathur, Franck Dernoncourt, Rajiv Jain, Vlad I Morariu, Ramit Sawhney, Preslav Nakov, and Dinesh Manocha. Docedit-v2: Document structure editing via multimodal llm grounding. *arXiv preprint arXiv:2410.16472*, 2024.
- 924 Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning, 2023. URL <https://arxiv.org/abs/2303.08128>.
- 925 Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014. URL <https://arxiv.org/abs/1409.3215>.
- 926 SWE-Agent Project. Swe-agent documentation, 2024.
- 927 swe-bench live. swe-bench-live, 2025. URL <https://swe-bench-live.github.io/>.
- 928 swebench. swebench, 2025. URL <https://www.swebench.com/original.html>.
- 929 swebenchmultilingual. swebenchmultilingual, 2025. URL <https://www.swebench.com/multilingual.html>.
- 930 swebenchverified. swebenchverified, 2025. URL <https://openai.com/index/introducing-swe-bench-verified/>.
- 931 Tabnine. Tabnine: Enterprise-grade AI code assistant. Technical report, Tabnine Ltd., 2024. URL <https://www.tabnine.com>.
- 932 Chang-Yu Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. Exploring chain of thought style prompting for text-to-sql. In *EMNLP 2023, Singapore, December 6-10, 2023*, pages 5376–5393. Association for Computational Linguistics, 2023.
- 933 Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. CHESS: contextual harnessing for efficient SQL synthesis. *CoRR*, abs/2405.16755, 2024.
- 934 Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 3473–3487. Association for Computational Linguistics, 2024.
- 935 Hongze Tan and Jianfei Pan. Gtpo and grp0-s: Token and sequence-level reward shaping with policy entropy. *arXiv preprint arXiv:2508.04349*, 2025.
- 936 Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark Gerstein. Biocoder: A benchmark for bioinformatics code generation with large language models, 2024. URL <https://arxiv.org/abs/2308.16458>.
- 937 Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F Bissyandé. Codeagent: Autonomous communicative agents for code review. *arXiv preprint arXiv:2402.02172*, 2024.
- 938 Xunzhu Tang, Jacques Klein, and Tegawendé F Bissyandé. Boosting open-source llms for program repair via reasoning transfer and llm-guided reinforcement learning. *arXiv preprint arXiv:2506.03921*, 2025.

- 939 Yuheng Tang, Hongwei Li, Kaijie Zhu, Michael Yang, Yangruibo Ding, and Wenbo Guo. Co-patcher: Collaborative software patching with component (s)-specific small reasoning models. *arXiv preprint arXiv:2505.18955*, 2025.
- 940 Yunhao Tang, Kunhao Zheng, Gabriel Synnaeve, and Rémi Munos. Optimizing language models for inference time objectives using reinforcement learning, 2025. URL <https://arxiv.org/abs/2503.19595>.
- 941 Zhi-Yi Tang, Hong-Xin Zhang, Zhen-Dong Sha, Xu-Hui Liu, Can-Ran Wang, Zhao-Wei Li, Feng-Lin Li, Gang Zhao, Jia-Ju He, and Yue-Hui Chen. Redcodeagent: Automatic red-teaming agent against diverse code agents, 2024.
- 942 Zilu Tang, Mayank Agarwal, Alex Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. Explain-then-translate: An analysis on improving program translation with self-generated explanations, 2023. URL <https://arxiv.org/abs/2311.07070>.
- 943 Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1741–1788, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.119. URL <https://aclanthology.org/2023.findings-emnlp.119>.
- 944 Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, et al. Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks. *arXiv preprint arXiv:2505.16901*, 2025.
- 945 Shimin Tao, Weibin Meng, Yimeng Chen, Yichen Zhu, Ying Liu, Chunling Du, Tao Han, Yongpeng Zhao, Xiangguang Wang, and Hao Yang. Logstamp: Automatic online log parsing based on sequence labelling. *SIGMETRICS Perform. Evaluation Rev.*, 49(4):93–98, 2022.
- 946 Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. On the evaluation of commit message generation models: An experimental study, 2021. URL <https://arxiv.org/abs/2107.05373>.
- 947 Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993, 2024.
- 948 Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.
- 949 Aider Team. Aider: Ai pair programming in your terminal. <https://aider.chat/>, 2024. Accessed: 2024.
- 950 AlphaCode Team. Alphacode 2 technical report. Technical report, Google DeepMind, December 2023. URL https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf.
- 951 Augment Code Team. Augment code: The most powerful ai software development platform. <https://www.augmentcode.com/>, 2024. Accessed: 2024.

- 952 Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, and et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024. URL <https://arxiv.org/abs/2403.05530>.
- 953 Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, Soroosh Mariooryad, Yifan Ding, Xinyang Geng, Fred Alcober, Roy Frostig, Mark Omernick, Lexi Walker, Cosmin Paduraru, Christina Sorokin, Andrea Tacchetti, Colin Gaffney, Samira Daruki, Olcan Sercinoglu, Zach Gleicher, Juliette Love, Paul Voigtlaender, Rohan Jain, Gabriela Surita, Kareem Mohamed, Rory Blevins, Junwhan Ahn, Tao Zhu, Kornraphop Kawintiranon, Orhan Firat, Yiming Gu, Yujing Zhang, Matthew Rahtz, Manaal Faruqui, Natalie Clay, Justin Gilmer, JD Co-Reyes, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024. URL <https://arxiv.org/abs/2403.05530>.
- 954 Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, and et al. Gemini: A family of highly capable multimodal models, 2025. URL <https://arxiv.org/abs/2312.11805>.
- 955 Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Jack Krawczyk, et al. Gemini: A family of highly capable multimodal models, 2025. URL <https://arxiv.org/abs/2312.11805>.
- 956 Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, et al. Gemma 3 technical report, 2025. URL <https://arxiv.org/abs/2503.19786>.
- 957 Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.
- 958 Ling Team, Wenting Cai, Yuchen Cao, Chaoyu Chen, Chen Chen, Siba Chen, Qing Cui, Peng Di, Junpeng Fang, Zi Gong, et al. Every sample matters: Leveraging mixture-of-experts and high-quality data for efficient and accurate code llm. *arXiv preprint arXiv:2503.17793*, 2025.
- 959 Manus Team. Leave it to manus. <https://manus.im/>, 2025.
- 960 OpenAI Team. Introducing deep research. <https://openai.com/index/introducing-dee-p-research/>, 2025.
- 961 Qwen Team. Codeqwen1.5. <https://huggingface.co/Qwen/CodeQwen1.5-7B>, 2024.

- 962 Qwen Team. Introducing qwen1.5, February 2024. URL <https://qwenlm.github.io/blog/qwen1.5/>.
- 963 Qwen Team. Qwen3-next: Towards ultimate training & inference efficiency, September 2025. URL <https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list>.
- 964 Refact Team. Refact.ai. <https://refact.ai/>, 2025. Accessed: 2025.
- 965 The OpenBlock Team. Openblock secures #2 on terminal bench with frontier agent ob-1, August 2025. URL <https://www.openblocklabs.com/research/terminal-bench>.
- 966 The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025. URL <https://github.com/laude-institute/terminal-bench>.
- 967 Shailja Thakur, Baleegh Ahmad, Hammond Fan, et al. VeriGen: A large language model for verilog code generation. *arXiv preprint arXiv:2308.00708*, 2023.
- 968 Surendrabikram Thapa, Usman Naseem, and Mehwish Nasim. From humans to machines: can chatgpt-like llms effectively replace human annotators in nlp tasks. In *Workshop Proceedings of the 17th International AAAI Conference on Web and Social Media*. Association for the Advancement of Artificial Intelligence, 2023.
- 969 Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists, 2024. URL <https://arxiv.org/abs/2407.13168>.
- 970 Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. Debugbench: Evaluating debugging capability of large language models, 2024. URL <https://arxiv.org/abs/2401.04621>.
- 971 Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. How secure is ai-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(2):47, 2024. ISSN 1573-7616. doi: 10.1007/s10640-024-10590-1. URL <https://doi.org/10.1007/s10664-024-10590-1>.
- 972 Norbert Tihanyi, Yiannis Charalambous, Ridhi Jain, Mohamed Amine Ferrag, and Lucas C Cordeiro. A new era in software security: Towards self-healing software via large language models and formal verification. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 136–147. IEEE, 2025.
- 973 Frank Tip, Jonathan Bell, and Max Schäfer. Llmorpheus: Mutation testing using large language models. *IEEE Transactions on Software Engineering*, 2025.
- 974 Lindia Tjuatja, Valerie Chen, Tongshuang Wu, Ameet Talwalkar, and Graham Neubig. Do llms exhibit human-like response biases? a case study in survey design. *Transactions of the Association for Computational Linguistics*, 12:1011–1026, 2024.
- 975 TogetherAI. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl. <https://www.together.ai/blog/deepswe>, 2025. Accessed: 2025.

- 976 Toggle Project. Toggle: Token-level localization for automated program repair. <https://github.com/Toggle-APR/toggle>, 2024.
- 977 Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184*, 2024.
- 978 Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models, 2024. URL <https://arxiv.org/abs/2410.02184>.
- 979 Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- 980 Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikell, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL <https://arxiv.org/abs/2307.09288>.
- 981 Towards Data Science. Why i stopped using cursor and reverted to vscode. <https://towardsdatascience.com/vscode-is-the-best-ai-powered-ide>, 2025.
- 982 Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A Standard Interface for Reinforcement Learning Environments. *arXiv e-prints*, art. arXiv:2407.17032, July 2024. doi: 10.48550/arXiv.2407.17032.
- 983 TRAE. TRAE - collaborate with intelligence. Technical report, TRAE, 2025. URL <https://trae.ai>.
- 984 Hieu Tran, Ngoc M. Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. Recovering variable names for minified code with usage contexts. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1165–1175. IEEE / ACM, 2019.
- 985 Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Code less, align more: Efficient llm fine-tuning for code generation with data pruning, 2024. URL <https://arxiv.org/abs/2407.05040>.
- 986 Songjun Tu, Jiahao Lin, Xiangyu Tian, Qichao Zhang, Linjing Li, Yuqian Fu, Nan Xu, Wei He, Xiangyuan Lan, Dongmei Jiang, et al. Enhancing llm reasoning with iterative dpo: A comprehensive empirical investigation. *arXiv preprint arXiv:2503.12854*, 2025.

- 987 Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.
- 988 Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. Autodev: Automated ai-driven development. *arXiv preprint arXiv:2403.08299*, 2024.
- 989 Mansi Uniyal, Mukul Singh, Gust Verbruggen, Sumit Gulwani, and Vu Le. One-to-many testing for code generation from (just) natural language. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 15397–15402, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.902. URL <https://aclanthology.org/2024.findings-emnlp.902/>.
- 990 Aidar Valeev, Roman Garaev, Vadim Lomshakov, Irina Piontkovskaya, Vladimir Ivanov, and Israel Adewuyi. Yabloco: Yet another benchmark for long context code generation, 2025. URL <https://arxiv.org/abs/2505.04406>.
- 991 Thomas Valentin, Ardi Madadi, Gaetano Sapia, and Marcel Böhme. Estimating correctness without oracles in llm-based code generation, 2025. URL <https://arxiv.org/abs/2507.00057>.
- 992 Tim Van Erven and Peter Harremos. Rényi divergence and kullback-leibler divergence. *IEEE Transactions on Information Theory*, 60(7):3797–3820, 2014.
- 993 S. R. P. van Hal, M. Post, and K. Wendel. Generating commit messages from git diffs, 2019. URL <https://arxiv.org/abs/1911.11690>.
- 994 Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*, 2019.
- 995 Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. Recovering clear, natural identifiers from obfuscated JS names. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 683–693. ACM, 2017.
- 996 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547de91fb053c1c4a845aa-Paper.pdf.
- 997 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.
- 998 Awid Vaziry, Sandro Rodriguez Garzon, and Axel Küpper. Towards multi-agent economies: Enhancing the a2a protocol with ledger-anchored identities and x402 micropayments for ai agents. *arXiv preprint arXiv:2507.19550*, 2025.

- 999** Visual Studio Blog. Top 5 github copilot features in visual studio from microsoft ignite 2024. <https://devblogs.microsoft.com/visualstudio/top-5-github-copilot-features/>, 2025.
- 1000** Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *ArXiv*, abs/2407.14044, 2024. URL <https://api.semanticscholar.org/CorpusID:271310399>.
- 1001** Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407, 2018.
- 1002** Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael Lyu. Divide-and-conquer: Generating ui code from screenshots. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3729364. URL <https://doi.org/10.1145/3729364>.
- 1003** Zhipeng Wan, Anda Cheng, Yinggui Wang, and Lei Wang. Information leakage from embedding in large language models, 2024.
- 1004** Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for linux containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102. IEEE, 2017.
- 1005** Amuguleng Wang, Yilagui Qi, and Dahu Baiyila. C-bert: A mongolian reverse dictionary based on fused lexical semantic clustering and bert. *Alexandria Engineering Journal*, 111: 385–395, 2025.
- 1006** Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7567–7578. Association for Computational Linguistics, 2020.
- 1007** Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. MAC-SQL: A multi-agent collaborative framework for text-to-sql. *CoRR*, abs/2312.11242, 2023.
- 1008** Bo Wang, Pengyang Wang, Chong Chen, Qi Sun, Jieke Shi, Chengran Yang, Ming Deng, Youfang Lin, Zhou Yang, and David Lo. Mut4all: Fuzzing compilers via llm-synthesized mutators learned from bug reports, 2025. URL <https://arxiv.org/abs/2507.19275>.
- 1009** Boshi Wang, Sewon Min, Xiang Deng, Jiaming Shen, You Wu, Luke Zettlemoyer, and Huan Sun. Towards understanding chain-of-thought prompting: An empirical study of what matters. *arXiv preprint arXiv:2212.10001*, 2022.
- 1010** Chaozheng Wang, Zhenhao Nong, Cuiyun Gao, Zongjie Li, Jichuan Zeng, Zhenchang Xing, and Yang Liu. Enriching query semantics for code search with reinforcement learning. *Neural Networks*, 145:22–32, 2022.
- 1011** Charles L Wang, Trisha Singhal, Ameya Kelkar, and Jason Tuo. Mi9–agent intelligence protocol: Runtime governance for agentic ai systems. *arXiv preprint arXiv:2508.03858*, 2025.

- 1012** Che Wang, Jiashuo Zhang, Jianbo Gao, Libin Xia, Zhi Guan, and Zhong Chen. Contract-tinker: Llm-empowered vulnerability repair for real-world smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2350–2353, 2024.
- 1013** Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv: 2305.16291*, 2023.
- 1014** Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- 1015** Hanbin Wang, Xiaoxuan Zhou, Zhipeng Xu, Keyuan Cheng, Yuxin Zuo, Kai Tian, Jingwei Song, Junting Lu, Wenhui Hu, and Xueyang Liu. Code-vision: Evaluating multimodal llms logic understanding and code generation capabilities. *arXiv preprint arXiv:2502.11829*, 2025.
- 1016** Haoran Wang, Zhipeng Wan, Yinggui Wang, and Lei Wang. Privacy-aware decoding: Mitigating privacy leakage of large language models in retrieval-augmented generation, 2025.
- 1017** Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–30, 2021.
- 1018** Jicheng Wang, Yifeng He, and Hao Chen. Repogenreflex: Enhancing repository-level code completion with verbal reinforcement and retrieval-augmented generation. *arXiv preprint arXiv:2409.13122*, 2024.
- 1019** Jixin Wang, Haonan Li, Jiayuan Chen, Zeliang Yu, Zongze Li, Wenyu Zhu, Li Li, and Qing Liao. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval, 2024.
- 1020** Jixin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395*, 2024.
- 1021** Kaiwen Wang, Rahul Kidambi, Ryan Sullivan, Alekh Agarwal, Christoph Dann, Andrea Michi, Marco Gelmi, Yunxuan Li, Raghav Gupta, Avinava Dubey, et al. Conditional language policy: A general framework for steerable multi-objective finetuning. *arXiv preprint arXiv:2407.15762*, 2024.
- 1022** Le Wang, Zhiliang Wang, Zepu Zong, Yuxin Huang, Min Yang, Ruipu Li, Runze Wang, Zhaoyue Cheng, Jing Liu, Xingyu Ma, et al. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. *arXiv preprint arXiv:2405.01358*, 2024.
- 1023** Lei Wang, Chengbang Ma, Xueyang Feng, Zeyu Zhang, Hao ran Yang, Jingsen Zhang, Zhi-Yang Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji rong Wen. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.*, 2023. doi: 10.1007/s11704-024-40231-1.

- 1024** Liran Wang, Xunzhu Tang, Yichen He, Changyu Ren, Shuhua Shi, Chaoran Yan, and Zhoujun Li. Delving into commit-issue correlation to enhance commit message generation models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 710–722. IEEE, 2023.
- 1025** Liyang Wang, Yu Cheng, Xingxin Gu, and Zhizhong Wu. Design and optimization of big data and machine learning-based risk monitoring system in financial markets. *arXiv preprint arXiv:2407.19352*, 2024.
- 1026** Min-Hui Wang, Groundhog-Day, Zhaowei Li, Jia-Ju He, Jia-Hao Ji, Yang-Kai-Hsiang, Feng-Lin Li, Wen-Hao Chen, and Zhe-Wei Lin. Redcoder: Automated multi-turn red teaming for code llms, 2024.
- 1027** Peiran Wang, Yang Liu, Yunfei Lu, Yifeng Cai, Hongbo Chen, Qingyou Yang, Jie Zhang, Jue Hong, and Ye Wu. Agentarmor: Enforcing program analysis on agent runtime trace to defend against prompt injection. *arXiv preprint arXiv:2508.01249*, 2025.
- 1028** Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL 2024, Bangkok, Thailand, August 11-16, 2024, pages 9426–9439. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.510. URL <https://doi.org/10.18653/v1/2024.acl-long.510>.
- 1029** Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. Qwen2-vl: Enhancing vision-language model’s perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*, 2024.
- 1030** Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xionghui Chen, Jianxin Yang, Zhenru Zhang, Yuqiong Liu, An Yang, Andrew Zhao, Yang Yue, Shiji Song, Bowen Yu, Gao Huang, and Junyang Lin. Beyond the 80/20 Rule: High-Entropy Minority Tokens Drive Effective Reinforcement Learning for LLM Reasoning, June 2025. URL <http://arxiv.org/abs/2506.01939>. arXiv:2506.01939 [cs] Read_Status: New Read_Status_Date: 2025-06-15T15:30:33.418Z.
- 1031** Shuai Wang, Liang Ding, Li Shen, Yong Luo, Bo Du, and Dacheng Tao. Oop: Object-oriented programming evaluation benchmark for large language models, 2024. URL <https://arxiv.org/abs/2401.06628>.
- 1032** Wei Wang, Jiang Liu, Kai Zhang, et al. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*, 2023. URL <https://arxiv.org/abs/2508.00083>.
- 1033** Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 146–158, 2023.
- 1034** Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, et al. Internvl3.5: Advancing open-source multimodal models in versatility, reasoning, and efficiency. *arXiv preprint arXiv:2508.18265*, 2025.

- 1035 Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. TESTEVAL: Benchmarking Large Language Models for Test Case Generation, February 2025. URL <http://arxiv.org/abs/2406.04531>. arXiv:2406.04531 [cs].
- 1036 Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S Yu, and Guandong Xu. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering*, 48(1):102–119, 2020.
- 1037 Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. Aegis: An agent-based framework for general bug reproduction from issue descriptions. *arXiv preprint arXiv:2411.18015*, 2024.
- 1038 Xinchen Wang, Pengfei Gao, Chao Peng, Ruida Hu, and Cuiyun Gao. Codevisionary: An agent-based framework for evaluating large language models in code generation, 2025. URL <https://arxiv.org/abs/2504.13472>.
- 1039 Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.
- 1040 Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.
- 1041 Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- 1042 Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 165–177. IEEE Computer Society, 2024.
- 1043 Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Weiran Xu, Jingang Wang, Mengdi Zhang, and Xunliang Cai. DolphCoder: Echo-locating code large language models with diverse and multi-objective instruction tuning. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4706–4721, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.259. URL <https://aclanthology.org/2024.acl-long.259>.
- 1044 Yejie Wang, Keqing He, Dayuan Fu, Zhuoma GongQue, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, Jingang Wang, Mengdi Zhang, Xunliang Cai, and Weiran Xu. How do your code LLMs perform? empowering code instruction tuning with really good data. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14027–14043, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.777. URL <https://aclanthology.org/2024.emnlp-main.777>.
- 1045 Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. *arXiv preprint arXiv:2506.03136*, 2025.

- 1046** Yixuan Wang, Xianzhen Luo, Fuxuan Wei, Yijun Liu, Qingfu Zhu, Xuanyu Zhang, Qing Yang, Dongliang Xu, and Wanxiang Che. Make some noise: Unlocking language model parallel inference capability through noisy training. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Miami, Florida, USA, November 2024. Association for Computational Linguistics. URL <https://aclanthology.org/2024.emnlp-main.718/>.
- 1047** Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL <https://aclanthology.org/2021.emnlp-main.685/>.
- 1048** Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.68. URL <https://aclanthology.org/2023.emnlp-main.68/>.
- 1049** Yue Wang, Hung Le, Akhilesh Deepak Gotmare, et al. CodeT5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.
- 1050** Yutong Wang, Pengliang Ji, Chaoqun Yang, Kaixin Li, Ming Hu, Jiaoyang Li, and Guillaume Sartoretti. Mcts-judge: Test-time scaling in llm-as-a-judge for code correctness evaluation, 2025. URL <https://arxiv.org/abs/2502.12468>.
- 1051** Zeyun Wang, Kaibo Liu, Ge Li, and Zhi Jin. Hits: High-coverage llm-based unit test generation via method slicing, 2024. URL <https://arxiv.org/abs/2408.11324>.
- 1052** Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, Tianyu Liu, and Weiran Xu. Ojbench: A competition level code benchmark for large language models, 2025. URL <https://arxiv.org/abs/2506.16395>.
- 1053** Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. Mconala: A benchmark for code generation from multiple natural languages. In Andreas Vlachos and Isabelle Augenstein, editors, *Findings of the Association for Computational Linguistics: EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, pages 265–273. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-EACL.20. URL <https://doi.org/10.18653/v1/2023.findings-eacl.20>.
- 1054** Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 1271–1290. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-EMNLP.89. URL <https://doi.org/10.18653/v1/2023.findings-emnlp.89>.
- 1055** Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. Codecontests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*, 2025.

- 1056** Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. Coderag-bench: Can retrieval augment code generation? In *North American Chapter of the Association for Computational Linguistics*, 2024. URL <https://api.semanticscholar.org/CorpusID:270620678>.
- 1057** Francis Rhys Ward, Matt MacDermott, Francesco Belardinelli, Francesca Toni, and Tom Everitt. The reasons that agents act: Intention and instrumental goals. *arXiv preprint arXiv:2402.07221*, 2024.
- 1058** Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does llm safety training fail? *Advances in Neural Information Processing Systems*, 36:80079–80110, 2023.
- 1059** Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, Ke Wang, et al. Equibench: Benchmarking large language models' understanding of program semantics via equivalence checking. *arXiv preprint arXiv:2502.12466*, 2025.
- 1060** Bingyang Wei. Requirements are all you need: From requirements to code with llms. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 416–422. IEEE, 2024.
- 1061** Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32, 2019.
- 1062** Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. URL <https://openreview.net/forum?id=yzkSU5zdwD>.
- 1063** Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.
- 1064** Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.
- 1065** Wenjie Wei, Yuqi Li, Tianyu Chen, et al. AutoSpec: Enchanting program specification synthesis by large language models using static analysis and program verification. In *CAV*, 2024.
- 1066** Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023.
- 1067** Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering Code Generation with OSS-Instruct, 2024.

- 1068** Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carboneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.
- 1069** Dean Weissman. A microarchitectural threat analysis of us federal cloud environments. *arXiv preprint arXiv:2311.13327*, 2023.
- 1070** Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. Unlocking reasoning potential in large langauge models by scaling code-form planning. *arXiv preprint arXiv:2409.12452*, 2024.
- 1071** Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, et al. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR, 2022.
- 1072** Kyle Wiggers. Amazon codewhisperer is now called q developer. <https://techcrunch.com/2024/04/30/amazon-codewhisperer-q-developer/>, 2024.
- 1073** Wikipedia. Github copilot. https://en.wikipedia.org/wiki/GitHub_Copilot, 2024.
- 1074** Jason Williams, Antoine Raux, Deepak Ramachandran, and Alan Black. The dialog state tracking challenge. In Maxine Eskenazi, Michael Strube, Barbara Di Eugenio, and Jason D. Williams, editors, *Proceedings of the SIGDIAL 2013 Conference*, pages 404–413, Metz, France, August 2013. Association for Computational Linguistics. URL <https://aclanthology.org/W13-4065/>.
- 1075** Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567, 2013. doi: 10.1109/ASE.2013.6693113.
- 1076** Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389, 2015. doi: 10.1109/SANER.2015.7081848.
- 1077** Ryan Wong, Jiawei Wang, Junjie Zhao, Li Chen, Yan Gao, Long Zhang, Xuan Zhou, Zuo Wang, Kai Xiang, Ge Zhang, et al. Widesearch: Benchmarking agentic broad info-seeking. *arXiv preprint arXiv:2508.07999*, 2025.
- 1078** Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled C code with large language models. *CoRR*, abs/2310.06530, 2023.
- 1079** Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models, 2023. URL <https://arxiv.org/abs/2303.04671>.
- 1080** Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. *arXiv preprint arXiv:2405.07990*, 2024.

- 1081** Fan Wu, Cuiyun Gao, Shuqing Li, Xin-Cheng Wen, and Qing Liao. Mllm-based ui2code automation guided by ui layout information. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1123–1145, 2025.
- 1082** Fang Wu, Weihao Xuan, Ximing Lu, Zaid Harchaoui, and Yejin Choi. The invisible leash: Why rlvr may not escape its origin. *arXiv preprint arXiv:2507.14843*, 2025.
- 1083** Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. Deceptprompt: Exploiting llm-driven code generation via adversarial natural language instructions. *arXiv preprint arXiv:2312.04730*, 2023.
- 1084** Haoze Wu, Yunzhi Yao, Wenhao Yu, and Ningyu Zhang. Recode: Updating code api knowledge with reinforcement learning. *arXiv preprint arXiv:2506.20495*, 2025.
- 1085** Hefeng Wu, Yandong Chen, Lingbo Liu, Tianshui Chen, Keze Wang, and Liang Lin. Sqlnet: Scale-modulated query and localization network for few-shot class-agnostic counting. *CoRR*, abs/2311.10011, 2023.
- 1086** Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey P Bigham, and Jeffrey Nichols. Uicoder: Finetuning large language models to generate user interface code through automated feedback. *arXiv preprint arXiv:2406.07739*, 2024.
- 1087** Jialong Wu, Wenbiao Yin, Yong Jiang, Zhenglin Wang, Zekun Xi, Runnan Fang, Linhai Zhang, Yulan He, Deyu Zhou, Pengjun Xie, and Fei Huang. Webwalker: Benchmarking llms in web traversal, 2025. URL <https://arxiv.org/abs/2501.07572>.
- 1088** Qishen Wu, Xinyin Chen, Can Jiang, Jing Liu, Yang Li, Yilong Li, and Nicholas Jing Yuan. Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning. *arXiv preprint arXiv:2405.18783*, 2024.
- 1089** Shican Wu, Xiao Ma, Dehui Luo, Lulu Li, Xiangcheng Shi, Xin Chang, Xiaoyun Lin, Ran Luo, Chunlei Pei, Changying Du, Zhi-Jian Zhao, and Jinlong Gong. Automated literature research and review-generation method based on large language models. *National Science Review*, 12(6), April 2025. ISSN 2053-714X. doi: 10.1093/nsr/nwaf169. URL <http://dx.doi.org/10.1093/nsr/nwaf169>.
- 1090** Tong Wu, Zhihao Fan, Xiao Liu, Yeyun Gong, Yelong Shen, Jian Jiao, Hai-Tao Zheng, Juntao Li, Zhongyu Wei, Jian Guo, Nan Duan, and Weizhu Chen. Ar-diffusion: Auto-regressive diffusion model for text generation, 2023. URL <https://arxiv.org/abs/2305.09515>.
- 1091** Yifan Wu, Lutao Yan, Leixian Shen, Yunhai Wang, Nan Tang, and Yuyu Luo. Chartinsights: Evaluating multimodal large language models for low-level chart question answering, 2024. URL <https://arxiv.org/abs/2405.07001>.
- 1092** Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation (2023). URL <https://arxiv.org/abs/2308.15276>.
- 1093** Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google’s neural machine translation system: Bridging the gap between human and machine translation, 2016. URL <https://arxiv.org/abs/1609.08144>.

- 1094 Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct, 2024. URL <https://arxiv.org/abs/2407.05700>.
- 1095 Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. Reinam: reinforcement learning for input-grammar inference. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 488–498, 2019.
- 1096 Zhenhe Wu, Zhongqiu Li, Mengxiang Li, Jie Zhang, Zhongjiang He, Jian Yang, Yu Zhao, Ruiyu Fang, Yongxiang Li, Zhoujun Li, and Shuangyong Song. MR-SQL: multi-level retrieval enhances inference for llm in text-to-sql. *DASFAA*, 2025.
- 1097 Zhenhe Wu, Zhongqiu Li, Jie Zhang, Zhongjiang He, Jian Yang, Yu Zhao, Ruiyu Fang, Bing Wang, Hongyan Xie, Shuangyong Song, and Zhoujun Li. UCS-SQL: uniting content and structure for enhanced semantic bridging in text-to-sql. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 8156–8168. Association for Computational Linguistics, 2025.
- 1098 Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024.
- 1099 Zhiyu Wu, Xiaokang Chen, Zizheng Pan, Xingchao Liu, Wen Liu, Damai Dai, Huazuo Gao, Yiyang Ma, Chengyue Wu, Bingxuan Wang, Zhenda Xie, Yu Wu, Kai Hu, Jiawei Wang, Yaofeng Sun, Yukun Li, Yishi Piao, Kang Guan, Aixin Liu, Xin Xie, Yuxiang You, Kai Dong, Xingkai Yu, Haowei Zhang, Liang Zhao, Yisong Wang, and Chong Ruan. Deepseek-vl2: Mixture-of-experts vision-language models for advanced multimodal understanding, 2024. URL <https://arxiv.org/abs/2412.10302>.
- 1100 Zian Wu, Yixuan Jiang, Yiyuan Zhou, and He Wang. Large language models for code: Security hardening and adversarial testing. *arXiv preprint arXiv:2311.00033*, 2023.
- 1101 xAI. Announcing grok. <https://x.ai/news/announcing-grok>, 2023.
- 1102 xAI. Announcing grok-1.5: Advancing long-context understanding and math and coding. <https://x.ai/news/announcing-grok-1-5>, 2024.
- 1103 xAI. Launching grok 2 and a suite of AIs. <https://x.ai/news/grok-2>, 2024. Includes HumanEval pass@1 results for Grok-1.5, Grok-2-mini, and Grok-2.
- 1104 xAI. Api documentation: Models and endpoints. <https://docs.x.ai/docs/models-and-endpoints>, 2025. Lists grok-code-fast-1 as a code-specialized model.
- 1105 xAI. Grok 4. <https://x.ai/news/grok-4>, 2025.
- 1106 xAI. Grok 4 fast. <https://x.ai/news/grok-4-fast>, 2025.
- 1107 Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng,

Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv: 2309.07864*, 2023.

- 1108 Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2630–2642. IEEE, 2023.
- 1109 Chunqiu Silva Xia and Zhang Lingming. Repairllama: Efficient representations for automated program repair. *arXiv preprint arXiv:2312.15698*, 2024.
- 1110 Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.
- 1111 Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.
- 1112 Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for 0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.
- 1113 Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. Revisiting the plastic surgery hypothesis via large language models. *arXiv preprint arXiv:2303.10494*, 2023.
- 1114 Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *CoRR*, abs/2407.01489, 2024. doi: 10.48550/ARXIV.2407.01489. URL <https://doi.org/10.48550/arXiv.2407.01489>.
- 1115 Chunqiu Steven Xia, Yifeng Wei, and Lingming Zhang. AlphaRepair: Code generation from bugs. *ICSE*, 2024.
- 1116 Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Hufeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms, 2025. URL <https://arxiv.org/abs/2504.14655>.
- 1117 Wei Xiang, Hanfei Zhu, Suqi Lou, Xinli Chen, Zhenghua Pan, Yuping Jin, Shi Chen, and Lingyun Sun. Simuser: Generating usability feedback by simulating various users interacting with mobile applications. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, CHI ’24, New York, NY, USA, 2024*. Association for Computing Machinery. ISBN 9798400703300. doi: 10.1145/3613904.3642481. URL <https://doi.org/10.1145/3613904.3642481>.
- 1118 Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R Lyu. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. *arXiv preprint arXiv:2411.03292*, 2024.
- 1119 Shuhong Xiao, Yunnong Chen, Jiazhi Li, Liuqing Chen, Lingyun Sun, and Tingting Zhou. Prototype2code: End-to-end front-end code generation from ui design prototypes. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 88353, page V02BT02A038. American Society of Mechanical Engineers, 2024.

- 1120** Tong Xiao, Zhe Quan, Zhi-Jie Wang, Kaiqi Zhao, and Xiangke Liao. LPV: A log parser based on vectorization for offline and online log parsing. In Claudia Plant, Haixun Wang, Alfredo Cuzzocrea, Carlo Zaniolo, and Xindong Wu, editors, *20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020*, pages 1346–1351. IEEE, 2020.
- 1121** Yuan-An Xiao, Weixuan Wang, Dong Liu, Junwei Zhou, Shengyu Cheng, and Yingfei Xiong. Predicatefix: Repairing static analysis alerts with bridging predicates. *arXiv preprint arXiv:2503.12205*, 2025.
- 1122** Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swefixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*, 2025.
- 1123** Danning Xie, Mingwei Zheng, Xuwei Liu, Jiannan Wang, Chengpeng Wang, Lin Tan, and Xiangyu Zhang. Core: Benchmarking llms code reasoning capabilities through static analysis tasks. *arXiv preprint arXiv:2507.05269*, 2025.
- 1124** Rui Xie, Zhengran Zeng, Zhuohao Yu, Chang Gao, Shikun Zhang, and Wei Ye. Codeshell technical report. *CoRR*, abs/2403.15747, 2024. doi: 10.48550/ARXIV.2403.15747. URL <https://doi.org/10.48550/arXiv.2403.15747>.
- 1125** Wenxuan Xie, Gaochen Wu, and Bowen Zhou. MAG-SQL: multi-agent generative approach with soft schema linking and iterative sub-sql refinement for text-to-sql. *CoRR*, abs/2408.07930, 2024.
- 1126** Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Proc. ACM Manag. Data*, 3(3):194:1–194:24, 2025.
- 1127** Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. Repost: Scalable repository-level coding environment construction with sandbox testing, 2025.
- 1128** Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. Repost: Scalable repository-level coding environment construction with sandbox testing, 2025. URL <https://arxiv.org/abs/2503.07358>.
- 1129** Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, et al. Dream-coder 7b: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*, 2025.
- 1130** Li Xin-Ye, Du Ya-Li, and Li Ming. Enhancing llms in long code translation through instrumentation and program state alignment, 2025. URL <https://arxiv.org/abs/2504.02017>.
- 1131** Jun Xing, Mayur Bhatia, Sahil Phulwani, Darshan Suresh, and Rafik Matta. Hackerrank-astro: Evaluating correctness & consistency of large language models on cross-domain multi-file project problems, 2025. URL <https://arxiv.org/abs/2502.00226>.
- 1132** Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*, 2023.

- 1133 Chengzhi Xu, Yuyang Wang, Lai Wei, Lichao Sun, and Weiran Huang. Improved iterative refinement for chart-to-code generation via structured instruction. *arXiv preprint arXiv:2506.14837*, 2025.
- 1134 Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Transactions on Software Engineering*, 2025.
- 1135 Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. Ckgfuzzer: Llm-based fuzz driver generation enhanced by code knowledge graph, 2024. URL <https://arxiv.org/abs/2411.11532>.
- 1136 Hongshen Xu, Zichen Zhu, Lei Pan, Zihan Wang, Su Zhu, Da Ma, Ruisheng Cao, Lu Chen, and Kai Yu. Reducing tool hallucination via reliability alignment. In *Forty-second International Conference on Machine Learning*, 2025. URL <https://openreview.net/forum?id=WeOLZmDXyA>.
- 1137 Jiacheng Xu, Bo Pang, Jin Qu, Hiroaki Hayashi, Caiming Xiong, and Yingbo Zhou. CLOVER: A Test Case Generation Benchmark with Coverage, Long-Context, and Verification, February 2025. URL <http://arxiv.org/abs/2502.08806>. arXiv:2502.08806 [cs].
- 1138 Kai Xu, YiWei Mao, XinYi Guan, and ZiLong Feng. Web-bench: A llm code benchmark based on web standards and frameworks. *arXiv preprint arXiv:2505.07473*, 2025.
- 1139 Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica S Lam. Grounding open-domain instructions to automate web support tasks. *arXiv preprint arXiv:2103.16057*, 2021.
- 1140 Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution. *arXiv preprint arXiv:2408.13001*, 2024.
- 1141 Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. Commit message generation for source code changes. In *IJCAI*, 2019.
- 1142 Shiyi Xu, Yiwen Hu, Yingqian Min, Zhipeng Chen, Wayne Xin Zhao, and Ji-Rong Wen. Icpc-eval: Probing the frontiers of llm reasoning with competitive programming contests, 2025. URL <https://arxiv.org/abs/2506.04894>.
- 1143 Tongtong Xu, Liushan Chen, Yu Pei, Tian Zhang, Minxue Pan, and Carlo A Furia. Restore: Retrospective fault localization enhancing automated program repair. *IEEE Transactions on Software Engineering*, 48(1):309–326, 2020.
- 1144 Wei Xu and Alexander I. Rudnicky. Task-based dialog management using an agenda. In *ANLP-NAACL 2000 Workshop: Conversational Systems*, 2000. URL <https://aclanthology.org/W00-0309/>.
- 1145 Wendong Xu, Jing Xiong, Chenyang Zhao, Qiujiang Chen, Haoran Wang, Hui Shen, Zhongwei Wan, Jianbo Dai, Taiqiang Wu, He Xiao, Chaofan Tao, Z. Morley Mao, Ying Sheng, Zhijiang Guo, Hongxia Yang, Bei Yu, Lingpeng Kong, Quanquan Gu, and Ngai Wong. Swingarena: Competitive programming arena for long-context github issue solving, 2025. URL <https://arxiv.org/abs/2505.23932>.

- 1146 Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.
- 1147 Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. Lmpa: Improving decompilation by synergy of large language model and program analysis. *CoRR*, abs/2306.02546, 2023.
- 1148 Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. Prosecc: Fortifying code llms with proactive security alignment. *arXiv preprint arXiv:2411.12882*, 2024.
- 1149 Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*, 2024.
- 1150 Yuzhuang Xu, Shuo Wang, Peng Li, Fuwen Luo, Xiaolong Wang, Weidong Liu, and Yang Liu. Exploring Large Language Models for Communication Games: An Empirical Study on Werewolf. *arXiv e-prints*, art. arXiv:2309.04658, September 2023. doi: 10.48550/arXiv.2309.04658.
- 1151 Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kod-code: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*, 2025.
- 1152 Zhengzhuo Xu, Sinan Du, Yiyan Qi, Chengjin Xu, Chun Yuan, and Jian Guo. Chartbench: A benchmark for complex visual reasoning in charts, 2024. URL <https://arxiv.org/abs/2312.15915>.
- 1153 Min Xue, Artur Andrzejak, and Marla Leuther. An interpretable error correction method for enhancing code-to-code translation. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=fVxIEHGNVT>.
- 1154 Yifan Xue et al. Classeeval-t: Cross-language class-level benchmark for code generation. In *ICSE*, 2024.
- 1155 Ankit Yadav, Himanshu Beniwal, and Mayank Singh. Pythonsaga: Redefining the benchmark to evaluate code generating llms, 2024. URL <https://arxiv.org/abs/2401.03855>.
- 1156 Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th annual computer security applications conference*, pages 359–368, 2012.
- 1157 Chuanhao Yan, Fengdi Che, Xuhan Huang, Xu Xu, Xin Li, Yizhi Li, Xingwei Qu, Jingzhe Shi, Zhuangzhuang He, Chenghua Lin, Yaodong Yang, Binhang Yuan, Hang Zhao, Yu Qiao, Bowen Zhou, and Jie Fu. Re:form – reducing human priors in scalable formal software verification with rl in llms: A preliminary study on dafny, 2025. URL <https://arxiv.org/abs/2507.16331>.
- 1158 Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 5067–5089. Association for Computational Linguistics, 2023. URL <https://aclanthology.org/2023.findings-emnlp.337>.

- 1159** Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and Shuguang Deng. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation, 2024. URL <https://arxiv.org/abs/2311.08588>.
- 1160** Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.
- 1161** Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. Vert: Verified equivalent rust transpilation with few-shot learning. *arXiv preprint arXiv:2404.18852*, 26, 2024.
- 1162** An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024. URL <https://arxiv.org/abs/2407.10671>.
- 1163** An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- 1164** Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. Morepair: Teaching llms to repair code via multi-objective fine-tuning. *arXiv preprint arXiv:2404.12636*, 2024.
- 1165** Boyang Yang, Haoye Tian, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, and Bach Le. Enhancing repository-level software repair via repository-aware knowledge graphs. *arXiv preprint arXiv:2503.21710*, 2025.
- 1166** Bruce Yang, Xinfeng He, Huan Gao, Yifan Cao, Xiaofan Li, and David Hsu. Codeagents: A token-efficient framework for codified multi-agent reasoning in llms. *arXiv preprint arXiv:2507.03254*, 2025.
- 1167** Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. Enhancing llm-based test generation for hard-to-cover branches via program analysis. *arXiv*, abs/2404.04966, 2025. URL <https://api.semanticscholar.org/CorpusID:270928236>.
- 1168** Cheng Yang, Chufan Shi, Yixin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, et al. Chartmimic: Evaluating lmm’s cross-modal reasoning capability via chart-to-code generation. *arXiv preprint arXiv:2406.09961*, 2024.
- 1169** Cheng Yang, Chufan Shi, Yixin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran XU, Xinyu Zhu, Siheng Li, Yuxiang Zhang, Gongye Liu, Xiaomei Nie, Deng Cai, and Yujiu Yang. Chartmimic: Evaluating LMM’s cross-modal reasoning capability via chart-to-code generation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=sGpCzsfd1K>.

- 1170 Chengran Yang, Hong Jin Kang, Jieke Shi, and David Lo. Acecode: A reinforcement learning framework for aligning code efficiency and correctness in code language models, 2024. URL <https://arxiv.org/abs/2412.17264>.
- 1171 Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):709–735, October 2024. ISSN 2475-1421. doi: 10.1145/3689736. URL <http://dx.doi.org/10.1145/3689736>.
- 1172 Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and Julian McAuley. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms, 2025. URL <https://arxiv.org/abs/2502.19411>.
- 1173 Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. DocAgent: A Multi-Agent System for Automated Code Documentation Generation. *arXiv e-prints*, art. arXiv:2504.08725, April 2025. doi: 10.48550/arXiv.2504.08725.
- 1174 Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Transactions on Software Engineering*, 2024.
- 1175 Guang Yang, Wei Zheng, Xiang Chen, Dong Liang, Peng Hu, Yukui Yang, Shaohua Peng, Zhenghan Li, Jiahui Feng, Xiao Wei, et al. Large language model for verilog code generation: Literature review and the road ahead. 2025.
- 1176 Jian Yang, Shuming Ma, Haoyang Huang, Dongdong Zhang, Li Dong, Shaohan Huang, Alexandre Muzio, Saksham Singhal, Hany Hassan, Xia Song, and Furu Wei. Multilingual machine translation systems from microsoft for WMT21 shared task. In Loïc Barrault, Ondřej Bojar, Fethi Bougares, Rajen Chatterjee, Marta R. Costa-jussà, Christian Federmann, Mark Fishel, Alexander Fraser, Markus Freitag, Yvette Graham, Roman Grundkiewicz, Paco Guzman, Barry Haddow, Matthias Huck, Antonio Jimeno-Yepes, Philipp Koehn, Tom Kocmi, André F. T. Martins, Makoto Morishita, and Christof Monz, editors, *Proceedings of the Sixth Conference on Machine Translation, WMT@EMNLP 2021, Online Event, November 10-11, 2021*, pages 446–455. Association for Computational Linguistics, 2021. URL <https://aclanthology.org/2021.wmt-1.54>.
- 1177 Jian Yang, Jiaxi Yang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Binyuan Hui, and Junyang Lin. Evaluating and aligning codellms on human preference. *arXiv preprint arXiv:2412.05210*, 2024.
- 1178 Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiya Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, Binyuan Hui, and Junyang Lin. Execrepobench: Multi-level executable code completion evaluation, 2024. URL <https://arxiv.org/abs/2412.11990>.
- 1179 Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiya Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, et al. Execrepobench: Multi-level executable code completion evaluation. *arXiv preprint arXiv:2412.11990*, 2024.
- 1180 Jian Yang, Wei Zhang, Shukai Liu, Linzheng Chai, Yingshui Tan, Jiaheng Liu, Ge Zhang, Wangchunshu Zhou, Guanglin Niu, Zhoujun Li, et al. Ifevalcode: Controlled code generation. *arXiv preprint arXiv:2507.22462*, 2025.

- 1181 Jian Yang, Wei Zhang, Yibo Miao, Shanghaoran Quan, Zhenhe Wu, Qiyao Peng, Liqun Yang, Tianyu Liu, Zeyu Cui, Binyuan Hui, and Junyang Lin. Qwen2.5-xcoder: Multi-agent collaboration for multilingual code instruction tuning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 13121–13131. Association for Computational Linguistics, 2025. URL <https://aclanthology.org/2025.acl-long.642/>.
- 1182 Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*, 2023.
- 1183 John Yang, OpenAI, et al. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *arXiv preprint arXiv:2306.14898*, 2023.
- 1184 John Yang, Carlos E. Jimenez, Alexander Wettig, et al. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.
- 1185 John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024. URL <https://arxiv.org/abs/2410.03859>.
- 1186 John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=riTiq3i21b>.
- 1187 John Yang, Kilian Leret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025.
- 1188 John Yang, Kilian Leret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL <https://arxiv.org/abs/2504.21798>.
- 1189 Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik Chaudhari, George Karypis, and Huzeifa Rangwala. Agentoccam: A simple yet strong baseline for llm-based web agents. *arXiv preprint arXiv:2410.13825*, 2024.
- 1190 L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen. An empirical study of unit test generation with large language models. *CoRR*, abs/2406.18181, 2024. URL <https://arxiv.org/abs/2406.18181>.
- 1191 Lei Yang, Renren Jin, Ling Shi, Jianxiang Peng, Yue Chen, and Deyi Xiong. Probench: Benchmarking large language models in competitive programming, 2025. URL <https://arxiv.org/abs/2502.20868>.
- 1192 Liqun Yang, Chaoren Wei, Jian Yang, Jinxin Ma, Hongcheng Guo, Long Cheng, and Zhoujun Li. Seq2seq-afl: Fuzzing via sequence-to-sequence model. *Int. J. Mach. Learn. Cybern.*, 15(10):4403–4421, 2024. doi: 10.1007/S13042-024-02153-Z. URL <https://doi.org/10.1007/s13042-024-02153-z>.

- 1193 Qinwei Yang, Xueqing Liu, Yan Zeng, Ruocheng Guo, Yang Liu, and Peng Wu. Learning the optimal policy for balancing short-term and long-term rewards. *Advances in Neural Information Processing Systems*, 37:36514–36540, 2024.
- 1194 Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. Gpt4tools: Teaching large language model to use tools via self-instruction, 2023. URL <https://arxiv.org/abs/2305.18752>.
- 1195 Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E. Gonzalez, and Ion Stoica. Rethinking benchmark and contamination for language models with rephrased samples, 2023. URL <https://arxiv.org/abs/2311.04850>.
- 1196 Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2025. URL <https://arxiv.org/abs/2412.06464>.
- 1197 Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length, 2025. URL <https://arxiv.org/abs/2406.06484>.
- 1198 Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. Coast: Enhancing the code debugging ability of llms through communicative agent based data synthesis, 2025. URL <https://arxiv.org/abs/2408.05006>.
- 1199 Wenhan Yang, Spencer Stice, Ali Payani, and Baharan Mirzasoleiman. Bootstrapping llm robustness for vlm safety via reducing the pretraining modality gap. *arXiv preprint arXiv:2505.24208*, 2025.
- 1200 Xudong Yang, Yifan Wu, Yizhang Zhu, Nan Tang, and Yuyu Luo. Askchart: Universal chart understanding through textual enhancement, 2024. URL <https://arxiv.org/abs/2412.19146>.
- 1201 Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. Mm-react: Prompting chatgpt for multimodal reasoning and action, 2023. URL <https://arxiv.org/abs/2303.11381>.
- 1202 Zheyuan Yang, Zexi Kuang, Xue Xia, and Yilun Zhao. Can LLMs Generate High-Quality Test Cases for Algorithm Problems? TestCase-Eval: A Systematic Evaluation of Fault Coverage and Exposure, June 2025. URL <http://arxiv.org/abs/2506.12278>. arXiv:2506.12278 [cs].
- 1203 Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *arXiv preprint arXiv:2403.07506*, 2024.
- 1204 Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, et al. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045*, 2025.
- 1205 Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.
- 1206 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2022.

- 1207** Shunyu Yao, Howard Chen, Austin W. Hanjie, Runzhe Yang, and Karthik Narasimhan. COLLIE: Systematic Construction of Constrained Text Generation Tasks. *arXiv e-prints*, art. arXiv:2307.08689, July 2023. doi: 10.48550/arXiv.2307.08689.
- 1208** Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL <https://arxiv.org/abs/2305.10601>.
- 1209** Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
- 1210** Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. taubench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.
- 1211** Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.
- 1212** Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. Coacor: Code annotation for code retrieval with reinforcement learning. In *The world wide web conference*, pages 2203–2214, 2019.
- 1213** He Ye and Martin Monperrus. Iter: Iterative neural repair for multi-location patches. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13, 2024.
- 1214** He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.
- 1215** He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering*, pages 1506–1518, 2022.
- 1216** Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. Limo: Less is more for reasoning. *arXiv preprint arXiv:2502.03387*, 2025.
- 1217** Xin Yin, Chao Ni, Tien N. Nguyen, Shaohua Wang, and Xiaohu Yang. Rectifier: Code translation with corrector via llms, 2024. URL <https://arxiv.org/abs/2407.07472>.
- 1218** Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286, 2024.
- 1219** Zheng-Xin Yong, Cristina Menghini, and Stephen H. Bach. Low-resource languages jailbreak gpt-4, 2023.
- 1220** Steve Young, Milica Gašić, Blaise Thomson, and Jason D. Williams. Pomdp-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101(5):1160–1179, 2013. doi: 10.1109/JPROC.2012.2225812.

- 1221** Dongjun Yu, Xiao Yan, Zhenrui Li, Jipeng Xiao, Haochuan He, Yongda Yu, Hao Zhang, Guoping Rong, and Xiaobo Huang. Synthcoder: A synthetical strategy to tune llms for code completion, 2025. URL <https://arxiv.org/abs/2508.15495>.
- 1222** Dongjun Yu, Xiao Yan, Zhenrui Li, Jipeng Xiao, Haochuan He, Yongda Yu, Hao Zhang, Guoping Rong, and Xiaobo Huang. Synthcoder: A synthetical strategy to tune llms for code completion. *arXiv preprint arXiv:2508.15495*, 2025.
- 1223** Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.
- 1224** Haonan Yu, Zihan Xu, Wei Tan, Yixuan Du, Kun Zhang, and Zhaoran Li. Towards safe reinforcement learning with a safety editor policy. *arXiv preprint arXiv:2205.15283*, 2022.
- 1225** Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.
- 1226** Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Avnish Narayan, Hayden Shively, Adithya Bellathur, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning. *arXiv e-prints*, art. arXiv:1910.10897, October 2019. doi: 10.48550/arXiv.1910.10897.
- 1227** Weichen Yu, Ravi Mangal, Terry Zhuo, Matt Fredrikson, and Corina S Pasareanu. A mixture of linear corrections generates secure code. *arXiv preprint arXiv:2507.09508*, 2025.
- 1228** Yongda Yu, Guoping Rong, Haifeng Shen, He Zhang, Dong Shao, Min Wang, Zhao Wei, Yong Xu, and Juhong Wang. Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. *ACM transactions on software engineering and methodology*, 34(1):1–26, 2024.
- 1229** Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. WaveCoder: Widespread And Versatile Enhancement For Code Large Language Models By Instruction Tuning, 2024.
- 1230** Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation, 2024. URL <https://arxiv.org/abs/2412.21199>.
- 1231** Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, Zhenghao Liu, Bowen Zhou, Hao Peng, Zhiyuan Liu, and Maosong Sun. Advancing llm reasoning generalists with preference trees, 2024. URL <https://arxiv.org/abs/2404.02078>.
- 1232** Mingyue Yuan, Jieshan Chen, Yongquan Hu, Sidong Feng, Mulong Xie, Gelareh Mohammadi, Zhenchang Xing, and Aaron Quigley. Towards human-ai synergy in ui design: Enhancing multi-agent based ui generation with intent clarification and alignment. *arXiv preprint arXiv:2412.20071*, 2024.
- 1233** Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. Easytool: Enhancing llm-based agents with concise tool instruction, 2024. URL <https://arxiv.org/abs/2401.06201>.

- 1234 Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. Circle: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 678–690, 2022.
- 1235 Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, et al. debug-gym: A text-based environment for interactive debugging. *arXiv preprint arXiv:2503.21557*, 2025.
- 1236 Yufeng Yuan, Yu Yue, Ruofei Zhu, Tiantian Fan, and Lin Yan. What’s behind ppo’s collapse in long-cot? value optimization holds the secret. *arXiv preprint arXiv:2503.01491*, 2025.
- 1237 Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models, 2023. URL <https://arxiv.org/abs/2308.01825>.
- 1238 Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. Semantic alignment-enhanced code translation via an llm-based multi-agent system. *arXiv preprint arXiv: 2409.19894*, 2024.
- 1239 Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation, 2024. URL <https://arxiv.org/abs/2305.04207>.
- 1240 Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. ZebraLogic: On the Scaling Limits of LLMs for Logical Reasoning. *arXiv e-prints*, art. arXiv:2502.01100, February 2025. doi: 10.48550/arXiv.2502.01100.
- 1241 Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, et al. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*, 2025.
- 1242 Sukmin Yun, Rusuru Thushara, Mohammad Bhat, Yongxin Wang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, Haonan Li, Preslav Nakov, et al. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms. *Advances in neural information processing systems*, 37:112134–112157, 2024.
- 1243 Abhay Zala, Han Lin, Jaemin Cho, and Mohit Bansal. Diagrammergpt: Generating open-domain, open-platform diagrams via llm planning. *arXiv preprint arXiv:2310.12128*, 2023.
- 1244 Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: Continual pre-training on sketches for library-oriented code generation. In *The 2022 International Joint Conference on Artificial Intelligence*, 2022.
- 1245 Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. Swe-bench-java: A github issue resolving benchmark for java, 2024. URL <https://arxiv.org/abs/2408.14354>.

- 1246** Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, Zhiguang Yang, Yongji Wang, Qianxiang Wang, and Lizhen Cui. Codes: Natural language to code repository via multi-layer sketch. *CoRR abs/2403.16443*, 2024. URL <https://arxiv.org/abs/2403.16443>.
- 1247** Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL <https://arxiv.org/abs/2504.02605>.
- 1248** Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifan An, Yilin Niu, Yuanhao Wen, Yushi Bai, Zhengxiao Du, Zihan Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, Bowen Xu, Can Huang, Casey Zhao, Changpeng Cai, Chao Yu, Chen Li, Chendi Ge, Chenghua Huang, Chenhui Zhang, Chenxi Xu, Chenzheng Zhu, Chuang Li, Congfeng Yin, Daoyan Lin, Dayong Yang, Dazhi Jiang, Ding Ai, Erle Zhu, Fei Wang, Gengzheng Pan, Guo Wang, Hailong Sun, Haitao Li, Haiyang Li, Haiyi Hu, Hanyu Zhang, Hao Peng, Hao Tai, Haoke Zhang, Haoran Wang, Haoyu Yang, He Liu, He Zhao, Hongwei Liu, Hongxi Yan, Huan Liu, Huilong Chen, Ji Li, Jiajing Zhao, Jiamin Ren, Jian Jiao, Jiani Zhao, Jianyang Yan, Jiaqi Wang, Jiayi Gui, Jiayue Zhao, Jie Liu, Jijie Li, Jing Li, Jing Lu, Jingsen Wang, Jingwei Yuan, Jingxuan Li, Jingzhao Du, Jinhua Du, Jinxin Liu, Junkai Zhi, Junli Gao, Ke Wang, Lekang Yang, Liang Xu, Lin Fan, Lindong Wu, Lintao Ding, Lu Wang, Man Zhang, Minghao Li, Minghuan Xu, Mingming Zhao, Mingshu Zhai, Pengfan Du, Qian Dong, Shangde Lei, Shangqing Tu, Shangtong Yang, Shaoyou Lu, Shijie Li, Shuang Li, Shuang-Li, Shuxun Yang, Sibo Yi, Tianshu Yu, Wei Tian, Weihan Wang, Wenbo Yu, Weng Lam Tam, Wenjie Liang, Wentao Liu, Xiao Wang, Xiaohan Jia, Xiaotao Gu, Xiaoying Ling, Xin Wang, Xing Fan, Xingru Pan, Xinyuan Zhang, Xinze Zhang, Xiuqing Fu, Xunkai Zhang, Yabo Xu, Yandong Wu, Yida Lu, Yidong Wang, Yilin Zhou, Yiming Pan, Ying Zhang, Yingli Wang, Yingru Li, Yinpei Su, Yipeng Geng, Yitong Zhu, Yongkun Yang, Yuhang Li, Yuhao Wu, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yuxuan Zhang, Zezhen Liu, Zhen Yang, Zhengda Zhou, Zhongpei Qiao, Zhuoer Feng, Zhuorui Liu, Zichen Zhang, Zihan Wang, Zijun Yao, Zikang Wang, Ziqiang Liu, Ziwei Chai, Zixuan Li, Zuodong Zhao, Wenguang Chen, Jidong Zhai, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL <https://arxiv.org/abs/2508.06471>.
- 1249** Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, Zhang-Wei Hong, et al. Satori-swe: Evolutionary test-time scaling for sample-efficient software engineering. *arXiv preprint arXiv:2505.23604*, 2025.
- 1250** Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhua Chen. Acecoder: Acing coder rl via automated test-case synthesis. *arXiv preprint arXiv:2408.06733*, 2024.
- 1251** Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhua Chen. Acecoder: Acing coder rl via automated test-case synthesis, 2025a. *arXiv preprint arXiv:2502.01718*, 2025. URL <https://arxiv.org/abs/2502.01718>.

- 1252 Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and Yahui Zhou. Skywork-swe: Unveiling data scaling laws for software engineering in llms, 2025.
- 1253 Q-F Zeng, F Liu, Y Fan, Z Guo, and J. M. Zhang. Shieldagent: Shielding agents via verifiable safety policy reasoning. *arXiv preprint arXiv:2405.19253*, 2024.
- 1254 Sihang Zeng, Kai Tian, Kaiyan Zhang, Yuru Wang, Junqi Gao, Runze Liu, Sa Yang, Jingxuan Li, Xinwei Long, Jiaheng Ma, et al. Reviewrl: Towards automated scientific review with rl. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 16942–16954, 2025.
- 1255 Jirong Zha, Yuxuan Fan, Xiao Yang, Chen Gao, and Xinlei Chen. How to Enable LLM with 3D Capacity? A Survey of Spatial Reasoning in LLM. *arXiv e-prints*, art. arXiv:2504.05786, April 2025. doi: 10.48550/arXiv.2504.05786.
- 1256 Jiaming Zhan, Yantao Liu, Yiran Liu, Fuchun Peng, Zhaofeng He, and Wenbo Guo. Safe lora: the silver lining of reducing safety risks when fine-tuning large language models, 2024.
- 1257 Zizheng Zhan, Ken Deng, Xiaojiang Zhang, Jinghui Wang, Huaixi Tang, Zhiyi Lai, Haoyang Huang, Wen Xiang, Kun Wu, Wenhao Zhuang, et al. Kat-coder technical report. *arXiv preprint arXiv:2510.18779*, 2025.
- 1258 Alexander Zhang, Marcus Dong, Jiaheng Liu, Wei Zhang, Yejie Wang, Jian Yang, Ge Zhang, Tianyu Liu, Zhongyuan Peng, Yinghui Tan, Yuanxing Zhang, Zhexu Wang, Weixun Wang, Yancheng He, Ken Deng, Wangchunshu Zhou, Wenhao Huang, and Zhaoxiang Zhang. Codecriticbench: A holistic code critique benchmark for large language models, 2025. URL <https://arxiv.org/abs/2502.16614>.
- 1259 Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Shihui Hu, Dengpeng Wu, Guanhua Huang, Kejiao Li, Qi Yi, et al. Artifactsbench: Bridging the visual-interactive gap in llm code generation evaluation. *arXiv preprint arXiv:2507.04952*, 2025.
- 1260 Chenyuan Zhang, Cristian Rojas Cardenas, Hamid Rezatofighi, Mor Vered, and Buser Say. Probabilistic active goal recognition. *arXiv preprint arXiv:2507.21846*, 2025.
- 1261 D Zhang, J Yang, T Zhang, R Wu, and Z Li. Human-in-the-loop imitation learning for advanced robotics. *Nature Machine Intelligence*, 2024.
- 1262 Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale. In *The Twelfth International Conference on Learning Representations*, 2024.
- 1263 Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023. URL <https://arxiv.org/abs/2303.12570>.
- 1264 Fengji Zhang, Linquan Wu, Huiyu Bai, Guancheng Lin, Xiao Li, Xiao Yu, Yue Wang, Bei Chen, and Jacky Keung. Humaneval-v: Benchmarking high-level visual reasoning with complex diagrams in coding tasks. *arXiv preprint arXiv:2410.12381*, 2024.
- 1265 Hang Zhang, Yanxin Shen, Lun Wang, Chuanqi Shi, Shaoshuai Du, Yiyi Tao, and Yixian Shen. Comparative analysis of large language models for context-aware code completion using safim framework, 2025. URL <https://arxiv.org/abs/2502.15243>.

- 1266** J. Zhang and et al. Hyperagent: Generalist multi-agent system for repository-level code generation. *arXiv preprint arXiv:2406.11223*, 2024. URL <https://arxiv.org/abs/2406.11223>.
- 1267** Jie Zhang, Dongrui Liu, Chen Qian, Linfeng Zhang, Yong Liu, Yu Qiao, and Jing Shao. REEF: representation encoding fingerprints for large language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL <https://openreview.net/forum?id=SnDmPkOJ0T>.
- 1268** Jing Zhang, Jian Zhang, and Zhao Jiang. Guard: Dual-agent based backdoor defense on chain-of-thought in neural code generation. *arXiv preprint arXiv:2405.16723*, 2024.
- 1269** Kaiyan Zhang, Yuxin Zuo, Bingxiang He, Youbang Sun, Runze Liu, Che Jiang, Yuchen Fan, Kai Tian, Guoli Jia, Pengfei Li, et al. A survey of reinforcement learning for large reasoning models. *arXiv preprint arXiv:2509.08827*, 2025.
- 1270** Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. Codedpo: Aligning code models with self generated and verified source code. *arXiv preprint arXiv:2410.05605*, 2024.
- 1271** Kechi Zhang, Ge Li, Jia Li, Yihong Dong, and Zhi Jin. Focused-dpo: Enhancing code generation through focused preference optimization on error-prone points. *arXiv preprint arXiv:2502.11475*, 2025.
- 1272** Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Swe-flow: Synthesizing software engineering data in a test-driven manner. *arXiv preprint arXiv:2506.09003*, 2025.
- 1273** Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Jun-hao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.
- 1274** Linghao Zhang, Junhao Wang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Jiaheng Wen, Chengxing Xie, Maoquan Wang, Yufan Huang, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Di-bench: Benchmarking large language models on dependency inference with testable repositories at scale, 2025. URL <https://arxiv.org/abs/2501.13699>.
- 1275** Linhao Zhang, Daoguang Zan, Quanshun Yang, Zhirong Huang, Dong Chen, Bo Shen, Tianyu Liu, Yongshun Gong, Pengjie Huang, Xudong Lu, et al. Codev: Issue resolving with visual data. *arXiv preprint arXiv:2412.17315*, 2024.
- 1276** Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 535–547. IEEE, 2023.
- 1277** Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. TestBench: Evaluating Class-Level Test Case Generation Capability of Large Language Models, September 2024. URL <http://arxiv.org/abs/2409.17561>. arXiv:2409.17561 [cs].

- 1278 Shenglin Zhang, Weibin Meng, Jiahao Bu, Sen Yang, Ying Liu, Dan Pei, Jun Xu, Yu Chen, Hui Dong, Xianping Qu, and Lei Song. Syslog processing for switch failure diagnosis and prediction in datacenter networks. In *25th IEEE/ACM International Symposium on Quality of Service, IWQoS 2017, Vilanova i la Geltrú, Spain, June 14-16, 2017*, pages 1–10. IEEE, 2017.
- 1279 Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. Instruction tuning for large language models: A survey, 2024. URL <https://arxiv.org/abs/2308.10792>.
- 1280 Simiao Zhang, Jiaping Wang, Guoliang Dong, Jun Sun, Yueling Zhang, and Geguang Pu. Experimenting a new programming practice with llms. *CoRR abs/2401.01062*, 2024. URL <https://arxiv.org/abs/2401.01062>.
- 1281 Wei Zhang, Jack Yang, Renshuai Tao, Lingzheng Chai, Shawn Guo, Jiajun Wu, Xiaoming Chen, Ganqu Cui, Ning Ding, Xander Xu, Hu Wei, and Bowen Zhou. V-gamegym: Visual game generation for code large language models, 2025. URL <https://arxiv.org/abs/2509.20136>.
- 1282 Wei Zhang, Jian Yang, Jiaxi Yang, Ya Wang, Zhoujun Li, Zeyu Cui, Binyuan Hui, and Junyang Lin. Turning the tide: Repository-based code reflection, 2025. URL <https://arxiv.org/abs/2507.09866>.
- 1283 Xinlu Zhang, Zhiyu Zoey Chen, Xi Ye, Xianjun Yang, Lichang Chen, William Yang Wang, and Linda Ruth Petzold. Unveiling the impact of coding data instruction fine-tuning on large language models reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25949–25957, 2025.
- 1284 Yifei Zhang, Huan Li, Zhiqiang Zhang, et al. Deep learning for code generation: A survey. *arXiv preprint arXiv:2302.06791*, 2023. URL <https://arxiv.org/abs/2302.06791>.
- 1285 Yiming Zhang, Zhengzi Liu, Xiang Chen, et al. Practices and challenges of using GitHub Copilot: An empirical study. In *Proceedings of the 45th International Conference on Software Engineering*, pages 2435–2447, 2023.
- 1286 Yinger Zhang, Hui Cai, Xeirui Song, Yicheng Chen, Rui Sun, and Jing Zheng. Reverse chain: A generic-rule for llms to master multi-api planning, 2024. URL <https://arxiv.org/abs/2310.04474>.
- 1287 Yue Zhang et al. Vulrepair: Vulnerability patching with llms. <https://github.com/VulRepair/vulrepair>, 2024.
- 1288 Yueke Zhang, Yifan Zhang, Kevin Leach, and Yu Huang. Formalgrad: Integrating formal methods with gradient-based llm refinement. *arXiv preprint arXiv:2508.10059*, 2025.
- 1289 Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.
- 1290 Yuxiang Zhang, Jing Chen, Junjie Wang, Yaxin Liu, Cheng Yang, Chufan Shi, Xinyu Zhu, Zihao Lin, Hanwen Wan, Yujiu Yang, Tetsuya Sakai, Tian Feng, and Hayato Yamana. ToolBeHonest: A multi-level hallucination diagnostic benchmark for tool-augmented large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11388–11422, Miami, Florida, USA, November 2024. Association for Computational

Linguistics. doi: 10.18653/v1/2024.emnlp-main.637. URL <https://aclanthology.org/2024.emnlp-main.637/>.

- 1291 Zhaoxu Zhang, Robert Winn, Yu Zhao, Tingting Yu, and William GJ Halfond. Automatically reproducing android bug reports using natural language processing and reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 411–422, 2023.
- 1292 Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*, 2023.
- 1293 Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. Unifying the perspectives of NLP and software engineering: A survey on language models for code. *Trans. Mach. Learn. Res.*, 2024, 2024.
- 1294 Ziyin Zhang, Chaoyu Zhao, Bingchang Chen, et al. Large language models for code: A survey. *arXiv preprint arXiv:2410.01241*, 2024.
- 1295 Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409, 2024. doi: 10.48550/ARXIV.2406.11409. URL <https://doi.org/10.48550/arXiv.2406.11409>.
- 1296 Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. Understanding programs by exploiting (fuzzing) test cases. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 10667–10679. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-ACL.678. URL <https://doi.org/10.18653/v1/2023.findings-acl.678>.
- 1297 Junjie Zhao, Xiang Chen, Guang Yang, and Yiheng Shen. Automatic smart contract comment generation via large language models and in-context learning. *Information and Software Technology*, 168:107405, 2024. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2024.107405>. URL <https://www.sciencedirect.com/science/article/pii/S0950584924000107>.
- 1298 Ruochen Zhao, Xingxuan Li, Yew Ken Chia, Bosheng Ding, and Lidong Bing. Can chatgpt-like generative models guarantee factual accuracy? on the mistakes of new generation search engines. *arXiv preprint arXiv:2304.11076*, 2023.
- 1299 Sebastian Zhao, Alan Zhu, Hussein Mozannar, David Sontag, Ameet Talwalkar, and Valerie Chen. Codinggenie: A proactive llm-powered programming assistant. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 1168–1172, 2025.
- 1300 W. Zhao and et al. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024. URL <https://arxiv.org/abs/2412.01769>.

- 1301 Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2025. URL <https://arxiv.org/abs/2303.18223>.
- 1302 Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024. URL <https://arxiv.org/abs/2412.01769>.
- 1303 Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Zhiyuan Liu, and Maosong Sun. Chartcoder: Advancing multimodal large language model for chart-to-code generation. *arXiv preprint arXiv:2501.06598*, 2025.
- 1304 Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023. URL <https://arxiv.org/abs/2304.11277>.
- 1305 Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023.
- 1306 Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. Repair: Automated program repair with process-based feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 16415–16429, 2024.
- 1307 Terry Yue Zhaou, Qian Liu, Zijian Wang, Wasi U Ahmad, Binuian Hui, and Loubna Ben Allal. Nlp+ code: Code intelligence in language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, pages 9–11, 2025.
- 1308 Chiaming Zheng, Cao Yuan, Yang Song, Pin-Yu Chen, and Sijia Liu. Seal: Safety-enhanced aligned llm fine-tuning via bilevel data selection. *arXiv preprint arXiv:2405.18835*, 2024.
- 1309 Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation, 2025. URL <https://arxiv.org/abs/2406.06918>.
- 1310 Qinkai Zheng, Xiao Xia, Xu Zou, et al. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X. *KDD*, 2023.
- 1311 Qinkai Zheng, Tianyu Zhu, Boxuan Lin, et al. CodeGeeX4-ALL-9B: Open multilingual code generation model. *arXiv preprint arXiv:2407.10845*, 2024.
- 1312 Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhui Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*, 2024.
- 1313 Tianyu Zheng, Tianshun Xing, Qingshui Gu, Taoran Liang, Xingwei Qu, Xin Zhou, Yizhi Li, Zhoufutu Wen, Chenghua Lin, Wenhao Huang, Qian Liu, Ge Zhang, and Zejun Ma. First return, entropy-eliciting explore, 2025. URL <https://arxiv.org/abs/2507.07017>.

- 1314 Xin Zheng, Jie Lou, Boxi Cao, Xueru Wen, Yuqiu Ji, Hongyu Lin, Yaojie Lu, Xianpei Han, Debinger Zhang, and Le Sun. Critic-cot: Boosting the reasoning abilities of large language model via chain-of-thoughts critic. *arXiv preprint arXiv:2408.16326*, 2024.
- 1315 Yanzhao Zheng, Haibin Wang, Baohua Dong, Xingjun Wang, and Changshan Li. HIE-SQL: history information enhanced network for context-dependent text-to-sql semantic parsing. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2997–3007. Association for Computational Linguistics, 2022.
- 1316 Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyuan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In Yixin Cao, Yang Feng, and Deyi Xiong, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-demos.38. URL <https://aclanthology.org/2024.acl-demos.38/>.
- 1317 Yongwei Zheng, Y-t Li, P Wang, T Shi, Y Liu, G Pu, and J Ma. Sandboxeval: Towards securing test environment for untrusted code. *arXiv preprint arXiv:2405.08375*, 2024.
- 1318 Zihan Zheng, Zerui Cheng, Zeyu Shen, Shang Zhou, Kaiyuan Liu, Hansen He, Dongruixuan Li, Stanley Wei, Hangyi Hao, Jianzhu Yao, Peiyao Sheng, Zixuan Wang, Wenhao Chai, Aleksandra Korolova, Peter Henderson, Sanjeev Arora, Pramod Viswanath, Jingbo Shang, and Saining Xie. Livecodebench pro: How do olympiad medalists judge llms in competitive programming?, 2025. URL <https://arxiv.org/abs/2506.11928>.
- 1319 Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 851–870. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.49. URL <https://doi.org/10.18653/v1/2024.findings-acl.49>.
- 1320 Zhiyuan Zhong, Sinan Wang, Hailong Wang, Shaojin Wen, Hao Guan, Yida Tao, and Yepang Liu. Advancing bug detection in fastjson2 with large language models driven unit test generation. *arXiv preprint arXiv:2410.09414*, 2024.
- 1321 Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*, 2023.
- 1322 Pei Zhou, Jay Pujara, Xiang Ren, Xinyun Chen, Heng-Tze Cheng, Quoc V Le, Ed Chi, Denny Zhou, Swaroop Mishra, and Huaxiu Steven Zheng. Self-discover: Large language models self-compose reasoning structures. *Advances in Neural Information Processing Systems*, 37:126032–126058, 2024.
- 1323 Peilin Zhou, Bruce Leon, Xiang Ying, Can Zhang, Yifan Shao, Qichen Ye, Dading Chong, Zhiling Jin, Chenxuan Xie, Meng Cao, et al. Browsecemp-zh: Benchmarking web browsing ability of large language models in chinese. *arXiv preprint arXiv:2504.19314*, 2025.
- 1324 Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code, 2023. URL <https://arxiv.org/abs/2302.05527>.

- 1325 Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.
- 1326 Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL <https://arxiv.org/abs/2307.13854>.
- 1327 Tianyang Zhou, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. Llm-driven multi-step translation from c to rust using static analysis, 2025. URL <https://arxiv.org/abs/2503.12511>.
- 1328 Banghua Zhu, Evan Frick, Tianhao Wu, Hanlin Zhu, Karthik Ganesan, Wei-Lin Chiang, Jian Zhang, and Jiantao Jiao. Starling-7b: Improving helpfulness and harmlessness with rlaif. In *First Conference on Language Modeling*, 2024.
- 1329 Jinguo Zhu, Weiyun Wang, Zhe Chen, Zhaoyang Liu, Shenglong Ye, Lixin Gu, Hao Tian, Yuchen Duan, Weijie Su, Jie Shao, et al. Internvl3: Exploring advanced training and test-time recipes for open-source multimodal models. *arXiv preprint arXiv:2504.10479*, 2025.
- 1330 Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence, 2022. URL <https://arxiv.org/abs/2206.08474>, 88.
- 1331 Ming Zhu, Karthik Suresh, and Chandan K Reddy. Multilingual code snippets training for program translation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pages 11783–11790, 2022.
- 1332 Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 341–353, 2021.
- 1333 Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.
- 1334 Rongxin Zhu, Jey Han Lau, and Jianzhong Qi. Factual dialogue summarization via learning from large language models. *arXiv preprint arXiv:2406.14709*, 2024.
- 1335 Terry Yue Zhuo. Ice-score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2232–2242, 2024.
- 1336 Terry Yue Zhuo, Armel Randy Zebaze, Leandro Von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. Parameter-efficient instruction tuning code large language models: An empirical study. In *ICLR 2025 Third Workshop on Deep Learning for Code*.
- 1337 Terry Yue Zhuo, Zhuang Li, Yujin Huang, Fatemeh Shiri, Weiqing Wang, Gholamreza Haffari, and Yuan-Fang Li. On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex. *arXiv preprint arXiv:2301.12868*, 2023.

- 1338 Terry Yue Zhuo, Zhou Yang, Zhensu Sun, Yufei Wang, Li Li, Xiaoning Du, Zhenchang Xing, and David Lo. Source code data augmentation for deep learning: A survey. *arXiv preprint arXiv:2305.19915*, 2023.
- 1339 Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.
- 1340 Terry Yue Zhuo, Yujin Huang, Chunyang Chen, Xiaoning Du, and Zhenchang Xing. Bypassing guardrails: Lessons learned from red teaming chatgpt. *ACM Transactions on Software Engineering and Methodology*, 2025.
- 1341 Terry Yue Zhuo, Xiaolong Jin, Hange Liu, Juyong Jiang, Tianyang Liu, Chen Gong, Bhupesh Bishnoi, Vaisakhi Mishra, Marek Suppa, Noah Ziems, et al. Bigcodearena: Unveiling more reliable human preferences in code generation via execution. *arXiv preprint arXiv:2510.08697*, 2025.
- 1342 Terry Yue Zhuo, Dingmin Wang, Hantian Ding, Varun Kumar, and Zijian Wang. Cyber-zero: training cybersecurity agents without runtime. *arXiv preprint arXiv:2508.00910*, 2025.
- 1343 Terry Yue Zhuo, Dingmin Wang, Hantian Ding, Varun Kumar, and Zijian Wang. Training language model agents to find vulnerabilities with ctf-dojo. *arXiv preprint arXiv:2508.18370*, 2025.
- 1344 Andy Zou, Zifan Wang, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.
- 1345 Maddalena Zuccotto, Alberto Castellini, Davide La Torre, Lapo Mola, and Alessandro Farinelli. Reinforcement learning applications in environmental sustainability: a review. *Artificial Intelligence Review*, 57(4):88, 2024.