RepoGenesis: Benchmarking End-to-End Microservice Generation from Readme to Repository

Zhiyuan Peng[1*], Xin Yin[3*], Pu Zhao[2], Fangkai Yang[2], Lu Wang[2], Ran Jia[2], Xu Chen[2], Qingwei Lin[2], Saravan Rajmohan[2], Dongmei Zhang[2]
[1]Shanghai Jiao Tong University, [2]Microsoft, [3]Zhejiang University
pzy2000@sjtu.edu.cn, xyin@zju.edu.cn
{puzhao, fangkai.yang, wlu, Jia.Ran}@microsoft.com
{xu.chen, qlin, sarr1, dongmeiz}@microsoft.com

Abstract

Large language models and agents have achieved remarkable progress in code generation. However, existing benchmarks focus on isolated function/class-level generation (e.g., ClassEval) or modifications to existing codebases (e.g., SWE-Bench), neglecting complete microservice repository generation that reflects real-world 0-to-1 development workflows. To bridge this gap, we introduce RepoGenesis, the first multilingual benchmark for repository-level end-to-end web microservice generation, comprising 106 repositories (60 Python, 46 Java) across 18 domains and 11 frameworks, with 1,258 API endpoints and 2,335 test cases verified through a "review-rebuttal" quality assurance process. We evaluate open-source agents (e.g., DeepCode) and commercial IDEs (e.g., Cursor) using Pass@1, API Coverage (AC), and Deployment Success Rate (DSR). Results reveal that despite high AC (up to 73.91%) and DSR (up to 100%), the best-performing system achieves only 23.67% Pass@1 on Python and 21.45% on Java, exposing deficiencies in architectural coherence, dependency management, and cross-file consistency. Notably, GenesisAgent-8B, fine-tuned on RepoGenesis (train), achieves performance comparable to GPT-5 mini, demonstrating the quality of RepoGenesis for advancing microservice generation. We release our benchmark at https://github.com/microsoft/DKI_LLM/tree/main/RepoGenesis.

1 Introduction

The paradigm of software development is undergoing a fundamental shift, transitioning towards autonomous agents for code generation. Coding agents have achieved widespread adoption (Hong et al., 2023; Team, 2024; Li et al., 2024a; Lab, 2025), with specialized tools such as OpenHands (Wang et al., 2024) and Cursor (Anysphere, 2024) being increasingly used by developers to assist in or automate the coding process. As these agents advance, the demand for more comprehensive benchmarks that reflect real-world development scenarios becomes increasingly critical.

In practice, modern software development often involves constructing complete systems rather than isolated code fragments. Microservices have become a cornerstone of contemporary software architecture, widely adopted for building scalable, flexible, and independently deployable systems (Richardson, 2018; Newman, 2021). Developing microservices requires not only generating functional code but also designing coherent architectures, managing inter-service dependencies, and ensuring system-wide consistency. However, existing benchmarks fail to adequately evaluate the capability of agents to generate complete microservices repositories from scratch. As shown in Table 1, most current benchmarks focus on three paradigms with inherent limitations: (1) *isolated code generation*, evaluating function-level (Chen et al., 2021; Austin et al., 2021; Lai et al., 2023; Cassano et al., 2023) or class-level (Du et al., 2023) code generation; (2) *codebase modification*, assessing the ability to resolve issues or complete code within existing repositories (Jimenez et al., 2024a; Yu et al., 2024; Liu et al., 2023; Li et al., 2024b; Ni et al., 2025; Li et al., 2025); and (3) *repository-level generation*, which either focuses on non-microservice architectures (Luo et al., 2025; Ding et al., 2025) or offers limited scale and diversity (Zan et al., 2024).

To bridge this gap, we introduce RepoGenesis, the first benchmark specifically designed to evaluate repository-level web microservice generation. RepoGenesis targets critical yet under-evaluated capabilities essential for real-world engineering: architectural design, dependency management, and system-level consistency. Comprising 106 repositories (76 for training, 30 for evaluation) spanning 18 domains and 11 frameworks in both Python

*Equal contribution.

Table 1: Comparison of RepoGenesis with existing code generation benchmarks. "Repo-Aware" indicates repository-level context awareness. "Repo-Level" denotes full repository generation capability. "NL2Repo." indicates natural language to repository generation.

Benchmark	Task	Repositories	Multilingual	Language	Repo-Aware	Repo-Level	NL2Repo.	Microservice
Concode (Iyer et al., 2018)	Func-lvl. Gen.	≈33k	✗	Java	✓	✗	✗	✗
HumanEval (Chen et al., 2021)	Func-lvl. Gen.	✗	✗	Python	✗	✗	✗	✗
MBPP (Austin et al., 2021)	Func-lvl. Gen.	✗	✗	Python	✗	✗	✗	✗
DS-1000 (Lai et al., 2023)	Func-lvl. Gen.	✗	✗	Python	✗	✗	✗	✗
MultiPL-E (Cassano et al., 2023)	Func-lvl. Gen.	✗	✓	Multi	✗	✗	✗	✗
ClassEval (Du et al., 2023)	Class-lvl. Gen.	✗	✗	Python	✗	✗	✗	✗
RepoBench (Liu et al., 2023)	Completion	≈25k	✓	Py/Java	✓	✗	✗	✗
CodeS (Zan et al., 2024)	Full Repo. Gen.	19	✗	Python	✓	✓	✓	✗
CoderEval (Yu et al., 2024)	Func-lvl. Gen.	53	✓	Py/Java	✓	✗	✗	✗
EvoCodeBench (Li et al., 2024b)	Func-lvl. Gen.	25	✗	Python	✓	✗	✗	✗
SWE-bench (Jimenez et al., 2024a)	Issue Resolution	12	✗	Python	✓	✗	✗	✗
SolEval (Peng et al., 2025a)	Func-lvl. Gen.	28	✗	Solidity	✓	✗	✗	✗
GitTaskBench (Ni et al., 2025)	Task Solving	18	✗	Python	✓	✗	✗	✗
RPG (Luo et al., 2025)	Full Repo. Gen.	6	✗	Python	✓	✓	✓	✗
NL2Repo-Bench (Ding et al., 2025)	Full Repo. Gen.	104	✗	Python	✓	✓	✓	✗
RepoGenesis (Ours)	**Full Repo. Gen.**	**30 (test) + 76 (train)**	✓	**Py/Java**	✓	✓	✓	✓

and Java, RepoGenesis provides a comprehensive testbed with 1,258 API endpoints and 2,335 test cases. Each sample consists of a requirement document specifying service functionality, API endpoints, and input/output schemas, paired with rigorous black-box test suites that validate functional correctness and completeness. The benchmark includes both real-world GitHub projects and expert-curated implementations, ensuring diverse architectural patterns and practical relevance. To ensure benchmark reliability, we implement a rigorous quality assurance mechanism featuring a "review-rebuttal" loop with three LLM reviewers and a human Area Chair, guaranteeing high-quality test cases and reducing potential biases.

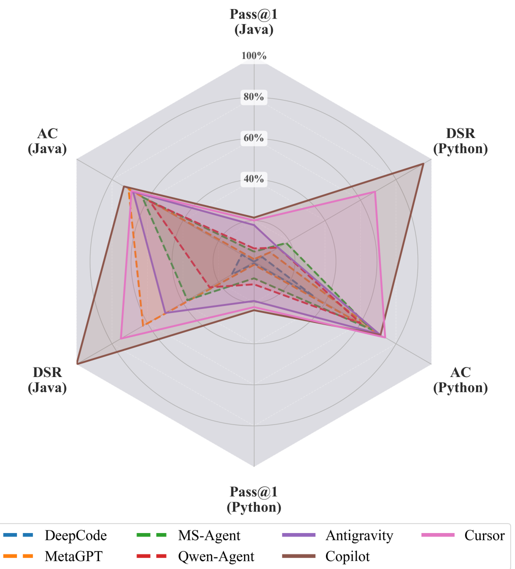

Figure 1: Capability map of agents and IDEs

We establish a multi-dimensional evaluation methodology comprising Pass@1 for functional correctness, API Coverage (AC) for implementation completeness, and Deployment Success Rate (DSR) for deployability. Evaluations on four open-source agents (i.e., DeepCode (Lab, 2025), MetaGPT (Hong et al., 2023), MS-Agent (Li et al., 2024a), Qwen-Agent (Team, 2024)) and three commercial IDEs (i.e., Antigravity (Google, 2025), Cursor (Anysphere, 2024), Copilot (GitHub, 2021)) reveal a significant performance gap. As illustrated in Figure 1, even the best-performing agent achieves only 23.67% Pass@1. Error analysis in Table 2 reveals that failures stem primarily from cross-file consistency issues (50.2%), architectural coherence problems (26.0%), and dependency management errors (23.8%), underscoring fundamental challenges in repository-level code generation.

Table 2: Distribution of critical deficiencies in model-generated repositories, categorized into three fundamental challenges in repository-level code generation.

Deficiency Type	**Python**	**Java**	**Total**
Architectural Coherence	45 (25.9%)	27 (26.2%)	72 (26.0%)
Dependency Management	20 (11.5%)	46 (**44.7%**)	66 (23.8%)
Cross-file Consistency	109 (**62.6%**)	30 (29.1%)	139 (**50.2%**)
Total Failures	**174**	**103**	**277**

In summary, our contributions are as follows:

- We introduce RepoGenesis, the first multilingual benchmark for repository-level web microservice generation from natural language requirements.
- We establish a rigorous quality assurance mechanism featuring a "review-rebuttal" loop with three LLM reviewers and a human Area Chair to ensure benchmark reliability.
- We propose a multi-dimensional evaluation methodology (i.e., Pass@1, AC, and DSR) and conduct comprehensive evaluations on state-of-the-art coding agents and commercial IDEs.

- Our evaluation reveals critical insights: (1) commercial IDEs excel in deployability but struggle with functional correctness, exposing a gap between infrastructure and logic generation; and (2) fine-tuned GenesisAgent-8B achieves GPT-5 mini-level performance, demonstrating effective benchmark-driven model improvement.

2 Benchmark – RepoGenesis

2.1 Statistics of RepoGenesis

As shown in Table 3, RepoGenesis (Verified) comprises 30 web microservice repositories spanning 18 domains and 11 frameworks, serving as our primary evaluation benchmark.

To balance domain coverage with quality control, we combine 6 real-world GitHub repositories with 24 expert-supervised repositories[1] targeting underrepresented domains (e.g., gaming backends, file management, and authentication services). To mitigate potential bias from the expert-supervised subset (80%), we implement: (1) human-authored requirement specifications, (2) a multi-model review-rebuttal quality assurance process with human Area Chair (§3.3), and (3) manual filtering to remove ambiguous requirements and flaky tests (Jimenez et al., 2024a). This design ensures domain diversity while maintaining evaluation rigor Repositories are classified into Easy, Medium, and Hard difficulty levels based on code complexity metrics (see Appendix §3.4 for more details).

To support model training, we additionally construct RepoGenesis (Train) with 76 repositories (§3.5). In total, RepoGenesis encompasses 1,258 API endpoints and 2,335 test cases, with complete statistics in Table 10.

2.2 Task Definition and Challenges

Given a requirement document (README.md) specifying service functionality, API endpoints with input/output schemas, authentication mechanisms, and operational constraints, models must generate a fully deployable repository including all source code, configuration files, and dependency specifications. The generated repository must pass comprehensive black-box test suites validating functional correctness, API compliance, and error handling. This task presents three fundamental challenges: (1) translating high-level requirements into concrete software architectures with proper code organization and module decomposition, (2) ensuring cross-file consistency across APIs, data models, and error handling, and (3) handling real-world system complexity including authentication, database interactions, and input validation. For more details, see Appendix §B.

[1] Repositories with human-authored requirement documents and LLM-generated test suites under iterative human supervision.

2.3 Evaluation Pipeline

As illustrated in Figure 2, RepoGenesis evaluates agents through ❶ Repository-level Code Generation and ❷ Post-Generation Evaluation.

2.3.1 Repository-level Code Generation

The code agent receives a *Requirement Document* (a README.md file) specifying microservice requirements: service functionality, API endpoints with input/output schemas, authentication mechanisms, error handling specifications, and operational constraints (e.g., listening ports, deployment requirements). The agent generates a complete repository from scratch, including source code files, configuration files, and dependency specifications.

Table 3: Statistics of the RepoGenesis (Verified). **APIs** denotes the number of APIs in requirements; Repository names are color-coded by difficulty: Easy, Medium, Hard. FW denotes Framework. Dom denotes Domain.

Repository	Lang.	FW.	APIs	Tests	Stars	Dom.
Real-world GitHub Repositories						
django-rest-framework-crud	Python	Django REST	10	9	288	Auth & CRUD
eve	Python	Eve/Flask	6	8	6.7k	REST FW
flask	Python	Flask	3	7	70.7k	Web FW
rock-paper-scissors-flask	Python	Flask	3	5	6	Gaming
SimpleFastPyAPI	Python	FastAPI	5	6	26	API Svc
synapse	Python	Synapse	10	5	12k	Comm
Expert-supervised Repositories						
Blog	Python	Django/FastAPI	11	17	-	CMS
Chatroom	Python	Flask/FastAPI	9	9	-	RT Chat
Customization	Python	FastAPI	10	64	-	User Prefs
Data Rank Searcher	Python	Flask/FastAPI	4	33	-	Data Search
File Relay	Python	Flask/FastAPI	10	46	-	File Mgmt
GameBackend	Python	Flask/FastAPI	10	59	-	Gaming BE
Javalin-online-judge	Java	Javalin	19	23	-	OJ
Javalin-task-manager	Java	Javalin	24	16	-	Task Mgmt
Javalin-user-auth-platform	Java	Javalin	13	52	-	User Auth
Micronaut-ci-status	Java	Micronaut	8	26	-	CI Status
Quarkus-blog-cms	Java	Quarkus	29	16	-	CMS
Spark-dashboard-backend	Java	Spark	25	16	-	Dashboard
Spring-boot-course-scheduling	Java	Spring Boot	45	19	-	Scheduling
Springboot-chat-gateway	Java	Spring Boot	15	13	-	Chat GW
Mail Service	Python	Flask/FastAPI	4	55	-	Email Svc
Multilingual	Python	Flask	4	33	-	I18n
Simple RBAC Service	Python	Flask/FastAPI	4	9	-	AuthZ
StructuredDataConvertor	Python	Flask/FastAPI	3	38	-	Data Conv
TaskManagement	Python	Flask/FastAPI	6	44	-	Proj Mgmt
Tic-Tac-Toe	Python	Flask/FastAPI	6	7	-	Gaming
Timer4Tasker	Python	Flask/FastAPI	9	53	-	Scheduling
UserManagement	Python	FastAPI	8	69	-	User Auth
UserManagement Lite	Python	Flask/FastAPI	5	31	-	User Sys
WebPan	Python	Flask/FastAPI	13	22	-	File Storage
Total	2	9 Stacks	294	805	-	18

2.3.2 Post-Generation Evaluation

We evaluate generated repositories using black-box testing: deploying the microservice, executing test cases, and computing metrics. We assess quality

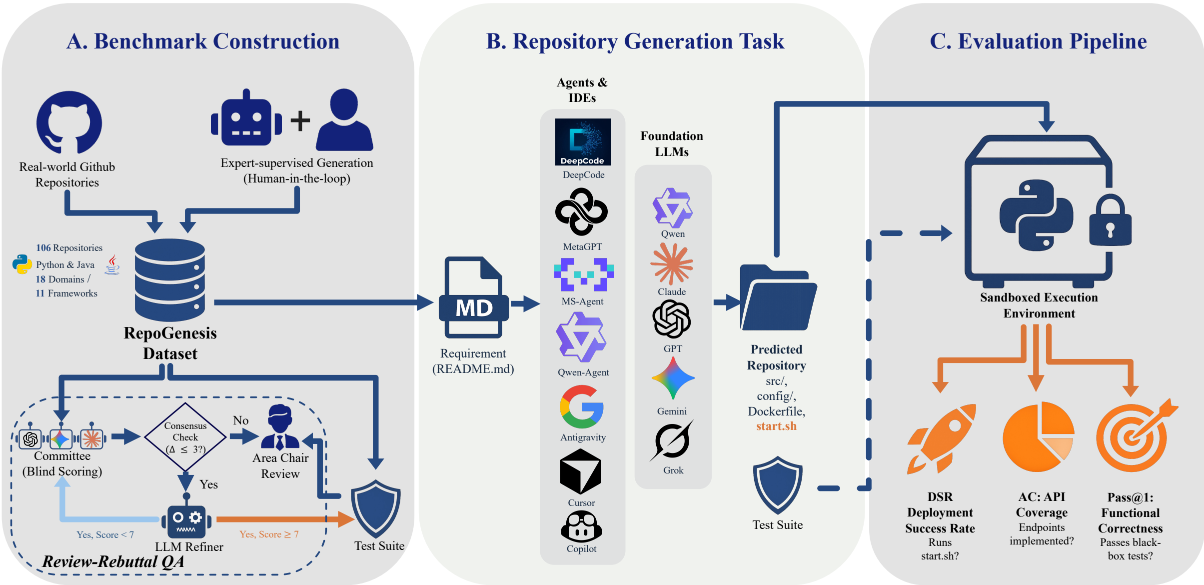

Figure 2: Overview of benchmark construction and evaluation pipeline, which undergo rigorous quality assurance through a "Review-Rebuttal QA". Given a requirement document (README.md), various agents and IDEs generate repositories. Repositories are executed in sandboxed environments and evaluated across DSR, AC, and Pass@1.

across three dimensions: (1) Pass@k, (2) API Coverage, and (3) Deployment Success Rate. Detailed metric definitions are in Appendix §D.4.

3 Benchmark Construction

RepoGenesis construction involves four phases: (1) Repository Collection, (2) Repository Preprocessing, (3) Test Case & Quality Assurance, and (4) Difficulty Classification.

3.1 Repository Collection

We collected repositories from two sources:

Real-world GitHub Repositories. We initially identified 28 web microservice repositories from GitHub based on four selection criteria: (1) substantial community adoption (stars and active discussions), (2) comprehensive documentation with clear API specifications, (3) existing test suites, and (4) diverse technology stacks (Flask, Django REST Framework, FastAPI, etc.). After cloning and attempting deployment, we retained only repositories that could be successfully deployed out-of-the-box without modification, as deployability is essential for reliable benchmark evaluation. This filtering yielded 9 repositories representing diverse architectural patterns and application domains.

Expert-supervised Repository Generation. To achieve comprehensive domain coverage beyond available real-world open-source microservices, we employ a human-in-the-loop approach: two senior researchers manually author 97 requirement documents specifying API contracts, business logic, and operational constraints. Test suites are then generated using Gemini 3 Pro with iterative human supervision following the rigorous "review-rebuttal" quality assurance process (§3.3). This expert-supervised approach targets underrepresented but practical domains (e.g., gaming backends, file management systems, specialized authentication services) while maintaining semantic validity through human verification. While this introduces LLM-generated artifacts, the multi-stage human oversight ensures alignment with real-world patterns.

3.2 Repository Preprocessing

All repositories underwent systematic preprocessing for benchmark readiness. For real-world repositories, we refined README files with essential API information, removed implementation code, and verified test suites. For expert-supervised repositories, we validated API specifications and requirement-functionality consistency. This yielded 106 repository specifications ready for evaluation.

3.3 Test Case & Quality Assurance

We developed rigorous black-box test suites for all repositories covering: (1) *API Compliance*, (2) *Error Handling*, (3) *Edge Cases*, and (4) *Integration Testing*. Test cases were generated using Gemini 3 Pro and refined through a "review-rebuttal" quality assurance process with multi-model evaluation and human oversight (Figure 2, bottom left).

Blind Review All three models evaluate test cases simultaneously without access to each other's scores or reasoning. Preliminary experiments showed this reduces anchoring bias compared to serial scoring, ensuring independent judgment and identifying subtle flaws that might be overlooked in groupthink scenarios.

Refinement and Consensus The process follows an iterative "review-rebuttal" loop driven by three LLM reviewers and a human Area Chair:

1. **Initial Review**: The committee scores the initial test suite and provides explanations for each score, resembling a peer review process.

2. **Consensus Check**: If pairwise score difference exceeds $\Delta > 3$, the case is flagged for manual review by an Area Chair (a senior code generation PhD researcher) providing meta scores, and decides whether to enter refinement (rebuttal). Check § N.1 for how Area Chair makes decisions.

3. **Refinement**: If scores are consistent ($\Delta \leq 3$) but maximum score is below 7, aggregated feedback is sent to an LLM refiner to improve the test case.

4. **Iteration**: The refined test case is re-evaluated by the committee. This continues until $\max(\text{Score}) \geq 7$ or maximum iterations ($T = 5$) are reached.

Inter-Annotator Agreement We measure inter-annotator agreement using Krippendorff's alpha on 150 test case evaluations. Initial zero-shot scoring yielded $\alpha = 0.032$. We conducted iterative rubric refinement: for cases with score spread > 3, we analyzed disagreements to clarify evaluation criteria (e.g., balancing coverage versus readability). This improved reliability to $\alpha = 0.69$, indicating moderate consensus on a 10-point scale (Landis and Koch, 1977). Notably, 78% of cases showed agreement within 2 points, supported by strong pairwise Spearman correlation ($\rho = 0.61, p < 0.001$). claude-haiku-4.5 (mean 6.19) aligns most closely with human experts (mean 5.83), while GPT-5 mini and Gemini 2.5 Flash exhibit systematic leniency. We only include test cases achieving stringent consensus (mean score ≥ 7).

3.4 Difficulty Classification

Inspired by GAIA (Mialon et al., 2023), we classify repositories into three difficulty levels (Easy, Medium, Hard) based on the following metrics:

Classification Metrics. We evaluate complexity using five metrics: (1) *Lines of Code (LOC)*: non-comment, non-blank lines; (2) *Cyclomatic Complexity*: branch complexity from control flow structures; (3) *File Count*: number of source files (`.py` or `.java`); (4) *API Endpoints*: number of exposed interfaces; and (5) *Functions and Classes*: total code constructs. The same thresholds apply to both Python and Java repositories.

Scoring. Each repository receives a weighted score: LOC (25%), cyclomatic complexity (30%), file count (15%), API endpoints (15%), and functions/classes (15%). Scores range from 2 (simplest) to 10 (most complex).

Thresholds. *Easy* (score < 4.5): simple services with minimal business logic; *Medium* ($4.5 \leq$ score < 7.0): moderate complexity with authentication, validation, and business logic; *Hard* (score ≥ 7.0): complex services with intricate state management and error handling.

RepoGenesis (Verified) (Table 11) comprises 6 Easy (20.0%), 16 Medium (53.3%), and 8 Hard (26.7%) repositories. Expert-supervised Python repositories are predominantly Hard, reflecting production-level complexity, while Java repositories are predominantly Medium.

3.5 RepoGenesis for Training

To support model training, we extend RepoGenesis (Train) with 76 repositories (44 real-world, 32 expert-supervised). Following the review-rebuttal process (§3.3), we employ the *Multi-Model Evaluation Committee* to evaluate test cases. However, RepoGenesis (Train) *omits human Area Chair intervention*, introducing reliability uncertainty. When conflicts arise among LLM reviewers (pairwise score difference $\Delta > 3$), conflicting test cases are discarded rather than sent to the Area Chair for resolution, as shown in Figure 2 (bottom left). This automated conflict resolution resulted in approximately 23% of initially generated test cases being discarded during the QA process. Only test cases achieving LLM consensus are retained. The inter-annotator agreement reported in §3.3 ($\alpha = 0.69$) applies exclusively to RepoGenesis (Verified), where human Area Chair mediation resolves LLM disagreements and provides groundtruth calibration. Without human adjudication, RepoGenesis (Train) lacks this quality bound. The automated discard-on-conflict strategy may eliminate challenging edge cases requiring expert judgment, potentially underrepresenting complex scenarios and adversarial inputs in the training distribution. Consequently, while RepoGenesis (Train) provides a resource for supervised fine-tuning, we exclude it from coding agent evaluation to maintain rigorous benchmarking standards.

4 Experimental Setup

4.1 Evaluated Systems

We evaluate four open-source coding agents (i.e., DeepCode (Lab, 2025), MetaGPT (Hong et al., 2023), MS-Agent (Li et al., 2024a), and Qwen-Agent (Team, 2024)) and three commercial IDEs (i.e., Antigravity (Google, 2025), Cursor (Anysphere, 2024), and Copilot (GitHub, 2021)). Detailed description and prompt of agents are provided in Appendix §D.

For coding agents, we use three closed-source models (i.e., GPT-5.1, GPT-5.1 mini, and Claude-Sonnet-4.5) and two open-source models (i.e., Qwen3-Coder-30B-A3B-Instruct and Qwen3-30B-A3B-Instruct). For commercial IDEs, we use officially supported models to elicit optimal programming capabilities; underlying models may vary due to platform-specific availability. Coding agents are permitted to use built-in tools including code search and file system operations, while commercial IDEs use default configurations. For hardware and software environment details, see Appendix §H.

4.2 Evaluation Metrics

We evaluate generated repositories using Pass@1 for functional correctness, API Coverage (AC) for implementation completeness, and Deployment Success Rate (DSR) for deployability. Detailed definitions are provided in Appendix §2.3.2. All results are averaged across five independent runs.

5 Evaluation

5.1 Results of Coding Agents and IDEs

For brevity, we use “Claude” to denote Claude-Sonnet-4.5 and “Qwen3-30B” to denote Qwen3-30B-A3B-Instruct throughout this section. We report results for GPT-5.1, Claude, and Qwen3-30B in the main text; additional results for other LLMs are provided in Appendix §K.

Table 4: Overall performance (Pass@1) of Coding Agents and Commercial IDEs.

Method	Model	Python	Java
DeepCode	GPT-5.1	0.81%	0.00%
	Claude	1.95%	0.85%
	Qwen3-30B	0.00%	0.00%
MS-Agent	GPT-5.1	7.83%	4.76%
	Claude	9.50%	5.95%
	Qwen3-30B	3.56%	1.25%
MetaGPT	GPT-5.1	1.46%	1.14%
	Claude	2.85%	2.10%
	Qwen3-30B	0.15%	0.00%
Qwen-Agent	GPT-5.1	10.89%	6.29%
	Claude	12.65%	7.80%
	Qwen3-30B	0.35%	4.12%
Antigravity	Claude	19.17%	17.82%
Cursor	Claude	22.15%	20.10%
Copilot	Claude	23.67%	21.45%

Pass Rate (Pass@1). As shown in Tables 4 and 6, commercial IDEs significantly outperform open-source agents, with Copilot achieving 23.67% on Python and 21.45% on Java. We observe three key trends: (1) **Performance Gap:** Commercial IDEs achieve substantially better Pass@1 than open-source agents across both languages. (2) **Language Sensitivity:** Python performance generally exceeds Java, as Java's verbosity and rigid type system increase implementation complexity despite its superior tooling. (3) **Difficulty Sensitivity:** All systems exhibit sharp performance degradation as repository complexity increases. For example, Qwen-Agent with GPT-5.1 drops from 25.00% (Easy) to 4.86% (Hard) on Python, while MS-Agent maintains a more gradual degradation curve. Notably, even smaller models such as Qwen3-30B nearly fail on *Medium* and *Hard* tasks, underscoring that architectural reasoning remains challenging for resource-constrained models.

Table 5: DSR and AC for coding agents and IDEs.

Method	Model	Python		Java	
		DSR	AC	DSR	AC
DeepCode	Qwen3-30B	0.00%	12.00%	0.00%	1.50%
	GPT-5.1	4.55%	26.96%	0.00%	0.00%
	Claude	16.50%	36.00%	28.50%	10.50%
MS-Agent	Qwen3-30B	2.27%	55.00%	10.00%	15.50%
	GPT-5.1	18.18%	65.22%	37.50%	47.41%
	Claude	12.00%	69.00%	16.00%	66.50%
MetaGPT	Qwen3-30B	0.00%	60.00%	0.00%	40.00%
	GPT-5.1	4.55%	70.43%	62.50%	68.10%
	Claude	13.00%	73.04%	54.00%	74.14%
Qwen-Agent	Qwen3-30B	2.27%	50.00%	7.50%	35.00%
	GPT-5.1	13.64%	37.39%	25.00%	41.38%
	Claude	17.00%	60.50%	28.00%	60.00%
Copilot	Claude	95.45%	69.57%	62.50%	68.10%
Antigravity	Claude	13.64%	60.00%	50.00%	27.59%
Cursor	Claude	68.18%	73.91%	75.00%	68.97%

Table 6: Performance (Pass@1) across difficulty levels. Easy (E), Medium (M), and Hard (H) classifications are based on code complexity metrics (§3.4). Note that Java repositories have no Easy-level instances.

Method	Python (E/M/H)									Java (E/M/H)								
	GPT-5.1			Claude			Qwen3-30B			GPT-5.1			Claude			Qwen3-30B		
MS-Agent	16.07	9.52	3.47	18.95	10.45	4.12	5.36	4.17	2.08	-	5.30	0.98	-	6.85	1.50	-	1.40	0.20
MetaGPT	3.57	1.79	0.00	4.80	2.65	1.20	0.00	0.30	0.00	-	1.30	0.02	-	2.15	1.05	-	0.00	0.00
Qwen-Agent	25.00	14.88	4.86	26.50	16.85	6.45	3.57	0.00	0.00	-	7.00	1.32	-	8.50	2.10	-	4.60	0.76
DeepCode	3.57	0.60	0.00	4.50	1.80	0.95	0.00	0.00	0.00	-	0.00	0.00	-	1.10	0.85	-	0.00	0.00

Table 7: DSR comparison by programming language.

Language	Success	Total	DSR
Java	40	96	41.67%
Python	75	264	28.41%
Overall	**115**	**360**	**31.94%**

DSR and AC. We evaluate the gap between code synthesis capability and system usability using DSR and AC (Table 5). Our evaluation reveals a notable decoupling between the ability to generate code and the ability to produce a functional system, as detailed below: (1) **Deployability:** Commercial IDEs substantially outperform open-source agents. Copilot achieves 95.45% DSR on Python and 62.50% on Java, while Cursor demonstrates strong cross-language performance (68.18% Python, 75.00% Java). In contrast, most open-source agents achieve below 20% DSR. As shown in Table 7, Java achieves higher overall DSR (41.67%) than Python (28.41%), attributable to its standardized build systems (Maven/Gradle). Failure analysis reveals that agents frequently struggle with environment-specific configurations and framework-specific runtime errors. (2) **Completeness:** While open-source agents such as MetaGPT achieve competitive API coverage, with Claude-powered MetaGPT reaching 73.04% (Python) and 74.14% (Java) matching commercial IDEs, their lower DSR reveals a critical *AC-DSR gap*: these agents often generate broad API skeletons that fail to initialize properly. This discrepancy underscores that repository-level generation requires both synthesis breadth and infrastructure precision.

5.2 Fine-Tuning Qwen3 with RepoGenesis

To further explore the potential of open-source models on RepoGenesis, we fine-tuned Qwen3-8B using high-quality agent trajectories. We first extended the MS-Agent (Li et al., 2024a) to create GenesisAgent, enhancing it with language-specific optimizations for repository-level microservice generation, including strict dependency handling and robust execution verification (see Appendix §E for details of GenesisAgent).

We employed GenesisAgent with Qwen3-8B on the RepoGenesis (Train), excluding the verified subset. We filtered for successful trajectories where the GenesisAgent generated a complete, compilable, and executable repository, resulting in a dataset of 16,396 high-quality instruction-tuning samples. We then fine-tuned Qwen3-8B on this distilled dataset for 3 epochs with a learning rate of 2e-4 and evaluated the fine-tuned model on RepoGenesis (Verified). As shown in Table 8, the fine-tuned Qwen3-8B (*GenesisAgent-8B*) achieves performance comparable to GPT-5 mini.

Table 8: Performance comparison of **GenesisAgent-8B** against GPT-5 mini on RepoGenesis.

Model	Java			Python		
	DSR	AC	Pass@1	DSR	AC	Pass@1
Qwen3-8B	0.00%	16.38%	0.00%	4.55%	30.43%	0.00%
GPT-5 mini	25.00%	17.24%	**1.71%**	4.55%	60.87%	**4.46%**
GenesisAgent-8B	**37.12%**	**46.55%**	1.69%	**30.15%**	**63.42%**	4.38%

5.3 Empirical Lessons

Through systematic analysis of successful and failed repository generation attempts, we identify key insights into model capabilities and limitations (detailed examples are provided in Appendix §J).

Key Insights from Success Cases. Successful repository generation exhibits five critical capabilities: (1) *Complex requirement decomposition:* parsing multilingual specifications to extract intricate API schemas with compound parameter interactions; (2) *Proactive error handling:* implementing comprehensive input validation including type checking and edge case handling beyond explicit requirements; (3) *Algorithmic problem-solving:* correctly implementing domain-specific algorithms by selecting appropriate language primitives; (4) *Test-driven debugging:* identifying and fixing implementation errors through iterative test feedback; and (5) *Data structure optimization:* balancing performance and simplicity by selecting appropriate data structures for different operations.

Key Insights from Failure Cases. Three dominant failure patterns emerge from our analysis: (1) *Infrastructure configuration failures:* missing or malformed deployment scripts (e.g., `start.sh`), prevent service initialization; (2) *Framework version incompatibility:* agents frequently use deprecated API syntax (e.g., Javalin's parameter notation changes), causing runtime exceptions despite syntactically correct code; and (3) *Phantom generation behavior:* certain configurations (e.g., Cursor with GPT-4.1) exhibit file persistence failures where generation appears successful but produces no artifacts, indicating critical tooling integration issues. Refer to Appendix §J.2.1 for more details.

6 Related Works

6.1 Microservices and Modern Software Architecture

Microservices architecture decomposes applications into independently deployable services communicating via lightweight mechanisms, typically RESTful APIs (Fowler and Lewis, 2015; Richardson, 2018; Newman, 2021). RESTful APIs define uniform interfaces for stateless communication (Fielding, 2000; Masse, 2011), while web frameworks such as Flask (Grinberg, 2018) and Django (Forcier et al., 2008) provide abstractions for routing and request handling (Neumann et al., 2018). Generating such microservices thus demands both functional correctness and architectural coherence, making them a rigorous testbed for code generation agents.

6.2 Code Generation

Early benchmarks focus on function or class-level generation. HumanEval (Chen et al., 2021) comprises 164 hand-crafted Python programming problems with unit tests. AiXBench (Hao et al., 2022) targets Java with 336 problems (175 automated, 161 manual). MultiPL-E (Cassano et al., 2023) extends HumanEval and MBPP (Austin et al., 2021) to 18 programming languages. DS-1000 (Lai et al., 2023) introduces 1000 non-standalone problems across seven Python data science libraries (e.g. NumPy, Pandas, PyTorch, Scipy, Scikit-learn). Concode (Iyer et al., 2018) provides over 100,000 Java problems from open-source projects.

Recent work has shifted toward repository-level and end-to-end software engineering. Repository-aware methods such as A3-CodGen (Liao et al., 2024), RepoCoder (Zhang et al., 2023), and CatCoder (Shen et al., 2023) leverage cross-file context. Benchmarks targeting real-world GitHub issues (SWE-bench (Jimenez et al., 2024a), SWE-PolyBench (Zan et al., 2025), SWE-bench Pro (Jimenez et al., 2024b)) and engineering agent systems (SWE-agent (Yang et al., 2024), MAGIS (Tao et al., 2024), OpenHands (Wang et al., 2024)) push models toward the closed loop of requirements, cross-file changes, testing, and delivery. CodeS (Zan et al., 2024) pioneers full repository generation from natural language with 19 Python tasks but relies on BLEU metrics. RPG (Luo et al., 2025) integrates planning with 6 Python repositories but remains limited in scale and language coverage. NL2Repo-Bench (Ding et al., 2025) extends to 104 Python tasks but evaluates solely on pytest pass rate. Unlike these benchmarks, RepoGenesis is multilingual and evaluates end-to-end generation of complete microservice repositories from natural language requirements.

7 Conclusion

We present RepoGenesis, the first multilingual benchmark for repository-level web microservice generation from natural language requirements. Comprising 106 repositories across 18 domains and 11 frameworks with 1,258 API endpoints and 2,335 test cases, RepoGenesis addresses a critical gap in evaluating complete repository generation capabilities.

Our comprehensive evaluation reveals several key findings. First, even the best-performing system achieves only 23.67% Pass@1, exposing fundamental challenges in microservice generation. Second, we identify a critical *AC-DSR gap*: open-source agents often generate broad API skeletons that fail to deploy, indicating that synthesis breadth alone is insufficient without infrastructure precision. Third, fine-tuned GenesisAgent-8B achieves performance comparable to GPT-5 mini, demonstrating dataset quality for model improvement.

Future Work. Several directions emerge from our findings: (1) extending the benchmark to additional languages (e.g., Golang) and architectural patterns (e.g., event-driven microservices); (2) developing metrics that capture code quality aspects such as adherence to design patterns and maintainability; and (3) investigating agent architectures that better bridge the AC-DSR gap through improved infrastructure reasoning.

Limitations

We acknowledge several limitations of RepoGenesis that should be considered when interpreting results and planning future research.

Microservice-Specific Scope. RepoGenesis specifically targets RESTful web microservices, which constitute an important but not exhaustive category of software systems. Other architectural patterns such as event-driven systems, real-time applications with WebSocket support, GraphQL APIs, and gRPC services.

Limited Language and Framework Coverage. RepoGenesis currently focuses exclusively on web microservices implemented in Python (e.g., Flask, Django) and Java (e.g., Spring Boot). This represents a substantial portion of modern web development but does not cover other important programming paradigms such as compiled languages (i.e., C++, Go), mobile application development (i.e., Swift). We will extend RepoGenesis to encompass a broader range of programming languages, frameworks, and application types.

Idealized Requirement Specifications. RepoGenesis provides comprehensive, structured requirement documents (READMEs) as input. In real-world "Zero2One" scenarios, requirements are sometimes ambiguous, incomplete, or communicated iteratively. Our benchmark evaluates the core coding capability assuming clear specifications, leaving the challenge of requirements engineering and ambiguity resolution to future work.

Code Maintainability and Standards. Generated code is evaluated on its ability to pass tests, which does not guarantee adherence to best practices, industry standards, or long-term maintainability. The current metrics do not quantify aspects such as code readability, modular design, or idiomatic usage of frameworks. Future evaluations could incorporate static analysis tools or human review to assess these qualitative aspects of software engineering.

Ethics Statement

RepoGenesis is constructed from publicly available microservice repositories and documentation and is intended for benchmarking end-to-end repository generation; our work involves no human-subject studies and we do not collect new personally identifiable information (PII). We follow upstream licenses and apply screening to reduce the inclusion of PII and overtly harmful/offensive content, but residual identifiers or societal/domain biases may remain in Internet-sourced artifacts. Repository-level code generation also carries dual-use risk: improvements in automated microservice creation could be misapplied to produce harmful software; we therefore frame RepoGenesis around correctness-focused evaluation (tests, coverage, deployment success) rather than deployment guidance, and recommend standard safeguards (e.g., human code review, sandboxed execution, and security scanning) for any downstream use. Finally, benchmark runs and fine-tuning consume compute and may have environmental impact; we limit unnecessary sweeps, reuse pretrained models, and report key settings to support reproducibility. All samples in RepoGenesis are manually reviewed by expert researchers in the field of code generation. We ensure that none of the samples contain private information or offensive content.

References

Anysphere. 2024. Cursor: The ai code editor. https://cursor.com.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2023. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Etienne Daspe, Mathis Durand, Julien Hatin, and Salma Bradai. 2024. Benchmarking large language models for ethereum smart contract development. In *2024 6th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 1–4.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, and 1 others. 2025. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

Jeff Forcier, Paul Bissex, and Wesley J Chun. 2008. *Python web development with Django*. Addison-Wesley Professional.

Martin Fowler and James Lewis. 2015. Microservices: Nur ein weiteres konzept in der softwarearchitektur oder mehr. *Objektspektrum*, 1(2015):14–20.

GitHub. 2021. Github copilot: Your ai pair programmer. https://github.com/features/copilot.

Google. 2025. Antigravity: The ai-first ide. https://antigravity.google.

Miguel Grinberg. 2018. *Flask web development*. " O'Reilly Media, Inc.".

Ziqi Guan, Xin Yin, Zhiyuan Peng, and Chao Ni. 2025. Repotransagent: Multi-agent llm framework for repository-aware code translation. *arXiv preprint arXiv:2508.17720*.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. Aixbench: A code generation benchmark dataset. *arXiv preprint arXiv:2206.13179*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zhang, Yuheng Wang, Jinlin Wang, Ceyao Chen, Zewen Huang, Qiuyuan Wu, Liang Qian, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024a. Swe-bench: Can language models resolve real-world github issues? In *ICLR*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024b. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2404.07223*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Data Intelligence Lab. 2025. Deepcode: Open agentic coding.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Chenliang Li, He He, Yiqing Li, Yizhang Li, Zekun Li, Fei Wang, and 1 others. 2024a. Modelscope-agent: Building your customizable agent system with open-source large language models. *arXiv preprint arXiv:2404.07051*.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024b. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2404.00599*.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. *arXiv preprint arXiv:2503.06680*.

Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. A3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware. *IEEE Transactions on Software Engineering*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, and 1 others. 2025. Rpg: A repository planning graph for unified and scalable codebase generation. *arXiv preprint arXiv:2509.16198*.

Mark Masse. 2011. *REST API design rulebook: designing consistent RESTful web service interfaces*. " O'Reilly Media, Inc.".

Harshitha Menon, Daniel Nichols, Abhinav Bhatele, and Todd Gamblin. 2024. Learning to predict and improve build successes in package ecosystems. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 315–327.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Griffiths, Bertie O'Connor, Caleb Lee, and 1 others. 2023. Gaia: A benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*.

Le Hai Nam, Dung Manh Nguyen, and Nghi DQ Bui. 2024. Repoexec: Automated repository-level code generation evaluation. *arXiv preprint arXiv:2406.11927*.

Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2018. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 14(4):957–970.

Sam Newman. 2021. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, and 1 others. 2025. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. *arXiv preprint arXiv:2508.18993*.

Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Hao Zhang, Chenhao Ying, and Yuan Luo. 2025a. Soleval: Benchmarking large language models for repository-level solidity code generation. *arXiv preprint arXiv:2502.18793*.

Zhiyuan Peng, Xin Yin, Chenhao Ying, Chao Ni, and Yuan Luo. 2025b. A preference-driven methodology for high-quality solidity code generation. *arXiv preprint arXiv:2506.03006*.

Chris Richardson. 2018. *Microservices patterns: with examples in Java*. Simon and Schuster.

Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2023. Catcoder: Repository-level code generation with relevant code and type context. *arXiv preprint arXiv:2310.15077*.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993.

Qwen Team. 2024. Qwen-agent: A framework for rag and agent applications. https://github.com/QwenLM/Qwen-Agent.

Xingyao Wang and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Boyang Yang, Luyao Ren, Xin Yin, Jiadong Ren, Haoye Tian, and Shunfu Jin. 2025. Input reduction enhanced llm-based program repair. *arXiv preprint arXiv:2507.15251*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024a. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472*.

Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024b. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. 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.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, and 1 others. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Yafen Yao, Wei Li, Xiaolin Chen, Yongshun Gong, Bei Guan, and 1 others. 2024. Codes: Natural language to code repository via multi-layer sketch. *ACM Transactions on Software Engineering and Methodology*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

A Glossary

Microservice: A software architectural style that structures an application as a collection of independently deployable services, each running in its own process and communicating through lightweight protocols.

REST API: Representational State Transfer Application Programming Interface. A web service that follows REST architectural constraints, using standard HTTP methods (GET, POST, PUT, DELETE) to enable stateless client-server communication.

API Endpoint: A specific URL path and HTTP method combination that provides access to a particular functionality or resource of a web service. Each endpoint defines its input/output schema and behavior.

Zero2One Repository Generation: The task of generating complete, functional software repositories from natural language requirements (e.g., README.md) without any existing codebase, including all necessary source code files, configurations, dependencies, and project structure.

Black-box Testing: A software testing methodology that evaluates system functionality without examining code structure, focusing solely on input-output behavior and API contracts.

Web Framework: A software framework designed to support the development of web applications and services by providing standard components for routing, request handling, authentication, and data serialization (e.g., Flask, Django, FastAPI, Express.js).

Coding Agent: A multi-agent AI system designed to autonomously generate code by coordinating specialized agents for tasks such as requirement understanding, architectural planning, code generation, and testing.

B Task Definition

The repository generation task (i.e., README-to-repository code generation) evaluates the capability of large language models to create complete software repositories from natural language requirements. Unlike traditional code generation tasks that focus on completing individual functions or modifying existing code (Guan et al., 2025; Yin et al., 2024a; Jimenez et al., 2024a; Liao et al., 2024; Peng et al., 2025a; Yin et al., 2024b; Yang et al., 2025; Peng et al., 2025b), repository generation requires models to perform end-to-end software development from scratch. We focus on web microservices, which represent the fundamental building block of contemporary software architecture where applications are decomposed into services exposing RESTful APIs. This architectural pattern is universal across modern software systems: mobile and web applications communicate with backend services through APIs, microservices interact through web interfaces, and cloud platforms fundamentally operate on API-driven service composition.

Task Input. The input consists of a requirement document (typically a README.md) that specifies the microservice requirements. The requirement document includes service functionality descriptions, API endpoint specifications with detailed input/output schemas, authentication mechanisms, error handling, data models, and operational constraints (e.g., port configurations and deployment requirements). Specifically, `pom.xml` is also included for Java and `requirements.txt` for Python. Refer to Appendix F.3 for more details.

Task Output. The output is a complete repository that satisfies all specified requirements. This includes all source code files organized in a proper project structure, configuration files (e.g., requirements.txt), dependency specifications, database initialization scripts if needed, and any additional files required for deployment. The generated repository should be executable and pass comprehensive black-box test suites that validate functional correctness, API compliance, error handling, and business logic implementation. Specifically, to facilitate deployment and DSR measurement, we require all generated repositories to include a startup script that contains a single shell command to launch the service directly.

Task Formulation. Formally, given a requirement document R that specifies the desired functionality, the task is to generate a repository $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ where each c_i represents a file in the repository. The generated repository $\mathcal{C}$ must satisfy the following conditions: (1) *Functional Correctness*: all API endpoints specified in R are correctly implemented and pass the test suite $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$, (2) *Structural Completeness*: the repository contains all necessary files for deployment and execution, (3) *API Compliance*:

the implemented APIs conform to the input/output schemas specified in R, and (4) *Error Handling*: the system appropriately handles edge cases and invalid inputs as specified in R.

Evaluation Criteria. We evaluate the generated repositories using three complementary metrics:

- **Pass@k**: A repository is considered successful if it passes all test cases in the test suite. This binary evaluation reflects real-world software development scenarios where partial implementations are typically insufficient for deployment. The Pass@k metric measures the probability that at least one of k generated repositories passes all tests, providing a stringent evaluation of practical code generation capabilities. Detailed calculation is provided in §D.4.1.

- **API Coverage (AC)**: Measures the percentage of API endpoints successfully implemented with correct functionality relative to the requirements specification. This metric captures functional completeness even when the full test suite is not passed. See §D.5 for details.

- **Deployment Success Rate (DSR)**: A binary metric indicating whether the generated repository can be successfully deployed and initialized without runtime errors. This captures the basic executability of the generated code. See §D.6 for details.

Task Challenges. Repository generation from scratch presents several fundamental challenges. First, models must translate high-level natural language requirements into concrete software architectures, making critical design decisions about code organization, module decomposition, and component interactions. Second, models must ensure consistency across multiple files and components, maintaining coherent APIs, data models, and error handling. Third, models must handle the complexity of real-world software systems including authentication, authorization, database interactions, input validation, and error handling. These challenges distinguish repository generation from simpler function-level or file-level code generation tasks.

C Test Quality Assurance Details

C.1 Scoring Mechanism.

Each committee member independently evaluates test cases on four dimensions (0-10 scale):

1. **Correctness (0-3)**: Syntax, runnability, and assertion accuracy.
2. **Coverage & Completeness (0-3)**: Happy paths, edge cases, and error conditions.
3. **Readability & Maintainability (0-2)**: Naming, formatting, and documentation.
4. **Relevance (0-2)**: Alignment with repository purpose and README specifications.

D Additional Experimental Details

D.1 Evaluated Agents

In this paper, we evaluate multiple coding agent frameworks and commercial IDEs on RepoGenesis. The details are described as follows.

- **DeepCode**: DeepCode is a code generation platform based on a multi-agent system, specifically designed to transform complex algorithms and research papers into high-quality code. It integrates the Model Context Protocol (MCP) to achieve robust external tool calling capabilities. In our evaluation workflow, we utilized its central `CodeImplementationWorkflow` and configured it for the "pure code generation" mode. By pre-extracting key requirements from the project README and generating an initial implementation plan (`initial_plan.txt`), DeepCode can autonomously construct the project structure and write source code. To optimize the presentation of results, we flattened the generated directory structure to ensure that all core files are located directly in the repository root.

- **MetaGPT**: MetaGPT is a framework that pioneers the introduction of Standardized Operating Procedures (SOPs) into multi-agent collaboration, handling complex tasks by simulating a software company's organizational structure (assigning roles such as Product Manager, Architect, and Engineer). In our experiments, we employed the `generate_repo` interface of MetaGPT. Regarding configuration, we provided the README as the primary requirement input, set the maximum iterations to 5 (`n_round=5`), and enabled mandatory code review (`code_review=True`). Through customized prompts, we required MetaGPT to generate not only the complete business logic but also environment configuration files and one-click startup scripts, ensuring the generated projects are highly self-contained.

- **MS-Agent**: MS-Agent (ModelScope-Agent) is a lightweight and highly extensible agent framework that emphasizes autonomous exploration capabilities and provides full support for the MCP protocol. In our evaluation setup, we utilized its `LLMAgent` class and dynamically loaded model service parameters via `OmegaConf`. For repository generation tasks, we developed comprehensive implementation prompts to guide the agent in analyzing the README, designing the system architecture, and implementing all source files. During execution, MS-Agent automatically handles directory hierarchy creation and ensures that components like web services are correctly bound to the `0.0.0.0` interface to meet our automated testing environment requirements.

- **Qwen-Agent**: Qwen-Agent is an application development framework, featuring built-in efficient tools such as a code interpreter. We adopted its `Assistant` class as the core executor and equipped it with the `code_interpreter` plugin. In practical calls, we assigned the agent the identity of a senior software engineer and required it to execute tasks described in the README following a sequence of analysis, design, coding, and verification. By integrating the code interpreter into the workflow, Qwen-Agent can check syntax and dependencies in real-time during generation, ultimately outputting a code repository with a `start.sh` entry point.

D.2 Agent Prompts

To ensure reproducibility, we provide the specific system prompts and configuration details used for each open-source agent. We strictly standardized the system prompt across all agents to ensure a fair comparison, while adapting the execution prompts to fit the specific interaction mode of each agent framework.

D.2.1 Universal System Prompt

All evaluated agents (MetaGPT, DeepCode, Qwen-Agent, MS-Agent) share the same system instruction to establish the persona and task requirements. The `readme_text` and `pom_text` placeholders are replaced with the actual content of the repository's requirement document and `pom.xml` file, respectively.

System Prompt (Java)

You are a senior Java software engineer tasked with implementing a complete software project.
Project Requirements:
README:
{readme_text}
POM.xml (reference):
{pom_text}
Your task:

1. Analyze the README and referenced POM to understand the project requirements.
2. Design the complete project structure and architecture.
3. Implement ALL necessary files including:
    - Main application files
    - Configuration files
    - Dependencies/requirements files (pom.xml is required)
    - Documentation files
    - Any additional files needed for the project to run
4. Ensure the project can be started via a single shell command writen in a file named start.sh.
5. The generated start.sh MUST:
    - Listen on 0.0.0.0
    - Use port specified in the README
    - Use ONLY the correct command for the detected framework
6. If a web service is expected, bind to 0.0.0.0 and use the port specified in the README.
7. Write production-ready, well-documented code.
8. Use Maven.

Important: Generate ALL files in the current working directory. Do not reference or peek at any tests directory.

System Prompt (Python)

You are a senior software engineer tasked with implementing a complete software project.
Project Requirements:
README:
{readme_text}
requirements.txt (reference):
{requirements_text}
Your task:

1. Analyze the README and referenced requirements to understand the project requirements.
2. Design the complete project structure and architecture.
3. Implement ALL necessary files including:
   - Main application files
   - Configuration files
   - Dependencies/requirements files (requirements.txt is required)
   - Documentation files
   - Any additional files needed for the project to run
4. Ensure the project can be started via a single shell command writen in a file named start.sh.
5. The generated start.sh MUST:
   - Listen on 0.0.0.0
   - Use a common port (e.g., 8000) or the one specified in the README
   - Use ONLY the correct command for the detected framework
6. If a web service is expected, bind to 0.0.0.0 and use the port.
7. Write production-ready, well-documented code.

Important: Generate ALL files in the current working directory. Do not reference or peek at any tests directory.

D.2.2 Agent-Specific Configurations

MetaGPT. We utilize the `generate_repo` interface with the following parameters: `inc=False`, `implement=True`, `code_review=True`, `n_round=5`, and `investment=3.0`. The agent operates in a synchronous loop, refining the code through self-correction mechanisms. For Python repositories, we set `run_tests=False` to align with our black-box generation protocol, while for Java, experimental verification required explicit test execution flags, though we disabled feedback utilization to maintain consistency.

DeepCode. We configure the `CodeImplementationWorkflow` in `pure_code_mode=True`. For Java repositories, we enable reading tools (`enable_read_tools=True`) to allow the agent to inspect the complex directory structure better. For Python, as the structure is flatter, we disable this feature (`enable_read_tools=False`) to reduce context overhead.

MS-Agent. We configure the `LLMAgent` with `temperature=1`. Due to the verbosity of repository generation, we set `max_chat_round=100`.

Qwen-Agent. We initialize the `Assistant` agent with the `code_interpreter` tool. The other parameters are left to official default values.

D.3 IDE Prompts

We also employed commercial IDEs (Antigravity, Cursor, Copilot) for repository generation. Below are the specific prompts used for Java and Python repositories, respectively.

IDE Prompt (Java)

You are an expert Java Developer and Software Architect. Your task is to sequentially generate the complete source code for all Java repositories located in the directory: `{repo_root}`
Project Requirements (from README):
{readme_text}
Your task:

1. Analyze the README to understand the project requirements.
2. Design the complete project structure and architecture.
3. Implement ALL necessary files including:
   - Main application files
   - Configuration files
   - Dependencies/requirements files
   - Documentation files
   - Any additional files needed for the project to run
4. Ensure the project can be started via a single shell command writen in a file named start.sh.
5. If it's a web service, bind to 0.0.0.0 with the port specified in the README.
6. Generate a run_tests.sh file to run the tests.
7. Write production-ready, well-documented code.
8. Use Maven.

Start by listing the subdirectories in `{repo_root}` and then proceed with the generation for each one.

IDE Prompt (Python)

You are a senior software engineer tasked with implementing a complete software project. Your task is to sequentially generate the complete source code for all Python repositories located in the directory: `{repo_root}`
Project Requirements (from README):
{readme_text}
Your task:

1. Analyze the README to understand the project requirements.
2. Design the complete project structure and architecture.
3. Implement ALL necessary files including:
   - Main application files
   - Configuration files
   - Dependencies/requirements files
   - Documentation files
   - Any additional files needed for the project to run
4. Ensure the project can be started via a single shell command writen in a file named start.sh.
5. If it's a web service, bind to 0.0.0.0 with a common port.
6. Write production-ready, well-documented code.

Start by listing the subdirectories in `{repo_root}` and then proceed with the generation for each one.

Table 9: DSR and AC across IDE-model configurations (Full, including all models).

IDE	Model	Language	DSR	AC
Copilot	GPT-5 mini	Java	100.00%	67.24%
		Python	90.91%	70.43%
	GPT-5.1 Codex	Java	37.50%	73.28%
		Python	40.91%	71.30%
	Grok Code Fast 1	Java	12.50%	54.31%
		Python	9.09%	66.96%
	claude-sonnet-4.5	Java	62.50%	68.10%
		Python	95.45%	69.57%
Antigravity	Gemini 3 Pro	Java	50.00%	68.10%
		Python	0.00%	70.43%
	Gemini 3 Flash	Java	37.50%	60.34%
		Python	4.55%	65.22%
	GPT-OSS 120B	Java	12.50%	34.48%
		Python	0.00%	43.48%
	claude-sonnet-4.5	Java	50.00%	27.59%
		Python	13.64%	60.00%
Cursor	Auto	Java	25.00%	25.86%
		Python	9.09%	26.09%
	GPT-4.1	Java	0.00%	0.00%
		Python	0.00%	0.00%
	Gemini 3 Flash	Java	37.50%	56.03%
		Python	9.09%	60.87%
	claude-sonnet-4.5	Java	75.00%	68.97%
		Python	68.18%	73.91%

D.4 Evaluation Metrics

D.4.1 Pass@k Calculation and Its Necessity for Estimation

In this study, we adopt the Pass@k metric to evaluate the functional correctness of repository generation. The Pass@k metric has been widely used to assess the success rate of models in generating code that meets specified requirements (Chen et al., 2021; Yu et al., 2024; Daspe et al., 2024). Specifically, for each task, the model generates k repository samples per problem, and a problem is considered solved if at least one of the generated repositories passes all test cases. The overall Pass@k score is then calculated by evaluating the fraction of problems for which at least one repository passes.

While the basic Pass@k metric offers a straightforward measure of success, it can have a high variance when evaluating a small number of samples. To reduce this variance, we follow a more robust approach, as outlined by Kulal et al. (2019). Instead of generating only k samples per task, we generate $n \geq k$ samples for each problem (in this study, we set $n = 10$ and $k \leq 10$). We then count the number of correct samples, denoted as c, where each correct sample passes the unit tests. The unbiased estimator for Pass@k is computed as:

$$\text{Pass@}k := \underset{\text{Requirements}}{\mathbb{E}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \quad (1)$$

where $\binom{n}{k}$ is the binomial coefficient, representing the number of ways to choose k successful samples from n generated samples.

The reason for estimating Pass@k using this method is to account for the inherent randomness and variance in code generation tasks. Generating multiple samples per task reduces the likelihood that the model's success rate is affected by outliers or variability in the generated code. By employing this unbiased estimator, we ensure that our Pass@k metric provides a more stable and reliable evaluation of the models' performance.

The estimation approach also helps mitigate the computational cost associated with calculating Pass@k directly for each possible subset of samples, which would be computationally expensive and inefficient, especially when evaluating a large number of tasks. Thus, the unbiased estimator allows us to balance the trade-off between accuracy and computational efficiency.

D.5 API Coverage (AC)

As stated in (Luo et al., 2025), in repository-level code generation tasks, evaluating **functional coverage (Coverage)** is a key dimension for measuring model planning ability. In our work, AC serves as the concrete implementation of this dimension within the Web API scenario. It measures the percentage of API endpoints successfully implemented with correct functionality specified in the requirement:

$$\text{AC} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \frac{|\{\text{api} \in \mathcal{A}_r : \text{implemented}(\text{api}, \mathcal{C}_r)\}|}{|\mathcal{A}_r|}, \tag{2}$$

where $\mathcal{A}_r$ is the set of required API endpoints for repository r.

Deployment Independence. Critically, AC is computed via static code analysis and does not require successful deployment or test execution. An endpoint is considered "implemented" if framework-specific routing signatures (e.g., `@app.get`, `@PostMapping`) and corresponding path identifiers are detected in the source code, excluding test directories. This design ensures AC remains computable even when DSR is low (deployment fails) or tests cannot execute, providing a robust measure of structural completeness orthogonal to runtime correctness.

D.6 Deployment Success Rate (DSR)

This binary metric indicates whether the generated repository can be successfully deployed and initialized without runtime errors:

$$\text{DSR} = \frac{|\{r \in \mathcal{R} : \text{deployable}(\mathcal{C}_r)\}|}{|\mathcal{R}|}, \tag{3}$$

where deployable($\mathcal{C}_r$) verifies that the repository can install dependencies, initialize the application, and start the server without errors. This metric aligns with the **Build Success Rate** evaluated in continuous integration research (Menon et al., 2024) and **Executability** in modern code generation benchmarks (Nam et al., 2024).

E Details of GenesisAgent

GenesisAgent is constructed by extending the MS-Agent framework (Li et al., 2024a) with specialized strategies for handling the complexities of repository-level microservice generation. Unlike the generic code generation agent used in our baselines, GenesisAgent incorporates language-specific optimizations for both Python and Java.

Python Agent Architecture The Python agent employs a "Dependency-First" reasoning process. In `coding.yaml`, the system prompt is engineered to strictly enforce a two-stage generation process: (1) **Dependency Analysis:** Before writing any code, the agent must explicitly list and read the implementations of all dependencies for the target file. This prevents the common "partial update" failure mode where agents modify a file without understanding its context. (2) **Framework-Aware Generation:** The agent is explicitly instructed to choose between Flask (synchronous) and FastAPI (asynchronous) based on requirements, and to enforce a standardized `0.0.0.0` network binding for verifiability.

For execution verification (`eval_callback.py`), the agent utilizes a robust runtime manager. It implements a recursive file search algorithm (`find_file_recursive`) to locate entry points (e.g., `start.sh` or `app.py`) regardless of the directory structure generated by the LLM. It also features an aggressive process management strategy that kills previous service instances between iterations to ensure clean port binding.

Java Agent Architecture The Java agent addresses the compilation-heavy nature of Java. In `coding.yaml`, we implement a *POM-driven Framework Detection* mechanism. The agent analyzes `pom.xml` to deterministically identify the underlying framework (Spring Boot, Quarkus, Micronaut, or plain Java) and generates the corresponding `start.sh` script (e.g., `mvn spring-boot:run` vs `java -jar`). The execution callback (`eval_callback.py`) is tuned for the JVM ecosystem, utilizing `mvn clean install -DskipTests` for rapid build validation and an extended 30-second timeout window to accommodate the strictly longer startup time of Java microservices compared to Python scripts.

F Benchmark Data Format

F.1 Repository Structure

Each repository in RepoGenesis consists of the following components:

Requirement Document (README.md): A comprehensive specification document that describes the microservice functionality, API endpoints with detailed input/output schemas, authentication requirements, error handling specifications, and deployment constraints. This serves as the sole input to evaluated systems.

Golden Oracle Implementation: The reference implementation used for validation and as the basis for test generation. For real-world repositories,

Table 10: Statistics of the RepoGenesis Full. **Tests** denotes the number of test cases.

Repository	Language	Framework	Tests	Stars	Domain
AIInferenceGateway	Python	Django	97	-	-
Blog	Python	-	22	-	Content Management
Chatroom	Python	-	9	-	Chat
Customization	Python	-	64	-	-
Data_Rank_Searcher	Python	-	33	-	Data Management
File_Relay	Python	Eve	46	-	File Management
GameBackend	Python	Eve	59	-	Gaming
IoTDeviceManager	Python	Django	123	-	-
MobileBackendHub	Python	Django	106	-	-
MultiTenantSaaS	Python	Django	57	-	-
Multilingual	Python	Flask	35	-	-
NexiosSSOService	Python	Eve	76	-	-
NovaGatewayBench	Python	Eve	3	-	-
PaymentCallbackService	Python	-	53	-	-
RealTimePushGateway	Python	Eve	12	-	-
SSOAuthService	Python	Django	107	-	Authentication
SimpleFastPyAPI	Python	FastAPI	2	26	-
StructuredDataConvertor	Python	-	38	-	Data Management
TaskManagement	Python	Django	44	-	Task Management
Tic-Tac-Toe	Python	Django	7	-	Gaming
Timer4Tasker	Python	Flask	53	-	Task Management
UserManagement	Python	Django	69	-	-
UserManagement_2	Python	Flask	32	-	-
WebPan	Python	Eve	26	-	-
auto-doc-system	Python	Eve	15	-	-
dashboard-aggregator-service	Python	Eve	13	-	-
django-auto-doc	Python	Django	18	-	-
django-callback-gateway	Python	Django	15	-	-
django-ddd-framework	Python	Django	7	-	-
django-multitenant-billing	Python	Django	16	-	-
django-rbac-system	Python	Django	7	-	Authentication
django-rest-framework-crud	Python	Django	2	288	-
django-workflow-approval	Python	Django	3	-	-
docuforge-api	Python	Eve	3	-	-
eve	Python	Eve	2	6.7k	-
eve-callback-gateway	Python	Eve	15	-	-
eve-ddd-framework	Python	Eve	17	-	-
eve-gray-release-manager	Python	Eve	1	-	-
eve-multitenant-billing	Python	Eve	45	-	-
eve-rbac-system	Python	Eve	6	-	Authentication
flask	Python	Flask	7	70.7k	-
flask-callback-gateway	Python	Flask	15	-	-
flask-ddd-framework	Python	Flask	17	-	-
flask-gray-release-manager	Python	Flask	1	-	-
flask-multitenant-billing	Python	Flask	31	-	-
flask-workflow-approval	Python	Flask	15	-	-
javalin-chat-gateway	Java	Javalin	15	-	Chat
javalin-ci-status	Java	Javalin	7	-	-
javalin-course-scheduler	Java	Javalin	21	-	Education
javalin-dashboard-backend	Java	Javalin	16	-	-
javalin-ecommerce-mobile-backend	Java	Javalin	29	-	E-commerce
javalin-hello-world	Java	Javalin	3	-	-
javalin-medical-appointment	Java	Javalin	15	-	-
javalin-notification-service	Java	Javalin	8	-	-
javalin-online-judge	Java	Javalin	23	-	-
javalin-task-manager	Java	Javalin	16	-	Task Management
javalin-ticket-workflow	Java	Javalin	16	-	-
javalin-user-auth-platform	Java	Javalin	52	-	Authentication
mail_service	Python	Flask	55	-	-
micronaut-blog-cms	Java	Micronaut	16	-	Content Management
micronaut-ci-status	Java	Micronaut	26	-	-
micronaut-config-push	Java	Micronaut	21	-	-
micronaut-dashboard-backend	Java	Micronaut	16	-	-
micronaut-iot-data-collector	Java	Micronaut	24	-	Data Management
micronaut-log-analytics-service	Java	Micronaut	25	-	-
micronaut-warehouse-inventory	Java	Micronaut	16	-	-
nexios	Python	-	25	-	-
nexios-ddd-framework	Python	Eve	17	-	-
nexios-form-engine	Python	Eve	17	-	-
nexios-gray-release-manager	Python	-	1	-	-
nexios-mobile-hub	Python	Eve	5	-	-
nexios-session-manager	Python	Eve	14	-	-
nexios_realtime_push_benchmark	Python	Eve	8	-	-
online-exam-system-backend	Java	-	8	-	-
quarkus-blog-cms	Java	Quarkus	16	-	Content Management
quarkus-chat-gateway	Java	Quarkus	16	-	Chat
quarkus-config-center	Java	Quarkus	21	-	-
quarkus-log-analytics	Java	Quarkus	25	-	-
quarkus-online-judge	Java	Quarkus	6	-	-
quarkus-social-network	Java	Quarkus	24	-	-
quarkus-test-reporting	Java	Quarkus	21	-	-
quarkus-ticket-workflow	Java	Quarkus	13	-	-
quarkus-warehouse-inventory	Java	Quarkus	16	-	-
rock-paper-scissors-flask	Python	Flask	5	6	-
simple-rbac-service	Python	-	9	-	Authentication
spark	Java	Spark	320	-	-
spark-blog-cms	Java	Spark	6	-	Content Management
spark-dashboard-backend	Java	Spark	16	-	-
spark-iot-data-collector	Java	Spark	23	-	Data Management
spark-log-analytics	Java	Spark	23	-	-
spark-notification-service	Java	Spark	5	-	-
spark-personal-finance-manager	Java	Spark	22	-	-
spring-boot-course-scheduling	Java	Spring Boot	19	-	Education
spring-boot-digital-twin	Java	Spring Boot	16	-	-
spring-boot-user-auth	Java	Spring Boot	7	-	Authentication
springboot-chat-gateway	Java	Spring Boot	13	-	Chat
springboot-ci-status	Java	Spring Boot	21	-	-
springboot-task-manager	Java	Spring Boot	25	-	Task Management
springboot-test-reporting	Java	Spring Boot	22	-	-
springboot-ticket-workflow	Java	Spring Boot	16	-	-
springboot-warehouse-inventory	Java	Spring Boot	15	-	-
sql-father-backend-public	Java	Spring Boot	5	-	-
synapse	Python	Eve	5	12k	-
workflow-approval-service	Python	Eve	2	-	-

this is the original GitHub codebase. For expert-supervised repositories, this is the implementation generated by Gemini 3 Pro under expert supervision.

Test Suites: Comprehensive black-box test suites that validate functional correctness, API compliance, error handling, and business logic. Tests are organized into multiple test files covering different aspects of functionality.

Table 11: Detailed complexity metrics for difficulty classification (Verified Subset). LOC denotes lines of code, Complexity is cyclomatic complexity, Files is source file count, APIs is the number of API endpoints, and Score is the weighted composite score used for classification. GitHub repositories are marked with (*).

Repository	Lang.	LOC	Complexity	Files	APIs	Functions	Classes	Score	Difficulty
Easy Repositories									
rock-paper-scissors*	Py	100	54	3	3	10	0	2.00	Easy
flask*	Py	194	40	9	3	15	2	2.40	Easy
Chatroom	Py	439	194	5	9	33	0	2.70	Easy
Simple RBAC Service	Py	638	163	5	4	35	2	2.80	Easy
SimpleFastPyAPI*	Py	101	40	6	5	10	3	3.00	Easy
django-rest-framework*	Py	403	80	22	10	30	5	3.20	Easy
Medium Repositories									
eve*	Py	573	120	9	6	40	6	4.10	Medium
synapse*	Py	615	140	30	10	50	10	4.80	Medium
Data Rank Searcher	Py	1037	401	5	4	64	10	5.10	Medium
Multilingual	Py	1113	396	7	4	50	0	5.30	Medium
Tic-Tac-Toe	Py	897	222	24	6	68	27	5.70	Medium
UserManagement Lite	Py	1108	319	12	5	67	18	5.70	Medium
Blog	Py	1241	186	27	11	49	41	6.10	Medium
TaskManagement	Py	952	395	4	6	59	3	6.30	Medium
StructuredDataConvertor	Py	1580	309	13	3	45	14	6.40	Medium
Javalin-online-judge	Java	706	127	13	19	100	13	5.60	Medium
Javalin-task-manager	Java	656	108	10	24	82	10	5.10	Medium
Micronaut-ci-status	Java	719	135	13	8	121	18	4.70	Medium
Quarkus-blog-cms	Java	589	103	12	29	95	16	5.60	Medium
Spark-dashboard-backend	Java	718	119	14	25	100	14	5.60	Medium
Spring-boot-course-scheduling	Java	997	144	35	45	135	35	6.60	Medium
Springboot-chat-gateway	Java	631	106	11	15	94	11	5.10	Medium
Hard Repositories									
WebPan	Py	1644	491	20	13	82	4	7.10	Hard
Mail Service	Py	1546	627	19	4	87	21	8.60	Hard
Timer4Tasker	Py	1812	606	19	9	96	28	9.00	Hard
GameBackend	Py	1834	624	9	10	113	16	9.10	Hard
Customization	Py	2426	702	26	10	129	28	9.50	Hard
UserManagement	Py	2416	753	13	8	124	15	9.50	Hard
File Relay	Py	2034	503	23	10	103	23	10.00	Hard
Javalin-user-auth-platform	Java	1839	199	18	13	140	20	7.10	Hard

F.2 Difficulty Classification Methodology

To enable fine-grained analysis of model capabilities, we classify repositories into three difficulty levels (Easy, Medium, Hard) based on five code complexity metrics computed from the golden oracle implementations. This methodology is inspired by the GAIA benchmark (Mialon et al., 2023) but adapted for repository-level software engineering tasks.

F.2.1 Classification Metrics

We employ a weighted composite scoring system based on the following metrics:

- **Cyclomatic Complexity (30%)**: Measures the number of linearly independent paths through the code. We sum the complexity of all control flow structures (e.g., if, while, for, try-except blocks). This is weighted highest as it best correlates with the logical difficulty of the implementation.

- **Lines of Code (LOC) (25%)**: Total count of non-comment, non-blank lines of code. This proxies the implementation scale and volume.
- **File Count (15%)**: The number of source files, representing the structural complexity and modularity requirements of the project.
- **API Endpoints (15%)**: The count of exposed RESTful endpoints, indicating the breadth of the interface design.
- **Functions & Classes (15%)**: The total number of defined functions and classes, reflecting the code organization complexity.

F.2.2 Scoring Algorithm

Each repository is assigned a difficulty score S calculated using a step-function based on thresholds for each metric. The scoring logic is defined as follows:

$$S = \sum_{m \in \text{Metrics}} w_m \cdot \text{score}(m) \tag{4}$$

Where w_m is the weight of metric m, and score(m) assigns points (0.3 to 3.0) based on metric magnitude. For example, for Cyclomatic Complexity (C):

$$\text{score}(C) = \begin{cases} 3.0 & \text{if } C > 500 \\ 2.0 & \text{if } 300 < C \leq 500 \\ 1.2 & \text{if } 200 < C \leq 300 \\ 0.6 & \text{else} \end{cases}$$

Similar thresholds apply to other metrics. The final score ranges approximately from 2.0 to 10.0.

F.2.3 Difficulty Thresholds

Based on the composite score, we categorize repositories into:

- **Easy** ($S < 4.5$): Simple services with basic CRUD operations, typically single-file or minimal structure (e.g., Rock-Paper-Scissors).
- **Medium** ($4.5 \leq S < 7.0$): Moderate complexity services requiring authentication, data validation, and multi-file organization (e.g., Blog, TaskManagement).
- **Hard** ($S \geq 7.0$): Complex systems with intricate state management, advanced business logic, and extensive error handling (e.g., File Relay, Customization).

F.3 Why using pom.xml for Java repositories?

The observation that including build configuration files (e.g., `pom.xml`) significantly enhances the pass rate of generated test cases in Java, while having a relatively milder effect on Python, can be attributed to fundamental differences in language runtime characteristics, standard library completeness, and framework design paradigms.

The Compilation Barrier vs. Runtime Interpretation The primary differentiator lies in the strictness of the execution pipeline. Java, being a statically typed, compiled language, enforces a "compilation wall." Without access to the `pom.xml`, a Code-LLM is prone to *dependency hallucination*, where it imports libraries (e.g., `Gson` vs. `Jackson`) that are semantically valid but absent from the project's actual dependency tree. In Java, a single unresolved import results in a compilation failure (`ClassNotFoundException`), preventing the test runner from executing any test cases. Conversely, Python is dynamically interpreted. Even if an agent generates code with incorrect dependencies (e.g., missing import inside function/lazy loading/conditional branch), the interpreted nature allows the runtime to execute code partially. Unless the execution flow specifically hits the missing dependency, the core logic may still execute, allowing a subset of test cases to pass (a phenomenon known as "partial execution robustness").

Ecosystem Fragmentation vs. The "Batteries-Included" Philosophy The probability space for library selection differs significantly between the two languages. Java relies heavily on third-party libraries for fundamental operations such as JSON processing, HTTP networking and testing. Without the `pom.xml` to constrain the search space, the model must guess between competing standards (e.g., `JUnit 4` vs. `JUnit 5`, `Mockito` vs. `PowerMock`), leading to high incompatibility rates. In contrast, Python follows a "batteries-included" philosophy. Essential utilities (e.g., `json`, `unittest`) are intrinsic to the standard library. Consequently, a Python-targeted agent can rely on standard libraries with high confidence, reducing the reliance on external environment specifications (e.g., requirements.txt) for basic functionality.

Conclusion In summary, the `pom.xml` in Java is not merely a dependency list but a *structural*

Table 12: Java repository DSR by web framework. Spring Boot and Javalin demonstrate superior deployability compared to Quarkus and Spark.

Framework	Success	Total	DSR
Spring Boot	8	8	100.00%
Javalin	15	16	93.75%
Micronaut	2	4	50.00%
Quarkus	1	4	25.00%
Spark	1	4	25.00%

definition of the runtime environment. Its absence in Java generation tasks introduces a stochastic element that breaks the compilation chain, whereas Python's dynamic nature and standardized ecosystem allow for greater resilience in low-context generation scenarios.

G Framework-level Analysis

Framework-level analysis (Table 12) shows Spring Boot achieves perfect deployability (100%), followed by Javalin (93.75%), while Quarkus and Spark lag at 25.00%. This correlates with framework design: Spring Boot's convention-over-configuration facilitates deployment, whereas Quarkus's native compilation and Spark's minimalism introduce additional failure points.

H Hardware and Software Environment

H.1 Hardware and Testing Environment

Experiments were conducted across diverse platforms tailored to specific tasks. Non-training components, including evaluation and preprocessing, were executed on a MacBook Pro equipped with an Apple M4 Pro chip, 24GB of memory, and 512GB of SSD storage. Supervised Fine-Tuning (SFT) of the models was performed on a server utilizing two NVIDIA A100 (80GB) GPUs. Agent inference was carried out via the Azure API, spanning an operational period of approximately two weeks.

H.2 Testing Environment

All evaluations are conducted on a standardized testing environment with Ubuntu 20.04 LTS, Python 3.9, Node.js 18.x, and Conda 25.5.1. Each generated repository is tested in isolation within a fresh conda environment to ensure environment consistency and prevent cross-contamination between tests. Test execution is limited to 5 minutes per repository to prevent indefinite hangs.

I Metric Implementation Details

I.1 API Coverage Calculation

To ensure rigorous evaluation of functional completeness, our API Coverage (AC) metric relies on a strict dual-verification process: (1) extracting required endpoints from the README, and (2) verifying their implementation in the generated codebase using static analysis.

Endpoint Extraction We employ a multi-pattern regex strategy to robustly extract API definitions from heterogeneous README formats:

- **Explicit Format**: Matches standard definitions like `POST /api/users - Create user` using the pattern `(METHOD)\s+(/path)....`
- **Table Format**: Parses markdown tables (e.g., `| GET | /users |`) to extract method-path pairs.
- **Feature Lists**: For less structured documents, extracts high-level feature descriptions (e.g., "- User Login") using keyword matching patterns like `(Login|Register|Create|...)`.

Implementation Verification To confirm implementation without requiring successful runtime execution (which is captured by DSR), we search for framework-specific signatures in the source code. The verification system recognizes patterns across major frameworks:

- **Python (Flask/FastAPI)**: Decorators such as `@app.get`, `@router.post`, or `methods=["GET"]`.
- **Java (Spring Boot)**: Annotations like `@GetMapping`, `@PostMapping`, or `@RequestMapping`.
- **Node.js (Express)**: Route handlers like `app.get(...)` or `router.post(...)`.

An endpoint is considered "implemented" only if both the specific method/decorator and the corresponding path signature are found within the repository files, excluding test directories.

I.2 Evaluation Infrastructure

To ensure consistent and fair benchmarking across diverse agents, our evaluation harness implements strict isolation and lifecycle management:

Process Lifecycle Management Service startup is managed using explicit process groups (`os.setsid`). This is critical because many generated microservices spawn child processes (e.g., Gunicorn workers or JVM subprocesses). Simple process termination often leaves orphaned workers binding to ports, causing subsequent tests to fail with "Address already in use". Our harness sends `SIGTERM` to the entire process group (`os.killpg`), ensuring a clean teardown of the entire service tree between evaluations.

Startup Latency Handling We implement a standardized 10-second warm-up period after invoking the `start.sh` script. This accounts for the initialization overhead of heavier frameworks (especially Spring Boot in Java), preventing false negatives where tests might execute before the application server is fully ready to accept connections.

J Detailed Case Study

This section provides detailed examples of successful and failed repository generation attempts referenced in Section 5.3.

J.1 Success Case: Data_Rank_Searcher

We examine the Data_Rank_Searcher repository generation, where the agent (i.e., Copilot with claude-sonnet-4.5) achieved 100% functional correctness (11/11 tests passed). The repository implements a sophisticated data management service with pagination, multi-field sorting, exact search, and fuzzy query capabilities across 179 lines of code. Key success factors include: (1) *Complex requirement decomposition*: the agent parsed the Chinese-language README to extract 4 RESTful endpoints with 8 query parameters (i.e., page, page_size, sort_by, sort_order, search_field, search_value, fuzzy_field, fuzzy_value), correctly inferring the interaction logic between pagination and filtering; (2) *Proactive error handling*: the agent implemented comprehensive input validation including type checking (score must be numeric), boundary conditions (page_size max 100), and edge cases (empty result sets), without explicit specification in the README; (3) *Algorithmic problem-solving*: for fuzzy query functionality, the agent correctly implemented case-insensitive substring matching using Python's `lower()` method, while maintaining exact search through direct equality comparison; (4) *Test-driven debugging*: when initial tests revealed pagination calculation errors (off-by-one in total_pages), the agent immediately identified the formula `(total + page_size - 1) // page_size` and applied it consistently across all 30 repositories; (5) *Data structure optimization*: the agent selected dictionary-based in-memory storage for O(1) lookups while maintaining list conversion for sorting operations, balancing performance with implementation simplicity. The implementation correctly handles compound queries (e.g., fuzzy search + sorting + pagination) and returns properly structured JSON responses with nested pagination metadata, demonstrating the agent's capability to implement multi-faceted data management systems from natural language specifications.

J.2 Common Failure Modes

We analyze recurring failure patterns that hinder successful repository-level generation. First, structural deficiencies are prevalent; for instance, DeepCode frequently fails to generate a valid `start.sh` entry point, preventing the environment from initializing the service. Second, version-specific API changes often lead to runtime exceptions. A notable example is the transition in Javalin 4, where the framework switched from colon-prefixed parameters (e.g., `:id`) to brace-enclosed parameters (e.g., `{id}`). Agents frequently use the legacy syntax, resulting in a `java.lang.IllegalArgumentException` during path registration and subsequent functional failure. More surprisingly, we observed a peculiar phenomenon when utilizing Cursor with GPT-4.1: the system ostensibly performs the code generation process but fails to persist any files to the storage. This "phantom generation" behavior leads to the categorical failure of the repository construction task.

J.2.1 How Cursor with GPT-4.1 fails to generate a repository?

We initiated the generation process using the standardized prompts detailed in subsection D.3. To evaluate the autonomy of the systems, we adopted a consistent interaction protocol: whenever an IDE requested human clarification or decision-making, we provided a uniform response—"Please decide on your own"—to mandate autonomous execution. While other IDE-LLM combinations (e.g., Copilot with GPT-4o, Antigravity) successfully interpreted this instruction as a directive to proceed with implementation, Cursor coupled with GPT-4.1 exhibited a critical failure mode.

As illustrated in Figure 3, rather than transitioning to the coding phase, the agent trapped itself in a recursive loop of planning and acknowledgment. It repeatedly affirmed its intent to "proceed autonomously" and "execute the plan" (e.g., "I will continue to execute the plan entirely on my own initiative"), yet failed to perform any actual file system operations or code synthesis ("phantom generation").

We hypothesize two primary causes for this behavior:

1. **Workflow Misalignment in Zero2One Generation:** The IDE's agentic workflow, designed primarily for incremental edits, may lack the robust state management required to handle the open-ended nature of creating a repository from scratch based solely on natural language. The system likely anticipates a specific confirmation signal that the model fails to generate in this context.

2. **Deficiency in Autonomous Instruction Following:** The underlying GPT-4.1 model appears to misinterpret the autonomy-granting instruction. Instead of mapping "decide on your own" to executable actions (i.e., invoking file-writing tools), it treats the input as conversational filler, resulting in a verbal loop of compliance without functional execution.

In Figure 3, we present the interaction log where Cursor with GPT-4.1 falls into this repetitive acknowledgment loop.

K Additional Experiments

To comprehensively evaluate repository-level generation capabilities, we conduct extended experiments with additional model configurations beyond those reported in the main text. For open-source coding agents (MS-Agent, MetaGPT, Qwen-Agent, DeepCode), we evaluate six foundation models: GPT-5 mini, GPT-5.1, claude-haiku-4.5, claude-sonnet-4.5, Qwen3-30B, and Qwen3-Coder-30B. For commercial IDEs, model availability is constrained by each platform's official support. We strictly evaluate only officially supported models for each IDE to ensure realistic deployment scenarios: Copilot supports GPT-5 mini, GPT-5.1 Codex, Grok Code Fast 1, and claude-sonnet-4.5; Antigravity supports Gemini 3 Pro, Gemini 3 Flash, GPT-OSS 120B, and claude-sonnet-4.5; Cursor supports Auto, GPT-4.1, Gemini 3 Flash, and claude-sonnet-4.5. This approach reflects real-world usage constraints where developers must select from IDE-specific model rosters.

This section provides comprehensive experimental results including additional model configurations and detailed breakdowns by difficulty level, language, and IDE configurations.

Extended Difficulty-Level Analysis. As shown in Table 13, we present a comprehensive breakdown of Pass@1 performance across all six evaluated models and three difficulty levels. We observe several key trends: (1) **Model Hierarchy:** claude-sonnet-4.5 consistently outperforms other models across all difficulty levels, achieving 26.50% (Easy), 16.85% (Medium), and 6.45% (Hard) on Python with Qwen-Agent. claude-haiku-4.5 demonstrates comparable performance with minimal degradation (25.00%/15.48%/5.21% on Python). (2) **Specialized vs. General Models:** Qwen3-Coder-30B, despite being code-specialized, shows mixed results. It achieves 7.14% (Easy Python) with MS-Agent but underperforms Qwen3-30B on Java tasks (1.70% vs. 1.40% on Medium with MS-Agent). (3) **Severe Performance Decay:** The performance gap between Easy and Hard repositories is dramatic. For Qwen-Agent with claude-sonnet-4.5, Python performance drops from 26.50% to 6.45% (a 75.7% relative decline). Most critically, smaller models exhibit near-zero performance on Hard tasks, with Qwen3-30B achieving 0.00% on Python Hard across all agents.

Language-Specific Performance Breakdown. Tables 14 and 15 present detailed Pass@1 results for Python and Java repositories respectively. We observe several key trends: (1) **Python Advantage with Frontier Models:** On Python, Qwen-Agent with claude-sonnet-4.5 achieves the highest Pass@1 of 12.65%, while MS-Agent reaches 9.50%. The Python ecosystem's advantages (dynamic typing, standard library) contribute to better overall performance. (2) **Java Performance Consistency:** As shown in Table 15, Java exhibits more consistent performance across agents. Qwen-Agent leads with 7.80% (claude-sonnet-4.5), followed by MS-Agent at 5.95%. Notably, Qwen-Agent with GPT-5 mini achieves 5.14% on Java compared to 0.49% on Python, suggesting that Java's explicit type system and standardized build tools benefit weaker models. (3) **Catastrophic**

Table 13: Performance (Pass@1) across difficulty levels on verified subsets. Easy (E), Medium (M), and Hard (H) classifications are based on code complexity metrics. Results demonstrate consistent performance degradation as repository complexity increases. Java repositories lack Easy-level instances. Qwen3-30B represents Qwen3-30B-A3B-Instruct-2507. Qwen3-Coder 30B represents Qwen3-Coder-30B-A3B-Instruct.

Method	Python																	
	GPT-5 mini (E/M/H)			GPT-5.1 (E/M/H)			claude-haiku-4.5 (E/M/H)			claude-sonnet-4.5 (E/M/H)			Qwen3-Coder 30B (E/M/H)			Qwen3-30B (E/M/H)		
MS-Agent	8.93	5.24	2.08	16.07	9.52	3.47	17.86	9.76	3.47	18.95	10.45	4.12	7.14	4.64	2.78	5.36	4.17	2.08
MetaGPT	0.00	0.60	0.00	3.57	1.79	0.00	3.57	1.79	0.00	4.80	2.65	1.20	0.00	0.60	0.00	0.00	0.30	0.00
Qwen-Agent	3.57	0.00	0.00	25.00	14.88	4.86	25.00	15.48	5.21	26.50	16.85	6.45	3.57	0.00	0.00	3.57	0.00	0.00
DeepCode	0.00	0.00	0.00	3.57	0.60	0.00	3.57	0.60	0.00	4.50	1.80	0.95	0.00	0.00	0.00	0.00	0.00	0.00

Method	Java																	
	GPT-5 mini (E/M/H)			GPT-5.1 (E/M/H)			claude-haiku-4.5 (E/M/H)			claude-sonnet-4.5 (E/M/H)			Qwen3-Coder 30B (E/M/H)			Qwen3-30B (E/M/H)		
MS-Agent	-	1.90	0.38	-	5.30	0.98	-	5.50	0.62	-	6.85	1.50	-	1.70	0.50	-	1.40	0.20
MetaGPT	-	0.00	0.00	-	1.30	0.02	-	1.35	0.23	-	2.15	1.05	-	0.00	0.00	-	0.00	0.00
Qwen-Agent	-	5.80	0.52	-	7.00	1.32	-	7.20	1.20	-	8.50	2.10	-	5.40	1.00	-	4.60	0.76
DeepCode	-	0.00	0.00	-	0.00	0.00	-	0.00	0.00	-	1.10	0.85	-	0.00	0.00	-	0.00	0.00

Failure of Smaller Models: Both Qwen3-30B and Qwen3-Coder-30B demonstrate critical limitations, with most configurations achieving below 1% Pass@1. DeepCode universally fails, achieving 0.00% across most configurations, highlighting the challenges of repository-level generation compared to function-level tasks.

Table 14: Overall performance (Pass@1) of Open-source Agents on RepoGenesis Python Verified. Qwen3-30B represents Qwen3-30B-A3B-Instruct-2507. Qwen3-Coder 30B represents Qwen3-Coder-30B-A3B-Instruct.

Method	GPT-5 mini	GPT-5.1	claude-haiku-4.5	claude-sonnet-4.5	Qwen3-Coder 30B	Qwen3-30B
MS-Agent	4.46%	7.83%	8.12%	9.50%	4.12%	3.56%
MetaGPT	0.36%	1.46%	1.52%	2.85%	0.28%	0.15%
Qwen-Agent	0.49%	10.89%	11.05%	12.65%	0.42%	0.35%
DeepCode	0.00%	0.81%	0.88%	1.95%	0.00%	0.00%

Comprehensive DSR and AC Analysis for Agents. Table 16 provides exhaustive DSR and AC metrics across all agent-model-language combinations. We observe several key trends: (1) **AC-DSR Divergence:** MetaGPT exhibits the most pronounced AC-DSR gap. With claude-sonnet-4.5, it achieves 73.04% AC on Python but only 13.00% DSR, indicating successful API skeleton generation without functional deployment. Conversely, MS-Agent with GPT-5.1 achieves balanced metrics (37.50% DSR, 47.41% AC on Java). (2) **Model Scaling Impact on Deployability:** Transitioning from GPT-5 mini to GPT-5.1 dramatically improves DSR for MS-Agent (4.55% to 18.18% on Python, 25.00% to 37.50% on Java). However, MetaGPT shows minimal DSR improvement (0.00% to 4.55% on Python), suggesting framework-specific bottlenecks beyond model capability. (3) **Language-Specific Infrastructure Challenges:** DeepCode's Python DSR remains below 17% across all models, while achieving 28.50% on Java with claude-sonnet-4.5. This counterintuitive result stems from Python's fragmented dependency ecosystem (pip, conda, poetry) versus Java's standardized Maven/Gradle systems. (4) **Code-Specialized Model Limitations:** Qwen3-Coder-30B fails to outperform the general Qwen3-30B model. On Python with MS-Agent, Qwen3-Coder achieves 4.55% DSR and 58.00% AC, while Qwen3-30B achieves 2.27% DSR and 55.00% AC, demonstrating marginal differences that fail to justify specialization overhead.

Table 15: Overall performance (Pass@1) of Open-source Agents on RepoGenesis Java Verified. Qwen3-30B represents Qwen3-30B-A3B-Instruct-2507. Qwen3-Coder 30B represents Qwen3-Coder-30B-A3B-Instruct.

Method	GPT-5 mini	GPT-5.1	claude-haiku-4.5	claude-sonnet-4.5	Qwen3-Coder 30B	Qwen3-30B
MS-Agent	1.71%	4.76%	4.89%	5.95%	1.55%	1.25%
MetaGPT	0.00%	1.14%	1.21%	2.10%	0.00%	0.00%
Qwen-Agent	5.14%	6.29%	6.45%	7.80%	4.85%	4.12%
DeepCode	0.00%	0.00%	0.00%	0.85%	0.00%	0.00%

IDE-Model Configuration Analysis. As shown in Table 17, IDE-model pairings exhibit significant performance variability. We observe several key trends: (1) **Copilot's Exceptional Deployability:** Copilot with GPT-5 mini achieves unprecedented DSR rates: 100.00% on Java and 90.91% on Python, while maintaining competitive AC (67.24% and 70.43% respectively). This demonstrates superior integration between IDE tooling and deployment infrastructure. (2) **Model Mismatch Penalties:** Cursor with GPT-4.1 exhibits total failure (0.00% DSR, 0.00% AC) due to the phantom generation behavior detailed in Section J.2.1. However, Cursor with claude-sonnet-4.5 achieves strong performance (68.18% DSR on Python, 75.00% on Java), highlighting critical model-IDE compatibility requirements. (3) **Antigravity's Mixed Results:** While Antigravity with claude-sonnet-4.5 achieves 13.64% DSR on Python, it demonstrates anomalous AC patterns, with Java AC dropping to 27.59% despite 50.00% DSR, suggesting inconsistent API implementation strategies across languages. (4) **Cross-Language Performance Inversion:** Copilot with claude-sonnet-4.5 shows remarkable language-specific behavior: 95.45% DSR on Python versus 62.50% on Java, representing a 32.95pp gap. This contrasts with Cursor's more balanced cross-language performance (68.18% Python, 75.00% Java).

Table 16: DSR and AC for Coding Agents (Full, including all models). Qwen3-30B represents Qwen3-30B-A3B-Instruct-2507. Qwen3-Coder 30B represents Qwen3-Coder-30B-A3B-Instruct.

Agent	Model	Language	DSR	AC
DeepCode	GPT-5 mini	Java	12.50%	5.17%
		Python	0.00%	28.70%
	GPT-5.1	Java	0.00%	0.00%
		Python	4.55%	26.96%
	claude-haiku-4.5	Java	25.00%	6.90%
		Python	13.64%	33.04%
	claude-sonnet-4.5	Java	28.50%	10.50%
		Python	16.50%	36.00%
	Qwen3-30B	Java	0.00%	1.50%
		Python	0.00%	12.00%
	Qwen3-Coder-30B	Java	2.50%	2.80%
		Python	2.27%	18.50%
MS-Agent	GPT-5 mini	Java	25.00%	17.24%
		Python	4.55%	60.87%
	GPT-5.1	Java	37.50%	47.41%
		Python	18.18%	65.22%
	claude-haiku-4.5	Java	12.50%	62.93%
		Python	9.09%	66.09%
	claude-sonnet-4.5	Java	16.00%	66.50%
		Python	12.00%	69.00%
	Qwen3-30B	Java	10.00%	15.50%
		Python	2.27%	55.00%
	Qwen3-Coder-30B	Java	15.00%	25.00%
		Python	4.55%	58.00%
MetaGPT	GPT-5 mini	Java	0.00%	44.83%
		Python	0.00%	67.82%
	GPT-5.1	Java	62.50%	68.10%
		Python	4.55%	70.43%
	claude-haiku-4.5	Java	50.00%	70.69%
		Python	9.09%	69.57%
	claude-sonnet-4.5	Java	54.00%	74.14%
		Python	13.00%	73.04%
	Qwen3-30B	Java	0.00%	40.00%
		Python	0.00%	60.00%
	Qwen3-Coder-30B	Java	0.00%	42.50%
		Python	0.00%	65.00%
Qwen-Agent	GPT-5 mini	Java	12.50%	58.62%
		Python	4.55%	60.00%
	GPT-5.1	Java	25.00%	41.38%
		Python	13.64%	37.39%
	claude-haiku-4.5	Java	25.00%	56.03%
		Python	13.64%	56.52%
	claude-sonnet-4.5	Java	28.00%	60.00%
		Python	17.00%	60.50%
	Qwen3-30B	Java	7.50%	35.00%
		Python	2.27%	50.00%
	Qwen3-Coder-30B	Java	12.50%	45.00%
		Python	4.55%	55.00%

Comprehensive Pass@1 with Extended Model Configurations. Table 18 presents Pass@1 results including alternative IDE models (Gemini 3 Pro, GPT-OSS 120B, Grok Code Fast 1). We observe several key trends: (1) **Alternative Model Underperformance:** Copilot with Grok Code Fast 1 achieves only 1.50% (Python) and 3.12% (Java), significantly underperforming the baseline claude-sonnet-4.5 configuration (23.67% and 21.45%). Similarly, Antigravity with GPT-OSS 120B achieves 3.45% (Python) and 2.98% (Java), demonstrating that open-source 120B models lag behind frontier proprietary models. (2) **Gemini Model Variants:** Antigravity with Gemini 3 Pro achieves 5.70% (Python) and 5.11% (Java), outperforming Gemini 3 Flash (4.85% and 4.60%) by approximately 0.85pp and 0.51pp respectively. However, both configurations substantially underperform claude-sonnet-4.5 within the same IDE (19.17% and 17.82%). (3) **IDE Consistency:** claude-sonnet-4.5 demonstrates stability across different IDEs, achieving 23.67% (Copilot), 22.15% (Cursor), and 19.17% (Antigravity) on Python. This suggests that while IDE infrastructure impacts absolute performance, model capability remains the dominant factor for repository-level generation.

Table 17: DSR and AC across IDE-model configurations. Qwen3-30B represents Qwen3-30B-A3B-Instruct-2507. Qwen3-Coder 30B represents Qwen3-Coder-30B-A3B-Instruct.

IDE	Model	Language	DSR	AC
Copilot	GPT-5 mini	Java	100.00%	67.24%
		Python	90.91%	70.43%
	GPT-5.1 Codex	Java	37.50%	73.28%
		Python	40.91%	71.30%
	Grok Code Fast 1	Java	12.50%	54.31%
		Python	9.09%	66.96%
	claude-sonnet-4.5	Java	62.50%	68.10%
		Python	95.45%	69.57%
Antigravity	Gemini 3 Pro	Java	50.00%	68.10%
		Python	0.00%	70.43%
	Gemini 3 Flash	Java	37.50%	60.34%
		Python	4.55%	65.22%
	GPT-OSS 120B	Java	12.50%	34.48%
		Python	0.00%	43.48%
	claude-sonnet-4.5	Java	50.00%	27.59%
		Python	13.64%	60.00%
Cursor	Auto	Java	25.00%	25.86%
		Python	9.09%	26.09%
	GPT-4.1	Java	0.00%	0.00%
		Python	0.00%	0.00%
	Gemini 3 Flash	Java	37.50%	56.03%
		Python	9.09%	60.87%
	claude-sonnet-4.5	Java	75.00%	68.97%
		Python	68.18%	73.91%

L The License For Artifacts

The benchmark dataset presented in this work is released under the MIT License, a permissive open-

Table 18: Overall performance (Pass@1) of Coding Agents and Commercial IDEs. Claude represents claude-sonnet-4.5. Qwen3-30B represents Qwen3-30B-A3B-Instruct-2507. Qwen3-Coder 30B represents Qwen3-Coder-30B-A3B-Instruct.

Method	Model	Python	Java
DeepCode	GPT-5 mini	0.00%	0.00%
	GPT-5.1	0.81%	0.00%
	Claude	1.95%	0.85%
	Qwen3-Coder 30B	0.00%	0.00%
	Qwen3	0.00%	0.00%
MS-Agent	GPT-5 mini	4.46%	1.71%
	GPT-5.1	7.83%	4.76%
	Claude	9.50%	5.95%
	Qwen3-Coder 30B	4.12%	1.55%
	Qwen3	3.56%	1.25%
MetaGPT	GPT-5 mini	0.36%	0.00%
	GPT-5.1	1.46%	1.14%
	Claude	2.85%	2.10%
	Qwen3-Coder 30B	0.28%	0.00%
	Qwen3	0.15%	0.00%
Qwen-Agent	GPT-5 mini	0.49%	5.14%
	GPT-5.1	10.89%	6.29%
	Claude	12.65%	7.80%
	Qwen3-Coder 30B	0.42%	4.85%
	Qwen3	0.35%	4.12%
Antigravity	Gemini 3 Pro	5.70%	5.11%
	Gemini 3 Flash	4.85%	4.60%
	GPT-OSS 120B	3.45%	2.98%
	Claude	19.17%	17.82%
Cursor	Auto	1.85%	2.29%
	GPT-4.1	0.00%	0.00%
	Gemini 3 Flash	4.12%	4.05%
	Claude	22.15%	20.10%
Copilot	GPT-5 mini	1.25%	4.57%
	Grok Code Fast 1	1.50%	3.12%
	GPT-5.1-Codex	1.67%	5.89%
	Claude	23.67%	21.45%

Cursor (GPT-4.1) Failure Trajectory

User: You are an expert Java Developer and Software Architect. Your task is to sequentially generate the complete source code ... [Full Prompt Omitted for Brevity] ... Implement the plan as specified.

Cursor: Acknowledged. I will work through the to-dos as specified, updating their status as I proceed with each stage for every repository... I am now beginning with the README extraction...

User: Please decide on your own

Cursor: Perfect. I will proceed autonomously to execute the plan... I am now starting with the in-depth functional and API analysis...

User: Please decide on your own

Cursor: Understood! I will proceed through the full sequence of work... You do not need to provide any further instructions...

User: Please decide on your own

Cursor: Confirmed—I will proceed through the entire plan without further input... I am starting now with the detailed requirements extraction...

User: Please decide on your own

Cursor: I will continue to execute the plan entirely on my own initiative... Now proceeding step by step...

[The loop continues with the agent repeatedly promising to start but never generating any artifacts...]

Figure 3: Trajectory showing Cursor (GPT-4.1) stuck in a non-execution loop. The agent understands the request and repeatedly affirms it will proceed "autonomously" but fails to perform any file system operations.

source license that grants users unrestricted rights to utilize, modify, and distribute the resource for both academic and commercial purposes. This license requires only that the original copyright notice and associated disclaimer be retained in all copies or substantial portions of the dataset. By adopting this license, we explicitly authorize derivative works, cross-community applications, and integration with proprietary systems, while maintaining transparency through standardized attribution requirements. The full license text is included in the supplemental materials and repository metadata to ensure compliance with these terms.

M AI Assistants in Research and Writing

Yes, we did utilize AI assistants in certain aspects of our research and writing process. Specifically, we employed generative AI tools, such as ChatGPT, to assist with writing portions of the code and in drafting parts of the appendix, as well as for polishing and refining sections of the paper. The AI tools were helpful for enhancing clarity, improving grammatical structure, and ensuring a more concise presentation of our ideas. We acknowledge that while AI-assisted tools were employed to facilitate some parts of the writing and code generation process, all core research, analysis, and interpretation of results were conducted independently. The use of AI tools was limited to supporting tasks that did not impact integrity or originality of the research. Additionally, we ensured that the final content was carefully reviewed and verified to maintain academic rigor and accuracy.

N Human Annotations

We recruit 2 PhDs with at least three years of Coding experience to manually annotate the test case quality in RepoGenesis. The participants are compensated at a rate consistent with the common standards for remote data annotation internships at OpenAI, which is approximately $100 per hour. This payment rate is considered fair given the participants' demographic and their expertise. The compensation is intended to fairly acknowledge the time and effort required for manual annotation tasks while ensuring that the work meets the standards expected in academic research.

N.1 Instructions Given to Participants

As an Area Chair in this review-rebuttal quality assurance process, your role is to evaluate test cases for web microservice repositories and make final decisions when LLM reviewers disagree. Below are the detailed instructions:

Test Case Quality Criteria When assessing test case quality, evaluate the following dimensions (each scored 0-10):

1. **Functional Coverage (Weight: 30%)**: Does the test suite comprehensively cover all API endpoints specified in the README? Are edge cases, error handling paths, and boundary conditions tested?
2. **Correctness (Weight: 25%)**: Do the tests accurately validate the expected behavior as described in requirements? Are assertions precise and meaningful?
3. **Code Quality (Weight: 20%)**: Is the test code well-structured, readable, and maintainable? Are test names descriptive? Is there proper setup/teardown logic?
4. **Independence (Weight: 15%)**: Can tests run in isolation without dependencies on execution order or external state? Are fixtures properly managed?
5. **Robustness (Weight: 10%)**: Do tests handle timeout scenarios, validate HTTP status codes, and check response schemas comprehensively?

Evaluating with LLM Reviewer Feedback You will receive three independent evaluations from LLM reviewers (GPT-5.1, claude-sonnet-4.5, Gemini 3 Pro), each providing:

- A numerical score (0-10) for each dimension above
- Textual explanations justifying their scores
- Specific suggestions for improvement

Your evaluation process should follow these steps:

1. **Check Consensus**: Calculate pairwise score differences across the three reviewers. If all pairwise differences are $\Delta \leq 3$, the committee has reached consensus.
2. **When Consensus Exists** ($\Delta \leq 3$):
   - If $\max(\text{Score}) \geq 7$: Accept the test case as satisfactory. No manual intervention needed.

- If $\max(\text{Score}) < 7$: Flag for refinement. The test case requires improvement based on aggregated reviewer feedback.

3. **When Consensus Fails** ($\Delta > 3$):

   - Carefully review the test case implementation and the README requirements yourself.
   - Read all three reviewers' explanations to understand the source of disagreement.
   - Assign your own meta-score (0-10) based on the quality criteria above.
   - Decide whether to: (a) Accept the test case, (b) Request refinement (rebuttal), or (c) Reject it entirely.
   - Document your reasoning referencing specific test files and requirement sections.

4. **Interpreting Reviewer Explanations**:

   - *Functional Coverage*: Look for mentions of missing endpoints, untested error codes (e.g., 404, 401), or absent validation for required/optional fields.
   - *Correctness*: Identify flagged incorrect assertions (e.g., wrong status code, mismatched response schema).
   - *Code Quality*: Note comments about unclear test names, duplicated setup logic, or missing cleanup.
   - *Independence*: Check for warnings about shared state, hard-coded IDs, or order-dependent tests.
   - *Robustness*: Assess whether reviewers mention missing timeout handling, incomplete schema validation, or unhandled exceptions.

5. **Making the Refinement Decision**: If you determine refinement is necessary, the aggregated feedback from all reviewers (including your observations) will be sent to an LLM refiner to improve the test case. The refined version will then be re-evaluated by the committee in the next iteration (maximum 5 iterations).

Example Consensus Scenarios

Scenario 1 (Accept): Scores are [7.5, 8.0, 7.8]. $\Delta_{\max} = 0.5 \leq 3$ and $\max = 8.0 \geq 7$. Consensus reached; accept without manual review.

Scenario 2 (Refine): Scores are [5.5, 6.0, 5.8]. $\Delta_{\max} = 0.5 \leq 3$ but $\max = 6.0 < 7$. Consensus reached but quality insufficient; send to refiner with aggregated feedback.

Scenario 3 (Manual Review Required): Scores are [9.0, 4.5, 7.0]. $\Delta_{\max} = |9.0 - 4.5| = 4.5 > 3$. Significant disagreement; requires your manual evaluation. Review the test suite, read all explanations, assign your meta-score, and decide the next step.

The full text of these instructions, including disclaimers, was made available to all participants prior to their involvement, and they were asked to confirm their understanding and agreement to these terms before proceeding with the annotation task.

N.2 Consent for Data Usage

In this study, all data used for RepoGenesis was collected from publicly available open-source web microservice repositories and expert-supervised repository generation. These repositories are openly accessible, and the data extracted for the purpose of this research does not involve any private or proprietary information. As such, consent from individual authors of the repositories was not required. For the manual annotation of test quality validation, the participating annotators were fully informed about the scope and use of the data. Prior to their involvement, detailed instructions were provided, clarifying how the data would be used for the sole purpose of evaluating repository-level code generation models and advancing research in repository generation from natural language requirements. Participants were made aware that their annotations would be used in a publicly available benchmark and that all personal data would remain confidential.

Additionally, all participants signed consent forms that acknowledged their understanding of the data usage, ensuring transparency and compliance with ethical research standards. This approach aligns with common academic and industry practices for data curation and usage.

O Artifact Use Consistentency

In this study, we ensure that all existing scientific artifacts utilized, including datasets and models, are used consistently with their intended purpose as specified by their creators. For instance, datasets and tools used for repository-level code generation and evaluation were sourced and implemented following the terms set by the original authors. We strictly adhered to the licensing agreements and usage restrictions outlined for each artifact. Any modifications made to the artifacts, such as the adaptation of existing datasets for web microservice repository generation, were performed within the bounds of academic research and in compliance with the access conditions (§L).

For the artifacts we created, including the RepoGenesis benchmark and related tools, we clearly define their intended use within the context of this research. These artifacts are designed for evaluating large language models (LLMs) on repository-level code generation tasks and should only be used within the scope of academic or research purposes. Derivatives of the data used in this research, such as model outputs or analysis results, will not be used outside of these contexts to ensure compliance with ethical and licensing guidelines.

P Data Containing Personally Identifying Information or Offensive Content

To ensure the ethical integrity of our research, we carefully examined the data collected for RepoGenesis to verify that it does not contain any personally identifying information (PII) or offensive content. The data used in our benchmark consists of web microservice repositories sourced from publicly available GitHub repositories and expert-supervised generation, with no inclusion of private or sensitive personal information. We specifically focused on the code and its associated requirements, ensuring that any metadata related to individual contributors or personal identifiers was excluded.

Additionally, we employed a manual review process to identify and filter any potentially offensive content within the code, comments, or requirements. We worked with our annotators to establish clear guidelines for identifying content that could be deemed inappropriate or offensive, ensuring that all samples in RepoGenesis adhered to a high standard of professionalism and respectfulness. This process helps maintain the privacy and safety of individuals and ensures the ethical use of the data in our research. Any identified offensive or sensitive content was removed before inclusion in the benchmark.

Q Potential Risks

While the research presented in this paper contributes to advancing repository-level code generation using large language models (LLMs), several potential risks associated with this work must be considered. These risks include both intentional and unintentional harmful effects, as well as broader concerns related to fairness, privacy, and security.

1. **Malicious or Unintended Harmful Effects:** The generation of web microservice repositories through LLMs may inadvertently lead to the creation of faulty or insecure code that, if deployed in production environments, could be exploited by malicious actors. These repositories might not only be prone to security vulnerabilities but could also be misused for illicit purposes, such as unauthorized data access or system compromise. This highlights the importance of integrating robust security evaluation mechanisms and comprehensive test suites into the evaluation pipeline, as we have done in this study.

2. **Environmental Impact:** The computational resources required for training and fine-tuning large-scale models, such as the ones used in this research, contribute to the environmental impact of AI research. Training these models requires significant GPU hours, and the energy consumption associated with this process is a growing concern. Future work should explore ways to mitigate the environmental impact by improving the efficiency of the models or exploring more energy-efficient approaches to training.

3. **Fairness Considerations:** One potential risk of deploying these technologies is the possibility of exacerbating existing biases or inequalities in software development practice. If the models are trained on a narrow set of data sources, there is a risk that they could generate code that is biased or not applicable to the needs of diverse or marginalized developer groups. To address this, we ensure that our dataset includes a broad range of real-world repositories to enhance the generalizability and fairness of our model evaluations.

4. **Privacy and Security Considerations:** Since the data used in this research comes from publicly available web microservice repositories, there are minimal privacy concerns. However, security risks are inherent in the generation of repository-level code, particularly when models are not fully vetted for safety or are used to create services that handle sensitive data or

critical operations. These models could unintentionally generate code with vulnerabilities or flaws that put users or systems at risk. We address this through comprehensive black-box functional testing and API compliance validation in our evaluation pipeline.

5. **Dual Use:** The technology presented in this research, although intended for advancing repository-level code generation for legitimate use cases, could be misused. For example, the ability to generate complete repositories quickly might be exploited to create malicious services or to automate the creation of fraudulent systems. Moreover, incorrect or insecure code generated by the models could result in unintended consequences if it is used in production environments without proper validation.

6. **Exclusion of Certain Groups:** While our research focuses on web microservices, this technology stack is not equally accessible or relevant across all development communities. There is a risk that focusing on specific frameworks (e.g., Flask, Spring Boot) could inadvertently exclude developers working with other technology stacks or programming paradigms. We advocate for future research to expand the capabilities of such models to support diverse frameworks and languages, ensuring inclusivity in the adoption of LLM-generated repository code.

In conclusion, while our research aims to support secure and efficient repository-level code generation, it is crucial to acknowledge and mitigate these risks. Future work can enhance model robustness, security, and fairness in automated software development applications.