Using A2A with SWE-bench for Analyzing Agent Capability

Manish Muttreja
manishmuttreja@gmail.com

Abstract

SWE-bench has emerged as the de facto standard for evaluating language model agents on real-world software engineering tasks. While recent systems achieve 65%+ resolution on SWE-bench Verified, the benchmark suffers from three critical limitations: (1) **data contamination**—models may have memorized repositories and patches during pretraining; (2) **patch-only scoring**—evaluation ignores the engineering process; and (3) **static test dependence**—fixed test suites can be overfit. I present **SWE-Bench-A2A**, an evaluation framework addressing these limitations through four techniques: *retro-holdout mutations* [6] to detect contamination, *adversarial testing* (fuzz, edge case, mutation testing) to expose patch fragility, *trajectory-based process scoring* capturing the full engineering workflow, and a *reproduction-first gate* enforcing understanding before patching. Cross-model evaluation with GPT-4o, GPT-4.1, GPT-5.2, Claude Sonnet 4.5 (27.7% avg), Claude Opus 4.1 (18.8%), and Claude 3 Haiku (18.5%) demonstrates framework generality across 5 models and 2 providers. Preliminary testing shows evidence of contamination (e.g., GPT-4o achieved 100%→0% on mutated `sklearn-14141`). I provide Dockerfiles, anti-contamination infrastructure, and CI scaffolding for AgentBeats integration.

1 Introduction

The rapid advancement of large language models (LLMs) has enabled a new class of *software engineering agents*—systems that can understand codebases, diagnose bugs, and generate patches with minimal human intervention. Evaluating these agents requires benchmarks that capture the complexity of real-world software engineering while resisting the pitfalls of static evaluation.

SWE-bench [1] represents a significant step forward, drawing from 2,294 real GitHub issues across 12 popular Python repositories. Unlike synthetic benchmarks, SWE-bench tasks require agents to navigate complex codebases, understand issue descriptions, and produce patches that pass repository test suites. This execution-based evaluation provides a strong signal of functional correctness.

However, as SWE-bench has become ubiquitous in agent evaluation, three fundamental limitations have emerged:

1. **Data Contamination**: The repositories in SWE-bench (Django, Flask, Scikit-learn, etc.) are among the most common in LLM training corpora. Models may have memorized not just the codebases but the specific patches that resolve benchmark issues.

2. **Patch-Only Scoring**: Current evaluation awards full credit for any patch that passes tests, ignoring whether the agent understood the problem. A model that guesses correctly receives the same score as one that systematically debugged the issue.

3. **Static Test Dependence**: Fixed test suites can be overfit through pattern matching without true understanding. Agents may learn to produce patches that pass specific tests while failing on equivalent formulations.

I present **SWE-Bench-A2A**, an evaluation framework that addresses these limitations through four key techniques:

- **Reproduction Gate**: Agents must first produce a failing test that reproduces the bug, demonstrating understanding before patching.

- **Process Scoring**: Beyond pass/fail, the framework captures full agent trajectories and computes multi-dimensional scores for correctness, process quality, efficiency, and adaptation.

- **Anti-Memorization**: I apply retro-holdout mutations [6] to SWE-bench, transforming codebases with semantic-preserving renames. A fresh issue harvester provides never-before-seen tasks.

- **Dynamic Testing**: Beyond repository tests, the framework supports fuzz testing, mutation testing, and adversarial probes to detect overfitting.

The framework implements the Agent-to-Agent (A2A) protocol, enabling modular composition of assessors (Green Agents) and participants (Purple Agents). This design allows any solver to be evaluated without modification, promoting reproducibility and fair comparison.

2 Related Work

2.1 Code Generation Benchmarks

Early code benchmarks like HumanEval [2] and MBPP [3] evaluate function-level generation from docstrings. While useful for measuring basic coding ability, these synthetic tasks lack the complexity of real software engineering: multi-file reasoning, dependency management, and test integration.

2.2 Repository-Level Evaluation

SWE-bench [1] pioneered repository-level evaluation using real GitHub issues. The SWE-bench Verified subset (500 instances) provides human-validated instances with clearer specifications. The official leaderboard[1] tracks state-of-the-art systems, with mini-SWE-agent achieving 65% resolved and SWE-agent 1.0 as the open-source SOTA. However, these "% resolved" metrics may not fully capture agent capability due to contamination and overfitting concerns. Concurrent work like DevBench [4] extends to multi-language settings.

2.3 Contamination and Memorization

Data contamination in LLM benchmarks has been extensively documented [5]. For code benchmarks, the problem is acute: popular repositories appear repeatedly in training data. Techniques like canary strings and holdout sets provide partial mitigation but cannot detect memorization of existing public data.

2.4 Process-Aware Evaluation

Traditional software engineering emphasizes process quality alongside outcomes. Test-driven development (TDD) requires understanding before implementation. The reproduction gate operationalizes this principle for agent evaluation.

[1] `https://www.swebench.com/`

3 Limitations of Current SWE-bench

3.1 Data Contamination

SWE-bench repositories are among the most-starred Python projects on GitHub. Analysis suggests substantial overlap with common training corpora:

- Django: 76k+ stars, extensive documentation
- Flask: 66k+ stars, widely referenced in tutorials
- Scikit-learn: 58k+ stars, standard ML library

Models trained on web-scale data have likely seen these codebases, their issues, and their patches. Performance on "unseen" tasks may reflect recall rather than reasoning.

3.2 Patch-Only Evaluation

Current scoring treats all passing patches equally:

$$\text{Score} = \mathbb{1}[\text{all tests pass}] \tag{1}$$

This binary metric ignores:

- Whether the agent understood the bug
- The quality of the debugging process
- Efficiency of the solution path
- Ability to handle ambiguity

3.3 Static Test Overfitting

Repository test suites, while valuable, have fixed specifications. Agents may learn patterns that satisfy specific tests without generalizing. A patch that passes `test_user_login` may fail on semantically equivalent `test_account_authentication`.

4 SWE-Bench-A2A Design

4.1 A2A Protocol Architecture

The framework implements the Agent-to-Agent protocol with two actor types:

Green Agent (Assessor) Orchestrates evaluation: provisions environments, dispatches tasks, verifies solutions, computes scores.

Purple Agent (Solver) Attempts tasks: receives issue descriptions, explores codebases, generates patches.

Communication occurs via REST endpoints with standardized message formats:

```
# Task creation
POST /a2a/task
{
  "title": "Fix bug #1234",
  "description": "...",
  "resources": {"repo": "...", "commit": "..."}
}

# Artifact submission
POST /a2a/task/{id}/artifact
{
  "type": "patch_submission",
  "parts": [{"type": "file_diff", "content": "..."}]
}
```

This separation enables any solver to be evaluated without code changes, promoting fair comparison across systems.

4.2 Reproduction Gate

Before accepting patches, the framework requires agents to demonstrate bug understanding through reproduction:

Algorithm 1 Reproduction Gate Protocol

Require: Issue description I, environment E

Agent submits reproduction script R
Execute R in unpatched E
if R does not fail **then**
  reject: "Reproduction must fail before patch"
end if
Agent submits patch P
Apply P to E
Run full test suite
return verification result

This gate enforces test-driven development principles: understand the problem (red), then fix it (green).

4.3 Trajectory-Based Process Scoring

The framework captures complete agent trajectories and computes multi-dimensional scores:

$$\begin{aligned} S = {} & 0.35\, s_{\text{correct}} + 0.20\, s_{\text{process}} \\ & + 0.15\, s_{\text{efficiency}} + 0.15\, s_{\text{collab}} \\ & + 0.10\, s_{\text{understand}} + 0.05\, s_{\text{adapt}} \end{aligned} \quad (2)$$

where the scoring dimensions are:

Category	Weight	Description
Correctness	0.35	Tests pass, patch applies
Process	0.20	Systematic exploration
Efficiency	0.15	Token/time usage
Collaboration	0.15	Information requests
Understanding	0.10	Reproduction quality
Adaptation	0.05	Response to feedback

Table 1: Scoring dimensions and weights

4.4 Anti-Memorization Strategies

4.4.1 Retro-Holdout Mutations

Following the retro-holdout methodology introduced by Haimes et al. [6], I apply semantic-preserving mutations to detect contamination:

- **Variable renaming**: `data` → `payload`
- **Function renaming**: `get_user` → `fetch_account`
- **Class renaming**: `UserManager` → `AccountHandler`
- **Comment perturbation**: Rephrase docstrings

Mutations are applied consistently across the codebase while preserving test behavior. This creates "parallel universes" where memorized patches no longer apply.

4.4.2 Fresh Issue Harvesting

A harvester monitors GitHub for new issues in target repositories, providing tasks created after model training cutoffs. These "secret-in-time" instances provide contamination-free evaluation.

4.5 Dynamic Testing

Beyond repository tests, the framework provides hooks for (not enabled by default in reported runs):

Fuzz Testing Property-based tests with random inputs

Mutation Testing Assert patches handle code mutations

Adversarial Probes LLM-generated edge cases

5 Implementation

5.1 System Architecture

The implementation consists of several key components:

- **A2A Server**: FastAPI-based REST API implementing the A2A protocol with endpoints for task management, artifact submission, and health checks.
- **Environment Orchestrator**: Docker-based container management with JIT provisioning, repository cloning, and commit checkout.
- **Verification Engine**: Patch application, test execution with timeout handling, and flaky test detection.
- **Trajectory Capture**: Action logging with database persistence and streaming support.
- **LLM Solver**: Integration with OpenAI/Anthropic APIs for reproduction script and patch generation. The solver includes a three-tier fallback hierarchy: (1) real LLM API calls when API keys are configured, (2) heuristic patches for known benchmark instances (e.g., django-11099), and (3) mock responses when no API access is available. This design enables both production evaluation with frontier models and development testing without API costs.

5.2 Docker Images

I provide Dockerfiles for containerizing the Green and Purple agents. Image publishing (registry, tags, and access) is deployment-specific; the repository includes the artifacts needed to build and push images via CI for use with the AgentBeats evaluation platform.

6 Experiments

6.1 Setup

I ran seven experiments to validate the framework:

- **Experiments 1–4**: Establish baseline metrics using semantic patch comparison across 3–100 instances with GPT-4o, GPT-4.1, and GPT-5.2.
- **Experiments 5–6**: Describe anti-contamination and adversarial testing methodology (infrastructure implemented, large-scale analysis pending).
- **Experiment 7**: Cross-provider evaluation with Claude model family (100 instances each).

Experiments 1–4 and 7 provide quantitative results; experiments 5–6 describe the methodology and infrastructure for future contamination/robustness analysis.

6.2 Experiment 1: Integration smoke test (3 Django instances)

Purpose: ensure end-to-end plumbing (environment provisioning, A2A dispatch, patch apply, test execution) works under Docker.

- `django__django-11099`: UsernameValidator trailing newline (passed via heuristic baseline)
- `django__django-11133`: HttpResponse charset handling (LLM patch failed to apply)
- `django__django-11179`: model_to_dict for unsaved model (LLM patch failed to apply)

Instance	Patch	Tests	Time	Source
django-11099	✓	3/3	74s	Heuristic
django-11133	×	0/0	73s	LLM
django-11179	×	0/0	71s	LLM
Total	33.3%	-	-	-

Table 2: Integration smoke test: confirms Docker + A2A pipeline; highlights solver fragility on diff formatting.

Important note: The 33.3% success rate is from a heuristic baseline, not the LLM solver. The LLM-only success rate was 0% (0/2 tasks where LLM was used). This confirms infrastructure correctness while highlighting LLM solver fragility on diff formatting.

6.3 Experiment 2: GPT-4o benchmark (20 instances)

Purpose: measure solver quality with a stronger model on a broader slice. Evaluation uses **semantic patch comparison** (code-change overlap) rather than strict line matching.

Representative outcomes (semantic match shown):

- **100%** `sklearn__sklearn-14141`: add `joblib` to `show_versions` deps (perfect semantic match).
- **100%** `django__django-13406`: queryset handling fix (perfect).
- **93%** `pallets__flask-5014`: blueprint registration fix (near-perfect).

Metric	Value
Tasks Tested	20
Correct File Identification	100% (20/20)
Perfect Solutions (100% match)	25% (5/20)
High Match (>50%)	35% (7/20)
Average Semantic Match	43.2%
Composite Score S (LLM-only)	0.43
Total Tokens	18,169
Total Cost	$0.120
Cost per Task	$0.006

Table 3: GPT-4o aggregate metrics on 20 SWE-bench Verified instances.

- **80%** `sympy__sympy-23534`: symbol handling (strong partial).
- **0–50%** Several SymPy/Django tasks: correct file localization but partial or divergent semantics.

Key findings:

1. **File localization remains perfect**: 100% correct files on 20/20 tasks, confirming strong navigation.
2. **Semantic quality is mixed**: 25% perfect, 35% high-match; average semantic match rises to 43.2% on the larger slice.
3. **Repository difficulty**: Django and Flask skew higher (multiple 100%/93% cases); SymPy and some Django tests remain challenging with 0–50% matches.
4. **Cost efficiency holds**: $0.006 per task with frontier model API calls.

Context vs. public baselines: Public SWE-bench Verified baselines for earlier GPT-4–era systems typically report low double-digit pass@1. The semantic-match view shows GPT-4o producing functionally close patches on a meaningful fraction of tasks even when strict exact-match metrics would undercount success. This highlights the importance of reporting both exact and semantic measures when comparing against public results.

6.4 Experiment 3: Multi-model comparison (10 instances)

Purpose: compare frontier models on identical tasks to reveal model-specific strengths. Each model processed the same 10 SWE-bench Verified instances.

Key findings:

Metric	GPT-4o	GPT-4.1	GPT-5.2
Perfect (F1=100%)	**1**	0	0
High Match ($\geq$50%)	**2**	2	0
Files Correct	**10/10**	9/10	10/10
Avg F1 Score	**18.3%**	17.2%	7.0%
Cost	**$0.063**	$0.068	$0.088

Table 4: Multi-model comparison on 10 identical SWE-bench tasks.

Instance	GPT-4o	GPT-4.1	GPT-5.2
sympy-22914	0%	**67%**	44%
sympy-23950	0%	**10%**	0%
sklearn-14141	**100%**	0%	0%
django-16082	0%	0%	0%
django-13406	**33%**	15%	7%
django-16429	0%	0%	0%
sympy-13757	0%	0%	0%
sympy-23534	0%	0%	0%
sympy-19040	0%	0%	0%
django-14534	50%	**80%**	18%

Table 5: Per-instance F1 scores across models (best per row in bold).

1. **GPT-4o leads at small scale**: Highest average F1 (18.3%) and only model with a perfect solution (sklearn-14141).
2. **Model-specific strengths**: GPT-4o uniquely solved sklearn-14141 perfectly; GPT-4.1 achieved highest score on sympy-22914 (67%) and django-14534 (80%).
3. **Consistent difficulty**: SymPy tasks (13757, 23534, 19040) remain challenging for all models (0% across the board).
4. **File localization robust**: 90–100% correct file identification across all models.

Note on sampling variance: GPT-5.2's low score (7.0%) on this 10-task sample vs. its higher score (44.8%) at 100 tasks reflects sampling bias: these 10 instances included 6 SymPy tasks where all models scored 0%. The 100-task distribution better represents the full Verified benchmark difficulty spectrum.

6.5 Experiment 4: Large-scale benchmark (100 instances)

Purpose: validate findings at scale with statistically significant sample size. GPT-4.1 and GPT-5.2 each processed 100 SWE-bench Verified instances.

Metric	GPT-4.1	GPT-5.2
Tasks Completed	99/100	**100/100**
Files Correct	82.8%	**88.0%**
High Match ($\geq$50%)	36.4%	**40.0%**
Avg Semantic Match	38.7%	**44.8%**
Total Cost	$1.30	**$1.12**
Cost per Task	$0.013	**$0.011**

Table 6: 100-task benchmark: GPT-5.2 outperforms GPT-4.1 at scale.

Key findings at scale:

1. **GPT-5.2 wins comprehensively**: Higher semantic match (44.8% vs 38.7%), more high-match solutions (40 vs 36), better file localization (88% vs 82.8%), and lower cost.

2. **Scale changes rankings**: At 10 tasks, GPT-4.1 led; at 100 tasks, GPT-5.2 dominates—demonstrating the importance of large-scale evaluation.

3. **Both models reliable**: 99–100% task completion shows production-ready robustness.

4. **Cost efficiency improves**: $0.011–0.013 per task at scale vs $0.006 in earlier runs reflects more complex tasks in the full distribution.

6.6 Experiment 5: Anti-Contamination Framework (Methodology)

Purpose: The framework implements retro-holdout mutations to detect contamination. While the infrastructure is complete, full evaluation at scale is pending.

Methodology: The anti-contamination pipeline applies semantic-preserving transformations (variable/function/class renaming, docstring rephrasing) to create "mutated" versions of SWE-bench instances. Performance drops between original and mutated versions indicate memorization.

Preliminary observation: In small-scale testing, we observed that GPT-4o achieved 100% semantic match on `sklearn-14141` (original) but 0% on the mutated version, suggesting exact patch memorization for this instance. This motivates larger-scale contamination studies.

Status: The retro-holdout infrastructure is implemented and available in the repository. Comprehensive contamination analysis across 100+ instances is planned for future work.

6.7 Experiment 6: Adversarial Testing Framework (Methodology)

Purpose: The framework implements adversarial testing to quantify patch robustness beyond test-pass metrics.

Methodology: The adversarial evaluator includes three components:

- **Fuzz Testing**: Property-based tests with random inputs to verify defensive coding

- **Adversarial Edge Cases**: LLM-generated edge cases (null inputs, boundary conditions, malformed data)

- **Mutation Testing**: Code mutations (operator swaps, boundary changes) to test patch resilience

Hypothesis: Patches that pass repository tests may still be fragile to real-world usage variations. Standard "% resolved" metrics may overstate robustness.

Status: The adversarial testing infrastructure is implemented and available in the repository (`src/adversarial/`). Comprehensive robustness analysis is planned for future work.

6.8 Experiment 7: Claude Model Family Benchmark (100 instances)

Purpose: Evaluate Anthropic's Claude model family on SWE-bench to compare against OpenAI models and validate framework generality across model providers.

Metric	Claude 3 Haiku	Opus 4.1	**Sonnet 4.5**
Tasks Completed	98/100	**100/100**	**100/100**
Avg Semantic Match	18.5%	18.8%	**27.7%**
High Match ($\geq$70%)	1	3	**8**
Perfect Match ($\geq$95%)	0	0	0
Total Tokens	92,810	114,760	103,511

Table 7: Claude model family comparison (100 instances each). Sonnet 4.5 leads.

Model	Tasks	Avg Match	High ($\geq$70%)	Provider
GPT-5.2	100/100	**44.8%**	**40**	OpenAI
GPT-4.1	99/100	38.7%	36	OpenAI
Claude Sonnet 4.5	100/100	27.7%	8	Anthropic
Claude 3 Haiku	98/100	18.5%	1	Anthropic
Claude Opus 4.1	100/100	18.8%	3	Anthropic

Table 8: Cross-provider model rankings on SWE-bench (100 instances).

Key findings:

1. **GPT-5.2 leads overall**: 44.8% avg match, significantly ahead of all Claude models.

2. **Claude Sonnet 4.5 leads Anthropic family**: 27.7% avg match with 8 high-quality solutions, outperforming both Opus 4.1 (18.8%) and Haiku (18.5%).

3. **Opus 4.1 comparable to Haiku**: Opus 4.1 achieves 18.8% avg match (vs Haiku's 18.5%), with 3 high-match solutions ($\geq$70%). Both are outperformed by Sonnet 4.5 (27.7%).

4. **Framework generality validated**: The A2A framework successfully evaluates 5 different models across 2 providers without modification.

Notable high-match results for Claude Sonnet 4.5:

- **Best**: sklearn-12585 (91.3%), django-11163 (91.7%), xarray-4629 (85.5%)

- **Strong**: django-13670 (65.2%), pytest-7205 (66.9%), django-11451 (91.3%)

This cross-provider evaluation demonstrates the A2A framework's generality—it can evaluate any LLM backend without modification.

6.9 Trajectory Analysis

For successful cases, the captured trajectory shows:

```
1. scenario_select -> instance_id
2. provision_environment -> [container_id]
3. dispatch_task -> [purple_task_id]
4. receive_artifact -> reproduction_script
5. receive_artifact -> patch_submission
6. verification -> passed (tests)
```

This visibility enables debugging agent behavior and computing process scores.

7 Impact of Novel Techniques

The framework provides tools to investigate three potential gaps in current SWE-bench evaluation:

7.1 Contamination Detection via Retro-Holdout

The retro-holdout methodology applies semantic-preserving mutations to detect memorization. Preliminary evidence:

- **Case study**: GPT-4o achieved 100% semantic match on `sklearn-14141` (original) but 0% on the mutated version, demonstrating exact patch memorization.

- **Implication**: Some "perfect" solutions may reflect recall rather than reasoning.

Status: Infrastructure implemented; large-scale contamination analysis is future work.

7.2 Robustness via Adversarial Testing

The adversarial framework tests patches against fuzz inputs, edge cases, and code mutations:

- **Hypothesis**: Patches passing repository tests may still be fragile to real-world usage variations.

- **Components**: Fuzz testing, LLM-generated edge cases, mutation testing.

Status: Infrastructure implemented (`src/adversarial/`); comprehensive analysis is future work.

7.3 Process Quality via Trajectory Scoring

Binary pass/fail ignores engineering quality. The multidimensional scoring captures:

- **Process**: Systematic exploration vs. random guessing

- **Understanding**: Reproduction quality before patching

- **Efficiency**: Token/time usage optimization

Contribution: Process scoring differentiates agents that systematically debug from those that guess correctly, enabling fairer comparison.

8 Evaluation Slices

I propose four evaluation slices for comprehensive assessment:

Verified Standard SWE-bench Verified instances

Mutated Retro-holdout transformed versions

Fresh Newly harvested issues (¡24h old)

Adversarial Instances with fuzz/mutation testing

Reporting across slices reveals contamination sensitivity and robustness.

9 Limitations and Future Work

9.1 Current Limitations

- **Python only**: Current implementation focuses on Python repositories
- **Model variance**: Performance varies significantly by both model and repository. At 100 tasks, GPT-5.2 leads overall but GPT-4o and GPT-4.1 excel on specific tasks. SymPy consistently challenges all models.
- **Semantic vs. exact matching**: The semantic comparison shows models often produce functionally equivalent patches that differ syntactically from expected solutions. Binary pass/fail evaluation may underestimate true capability.
- **Mutation coverage**: Retro-holdout not yet integrated in live evaluation flow
- **Dynamic test generation**: Fuzz/adversarial commands require per-repo configuration

9.2 Future Directions

1. Integrate additional frontier models (Gemini 2.0, Llama 3) for Purple agent comparison
2. Complete retro-holdout pipeline with semantic equivalence verification
3. Implement default fuzz command packs for common frameworks
4. Extend to multi-language evaluation (TypeScript, Rust)
5. Add visual/multimodal signals for UI-related bugs
6. Scale evaluation to full SWE-bench Verified (500+ instances)

10 Conclusion

SWE-Bench-A2A provides an evaluation framework and infrastructure to address three critical gaps in current SWE-bench evaluation:

1. Contamination Detection (Retro-Holdout): The framework applies semantic-preserving mutations to detect memorization. Preliminary evidence shows some models achieve perfect scores on original instances but fail completely on mutated versions (e.g., GPT-4o on `sklearn-14141`: 100%→0%), indicating memorization of specific patches.

2. Robustness Testing (Adversarial): The framework provides fuzz testing, adversarial edge cases, and mutation testing infrastructure. Hypothesis: current "% resolved" metrics may overstate true capability when robustness is considered.

3. Process Scoring (Trajectory): Binary pass/fail treats lucky guesses equally with systematic debugging. The multi-dimensional scoring (correctness, process, efficiency, adaptation) enables fairer comparison that rewards transferable engineering skills.

Key contributions:

- **Cross-provider evaluation**: Demonstrated framework generality across 5 models (GPT-4o, GPT-4.1, GPT-5.2, Claude Sonnet 4.5, Claude Opus 4.1, Claude 3 Haiku) and 2 providers
- **Infrastructure**: Retro-holdout mutation pipeline, adversarial testing suite, trajectory capture
- **AgentBeats integration**: Ready-to-run Docker images and CI scaffolding

I recommend the community adopt multi-slice reporting (Verified, Mutated, Adversarial) to provide more complete evaluation of agent capabilities.

Acknowledgments

I thank the SWE-bench team for their foundational work and the AgentBeats community for evaluation infrastructure.

References

[1] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Pinto, J. Kaplan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[4] DevBench Team. DevBench: A comprehensive benchmark for software development. *arXiv preprint*, 2024.

[5] O. Sainz, J. Campos, I. García-Ferrero, J. Etxaniz, O. Lopez de Lacalle, and E. Agirre. NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark. In *Findings of EMNLP*, 2023.

[6] J. Haimes, B. Peng, and S. Mukherjee. Benchmark Inflation: Revealing LLM Performance Gaps Using Retro-Holdouts. *arXiv preprint arXiv:2402.14857*, 2024.

[7] AgentBeats Platform. Agent registry for AI evaluation. `https://agentbeats.dev/`, 2024.

A A2A Protocol Specification

A.1 Agent Card Format

```
{
  "name": "SWE-bench Green Agent",
  "version": "1.0.0",
  "agent_id": "uuid",
  "capabilities": ["swebench_evaluation"],
  "endpoints": {
    "task": "/a2a/task",
    "health": "/health"
  }
}
```

A.2 Artifact Types

- `reproduction_script`: CODE artifact with failing test
- `patch_submission`: FILE_DIFF artifact with unified diff
- `assessment_result`: JSON artifact with verification results

B Scoring Formula Details

B.1 Correctness Score

$$s_{\text{correct}} = 0.6 \cdot \mathbb{1}[\text{pass}] + 0.3 \cdot \frac{\text{tests\_passed}}{\text{total\_tests}} + 0.1 \cdot \mathbb{1}[\text{patch\_applied}] \tag{3}$$

B.2 Process Score

$$s_{\text{process}} = 0.4 \cdot s_{\text{exploration}} + 0.3 \cdot s_{\text{reasoning}} + 0.3 \cdot s_{\text{reproduction}} \tag{4}$$

B.3 Efficiency Score

$$s_{\text{efficiency}} = 0.4 \cdot \frac{T_{\text{budget}} - T_{\text{used}}}{T_{\text{budget}}} + 0.4 \cdot \frac{N_{\text{budget}} - N_{\text{tokens}}}{N_{\text{budget}}} + 0.2 \cdot \frac{1}{1 + \text{atten}} \tag{5}$$