Toward Training Superintelligent Software Agents through *Self-Play SWE-RL*

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, Sida Wang

Presenter: Feng Zhu, Anupama Nair

2026.03.18

Georgia Tech

Current State of Software Agents

- **Foundation:** LLMs fine-tuned on vast repositories (SFT) and instruction-following data.
- **Current Benchmarks:** SWE-bench (Verified/Lite) is the gold standard, where agents fix real-world GitHub issues.
- **Current Workflow**: Most current agents (like the base CWM model) use a **Reason-Act-Verify** loop:
  1. Read issue description
  2. Explore codebase via Bash/Editor tools
  3. Generate a patch and run existing tests
- **The Bottleneck:** They are tethered to **natural language descriptions** and **pre-existing human tests**

Problem with Current Software Agents

Human-Curated Data

Rely on GitHub issues, PRs, human-written test suites, evaluation scripts for training

Curation Overhead

Unreliable without extensive human inspection
(→ SWE-bench Verified)

Scalability Barrier

Cannot scale indefinitely — replay human traces rather than discover new solutions

These agents primarily learn to replay human software development traces rather than independently discovering new problems and solutions.

Georgia Tech

Inspiration: Self-Play for Superintelligence

Self-Play Successes

AlphaZero
Superhuman Go/chess/shogi from only game rules

Absolute Zero
Self-proposes and solves coding tasks for reasoning

SPICE
Corpus-grounded self-play for general reasoning

The Missing Piece

"Zero" self-play cannot acquire knowledge beyond fixed environment rules and existing model knowledge.

Real-world codebases contain far richer knowledge than what can be inferred from a Python interpreter alone.

→ **Need grounded self-play on real repositories**

Georgia Tech

Self-play SWE-RL (SSR): Key Idea

A single LLM agent trained via RL in a self-play setting to iteratively inject and repair software bugs of increasing complexity.

Bug-Injection Agent

- Explores repository
- Discovers tests
- Injects realistic bugs
- Weakens tests to hide bugs

Bug-Solving Agent

- Receives formal test spec
- Interacts with buggy codebase
- Produces repair patch
- Validated against oracle tests

Same model, shared parameters, jointly trained with RL

Georgia Tech

Minimal Input Assumptions

What SSR Needs

Sandboxed Docker images with:

- Source code repository
- Installed dependencies

That's it.

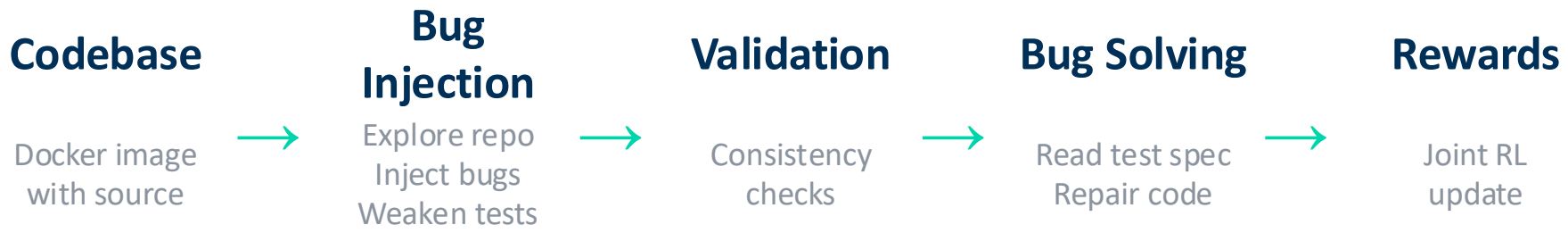
What SSR Does NOT Need

- Human-labeled issues
- Test suites or test runners
- Test parsers
- Language-specific infrastructure
- Issue descriptions (NL)
- Prior knowledge about frameworks

The bug-injection agent discovers how to run tests, creates parsers, and understands test structure entirely through environmental interaction.

Georgia Tech

SSR Pipeline Overview



← Bug-injection reward signal (based on consistency + solver solve rate) ←

← Bug-solving reward signal (binary: all tests pass?) ←

Higher-order bugs: Failed solver attempts create new bug artifacts → richer training distribution

Georgia Tech

Bug Artifact Specification

Each bug is formally specified via 5 files:

`test_script.sh`	Bash script to run the test suite
`test_files.txt`	List of oracle test files (reset before evaluation)
`test_parser.py`	Parses test output → JSON {test_id: pass/fail}
`bug_inject.diff`	Git diff patch that introduces bugs
`test_weaken.diff`	Git diff that weakens/removes tests to hide bug

The reversed test_weaken.diff serves as the formal specification for the solver — no NL issue description needed.

Georgia Tech

Test Weakening → Solver Specification

`test_weaken.diff`

Removes init=warm_start_params(m)

and assertion on delta shape

→ Hides the bug from tests

reverse

→

`oracle_test.diff (solver sees this)`

Adds init=warm_start_params(m)

and assertion on delta shape

→ Formal spec the solver must satisfy

test_weakening_patch.diff

```
@@ -957,6 +957,5 @@ class TestProphetWarmStart:
     daily_univariate_ts.iloc[:500], show_progress=False
 )
     m2 = Prophet(mcmc_samples=100, stan_backend=backend).fit(
-        daily_univariate_ts.iloc[:510], init=warm_start_params(m), show_progress=False
+        daily_univariate_ts.iloc[:510], show_progress=False
     )
-    assert m2.params["delta"].shape == (200, 25)
```

oracle_test_patch.diff (*weakening patch reversed:* `git diff buggy original -- {test_files}`)

```
@@ -957,5 +957,6 @@ class TestProphetWarmStart:
     daily_univariate_ts.iloc[:500], show_progress=False
 )
     m2 = Prophet(mcmc_samples=100, stan_backend=backend).fit(
-        daily_univariate_ts.iloc[:510], show_progress=False
+        daily_univariate_ts.iloc[:510], init=warm_start_params(m), show_progress=False
     )
+    assert m2.params["delta"].shape == (200, 25)
```

No NL issue synthesis → improvements on NL issue solving come purely from learning to write test-passing code

Georgia Tech

Bug-Injection Strategies

Direct Injection	Removal-Oriented	Removal + History
Naive prompting	Remove code hunks/files	Also revert git changes
No detailed guidance	Remove substantial code	Selectively revert past fixes
Tends to produce trivial	Require compatibility fixes	More realistic bug patterns
one-line modifications	Force solver to reconstruct	Greater diversity
Weakest	**Strong**	**Best**

Both removal and history bugs require the solver to understand repository structure and add/modify code.

Georgia Tech

Higher-Order Bugs

1st-order bug
From injection agent → **Solver attempts**
Fails to fix completely → **2nd-order bug**
Failed patch = new bug → **Solver retries**
On layered failure

Why Higher-Order Bugs?

Scalability: Running longer yields new distinct bugs rather than exhausting patterns

Realism: Mimics how developers unintentionally write buggy code

Diversity: Exposes the system to layered, interdependent failure patterns

Consistency Validation

1.	**Test files existence**	All test files must exist in original repo
2.	**Test parser validity**	Parser must produce correct JSON mappings
3.	**Test script validity**	All tests pass on original code, ≥ min_passing_tests
4.	**Bug scope**	Bug patch modifies ≥ min_changed_files
5.	**Bug validity**	≥ min_failing_tests must fail after injection
6.	**Test weakening validity**	Some failing tests pass after weakening
7.	**Inverse mutation testing**	Each modified file contributes to at least one test failure

Artifacts passing all checks → reformatted as SWE-bench instances with pass-to-pass and fail-to-pass specs

Georgia Tech

Constructing the Buggy Codebase

1	**Apply bug_inject.diff**	Introduce the bug into source code
2	**Apply test_weaken.diff**	Hide the bug from existing tests
3	**Apply failed pred_patch.diff**	(Optional) For higher-order bugs
4	**Reinitialize git**	rm -r .git && git init → prevent info leakage
5	**Buggy codebase ready**	Solver's starting environment

Git reinitialization prevents agents from cheating via git history inspection.

Georgia Tech

Patch Evaluation Pipeline

1. Tag original as ssr-original
2. Apply bug_inject.diff + test_weaken.diff
3. Apply predicted patch from solver
4. Restore test files from ssr-original
5. Run test_script.sh | python test_parser.py

Test file restoration ensures the solver cannot game tests — evaluation uses original oracle tests.

Georgia Tech

Reward Design

Bug-Injection Reward

`r = -1.0`
if consistency validation fails

`r = -α`
if s = 0 (too hard) or s = 1 (too easy)

`r = 1 - (1+α)s`
if 0 < s < 1 (ideal difficulty)

s = solve rate, α = 0.8

Bug-Solving Reward

`r = +1`
if all tests pass

`r = -1`
otherwise

Simple binary reward

Georgia Tech

Opposing Incentives

Solver

`E[r_solve] = 2s – 1  →  Benefits from higher s (easier bugs)`

Injector

`r_inject = 1 – (1+α)s  →  Benefits from lower s (harder bugs)`

But: s = 0 is penalized → injector must keep bugs solvable

→ Adversarial pressure pushes the injector to propose bugs near the frontier of the solver's current capability

Optimal Target Solve Rate

With G = 8 samples, the expected reward R(p) has a unique maximum $p^* \approx 0.2$
→ Challenger targets bugs that are hard but solvable ~20% of the time

Georgia Tech

Experimental Setup & Benchmarks

Base Model

CWM-sft
(32B parameter
Code World Model)

Benchmarks

SWE-bench Verified:
500 curated, solvable Python tasks

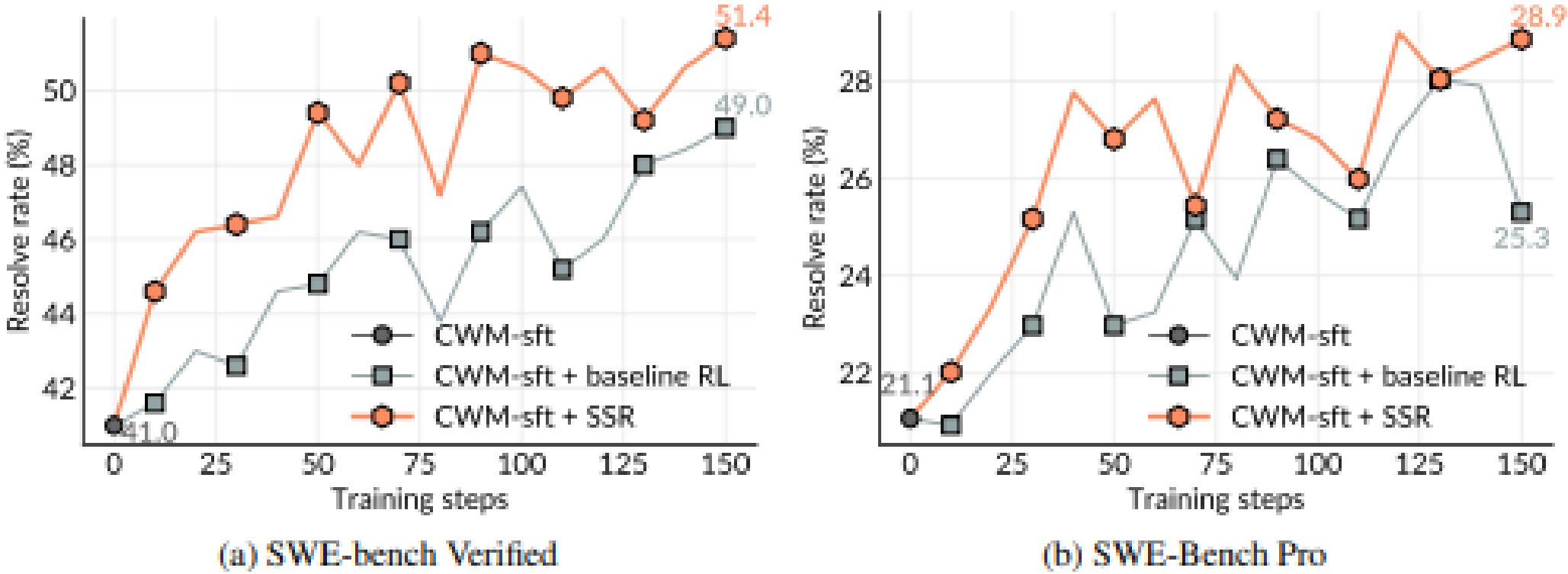
SWE-bench Pro:
featuring longer-horizon tasks and multi-file edits.

The Baseline

CWM-RL trained on human data
(actual GitHub issues/tests)
vs.
SSR (Self-Play)

Georgia Tech

Key Results: SSR vs. Human-Data Baseline



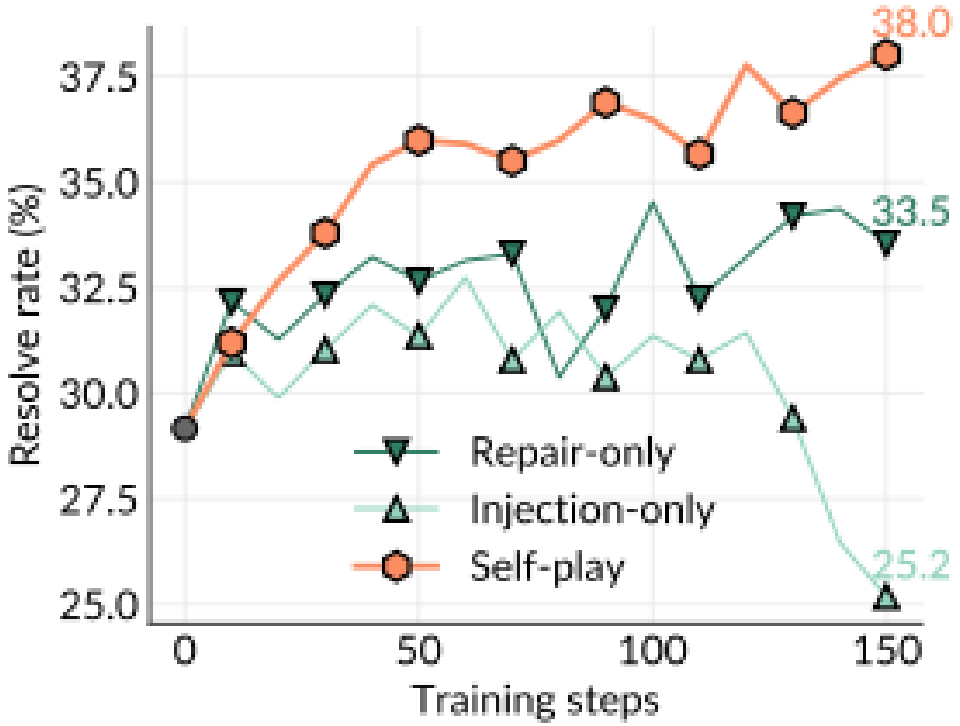
(a) SWE-bench Verified (b) SWE-Bench Pro

Figure 8: Baseline comparison over the course of training.

SSR proves that autonomous self-play can break the human data limits to reach better agent performance.

Georgia Tech

Ablation: Role of Self-Play

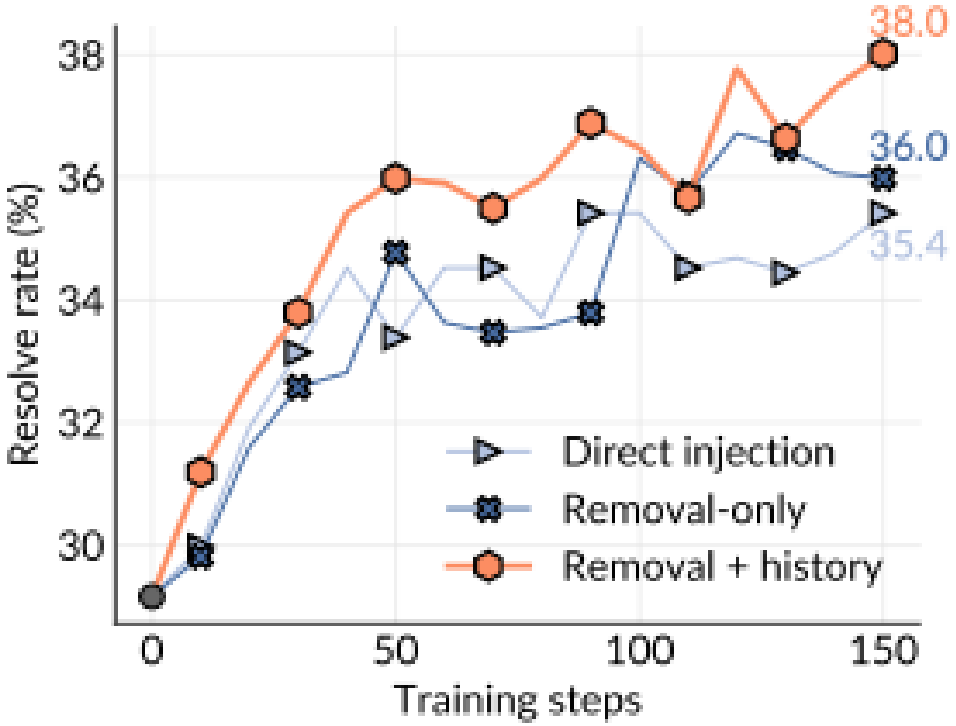


(a) Self-play components

Self-Play Learning requires Task Generation + Solving

Georgia Tech

Ablation: Bug Injection Strategy



(b) Bug-injection methods

Bug Injection quality determines learning quality

Georgia Tech

Discussion: Why Self-Play Works for Software

- **The Grounding Advantage:** Unlike LLM-only self-play, SSR is grounded in a **Dockerized sandbox**. The agent must generate code that actually compiles and tests that actually run.
- **Strategy Comparison:** The authors found that *how* you inject bugs matters:
  - **Naïve Injection:** Often leads to trivial, one-line changes that are too easy to solve.
  - **Removal + History (Best):** By deleting existing logic and providing the "history" of what was there, the agent learns to re-implement complex functionality.
- **The Hiding Mechanism:** A key innovation is **Test Weakening**. The agent doesn't just break code; it learns to modify existing tests, forcing the "Solver" to think harder.

Limitations

Test-Based Supervision Limits Scope

Relies on unit tests as oracle

Risk of Reward Hacking

May overfit to tests instead of true correctness

Weak/Noisy Reward

Sparse + Indirect feedback

Single Model for Dual Roles

Limits specialization

Georgia Tech

Emergent Properties & Future Directions

- **Superintelligence:** SSR shows a consistent upward trajectory. It doesn't plateau like SFT (Supervised Fine-Tuning) because it creates its own "infinite curriculum" of bugs.

- **Natural Language Transfer**: even though the model is trained on **code patches** (test-to-patch), it gets better at solving **Natural Language issues** (issue-to-patch).

- **Scaling Potential:** Can train on private or internal codebases that don't have human-written GitHub issues.

Conclusion

- **SSR** represents a shift from "Learning from Humans" to "Learning from the Code Itself."
- **Key Achievement:** Outperforming human-data baselines using zero human labels.
- **The Path Ahead:** This is the first step toward software agents that can maintain and evolve entire systems autonomously.

Thank You

Georgia Tech