Code execution for Code World Model training

Emily McMilin, FAIR, April 26, 2026

Outline

- Background: Code World model
- Code execution environments
- Code execution for world modeling
- Code execution for RL (training instances)
- RL for code execution (training instances)

Outline

- Background: Code World model
- Code execution environments
- Code execution for world modeling
- Code execution for RL (training instances)
- RL for code execution (training instances)

- Finite training data from static code
- ~infinite training data from executed code
  - for world modeling
- Human curation remains a bottleneck for building:
  - code execution environments
  - RL training instances
- Use code execution to ground training
  - ~zero sim-to-real gap for some deployment code applications
  - Use world modeling expensive, risky, deployment code applications
- Many unexplored applications of code world modeling capabilities:
  - neural debugging, RL efficiencies

Code World Model

Code World Model

CWM: An Open-Weights LLM for Research on Code Generation with World Models

Meta FAIR CodeGen Team

We release Code World Model (CWM), a 32-billion-parameter open-weights LLM, to advance research on code generation with world models. To improve code understanding beyond what can be learned from training on static code alone, we mid-train CWM on a large amount of observation-action trajectories from Python interpreter and agentic Docker environments, and perform extensive multi-task reasoning RL in verifiable coding, math, and multi-turn software engineering environments. With CWM, we provide a strong testbed for researchers to explore the opportunities world modeling affords for improving code generation with reasoning and planning in computational environments. We present first steps of how world models can benefit agentic coding, enable step-by-step simulation of Python code execution, and show early results of how reasoning can benefit from the latter. CWM is a dense, decoder-only LLM trained with a context size of up to 131 k tokens. Independent of its world modeling capabilities, CWM offers strong performance on general coding and math tasks: it reaches pass@1 scores of 65.8 % on SWE-bench Verified (with test-time scaling), 68.6 % on LiveCodeBench, 96.6 % on Math-500, and 76.0 % on AIME 2024. To support further research on code world modeling, we release model checkpoints after mid-training, SFT, and RL.

Date: September 29, 2025
Inference Code: github.com/facebookresearch/cwm
Model Weights: ai.meta.com/resources/models-and-libraries/cwm-downloads, huggingface.co/facebook/cwm, ../cwm-sft, ../cwm-pretrain

Core contributors

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng

Contributors

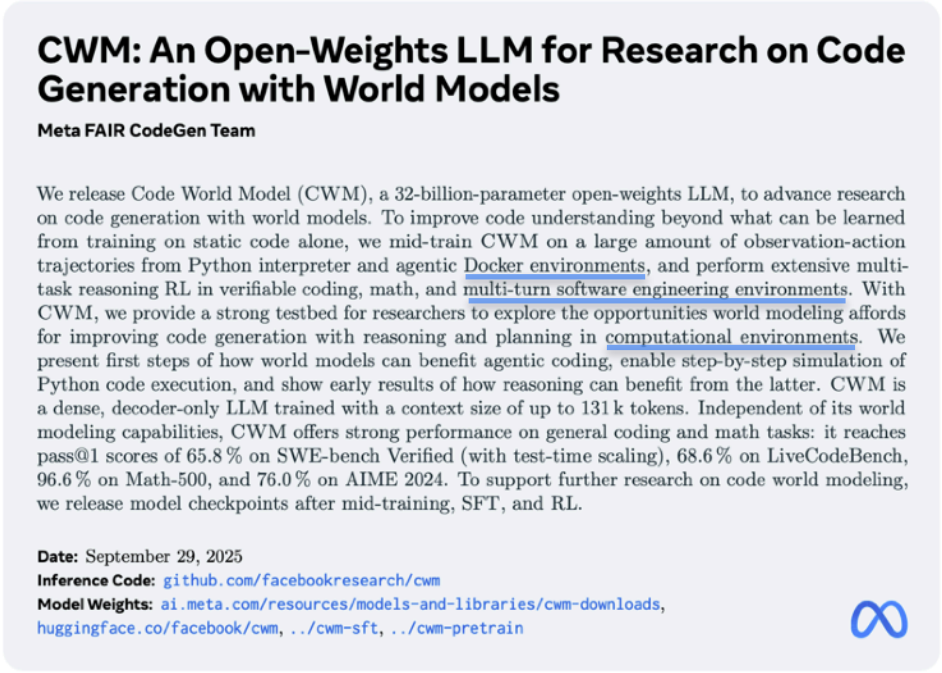
Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang

Senior core contributors

Taco Cohen, Yossi Adi, Gabriel Synnaeve

FAIR CodeGen team et al., CWM: An Open-Weights LLM for Research on Code Generation with World Models, 2025.

Code World Model



CWM: An Open-Weights LLM for Research on Code Generation with World Models

Meta FAIR CodeGen Team

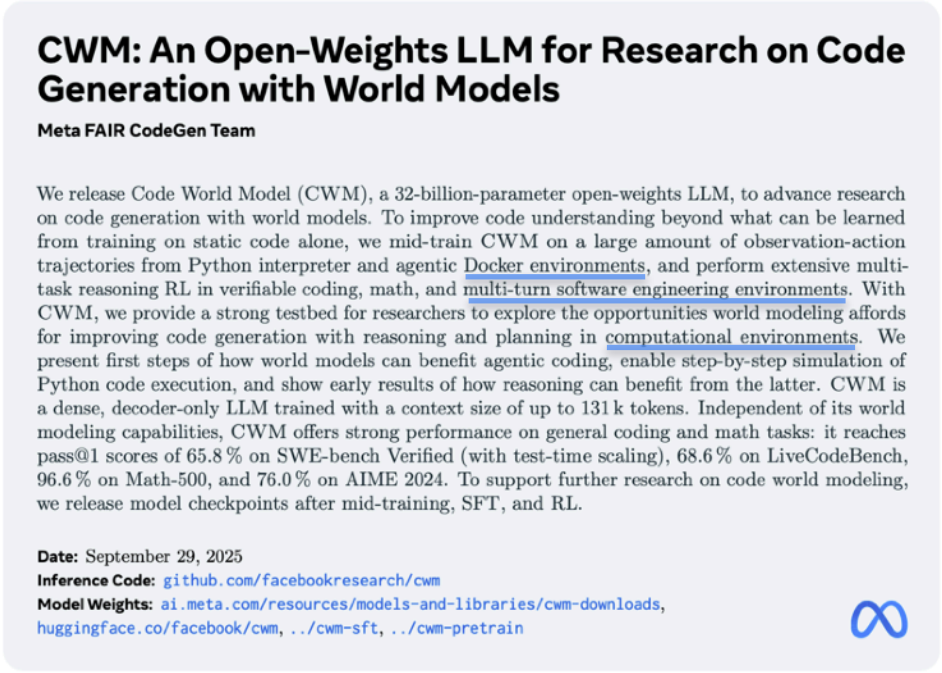
We release Code World Model (CWM), a 32-billion-parameter open-weights LLM, to advance research on code generation with world models. To improve code understanding beyond what can be learned from training on static code alone, we mid-train CWM on a large amount of observation-action trajectories from Python interpreter and agentic Docker environments, and perform extensive multi-task reasoning RL in verifiable coding, math, and multi-turn software engineering environments. With CWM, we provide a strong testbed for researchers to explore the opportunities world modeling affords for improving code generation with reasoning and planning in computational environments. We present first steps of how world models can benefit agentic coding, enable step-by-step simulation of Python code execution, and show early results of how reasoning can benefit from the latter. CWM is a dense, decoder-only LLM trained with a context size of up to 131 k tokens. Independent of its world modeling capabilities, CWM offers strong performance on general coding and math tasks: it reaches pass@1 scores of 65.8 % on SWE-bench Verified (with test-time scaling), 68.6 % on LiveCodeBench, 96.6 % on Math-500, and 76.0 % on AIME 2024. To support further research on code world modeling, we release model checkpoints after mid-training, SFT, and RL.

Date: September 29, 2025
Inference Code: github.com/facebookresearch/cwm
Model Weights: ai.meta.com/resources/models-and-libraries/cwm-downloads, huggingface.co/facebook/cwm, ../cwm-sft, ../cwm-pretrain

- Terminology:
  - “Executable repository image” == built repo saved to Docker image
  - “Training instance” == executable repo image + metadata for RL task
  - “RL training env” == E.g. for training policy (`.step`, `.reset`)

FAIR CodeGen team et al., CWM: An Open-Weights LLM for Research on Code Generation with World Models, 2025.

Code World Model



CWM: An Open-Weights LLM for Research on Code Generation with World Models

Meta FAIR CodeGen Team

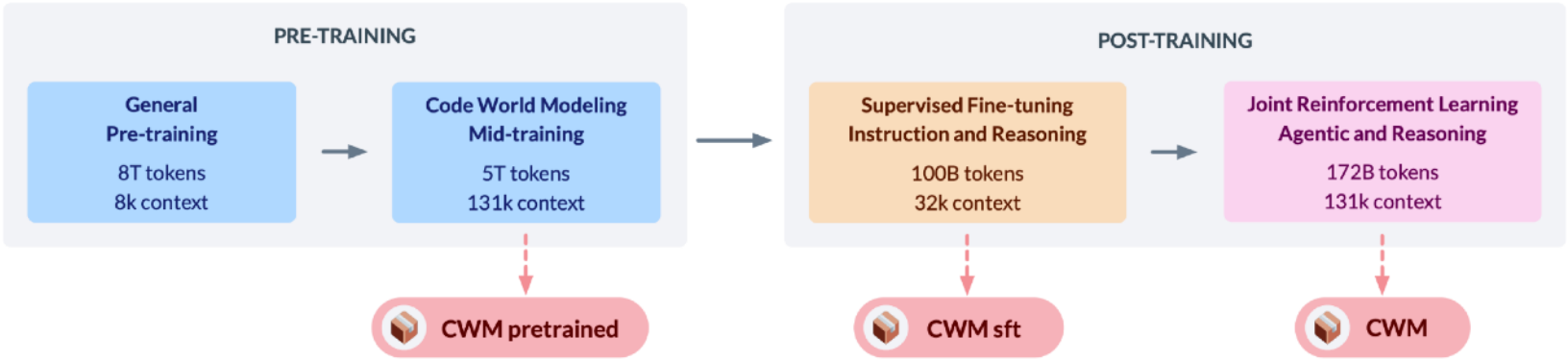
We release Code World Model (CWM), a 32-billion-parameter open-weights LLM, to advance research on code generation with world models. To improve code understanding beyond what can be learned from training on static code alone, we mid-train CWM on a large amount of observation-action trajectories from Python interpreter and agentic Docker environments, and perform extensive multi-task reasoning RL in verifiable coding, math, and multi-turn software engineering environments. With CWM, we provide a strong testbed for researchers to explore the opportunities world modeling affords for improving code generation with reasoning and planning in computational environments. We present first steps of how world models can benefit agentic coding, enable step-by-step simulation of Python code execution, and show early results of how reasoning can benefit from the latter. CWM is a dense, decoder-only LLM trained with a context size of up to 131 k tokens. Independent of its world modeling capabilities, CWM offers strong performance on general coding and math tasks: it reaches pass@1 scores of 65.8 % on SWE-bench Verified (with test-time scaling), 68.6 % on LiveCodeBench, 96.6 % on Math-500, and 76.0 % on AIME 2024. To support further research on code world modeling, we release model checkpoints after mid-training, SFT, and RL.

Date: September 29, 2025
Inference Code: github.com/facebookresearch/cwm
Model Weights: ai.meta.com/resources/models-and-libraries/cwm-downloads, huggingface.co/facebook/cwm, ../cwm-sft, ../cwm-pretrain

- Terminology:
  - “Executable repository image” == built repo saved to Docker image
  - “Training instance” == executable repo image + metadata for RL task
  - “RL training env” == E.g. for training policy (`.step`, `.reset`)
- Caveats:
  - Many other aspects of paper & model I’m not covering today.
  - All info I’m sharing is publicly available in referenced papers

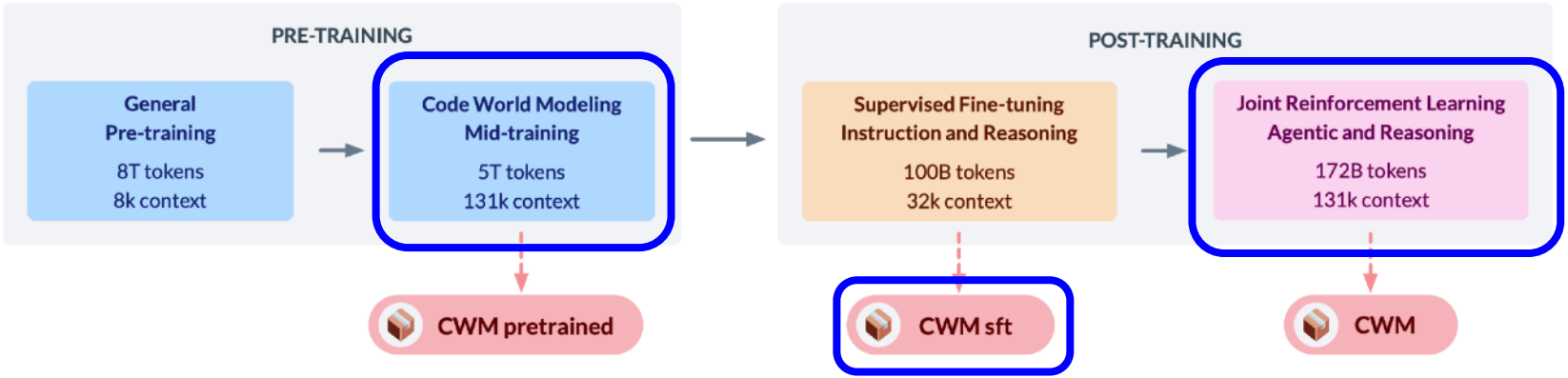
FAIR CodeGen team et al., CWM: An Open-Weights LLM for Research on Code Generation with World Models, 2025.

Code World Model: training stages



CWM 32B open-weights auto-regressive LLM

Code World Model: code-execution training stages



CWM 32B open-weights auto-regressive LLM

Code execution environments

Code execution environments

- “Executable repository image”:
  - A Docker image containing an arbitrary repository with dependences installed to enable running the code & tests, without additional setup.
  - Use containers:
    - For isolation (safety) and reproducibility (e.g for identical initial-state for all trajectories in GRPO group)

Code execution environments

- “Executable repository image”:
  - A Docker image containing an arbitrary repository with dependences installed to enable running the code & tests, without additional setup.
  - Use containers:
    - For isolation (safety) and reproducibility (e.g for identical initial-state for all trajectories in GRPO group)
- World modeling data generation: we require a diverse range of unique repositories:
  - Both pythonic and agentic tracing traverses git log commits
    - cap number commits per repo to limit redundancy, e.g (4:1)
  - Many available training sets have higher commit:repository ratio:
    - SWE-Gym (2.4k:11), R2E-Gym (4.6k:10), SWE-Smith (50k:128)
- ~= 200 unique repos, we required building +100x more unique repos

Pan et al., SWE-Gym, ICML 2025; Jain et al., R2E-Gym, 2025; Yang et al., SWE-smith, 2025.

Executable repository images: Scaling up

AI-assisted

- LLM tasked with setting up the dev env:
  - read the docs, pull in dependencies, ensure most tests pass.
- But:
  - Human-targeted instructions can be stale/wrong due to lack of verifiability.
  - Can be slow and costly

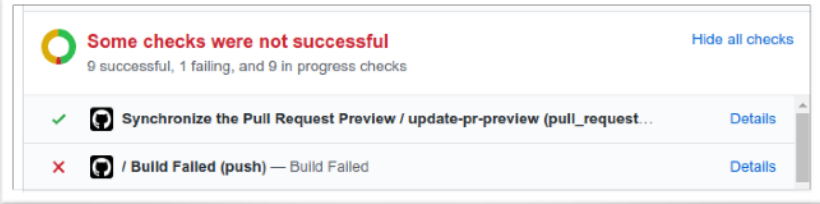
Executable repository images: Scaling up

AI-assisted

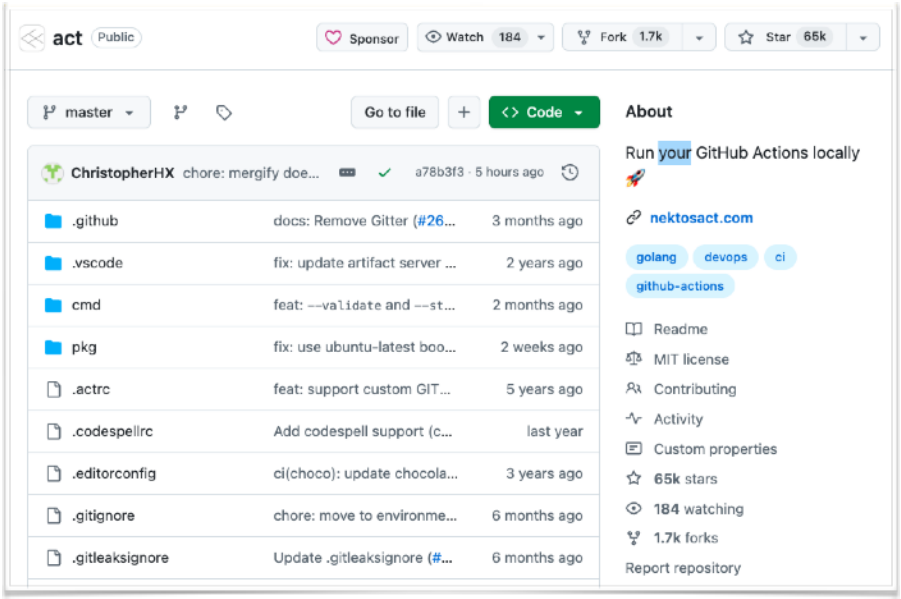
- LLM tasked with setting up the dev env:
  - read the docs, pull in dependencies, ensure most tests pass.
- But:
  - Human-targeted instructions can be stale/wrong due to lack of verifiability.
  - Can be slow and costly

CI-assisted

- Leverage continuous-integration:
  - workflows build arbitrary repos
- And:
  - Machine-targeted instructions must remain accurate for successful builds
    - GitHub signals failures, can't land PR.
  - Quick and cheap!



Executable repository images: Scaling up



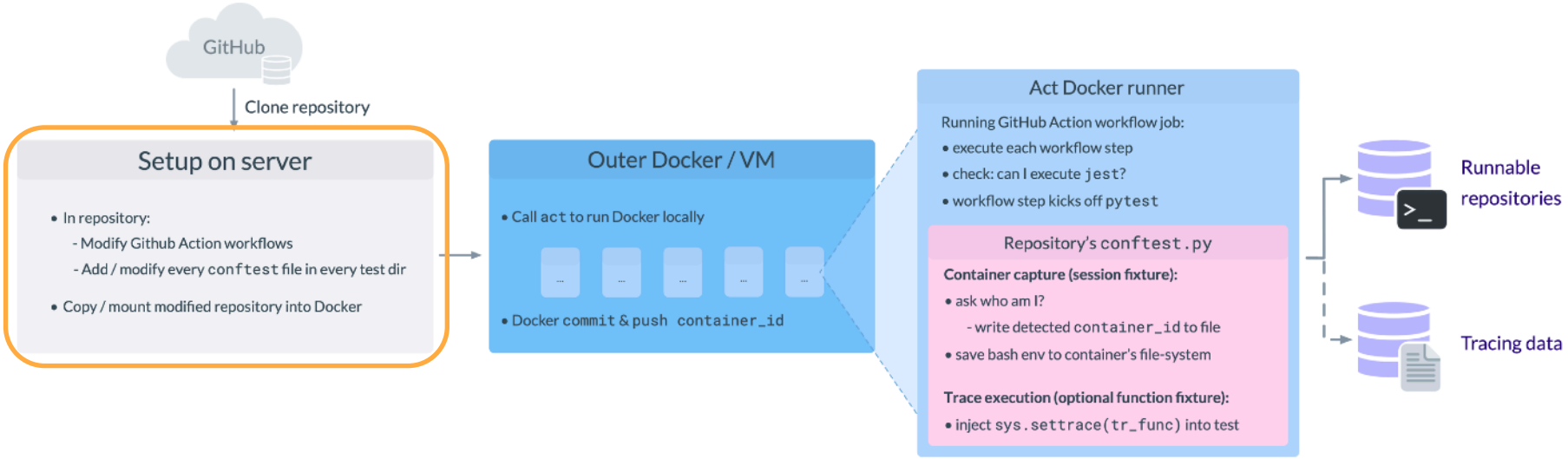
`act` library lets you run “your” own GitHub repo locally.

CI-assisted

- Leverage continuous-integration:
  - workflows build arbitrary repos
- And:
  - Machine-targeted instructions must remain accurate for successful builds
    - GitHub signals failures, can’t land PR.
  - Quick and cheap!

https://github.com/features/actions
https://github.com/nektos/act

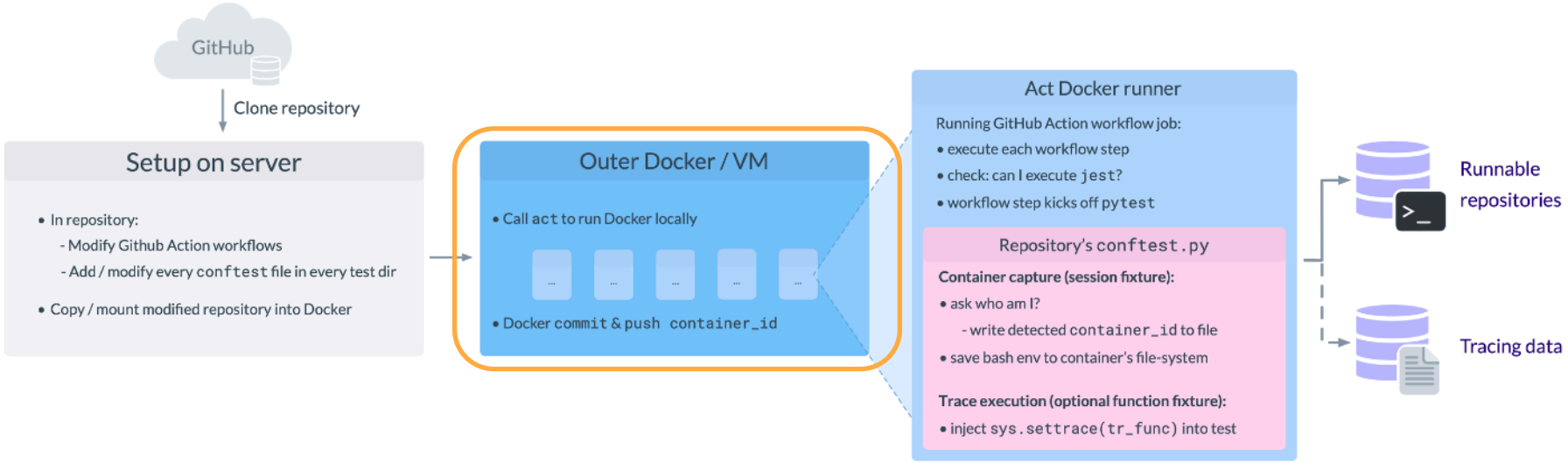
Executable repository images: CI-assisted (Activ)



In server, modify each GitHub repo to enable:

- continuation of CI after trivial error
- fail-fast termination of CI for fatal errors

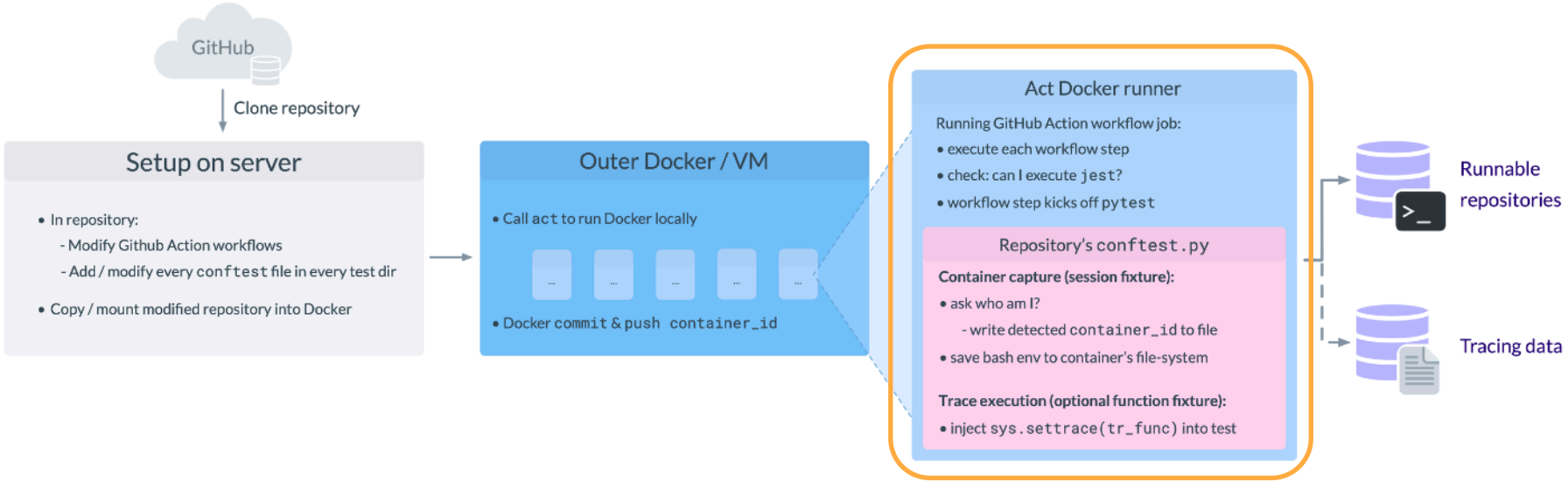
Executable repository images: CI-assisted (Activ)



In outer docker / VM, running `act` for local GitHub Actions:

- spins up multiple parallel Dockers: one container per workflow
- we seek the `built_container_id` for Docker successfully running unit tests

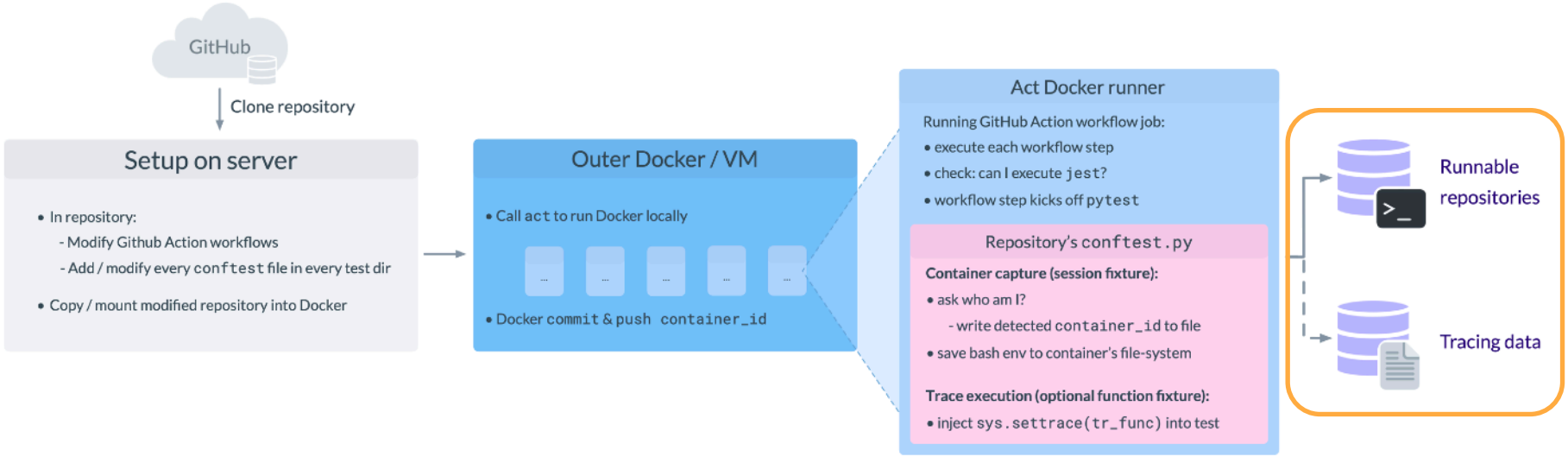
Executable repository images: CI-assisted (Activ)



Within each workflow (inner) Docker, find the `built_container_id` via:

- Non-pytest repos: detect executable test-framework binaries via mods made to the repo's GitHub Action workflows
- Pytest repos: detect if pytest running via mods made to the repo's unit tests (leverage conftest.py, to inject code into every test-session)

Executable repository images: CI-assisted (Activ)



Stored for later:

- ~35k unique built-repositories in container registry
- Store optional pythonic tracing data

Code execution for world modeling

Agentic traces

Solve the following issue by implementing the necessary code changes and submitting a patch file:

<issue_description> *{issue}* **</issue_description>**

The *[result]* argument of **<tool: submit>** should be the path to a patch file that resolves the issue … Once you've submitted at least once and are confident in your solution, provide a final response summarizing your work …

Your primary objective is to ensure patch correctness above all else - thoroughly explore the codebase, think hard, and leverage significant execution to verify correctness by writing comprehensive tests to validate your solution and running existing tests to prevent regressions. Only submit when you are genuinely confident in your patch's correctness …

<think> …	**<tool: bash>** `ls …`	…	**<think>** …	**<tool: edit>** …	…	**<think>** …	**<tool: bash>** `pytest …`	…	**<think>** …	…	…

<think> … **</think>**	**<tool: bash>** `git diff … > submission.patch` **</tool>**	(No output)
<think> … **</think>**	**<tool: submit>** `submission.patch` **</tool>**	The following patch content is marked as your final submission …
<think> … Now, I'll provide a summary … **</think>**	## Issue summary … ## Reproduction … ## Code changes …	

Agentic traces

- Initial state = executable repository image + prompt
- Action = tool call (`bash`, `edit`, `create`, `submit`)
- State/observation = Bash session stdout (inc. filesystem state)

Agentic traces for world modeling

Solve the following issue by implementing the necessary code changes and submitting a patch file:

<issue_description> *{issue}* **</issue_description>**

The *[result]* argument of **<tool: submit>** should be the path to a patch file that resolves the issue … Once you've submitted at least once and are confident in your solution, provide a final response summarizing your work …

Your primary objective is to ensure patch correctness above all else - thoroughly explore the codebase, think hard, and leverage significant execution to verify correctness by writing comprehensive tests to validate your solution and running existing tests to prevent regressions. Only submit when you are genuinely confident in your patch's correctness …

<think> …	**<tool: bash>** `ls` …	…	**<think>** …	**<tool: edit>** …	…	**<think>** …	**<tool: bash>** `pytest` …	…	**<think>** …	…	…
<think> … **</think>**	**<tool: bash>** `git diff … > submission.patch` **</tool>**	(No output)									
<think> … **</think>**	**<tool: submit>** `submission.patch` **</tool>**	The following patch content is marked as your final submission …									
<think> … Now, I'll provide a summary … **</think>**	## Issue summary … ## Reproduction … ## Code changes …										

Agentic traces for WM

- Initial state = executable repository image + prompt
- Action = tool call (`bash`, `edit`, `create`, `submit`)
- State/observation = Bash session stdout (inc. filesystem state)
- ~~Reward = 1 if all tests pass; 0 otherwise~~

Agentic traces for world modeling (ForagerAgent)

Solve the following issue by implementing the necessary code changes and submitting a patch file:

<issue_description> *[issue]* **</issue_description>**

The *[result]* argument of **<tool: submit>** should be the path to a patch file that resolves the issue … Once you've submitted at least once and are confident in your solution, provide a final response summarizing your work …

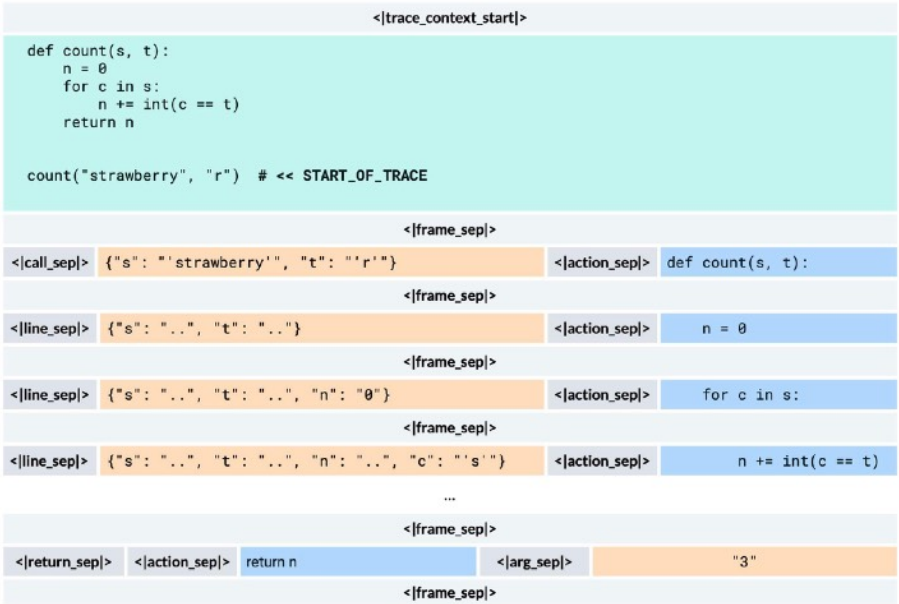
Your primary objective is to ensure patch correctness above all else - thoroughly explore the codebase, think hard, and leverage significant execution to verify correctness by writing comprehensive tests to validate your solution and running existing tests to prevent regressions. Only submit when you are genuinely confident in your patch's correctness …

<think> …	**<tool: bash>** `ls` …	…	**<think>** …	**<tool: edit>** …	…	**<think>** …	**<tool: bash>** `pytest` …	…	**<think>** …	…	…	
<think> … **</think>**	**<tool: bash>** `git diff … > submission.patch` **</tool>**	(No output)										
<think> … **</think>**	**<tool: submit>** `submission.patch` **</tool>**	The following patch content is marked as your final submission …										
<think> … Now, I'll provide a summary … **</think>**	## Issue summary … ## Reproduction … ## Code changes …											

Agentic traces for WM

- Initial state = executable repository image + mutation
- Action = tool call (`bash`, `edit`, `create`, `submit`)
- State/observation = Bash session stdout (inc. filesystem state)

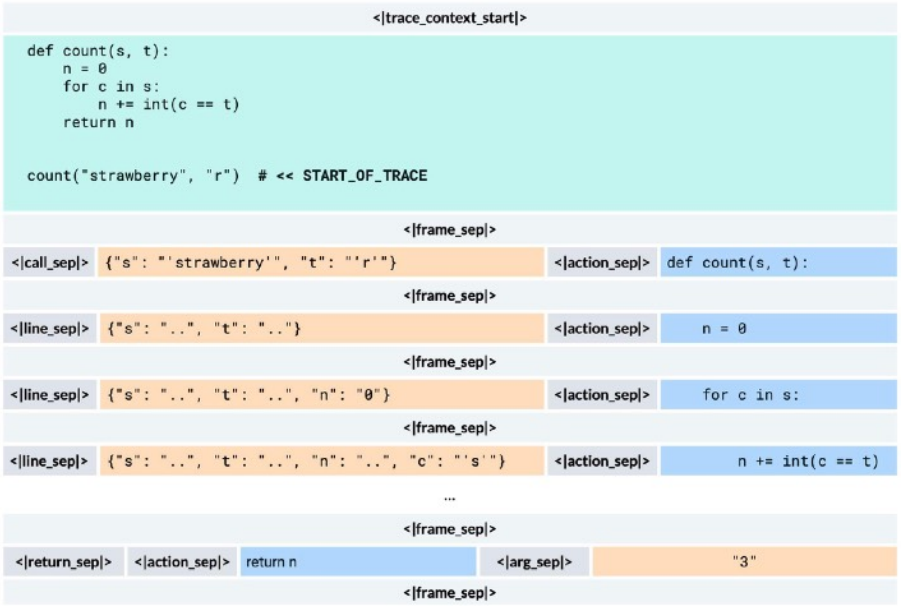
Pythonic traces intro



Pythonic traces

- Initial state = executable repository image + prompt
- Action = line of source code
- State/observation = stack frame: changes to local variable

Pythonic traces for world modeling

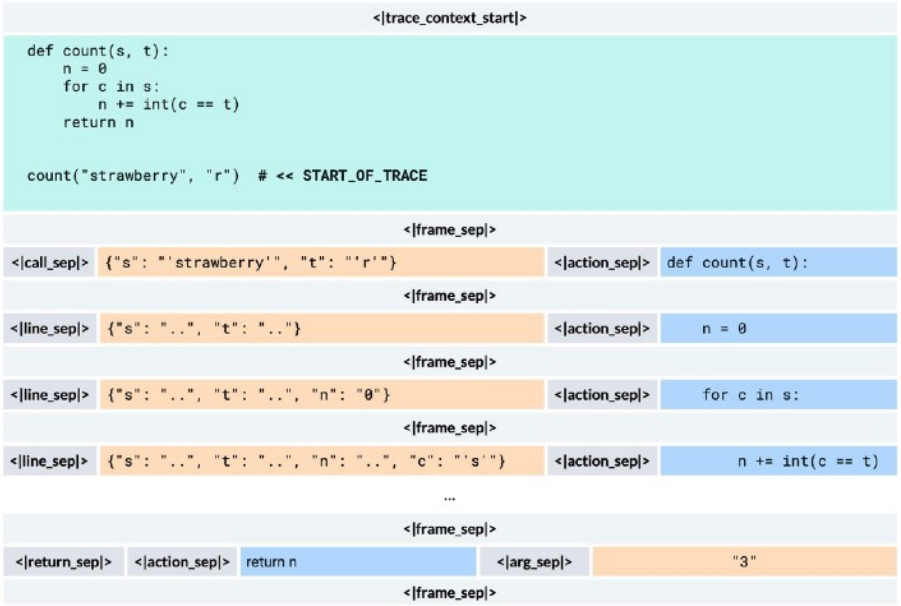


- We have executable code, we require inputs.
- Non-trivial to find interesting inputs with high code coverage.
- Fuzzing is an option, but can be time-consuming.
- How about executing all unit tests in arbitrary repos?
    - Tests have inputs & code coverage

Pythonic traces

- Initial state = executable repository image + no prompt?
- Action = line of source code
- State/observation = stack frame: changes to local variable

Pythonic traces for world modeling



Pythonic traces

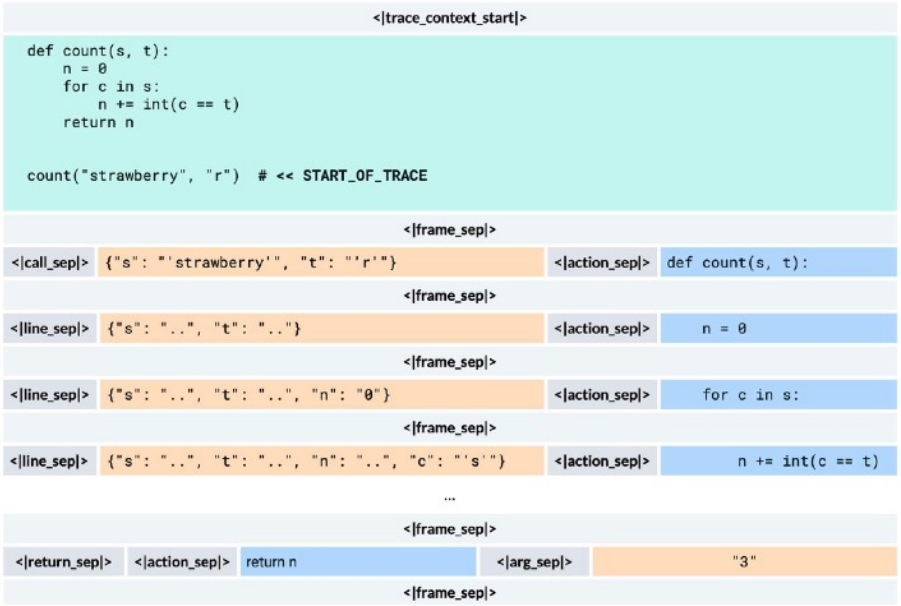
- Initial state = executable repository image + unit tests
- Action = line of source code
- State/observation = stack frame: changes to local variable

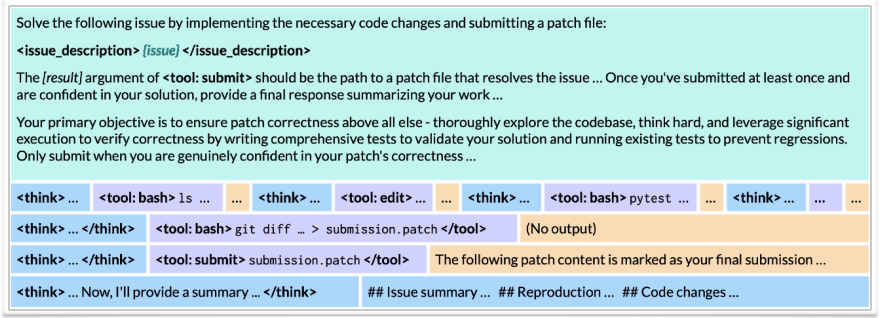
- We have executable code, we require inputs.
- Non-trivial to find interesting inputs with high code coverage.
- Fuzzing is an option, but can be time-consuming.
- How about executing all unit tests in arbitrary repos?
  - Tests have inputs & code coverage

- How to capture traces from tests in these arbitrary repos?
  - Leverage pytest's fixture configuration file (conftest.py)
  - Use to inject our tracing code into every test function in every repo
  - Run tests to kick off capture.

https://docs.pytest.org/en/stable/reference/fixtures.html#conftest-py-sharing-fixtures-across-multiple-files

Code execution for world modeling





Pythonic traces

- Initial state = executable repository image + unit tests
- Action = line of source code
- State/observation = stack frame: changes to local variable

Agentic traces

- Initial state = executable repository image + mutation
- Action = tool call (`bash`, `edit`, `create`, `submit`)
- State/observation = Bash session stdout (inc. filesystem state)

70k Pythonic repo-level traces & 3M Agentic (ForagerAgent) traces

Pythonic traces: inference

```
<|begin_of_text|><|trace_context_start|>
def f(n):
    p = ''
    if n%2 == 1:
        p+='sn'
    else:
        return n*n
    for x in range(1, n+1):
        if x%2 == 0:
            p+='to'
        else:
            p+='ts'
return p

def main():  # << START_OF_TRACE
    return f(1)
<|frame_sep|><|call_sep|>{}<|action_sep|>def main():
<|frame_sep|>
----END OF PROMPT----
<|line_sep|>{}<|action_sep|>    return f(1)
<|frame_sep|><|call_sep|>{"n": "1"}<|action_sep|>def f(n):
<|frame_sep|><|line_sep|>{"n": ".."}<|action_sep|>    p = ''
<|frame_sep|><|line_sep|>{"n": "..", "p": "''"}
<|action_sep|>    if n%2 == 1:
<|frame_sep|><|line_sep|>{"n": "..", "p": ".."}
<|action_sep|>        p+='sn'
<|frame_sep|><|line_sep|>{"n": "..", "p": "'sn'"}
<|action_sep|>    for x in range(1, n+1):
<|frame_sep|><|line_sep|>{"n": "..", "p": "..", "x": "1"}
<|action_sep|>        if x%2 == 0:
<|frame_sep|><|line_sep|>{"n": "..", "p": "..", "x": ".."}
<|action_sep|>            p+='ts'
<|frame_sep|><|line_sep|>{"n": "..", "p": "'snts'", "x": ".."}
<|action_sep|>    for x in range(1, n+1):
<|frame_sep|><|line_sep|>{"n": "..", "p": "..", "x": ".."}
<|action_sep|>    return p
<|frame_sep|><|return_sep|><|action_sep|>    return p<|arg_sep|>"'snts'"
<|frame_sep|><|return_sep|><|action_sep|>    return f(1)
<|arg_sep|>"'snts'"<|frame_sep|>
```

Full, line-by-line, execution trace prediction.

Gu et al., CRUXEval, 2024.

Pythonic traces: inference

For single-step execution, replace **<|line_sep|>** with **<|return_sep|>**

```
<|begin_of_text|><|trace_context_start|>
def f(n):
    p = ''
    if n%2 == 1:
        p+='sn'
    else:
        return n*n
    for x in range(1, n+1):
        if x%2 == 0:
            p+='to'
        else:
            p+='ts'
return p

def main():  # << START_OF_TRACE
    return f(1)
<|frame_sep|><|call_sep|>{}<|action_sep|>def main():
<|frame_sep|><|return_sep|>
----END OF PROMPT----
<|action_sep|>    return f(1)
<|arg_sep|>"'snts'"<|frame_sep|>
```

Single-step execution trace prediction.

```
<|begin_of_text|><|trace_context_start|>
def f(n):
    p = ''
    if n%2 == 1:
        p+='sn'
    else:
        return n*n
    for x in range(1, n+1):
        if x%2 == 0:
            p+='to'
        else:
            p+='ts'
return p

def main():  # << START_OF_TRACE
    return f(1)
<|frame_sep|><|call_sep|>{}<|action_sep|>def main():
<|frame_sep|>
----END OF PROMPT----
<|line_sep|>{}<|action_sep|>    return f(1)
<|frame_sep|><|call_sep|>{"n": "1"}<|action_sep|>def f(n):
<|frame_sep|><|line_sep|>{"n": ".."}<|action_sep|>    p = ''
<|frame_sep|><|line_sep|>{"n": "..", "p": "''"}
<|action_sep|>    if n%2 == 1:
<|frame_sep|><|line_sep|>{"n": "..", "p": ".."}
<|action_sep|>        p+='sn'
<|frame_sep|><|line_sep|>{"n": "..", "p": "'sn'"}
<|action_sep|>    for x in range(1, n+1):
<|frame_sep|><|line_sep|>{"n": "..", "p": "..", "x": "1"}
<|action_sep|>        if x%2 == 0:
<|frame_sep|><|line_sep|>{"n": "..", "p": "..", "x": ".."}
<|action_sep|>            p+='ts'
<|frame_sep|><|line_sep|>{"n": "..", "p": "'snts'", "x": ".."}
<|action_sep|>    for x in range(1, n+1):
<|frame_sep|><|line_sep|>{"n": "..", "p": "..", "x": ".."}
<|action_sep|>    return p
<|frame_sep|><|return_sep|><|action_sep|>    return p<|arg_sep|>"'snts'"
<|frame_sep|><|return_sep|><|action_sep|>    return f(1)
<|arg_sep|>"'snts'"<|frame_sep|>
```

Full, line-by-line, execution trace prediction.

Gu et al., CRUXEval, 2024.

Code execution for world modeling: ablations

PRs	Tracing	Forager	CruxEval-O↑	CruxEval-I↑	Oracle SBV NLL↓	Agentic SBV NLL (32k)↓	SBV↑
✗	✗	✗	45.4	44.1	0.64	0.39	14.6
✓	✗	✗	44.6	45.8	**0.55**	0.37	18.6
✓	✓	✗	73.9	51.5	**0.54**	0.38	18.4
✓	✓	✓	**74.5**	**54.8**	**0.54**	**0.29**	**22.1**

- Results for CruxEval-output, CruxEval-input, negative log-likelihood (NLL) on oracle SWE-bench Verified (SBV) patch, NLL on agentic SBV trajectories, and SBV pass@1 scores.

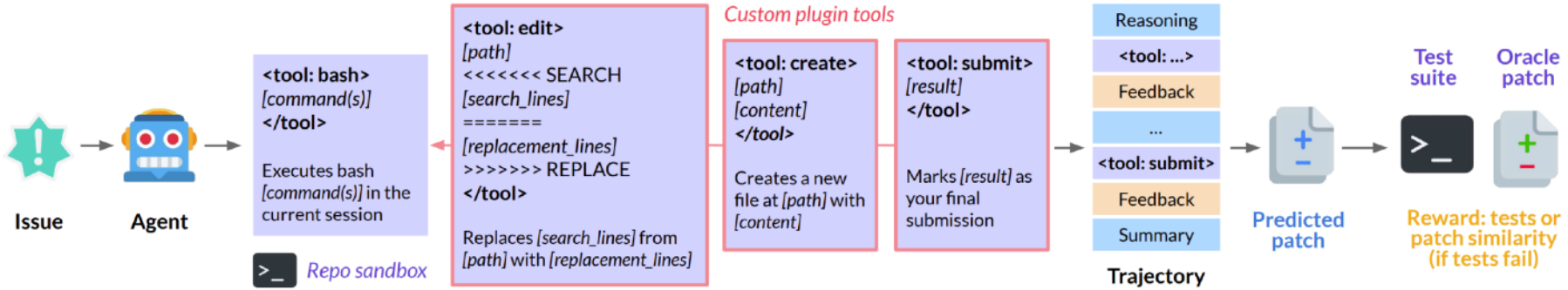
Code execution for world modeling: ablations

PRs	Tracing	Forager	CruxEval-O↑	CruxEval-I↑	Oracle SBV NLL↓	Agentic SBV NLL (32k)↓	SBV↑
✗	✗	✗	45.4	44.1	0.64	0.39	14.6
✓	✗	✗	44.6	45.8	**0.55**	0.37	18.6
✓	✓	✗	73.9	51.5	**0.54**	0.38	18.4
✓	✓	✓	**74.5**	**54.8**	**0.54**	**0.29**	**22.1**

- Results for CruxEval-output, CruxEval-input, negative log-likelihood (NLL) on oracle SWE-bench Verified (SBV) patch, NLL on agentic SBV trajectories, and SBV pass@1 scores.
- First pre-trained one 8B parameter model for 6T tokens, then added 1T additional training data:
  - PRs (datasets derived from GitHub PRs)
    - helps oracle SBV NLL and SBV pass@1, but not the agentic SBV trajectory NLL or CruxEval.
  - Pythonic traces (Tracing):
    - significantly improves CruxEval-input and -output prediction but leaves all SBV-related metrics unaffected.
  - Agentic traces (Forager):
    - improves agentic SBV NLL and SBV pass@1 scores by 3.7pp.
    - also improves, but less-so, CruxEval-input and -output prediction

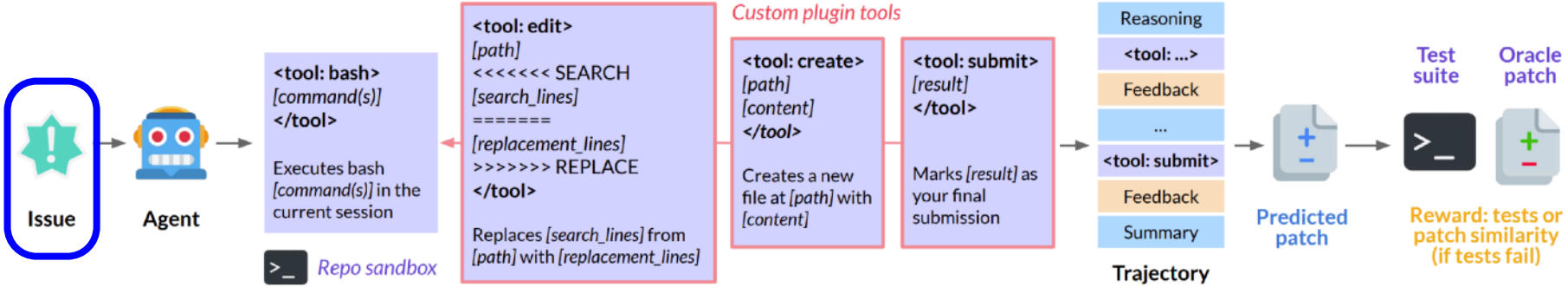
Code execution for reinforcement learning

Code execution for SWE-RL: training instances



Apply the fantastic SWE-Bench recipe to a executable repository image

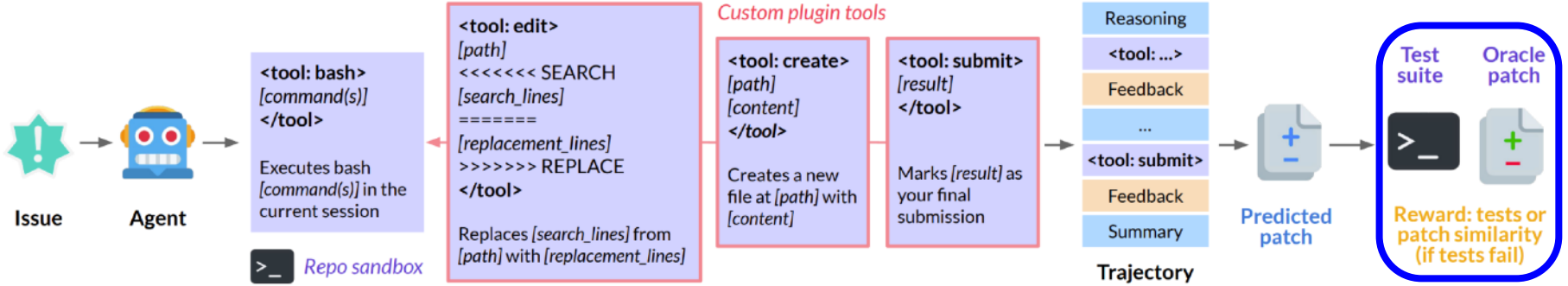
Code execution for SWE-RL: training instances



Apply the fantastic SWE-Bench recipe to a executable repository image

- Match (1:many) the executable (GitHub) repository image to it's (GitHub) Issues & PRs
- For the prompt: extract Issue description

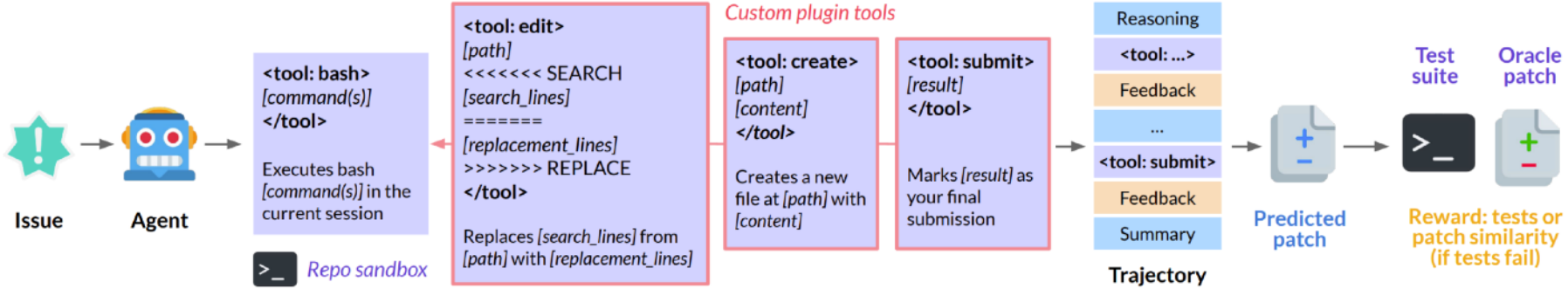
Code execution for SWE-RL: training instances



Apply the fantastic SWE-Bench recipe to a executable repository image

- Match (1:many) the executable (GitHub) repository image to it's (GitHub) Issues & PRs
- For the prompt: extract Issue description
- For the RLVR-signal:
  - Separate the PR's `oracle_patch` into `test_patch` and `solution_patch`
  - Iteratively `git apply` patches & run tests to determine lists:
    - `fail_to_pass` (test what should be addressed in policy's `pred_patch`)
    - `pass_to_pass` (test what should not be regressed in the policy's `pred_patch`)
- To extract RLVR signal to train time:
  - Write harness and repo-specific `log_parser` scripts
  - Convert arbitrary test suite's stdout logs into dict: `{test_name:PASS/FAIL}`
- Classify instance difficulty using the pass@k score from 32 samples & filter out too easy

Code execution for SWE-RL: training instances challenges

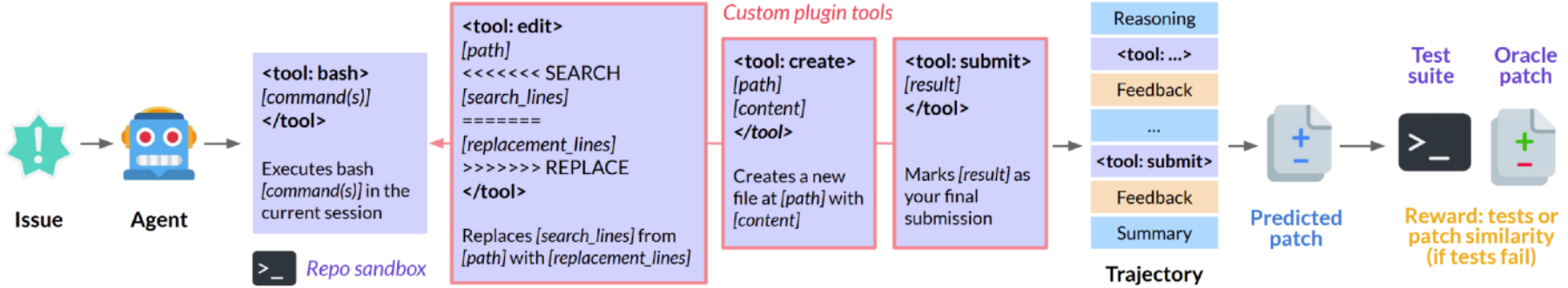


Human curation challenges:

- Not all executable repository images have Issues & PRs, that can be decomposed into test/solution_patch
- Recipe has a narrowing of specificity:
  - Issue: general description of problem —> PR: one specific solution
- In OpenAI's recent retiring of SWE-Bench Verified
  - "35.5% of the [138 SWE-bench Verified problems with low pass@k]… have strict test cases that enforce specific implementation details, invalidating many functionally correct submissions…"

https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/

Code execution for SWE-RL: training instances challenges

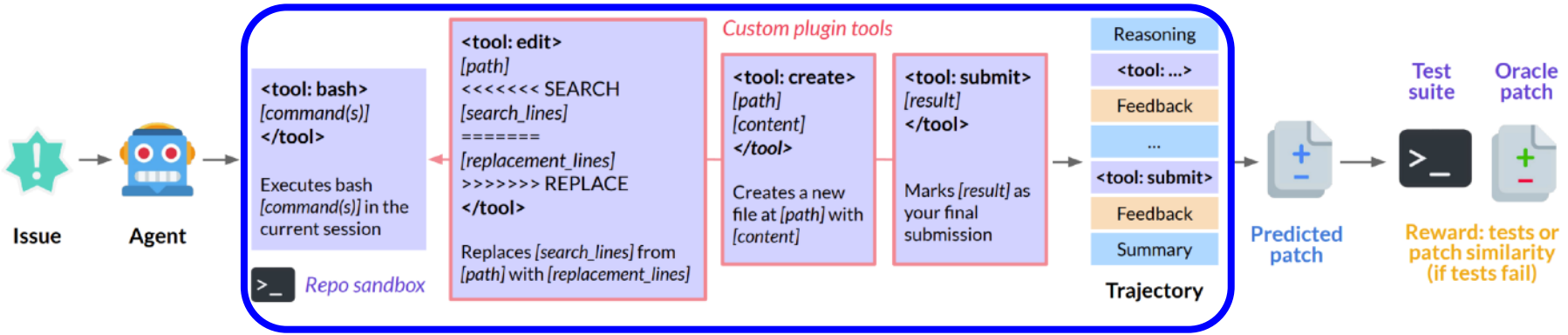


Human curation challenges:

- Not all executable repository images have Issues & PRs, that can be decomposed into test/solution_patch
- Recipe has a narrowing of specificity:
  - Issue: general description of problem —> PR: one specific solution
- In OpenAI's recent retiring of SWE-Bench Verified
  - "35.5% of the [138 SWE-bench Verified problems with low pass@k]… have strict test cases that enforce specific implementation details, invalidating many functionally correct submissions…"
- Costly pass@k required to separate the (too) easy from the hard (and perhaps impossible) instances.
- For GPT-5, OpenAI reports SWE-bench Verified results "on subset of n=477 verified tasks which have been validated on our internal infrastructure"
- Hackable: agent could use `git log` to find future or off-main-branch solutions to the Issue.

https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/, https://openai.com/index/introducing-gpt-5/ , https://github.com/SWE-bench/SWE-bench/issues/465

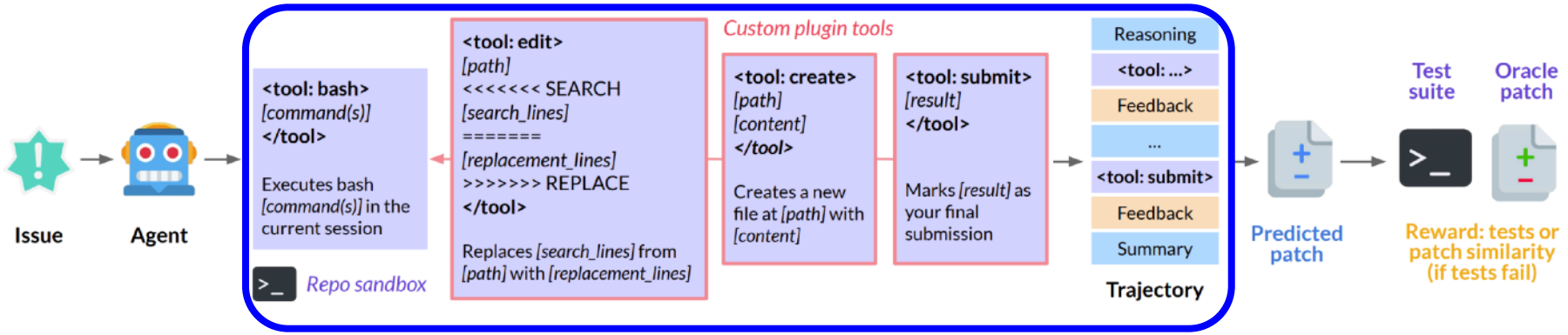
Code execution for SWE-RL: training environment



Despite challenges, we curate internally & from other sources, 12.6k unique instances for training

- Drop the policy into one of the instances with:
  - a stateful bash shell session running in a persistent server process
  - customized tools plugins (`edit`, `create`, `submit`)
    - can be “de-sugared” into simple bash commands
    - we also trained with a bash-only variant

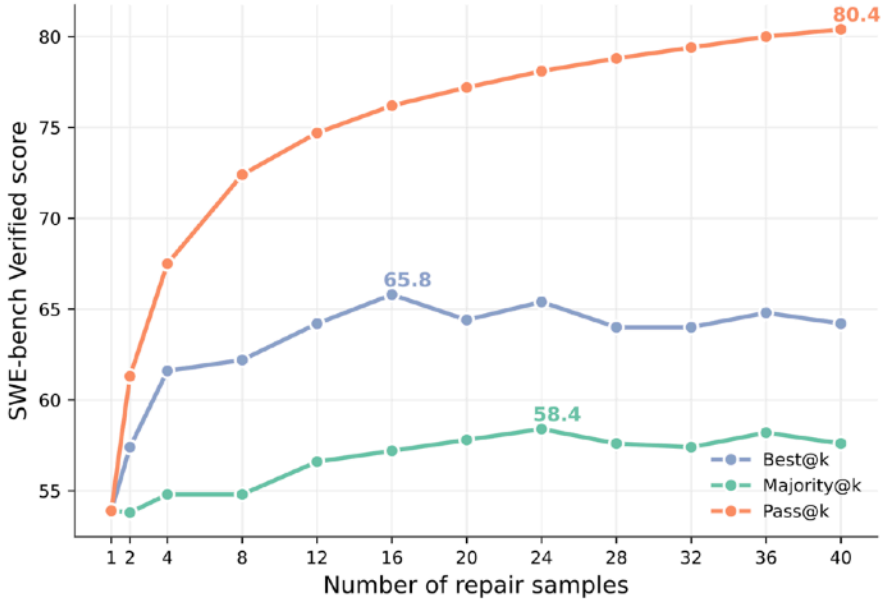
Code execution for SWE-RL: training environment



Despite challenges, we curate internally & from other sources, 12.6k unique instances for training

- Drop the policy into one of the instances with:
  - a stateful bash shell session running in a persistent server process
  - customized tools plugins (`edit`, `create`, `submit`)
    - can be “de-sugared” into simple bash commands
    - we also trained with a bash-only variant
- Multi-turn RL environment
  - System prompt + single user turn populated with the instance metadata (e.g. Issue text)
  - Subsequent user turns are replaced with stdout “feedback” from the bash session
  - Maximum of 128 turns over a context window of 131k tokens.
- Joint RL training (with non-SWE-RL envs) for ~26k step

Code execution for SWE-RL: test-time scaling



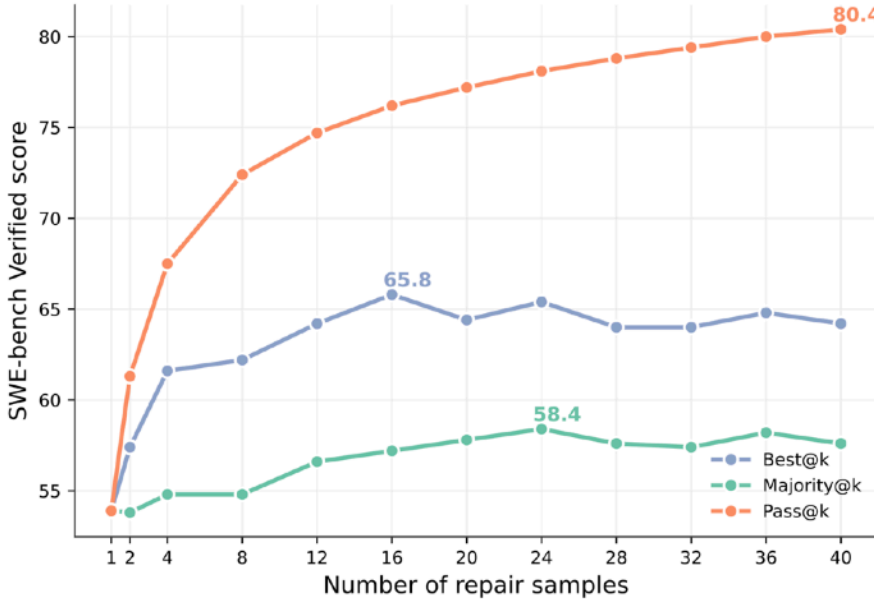
Test-Time-Scaling (TTS) on SWE-bench Verified

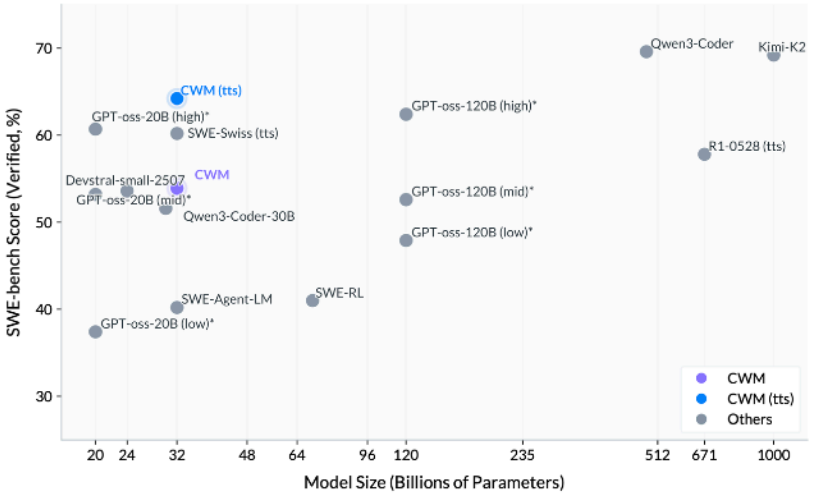
- Best@k:
  - For each instance generate:
    - k candidate solutions
    - 40 novel unit tests
  - Execute test on candidate solutions
  - submit single solution with highest pass-rate

- Pass@k:
  - pass@k improves monotonically with k
  - reaching a success rate of 80.4% at k= 40

- Majority@k:
  - based on exact string matching, without any test generation or execution
  - leads to a pass rate of 58.4 %

Code execution for SWE-RL: test-time scaling





RL for code execution (training instances)

RL for code execution (training instances)

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

Yuxiang Wei[13†*] **Zhiqing Sun**[2†] **Emily McMilin**[1†] **Jonas Gehring**[1] **David Zhang**[1]
Gabriel Synnaeve[1] **Daniel Fried**[14†] **Lingming Zhang**[3†] **Sida Wang**[1†]
[1]Meta FAIR [2]Meta TBD Lab [3]UIUC [4]CMU

RL for code execution (training instances)

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

Yuxiang Wei[13†*] **Zhiqing Sun**[2†] **Emily McMilin**[1†] **Jonas Gehring**[1] **David Zhang**[1]
Gabriel Synnaeve[1] **Daniel Fried**[14†] **Lingming Zhang**[3†] **Sida Wang**[1†]
[1]Meta FAIR [2]Meta TBD Lab [3]UIUC [4]CMU

Wei et al., Toward Training Superintelligent Software Agents through Self-Play SWE-RL, 2025.

RL for code execution (training instances)

- Previously claimed: ~Infinite training data from executed code
- But for RL training instances there remains a human curation bottleneck, many challenges

- Instead lets task an agent with creating RL training instances
- Provide agent with only the executable repository images

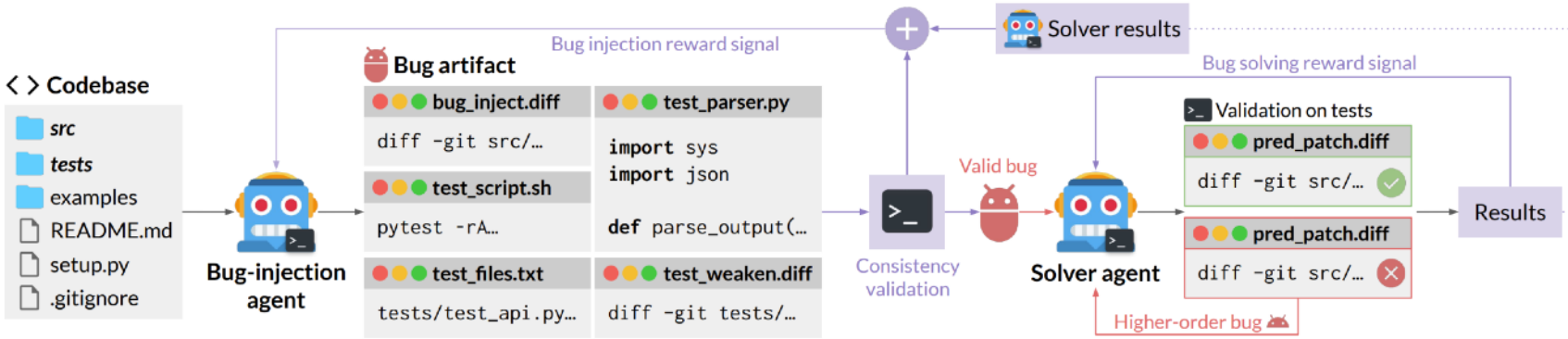
RL for code execution (training instances)

- Previously claimed: ~Infinite training data from executed code
- But for RL training instances there remains a human curation bottleneck, many challenges

- Instead lets task an agent with creating RL training instances
- Provide agent with only the executable repository images

- Inspired by AlphaGo/Zero:
  - Self-play SWE-RL (SSR)
- Single policy, adversarially prompted:
  - Bug-injector: create bugs that are hard to solve
  - Bug-Solver: fix the bug
- Not necessarily zero-sum:
  - Bug-injector: wants the bug to be solvable (but just barely)
  - Bug-Solver: wants to fix the bug!

RL for code execution (training instances): SWE-RL Self-play



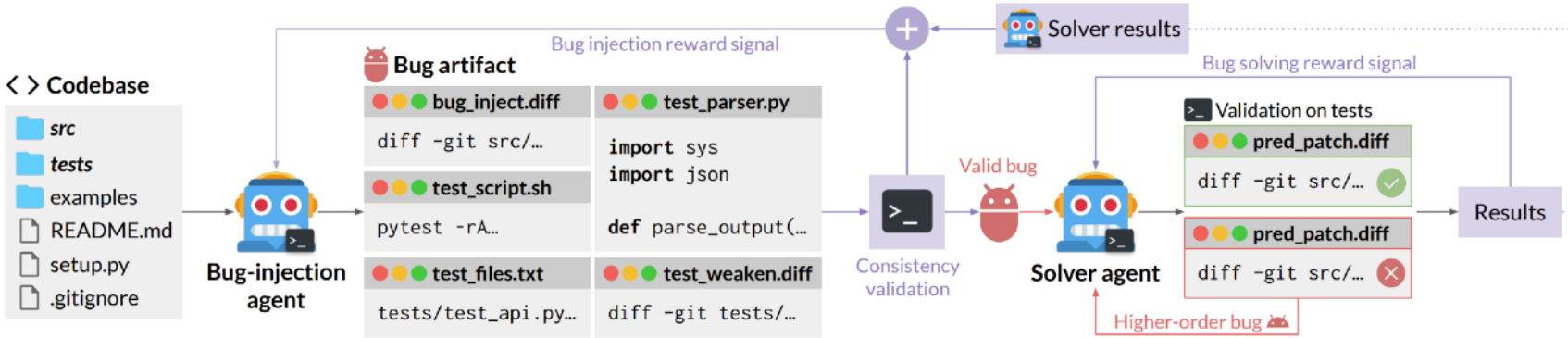
Bug-injector:

- bug_inject.diff: Modifies codebase to add a realistic bug, that causes existing tests to fail
- test_weaken.diff: Now make tests pass, despite the bug, as real world codebases have (currently uncaught) bugs

Bug is grounded in code-execution and artifacts:

- Uses `git log` to view/revert prior commits
- Unsolved (thus buggy) attempts by Solver are used as new (Higher-order) bugs.

RL for code execution (training instances): SWE-RL Self-play



Bug-injector:

- bug_inject.diff: Modifies codebase to add a realistic bug, that causes existing tests to fail
- test_weaken.diff: Now make tests pass, despite the bug, as real world codebases have (currently uncaught) bugs

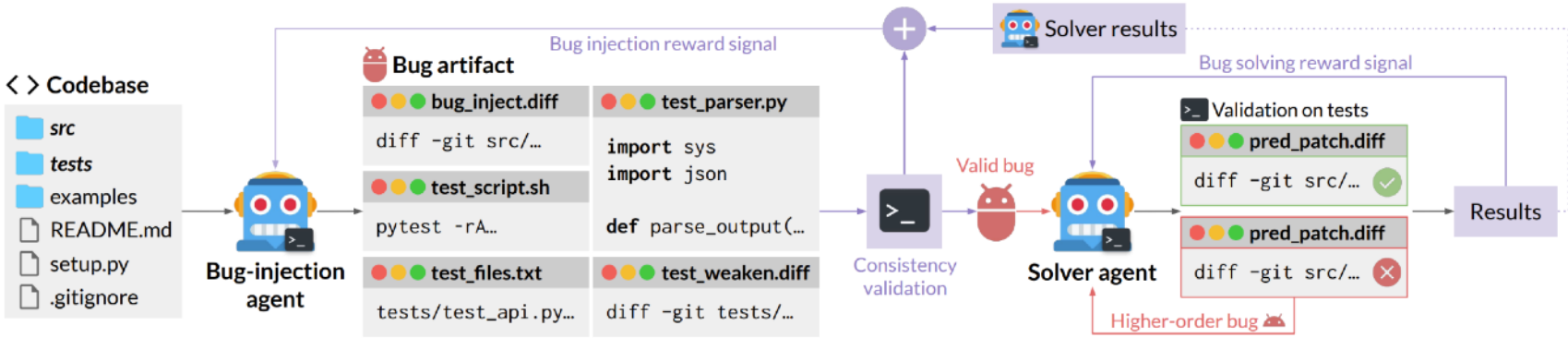
Bug is grounded in code-execution and artifacts:

- Uses `git log` to view/revert prior commits
- Unsolved (thus buggy) attempts by Solver are used as new (Higher-order) bugs.

Bug-solver:

- Placed in bash session in an executable repository image
- Instead of GitHub Issue, the reversed test_weaken.diff serves as the issue description in prompt
  - Akin to test-driven development

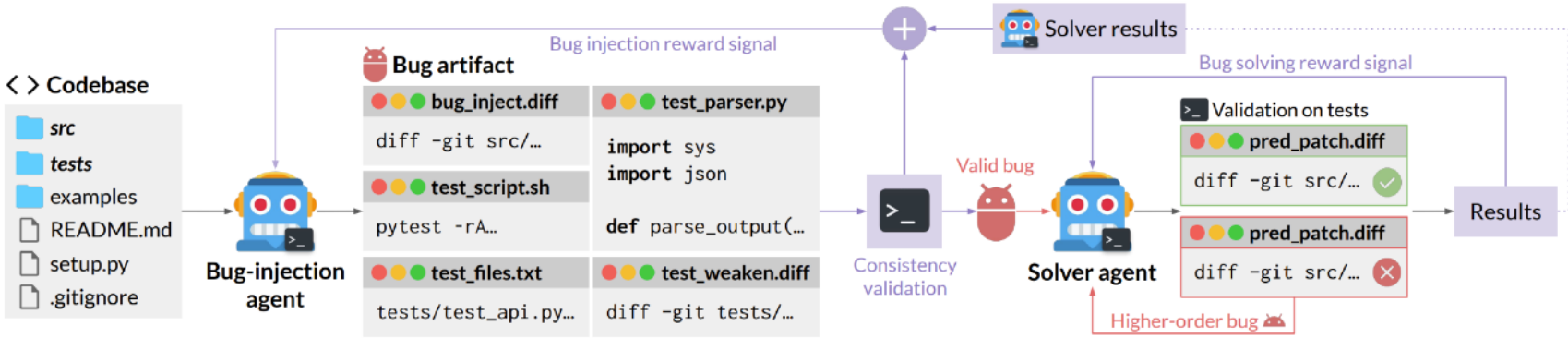
RL for code execution (training instances): No humans



No human curation:

- No GitHub Issue matching or underspecification: the reversed test-weakening patch as a specification of the gap in required behavior
  - Not ideal as can lead to reward-hacking behaviors (e.g., overfitting to the tests), but we didn't observe this in practice

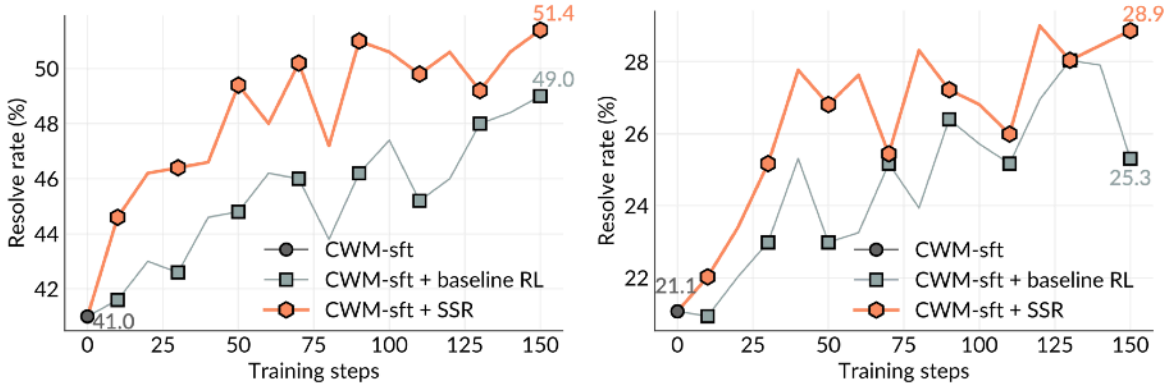
RL for code execution (training instances): No humans



No human curation:

- No GitHub Issue matching or underspecification: the reversed test-weakening patch as a specification of the gap in required behavior
  - Not ideal as can lead to reward-hacking behaviors (e.g., overfitting to the tests), but we didn't observe this in practice
- No GitHub PRs or test_patch required: Bug-injector saves the oracle_patch
- No log parsers required for grading: Bug-injector writes these too.
- No costly pass@k for separating easy/hard problems: Bug-injector rewarded to propose bugs near the frontier of the Solver's current capability

RL for code execution (training instances): Training and Results

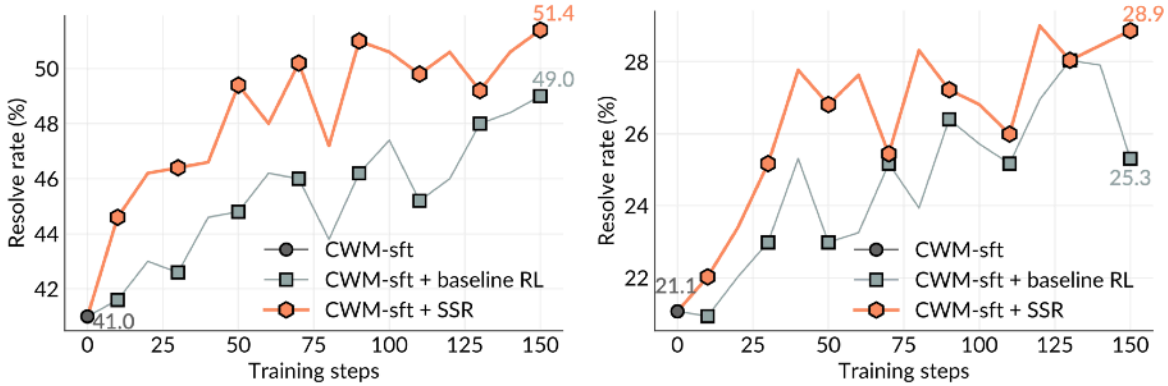


(a) SWE-bench Verified (b) SWE-Bench Pro

Training

- We train off of the 32B CWM-sft chkpt, using the same/very-similar infra, params, and 12.6k training instances as CWM, two models:
  - CWM-sft + baseline RL: same as SWE-RL in CWM, with access to human curated data, e.g. natural language issue descriptions, fail_to_pass tests
  - CWM-sft + SSR: provided only executable repository images, requiring the model to discover problems, formulate solutions, and validate them entirely through self-play
- Both use same executable repository images; difference is baseline RL access to additional human curated metadata

RL for code execution (training instances): Training and Results



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

Training

- We train off of the 32B CWM-sft chkpt, using the same/very-similar infra, params, and 12.6k training instances as CWM, two models:
  - CWM-sft + baseline RL: same as SWE-RL in CWM, with access to human curated data, e.g. natural language issue descriptions, fail_to_pass tests
  - CWM-sft + SSR: provided only executable repository images, requiring the model to discover problems, formulate solutions, and validate them entirely through self-play
- Both use same executable repository images; difference is baseline RL access to additional human curated metadata

Results

- Evaluation on SWE-bench Verified (500 instances) SWE-Bench Pro (public split, 731 instances).
- We perform one attempt for each problem without parallel test-time scaling or ranking.
- SSR can outperform baseline RL on both benchmarks across the entire training trajectory

Conclusion

Revisiting:

- For world modeling ~infinite training data from executed code
  - Establish some initial state and kick off execution
- While human curation remains a bottleneck, promising initial efforts to address this.
  - CI for executable repository image building
  - Self-play for RL instance building
- Via grounding training in code execution data, the CWM model:
  - has improved pythonic next state and output prediction.
  - and (hypothesized) improved efficiency in RL training given pre-training on the transition dynamics
- Please download the CWM weights and reach out with feedback!

Thanks for your time and attention
Please reach out: emcmilin@meta.com

Questions?