Towards a Neural Debugger for Python

Paper presentation at KTH

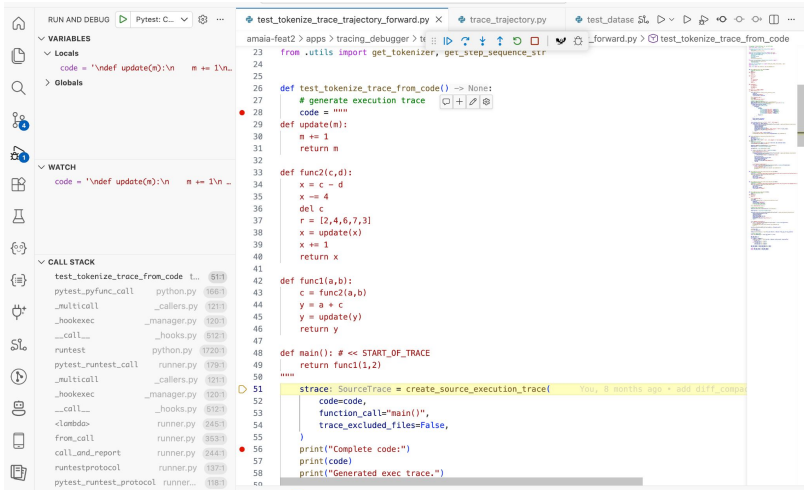
Maximilian Beck, maxbeck@meta.com, maxbeck.ai

with Jonas Gehring, Jannik Kossen, Gabriel Synnaeve

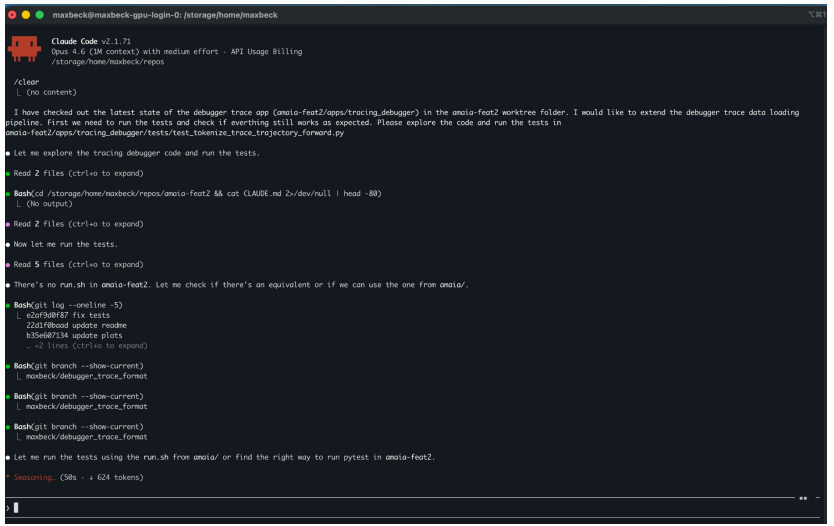
May 2026

Motivation

Coding 1-2 years ago



Coding today



Current code models:

- Use tools to find and execute relevant parts of a large program
- Reason about specific designs and their trade-offs

This work →

- **Understand program flow without executing the program**

Teaching LLMs semantics of code with execution traces

Approaches to expose LLMs to execution data:

- ➔ Include feedback of test results, error messages or runtime/shell outputs into RL
- ➔ Iterative generation loops

This work: **Train LLMs on execution traces, i.e. records of variable states and control flow transitions of executed programs**

Prominent example: Code World Model (CWM)

- ➔ 32B Coding model trained on execution traces during mid-training
- ➔ CWM can simulate line-by-line execution of Python programs, which turns CWM effectively into a "*neural interpreter*"
- ➔ Predicts every execution step (every loop iteration and variable change)

CWM: An Open-Weights LLM for Research on Code Generation with World Models
Meta FAIR CodeGen Team

➔ **can lead to very long sequences, limiting execution prediction to shorter programs & overlooks how human developers inspect programs**

Neural debuggers: Main idea

We introduce the concept of ***neural debuggers***:

➜ LLMs that can simulate program execution while providing core functionalities of debuggers

➜ **Main idea:** Interleave program states with actions inspired by traditional debuggers such as `step_into`, `step_return`, `step_over`, `breakpoint`

➜ Enables the model to control which part of the execution trace it predicts

Advantages over traditional debuggers:

- Simulate execution for non-executable or partially specified programs
- Efficient reinitialization of program state and execution context through prompting and regeneration
- Approximate prediction of inverse execution (which is ambiguous): Can infer or sample plausible preceding program states or inputs

Use cases in agentic workflows:

- Learned world models for simulated debugging environments
- Enable coding agents to control real debuggers

➜ This work: Validation of the concept, but no application to agentic workflows

Outline: Towards a Neural Debugger for Python

- Background on Python execution traces
- Neural debuggers
  - Formulation as MDP
  - Data pipeline
- Experimental results
- Limitations & Future work
- Conclusion

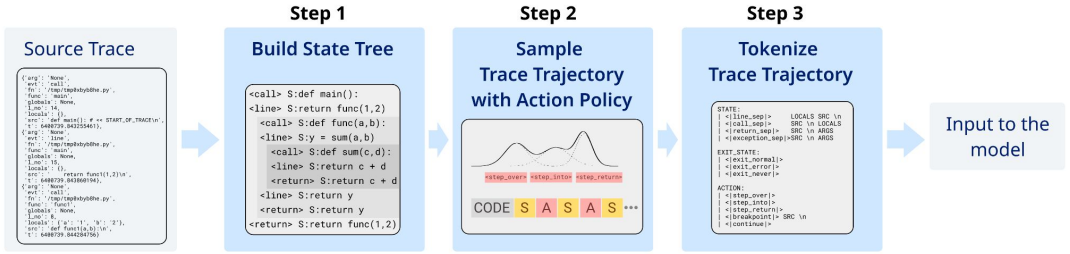


Figure 1 Neural Debugger Data Pipeline.

Background: Python execution traces

- Python program execution is organized as a sequence of frame stack objects
- Frame stack objects:
  - contain current local and global variables, source line & execution event (`line`, `call`, `return`, …)
  - stored as call stack
- Python provides access to frame stack objects via `sys.settrace(tracefunc)`, where `tracefunc(frame,event,arg)` is a custom trace function
  - trace functions are used to implement debuggers

➜ **We use a custom trace function to record the execution traces for our neural debuggers**

```
{'arg': 'None',
 'evt': 'line',
 'fn': '/tmp/tmp0xbyb8he.py',
 'func': 'main',
 'globals': None,
 'l_no': 15,
 'locals': {},
 'src': '    return func1(1,2)\n',
 't': 6400739.843860194},
```

```
{'arg': 'None',
 'evt': 'call',
 'fn': '/tmp/tmp0xbyb8he.py',
 'func': 'func1',
 'globals': None,
 'l_no': 8,
 'locals': {'a': '1', 'b': '2'},
 'src': 'def func1(a,b):\n',
 't': 6400739.844284756}
```

Recorded stack frame series

https://docs.python.org/3/library/sys.html#sys.settrace

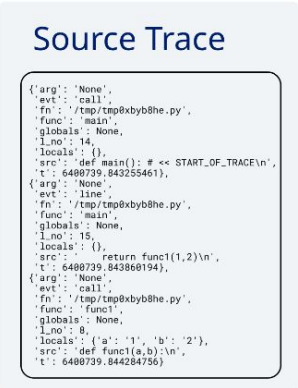
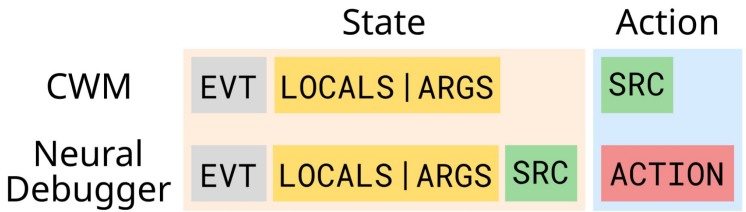


Figure 1 Neural Debugger Data Pipeline.

Neural debuggers as MDPs: State & Actions

- A debugger holds the state of the program being debugged and receives actions that control program execution.
- We formalize the interactive debugger as MDP given by the tuple $(\mathcal{S}, \mathcal{A}, P, R, s_0)$
  - $\mathcal{S}$: space of program states
  - $\mathcal{A}$: set of available debugger actions:

    { `<step_into>, <step_over>, <step_return>,`
    `<breakpoint> SRC, <continue>` }



  - P : $\mathcal{S} \times \mathcal{A} \to \mathcal{S}$ transition dynamics steered by the debugged program
  - R : reward function ← No reward function in this work (focus on state prediction)
  - s_0 : set of possible entry points

CWM: An Open-Weights LLM for Research on Code Generation with World Models. Meta FAIR CodeGen Team. 2025.

Neural debuggers as MDPs: State Tree

code.py

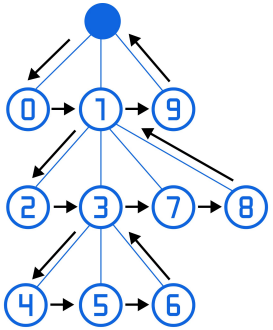
```
def sum(c,d):
    return c + d

def func(a,b):
    y = sum(a,b)
    return y

def main():
    return func(1,2)
```

Execution trace

```
<call> S:def main():
<line> S:return func(1,2)
<call> S:def func(a,b):
<line> S:y = sum(a,b)
<call> S:def sum(c,d):
<line> S:return c + d
<return> S:return c + d
<line> S:return y
<return> S:return y
<return> S:return func(1,2)
```

State Tree

```
⓪<call> S:def main():
①<line> S:return func(1,2)
  ②<call> S:def func(a,b):
  ③<line> S:y = sum(a,b)
    ④<call> S:def sum(c,d):
    ⑤<line> S:return c + d
    ⑥<return> S:return c + d
  ⑦<line> S:return y
  ⑧<return> S:return y
⑨<return> S:return func(1,2)
```

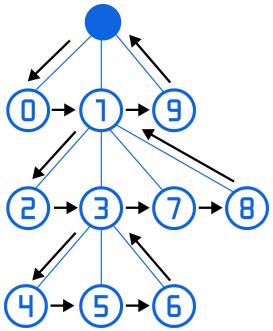


Build state tree from execution trace

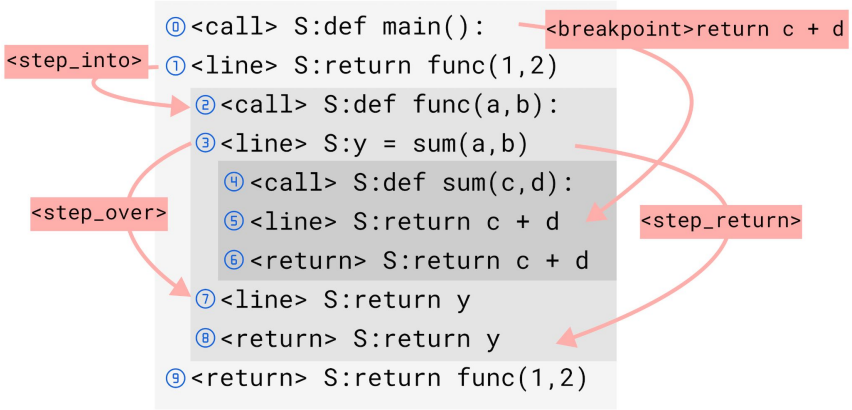
- The state tree reconstructs the runtime call stack by tracking the `call` and `return` events
- Program states from a function are child nodes of the calling `line` event node
- The depth of a node in the tree corresponds to the depth of the call stack at the given program state

Neural debuggers as MDPs: Transition model

- The transition model of the neural debugger is defined by traversal rules on the state tree
- Each debugger action results in a transition from a starting node to a target node
- Different actions have different starting and target nodes

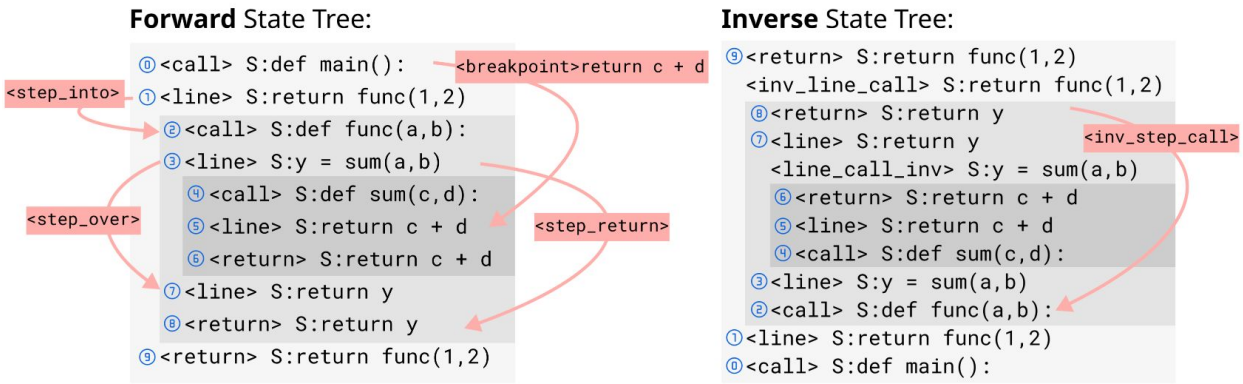


State Tree



Forward and inverse execution prediction

- Processing execution traces as state trees
  enables inverse execution prediction & inverse debugging
- **Inverse** state tree is the **reversed forward** state tree
  - Line events with function calls are duplicated with special event type
  - No `<breakpoint>` action, same `<step_into>`, `<step_over>`
  - `<step_return>` action is repurposed to `<inv_step_call>`



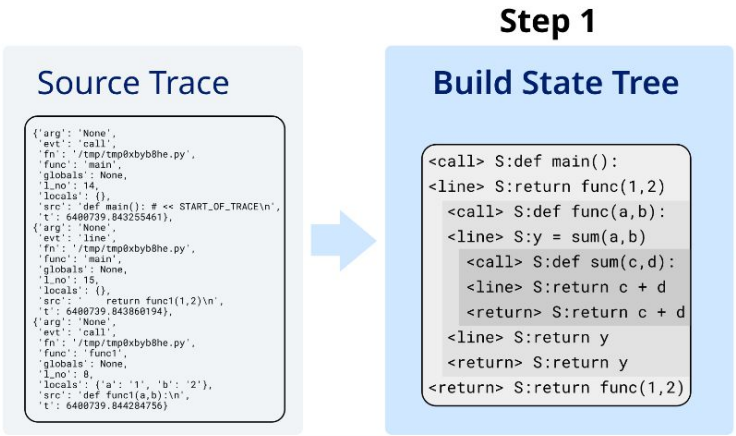
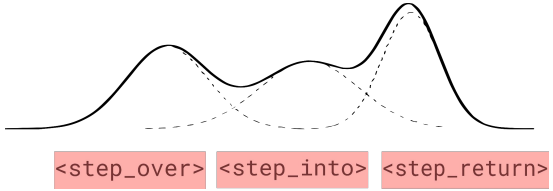


Figure 1 Neural Debugger Data Pipeline.

Debugger trace trajectory sampling

- From each state tree, we generate debugger trace trajectories by sampling actions random actions
- We sample from a mixture of categorical distributions
- By adjusting the distributions, we ensure action diversity and sufficient trajectory lengths



Policy Prob	step_into	step_over	step_return	breakpoint	continue
0.5	0.45	0.2	0.2	0.1	0.05
0.5	0.5	0.5	-	-	-

Table A.1 Action probabilities for the action policy mix used to generate our debugger trace dataset.

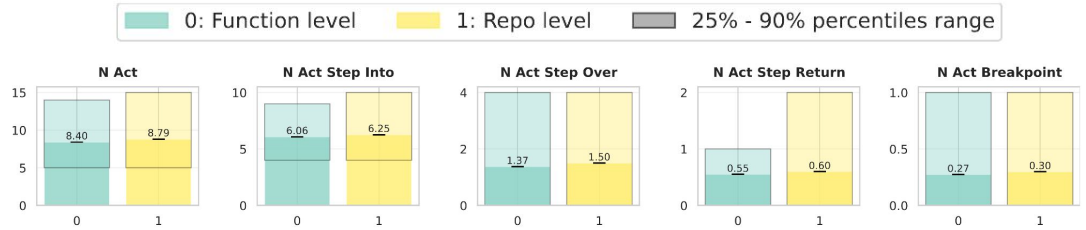


Figure 5 Average token, action and event counts of forward debugger trajectory datasets.

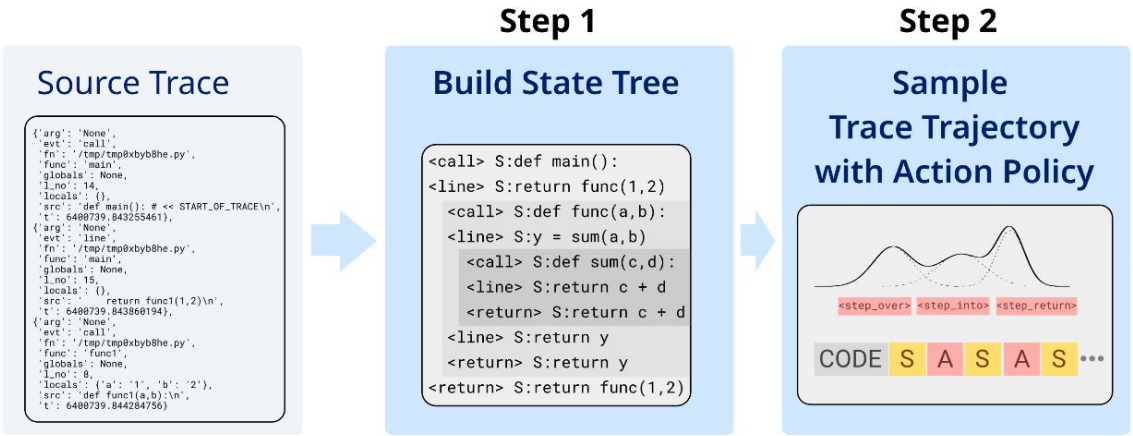
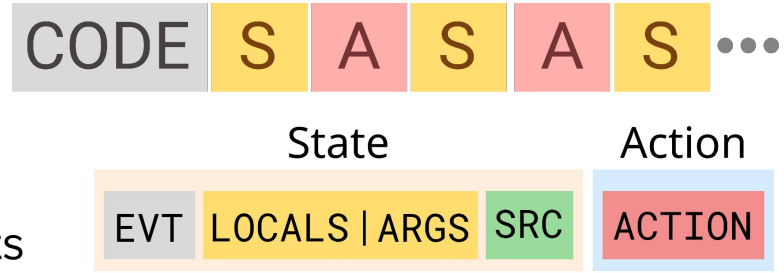


Figure 1 Neural Debugger Data Pipeline.

Tokenization of debugger trace trajectories

- Debugger trace trajectories are state-action sequences
- We use special separator tokens that mark the beginning of state and action segments



- LOCALS representation:
  - Serialize the the LOCALS dictionary as JSON
  - Display only local variables that have changed (for shorter sequences) and insert `"..":".."` to indicate omitted, unchanged entries
  - Convert arbitrary Python objects via their `__repr__()` methods.

Debugger trace example

```
<|begin_of_text|><|trace_context_start|>
def f(single_digit):
    result = []
    for c in range(1, 11):
        if c != single_digit:
            result.append(c)
    return result

def main():
    return f(5)

<|frame_sep|><|call_sep|>
<|src_sep|>def main():
<|arg_sep|>{}
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{}
<|src_sep|> return f(5)
<|action_sep|><|step_into|>

<|frame_sep|><|call_sep|>
<|src_sep|>def f(single_digit):
<|arg_sep|>{"single_digit": "5"}
<|action_sep|><|breakpoint|> return result

<|frame_sep|>
----END OF PROMPT----

<|line_sep|>
<|arg_sep|>{"single_digit": "5",
"result": "[1, 2, 3, 4, 6, 7, 8, 9, 10]", "c": "10"}
<|src_sep|> return result
```

Output prediction with **breakpoint** action.

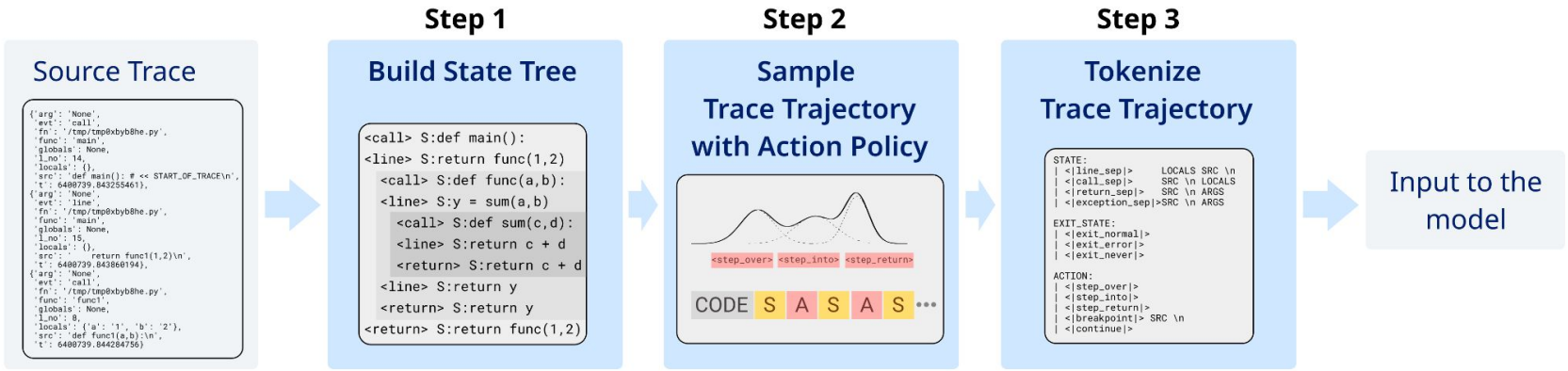


Figure 1 Neural Debugger Data Pipeline.

Experimental results

1) How does finetuning CWM, compare to pre-training smaller Transformer models on neural debugger traces?

2) What are prediction accuracies of individual state elements, i.e. where do neural debuggers make errors?

3) How do neural debuggers perform on input and output prediction downstream tasks?

Experimental setup

- Train neural debugger models by finetuning CWM (32B parameters) and pre-training small 1.8B Transformer models:
  - CWM 32B: fine tune on 50B tokens on debugger trace data only
  - 1.8B Transformer: train on 50B / 150B tokens with different data mixes (trace data, web data, code data)

- Execution trace dataset from CWM: function-level & repository-level
  - 120M functions, 21k repository images, 262k code contest solutions
  - We sample 15B repo-level and 100B function-level (forward + inverse) tokens

Repo-level contain more more function calls (i.e., call and return events), more exceptions, and longer sequences (mainly due to arbitrary Python objects) for the same number of actions

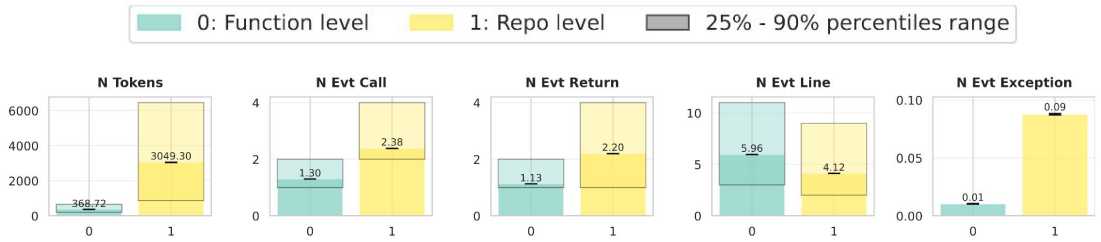
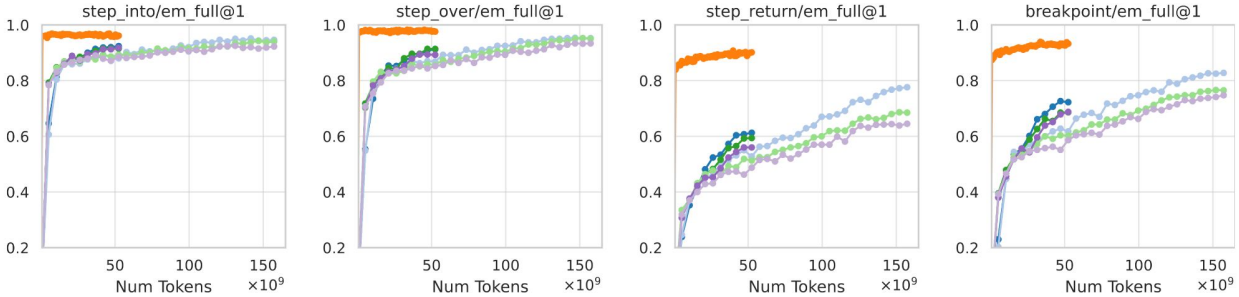


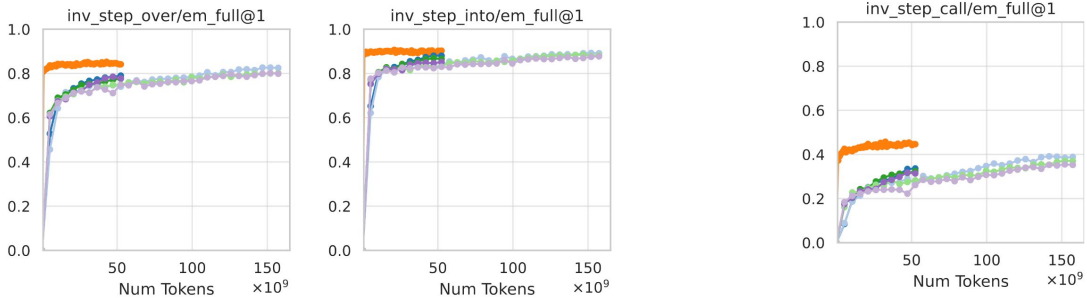
Figure 5 Average token, action and event counts of forward debugger trajectory datasets.

1) Finetuning and pre-training neural debuggers

Step actions **Jump actions**



(a) Forward prediction.



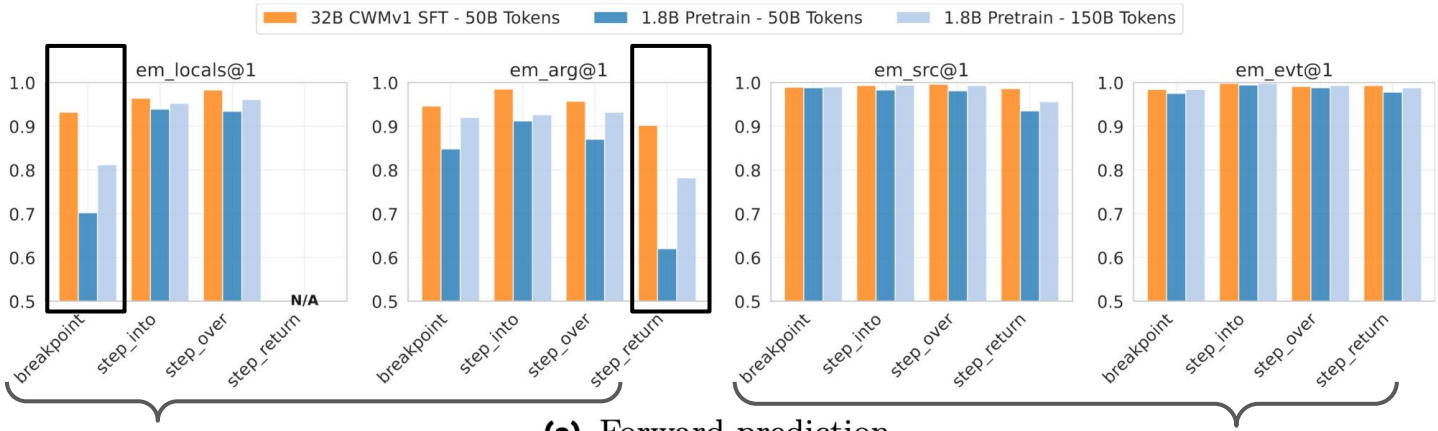
(b) Inverse prediction.

- ➜ Step actions are easier than jump actions
- ➜ Inverse execution is learnable
- ➜ Small models are good neural debuggers

Exact match results during training averaged over 800 trajectories from the function-level validation set (same conclusions for repo level)

2) Next program state prediction by program element

Where do neural debuggers make errors?



(a) Forward prediction.

Errors in local variables (esp. for jump actions)

Source lines & events are predicted reliably

Exact match (em) next state prediction by state component
averaged over 800 trajectories from the function-level validation set
(same conclusions for repo level)

3) Input and output prediction on CruxEval

➜ Neural debugger finetuned CWM model improves over original CWM model (achieved 58.1% on output, Table 8, Trace Step)

➜ Small neural debuggers trained from scratch achieve remarkable performance for their size

Model	Tokens	Input	Output	
		inv_step_call	step_return	breakpoint
1.8B Transformer Pretrain	50B	40.7	34.4	44.9
1.8B Transformer Pretrain	150 B	53.6	48.0	57.7
32B CWM Finetune	50B	66.5	77.9	83.2

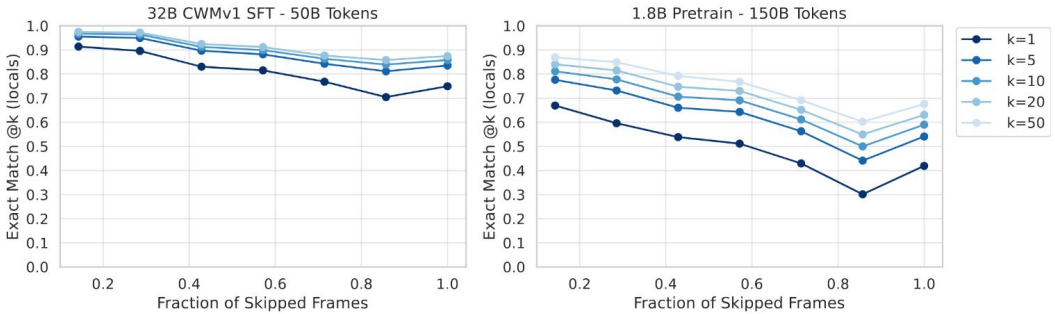
Table 1 CruxEval input and output pass@1 scores for single step prediction with neural debugger actions `inv_step_call`, `step_return`, and `breakpoint` (greedy decoding). All models are trained on debugger trace data only.

Prediction accuracy for increasing prediction horizon on CruxEval:

- Generate multiple prompts by inserting a variable number of step actions before breakpoint action
- Measure exact match @k scores

➜ Prediction accuracy decreases as the prediction horizon increases (steeper decline for smaller models)

➜ Larger sampling budgets (higher k) partially mitigate the accuracy drop



Output prediction with forward `breakpoint` action (exact match of local variable dict).

Limitations & Future work

➜ We have demonstrated the viability of our neural debugger concept & data pipeline.

- **No application to agentic program repair, reasoning & tool use with neural debuggers so far**

Future work:

- Expanding & improving data generation
  - More execution data & more programming language
  - Better action policies for data generation
- Improving inverse debugging
  - Model ambiguity with feasible value sets
  - Metrics that account for ambiguity
- Better Python object representations (than `__repr__()`)
  - Especially for complex data structures

Conclusion

- Training on execution trace data enables simulation of program execution with LLMs
- Neural debuggers enable line-by-line execution of computer programs conditioned on common debugger actions (`step_into`, `step_over`, `breakpoint`, `step_return`)
- We formalized the neural debugger as MDP, where states comprise program variables and source lines, and transitions are defined by traversal rules on a tree data structure reconstructed from the program's call stack
- Future work: Integrate neural debuggers into reasoning & agentic workflows

Thank you!

Check out the paper:
https://arxiv.org/abs/2603.09951

Maximilian Beck

maxbeck@meta.com, maxbeck.ai

Appendix

Model	Tokens	Input	Output	
		`inv_step_call`	`step_return`	`breakpoint`
1.8B Transformer Pretrain	50B	40.7	34.4	44.9
1.8B Transformer Pretrain	150 B	53.6	48.0	57.7
32B CWM Finetune	50B	66.5	77.9	83.2

Table 1 CruxEval input and output pass@1 scores for single step prediction with neural debugger actions `inv_step_call`, `step_return`, and `breakpoint` (greedy decoding). All models are trained on debugger trace data only.

Formal neural debugger language grammar

```
NEURAL_DEBUGGER_TRACE:
| <|trace_context|> CODE_CONTEXT STATE_ACTION { STATE_ACTION } <|trace_end|>

STATE_ACTION:
| <|frame_sep|> STATE <|action_sep|> ACTION
| <|frame_sep|> EXIT_STATE
```

Forward

```
STATE:
| <|line_sep|>      <|arg_sep|> LOCALS <|src_sep|> SRC \n
| <|call_sep|>      <|src_sep|> SRC \n <|arg_sep|> LOCALS
| <|return_sep|>    <|src_sep|> SRC \n <|arg_sep|> ARGS
| <|exception_sep|> <|src_sep|> SRC \n <|arg_sep|> ARGS

EXIT_STATE:
| <|exit_normal|>
| <|exit_error|>
| <|exit_never|>

ACTION:
| <|step_over|>
| <|step_into|>
| <|step_return|>
| <|breakpoint|> SRC \n
| <|continue|>
```

Inverse

```
STATE:
| <|inv_line_sep|>      <|src_sep|> SRC \n <|arg_sep|> LOCALS
| <|inv_call_sep|>      <|src_sep|> SRC \n <|arg_sep|> LOCALS
| <|inv_return_sep|>    <|src_sep|> SRC \n <|arg_sep|> ARGS
| <|inv_exception_sep|> <|src_sep|> SRC \n <|arg_sep|> ARGS

| <|inv_line_call_sep|> <|src_sep|> SRC \n

EXIT_STATE:
| <|inv_exit|>

ACTION:
| <|inv_step_over|>
| <|inv_step_into|>
| <|inv_step_call|>
| <|inv_continue|>
```

Figure 4 Formal neural debugger language grammar. | indicates an OR-statement, {} indicate none or more elements, and : denotes an assignment. Whitespaces are shown for illustration purposes only. <|.|> indicate special tokens, LOCALS is the local variable dictionary, ARGS are return or exception arguments, and SRC denotes the source line.

Length distribution of debugger trace dataset

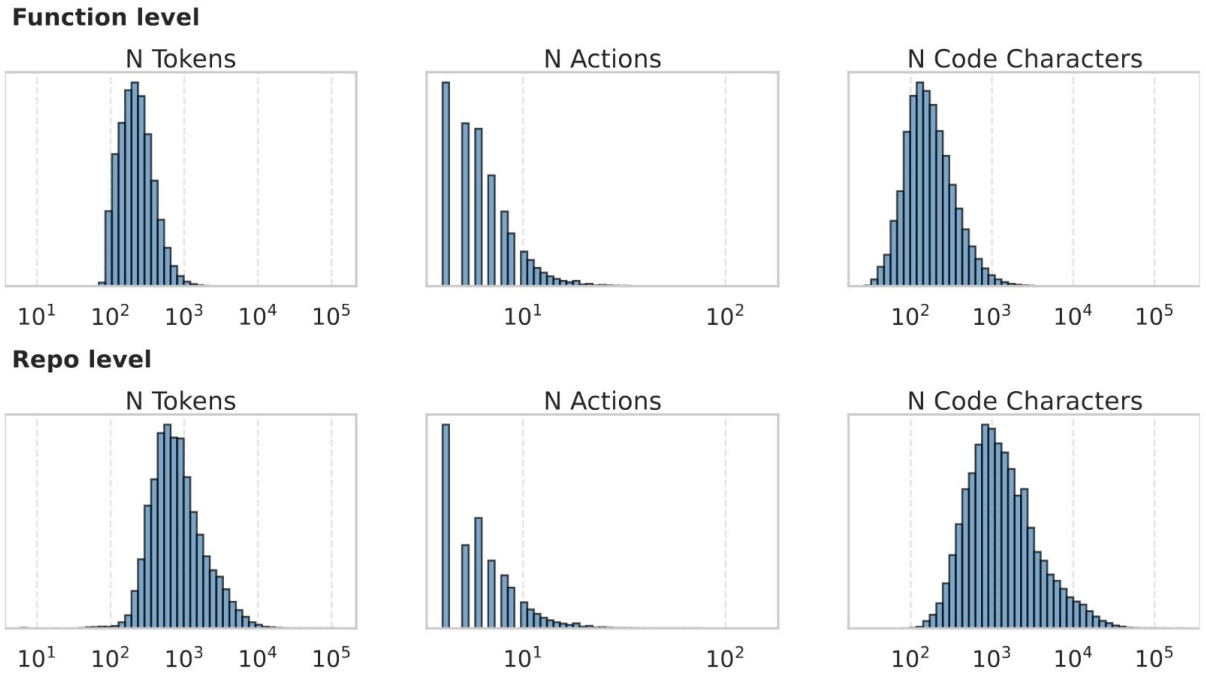


Figure A.1 Length distributions of the debugger trace dataset. We show the length distribution histograms of debugger trajectories in number of tokens and actions, and show the distribution of the code length in number of characters. Even though the number of actions are similar for function-level and repository-level data, the average token count for repository-level trajectories is significantly higher (see also Section 4.3). The reasons for the longer trajectories are more arbitrary python objects in their local variable state dictionaries and larger code context (see right column).

CruxEval prompts

```
<|begin_of_text|><|trace_context_start|>
def f(single_digit):
    result = []
    for c in range(1, 11):
        if c != single_digit:
            result.append(c)
    return result

def main():
    return f(5)

<|frame_sep|><|call_sep|>
<|src_sep|>def main():
<|arg_sep|>{}
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{}
<|src_sep|> return f(5)
<|action_sep|><|step_into|>

<|frame_sep|><|call_sep|>
<|src_sep|>def f(single_digit):
<|arg_sep|>{"single_digit": "5"}
<|action_sep|><|step_return|>

<|frame_sep|>
----END OF PROMPT----


<|return_sep|>
<|src_sep|> return result
<|arg_sep|>"[1, 2, 3, 4, 6, 7, 8, 9, 10]"
```

(a) Output prediction with **step_return** action.

```
<|begin_of_text|><|trace_context_start|>
def f(single_digit):
    result = []
    for c in range(1, 11):
        if c != single_digit:
            result.append(c)
    return result

def main():
    return f(5)

<|frame_sep|><|call_sep|>
<|src_sep|>def main():
<|arg_sep|>{}
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{}
<|src_sep|> return f(5)
<|action_sep|><|step_into|>

<|frame_sep|><|call_sep|>
<|src_sep|>def f(single_digit):
<|arg_sep|>{"single_digit": "5"}
<|action_sep|><|breakpoint|> return result

<|frame_sep|>
----END OF PROMPT----

<|line_sep|>
<|arg_sep|>{"single_digit": "5",
"result": "[1, 2, 3, 4, 6, 7, 8, 9, 10]", "c": "10"}
<|src_sep|> return result
```

(b) Output prediction with **breakpoint** action.

Figure B.3 CruxEval output prompt for the **step_return** and the **breakpoint** actions. The outcome of the **step_return** action is the frame with the return event and argument, while the outcome of the **breakpoint** action is the line event frame, including the local variables dictionary at the source line of the return statement. Predictions are generated by the 1.8 B parameter neural debugger trained on 150 B debugger trace tokens. Some line breaks are inserted for illustration purposes.

```
<|begin_of_text|><|trace_context_start|>
def f(single_digit):
    result = []
    for c in range(1, 11):
        if c != single_digit:
            result.append(c)
    return result

<|frame_sep|><|inv_return_sep|>
<|src_sep|> return result
<|arg_sep|>"[1, 2, 3, 4, 6, 7, 8, 9, 10]"
<|action_sep|><|inv_step_call|>

<|frame_sep|>
----END OF PROMPT----

<|inv_call_sep|>
<|src_sep|>def f(single_digit):
<|arg_sep|>{"single_digit": "5"}
```

CruxEval input prompt

Long Example

```
<|begin_of_text|><|trace_context_start|>
def count(s, t):
    n = 0
    for c in s:
        n += int(c == t)
    return n

def main():
    return count("berry", "r")
<|frame_sep|><|call_sep|>
<|src_sep|>def main():<|arg_sep|>{}
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{}<|src_sep|> return count("berry", "r")
<|action_sep|><|step_into|>

<|frame_sep|><|call_sep|>
<|src_sep|>def count(s, t):<|arg_sep|>{"s": "'berry'", "t": "'r'"}
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": ".."}<|src_sep|> n = 0
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..", "n": "0"}<|src_sep|> for c in s:
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..", "c": "'b'"}<|src_sep|> n += int(c == t)
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": ".."}<|src_sep|> for c in s:
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..", "c": "'e'"}<|src_sep|> n += int(c == t)
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": ".."}<|src_sep|> for c in s:
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..", "c": "'r'"}<|src_sep|> n += int(c == t)
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..", "n": "1"}<|src_sep|> for c in s:
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": ".."}<|src_sep|> n += int(c == t)
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..", "n": "2"}<|src_sep|> for c in s:
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..", "c": "'y'"}<|src_sep|> n += int(c == t)
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": ".."}<|src_sep|> for c in s:
<|action_sep|><|step_into|>

<|frame_sep|><|line_sep|>
<|arg_sep|>{"..": "..}<|src_sep|> return n
<|action_sep|><|step_into|>

<|frame_sep|><|return_sep|><|src_sep|> return n
<|arg_sep|>"2"
<|action_sep|><|step_into|>

<|frame_sep|><|return_sep|>
<|src_sep|> return count("berry", "r")<|arg_sep|>"2"
<|action_sep|><|step_into|>

<|frame_sep|><|exit_normal|><|trace_end|><|end_of_text|>
```

(a) Forward debugger trace.

```
<|begin_of_text|><|trace_context_start|>
def count(s, t):
    n = 0
    for c in s:
        n += int(c == t)
    return n
<|frame_sep|><|inv_return_sep|>
<|src_sep|> return n<|arg_sep|>"2"
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> return n<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> for c in s:<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> n += int(c == t)<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> for c in s:<|arg_sep|>{"..": "..", "c": "'r'"}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> n += int(c == t)<|arg_sep|>{"..": "..", "n": "1"}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> for c in s:<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> n += int(c == t)<|arg_sep|>{"..": "..", "n": "0"}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> for c in s:<|arg_sep|>{"..": "..", "c": "'e'"}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> n += int(c == t)<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> for c in s:<|arg_sep|>{"..": "..", "c": "'b'"}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> n += int(c == t)<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> for c in s:<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_line_sep|>
<|src_sep|> n = 0<|arg_sep|>{"..": ".."}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_call_sep|>
<|src_sep|>def count(s, t):<|arg_sep|>{"s": "'berry'", "t": "'r'"}
<|action_sep|><|inv_step_into|>

<|frame_sep|><|inv_exit_entry|><|trace_end|><|end_of_text|>
```

(b) Inverse debugger trace.

Figure C.1 An example of a forward and inverse debugger trace for a function counting the occurrences of `t` in `s`. We take only `step_into` or `inv_step_into` actions to visit every frame. While the forward debugger trace is deterministic given its inputs, the inverse debugger trace with return argument `n=2` is an example with infinitely many input combinations for `s` and `t`.