Appendix

A Experimenting with additional knowledge

Certain records are particularly challenging for our baseline agents, such as $\mathcal{R}_{12}$, which achieves its speedup via FlexAttention [Dong et al., 2024], a PyTorch module that enables performant implementation of custom attention variants and was released in August 2024, potentially after the knowledge cut-off of R1 and o3-mini. To determine whether the agents' poor performance on $\mathcal{R}_{12}$ was due to missing in-weights knowledge of this module, we inserted content from the blog post describing FlexAttention (including usage examples) as an additional hint to the agent (across all hint levels and agent variations). Table A.1 shows this additional hint actually negatively impacts performance on $\mathcal{R}_{12}$, suggesting that recent models may still struggle to correctly exploit external knowledge that was not present in their training corpus in more complex tasks.

Table A.1: FSR of $\mathcal{R}'_{12}$ worsens when FlexAttention docs are inserted in the model's context.

$\mathcal{R}'_{12}$	DeepSeek-R1	o3-mini
with docs	0.07±0.01	0.06±0.01
without docs	**0.09±0.01**	**0.10±0.01**

B Cumulative speedrun experiment

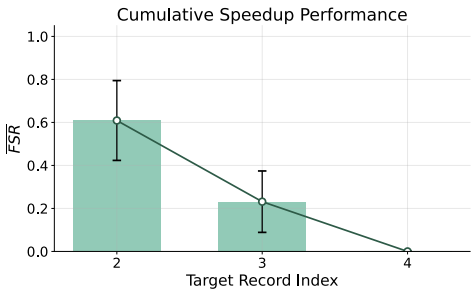

Figure B.1: Cumulative Speedup from initial codebase.

In this section, we test the models to see if they can reproduce the record described in the hint by building on the codebase they generated when reproducing previous records. Specifically, each task is formulated as a tuple of $\langle \mathcal{R}'_{i-1}, \mathcal{R}_i, t_i, m \rangle$ where the agent will be given the codebase it generated for the previous task $\mathcal{R}'_{i-1}$ and the hint level m for reproducing the next record $\mathcal{R}_i$, where the performance is measured by the FSR metric. This is a challenging yet realistic extension of the reproducibility task where the agent seeks to cumulatively improve from the initial codebase. We evaluate the best-performing model (o3-mini) with the best search scaffold (multi-AIDE) from our previous evaluations, with access to all hint levels (L1 + L2 + L3). Results averaged across three seeds are presented in Figure B.1. The agent recovers approximately 60% of the ground-truth speedup for $\mathcal{R}'_2$ starting from $\mathcal{R}_1$. Yet its performance drops significantly afterwards, with $\mathcal{R}'_3$ recovering only around 20% of the speed-up, compared to the 60% of speed-up recovered when starting from the ground-truth $\mathcal{R}_2$ (see Figure 6). By only the third record, the agent's solution $\mathcal{R}'_4$ fails to reproduce any speedup compared to $\mathcal{R}_4$.

C Reproducing ground-truth speedruns on our hardware

Figure C.1 compares the training times reported[2] with the training time of running the same code on our AWS cluster, where we report the mean and standard deviation of three runs. We can see that the two curves track closely each other and, as expected, there is no training time decrease for the $\mathcal{R}_6 \to \mathcal{R}_7$ transition which corresponds to the PyTorch upgrade (we are using the upgraded version for $\mathcal{R}_1$ through $\mathcal{R}_6$ as we were not aware which one was the previous PyTorch version).

[2] https://github.com/KellerJordan/modded-nanogpt?tab=readme-ov-file#world-record-history

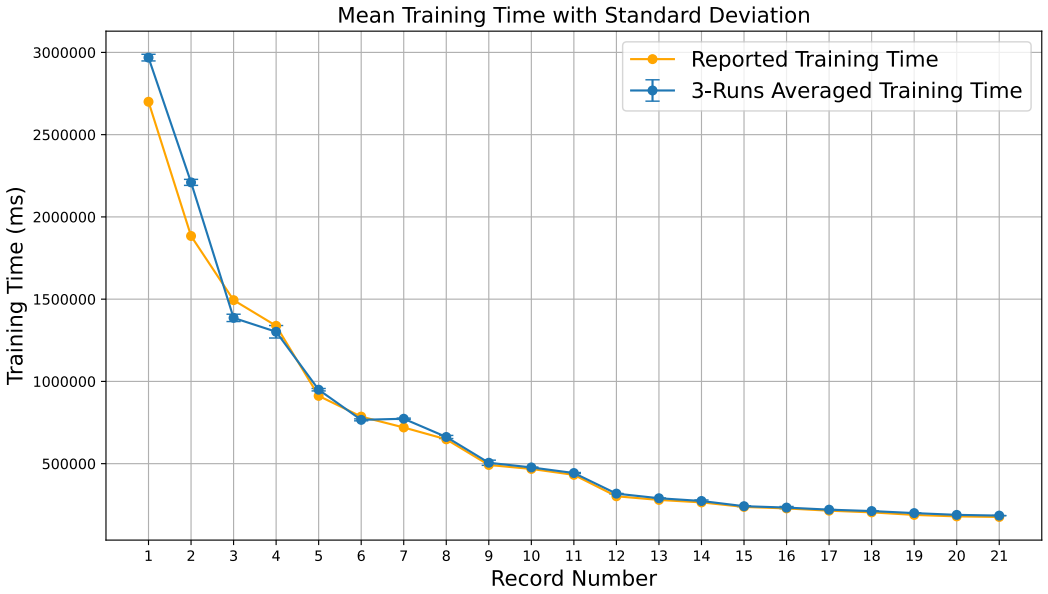

Figure C.1: Running the human speedrun records.

D Additional results for reproducing individual records

Figures D.1, D.2, D.3, D.4 depict mean FSR of DeepSeek-R1 and o3-mini agents when aggregating by search scaffold and hint level. The metrics are reported as 95% confidence intervals bootstrapped from 3 seeds, with IQM being the interquartile mean and the optimality gap being the difference from the best possible performance. We used the `rliable`[3] library for the evaluation of our runs across multiple search scaffolds and hint levels.

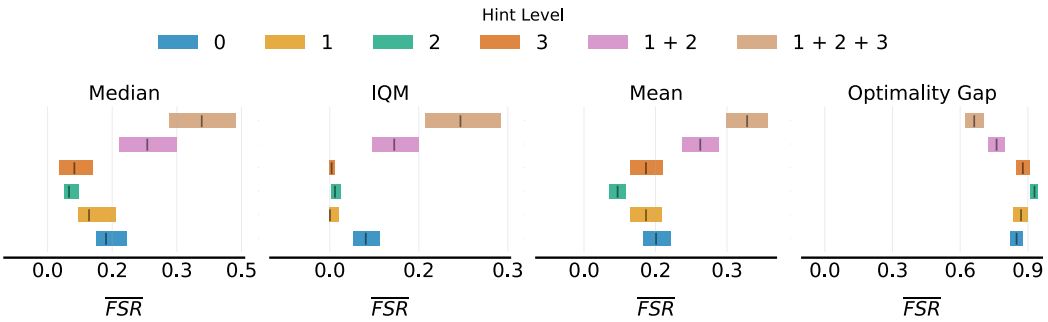

Figure D.1: Aggregate performance of DeepSeek-R1 agents by hint level, reported as 95% confidence intervals, bootstrapped from 3 seeds. We observe that DeepSeek-R1 agents perform better when instructed with pseudocode hints.

[3] https://github.com/google-research/rliable

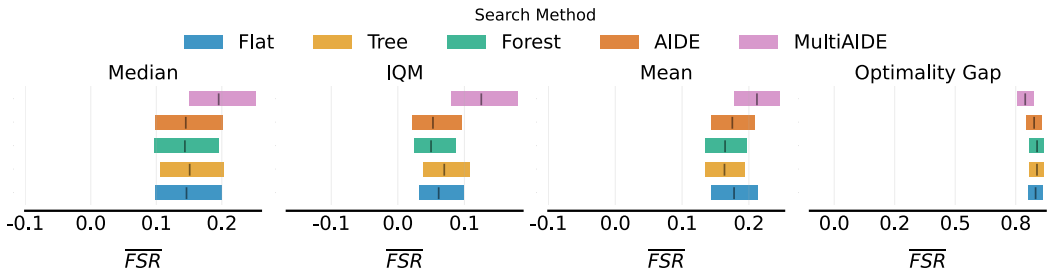

Figure D.2: Aggregate performance of DeepSeek-R1 agents by search scaffold, reported as 95% confidence intervals, bootstrapped from 3 seeds. The agent maximizes speedup recovery when using the multi-AIDE scaffold

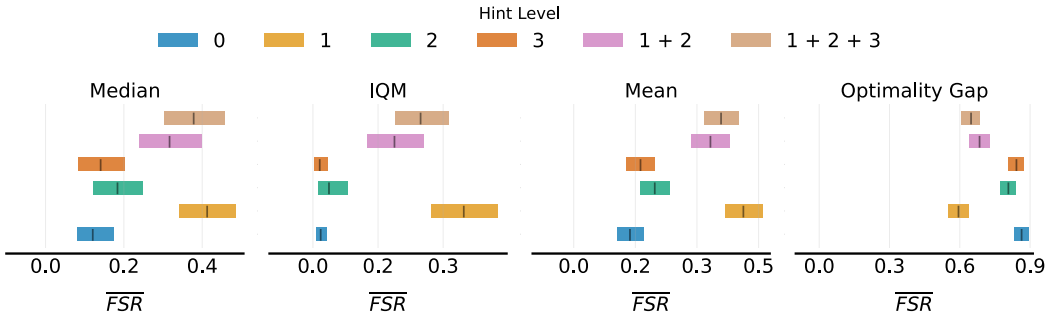

Figure D.3: Aggregate performance of o3-mini agents by hint level, reported as 95% confidence intervals, bootstrapped from 3 seeds. For o3-mini agents the hints combining pseudocode, text and mini-paper yield better results

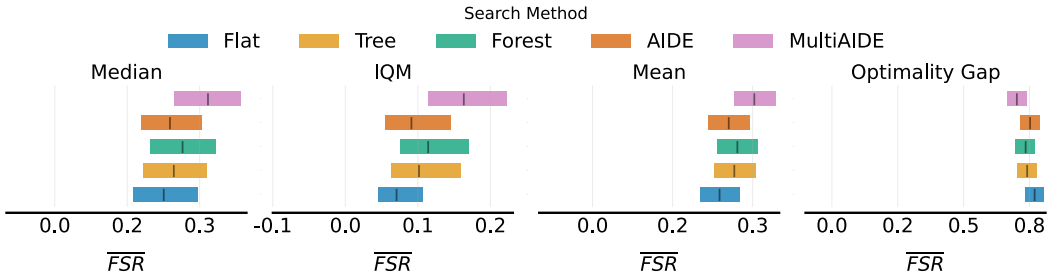

Figure D.4: Aggregate performance of o3-mini agents by search scaffold, reported as 95% confidence intervals, bootstrapped from 3 seeds. The agent demonstrates its best performance with the multi-AIDE scaffold.

Figures D.5, D.6, D.7, D.8, D.9 show FSR results for individual records for the flat, tree, forest, AIDE and multi-AIDE scaffolds, respectively. The agent encounters more difficulty in recovering speedups at later records, which is expected as minimising training time requires more complex changes later on.

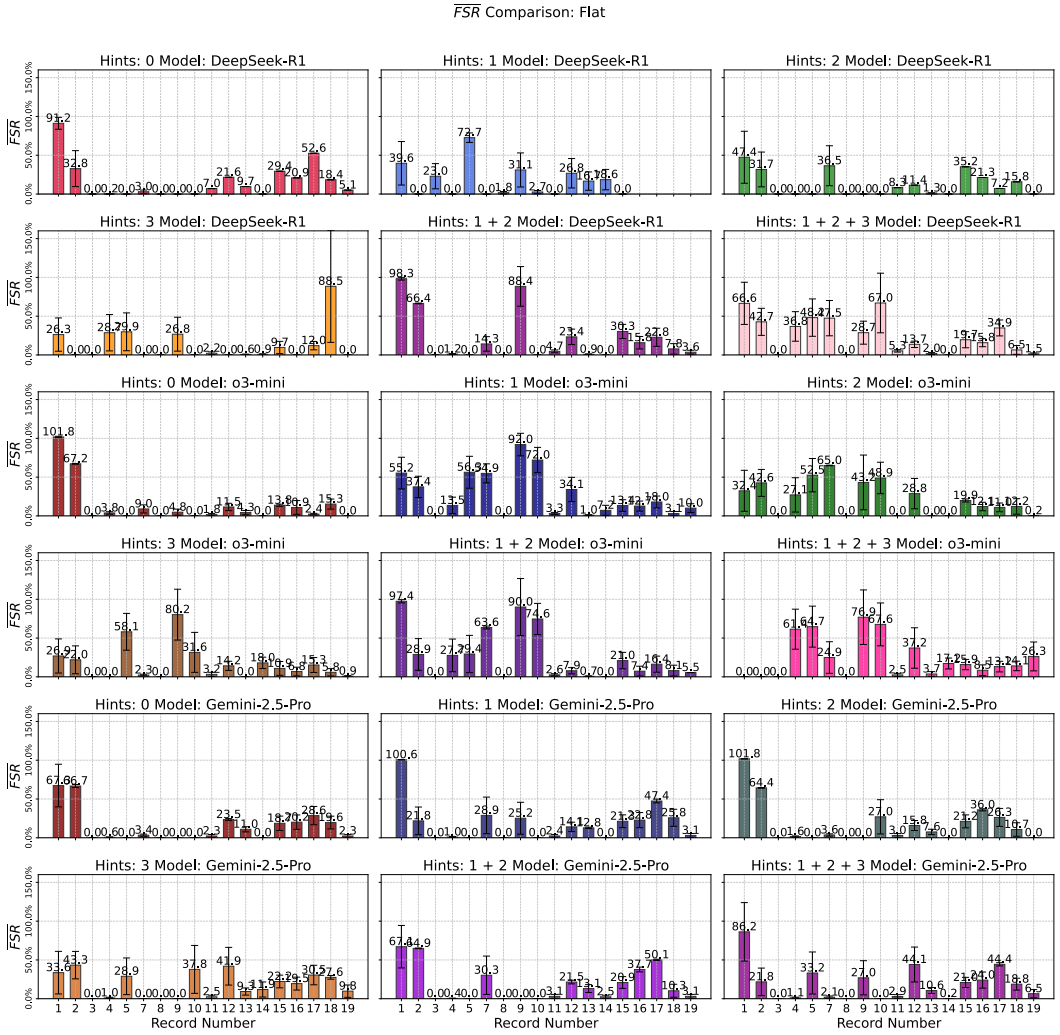

Figure D.5: FSR results (mean and std over 3 runs) for each record, hint format, and model when using the flat search scaffold.

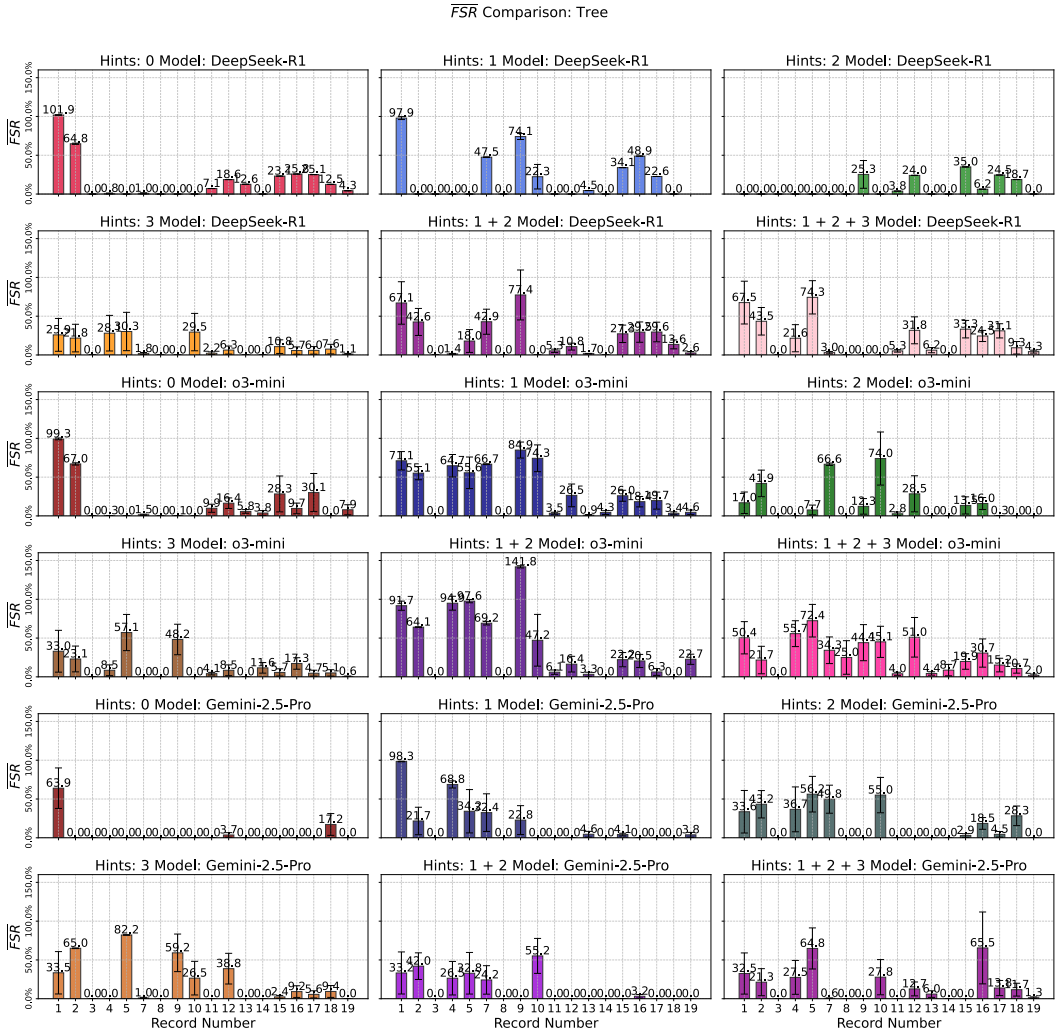

Figure D.6: FSR results (mean and std over 3 runs) for each record, hint format, and model when using the tree search scaffold.

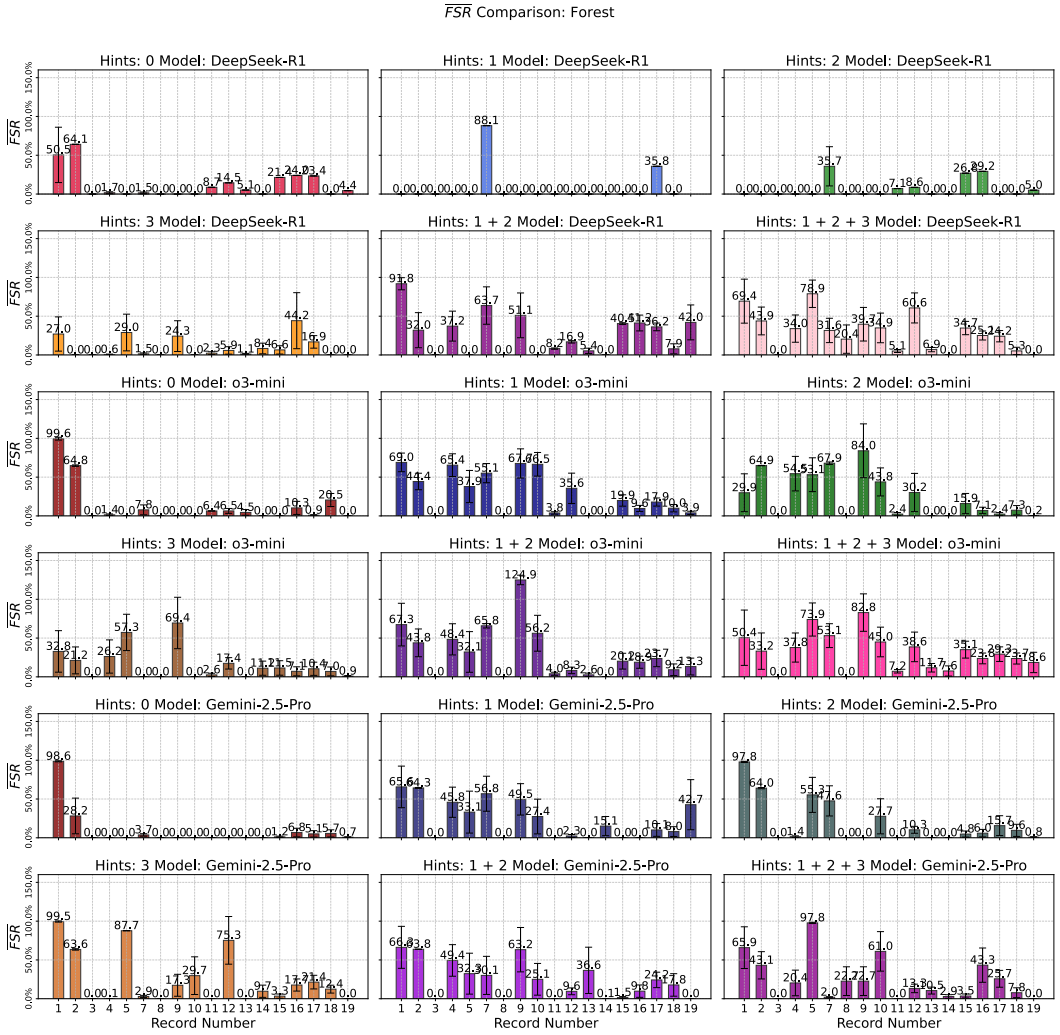

Figure D.7: FSR results (mean and std over 3 runs) for each record, hint format, and model when using forest search scaffold.

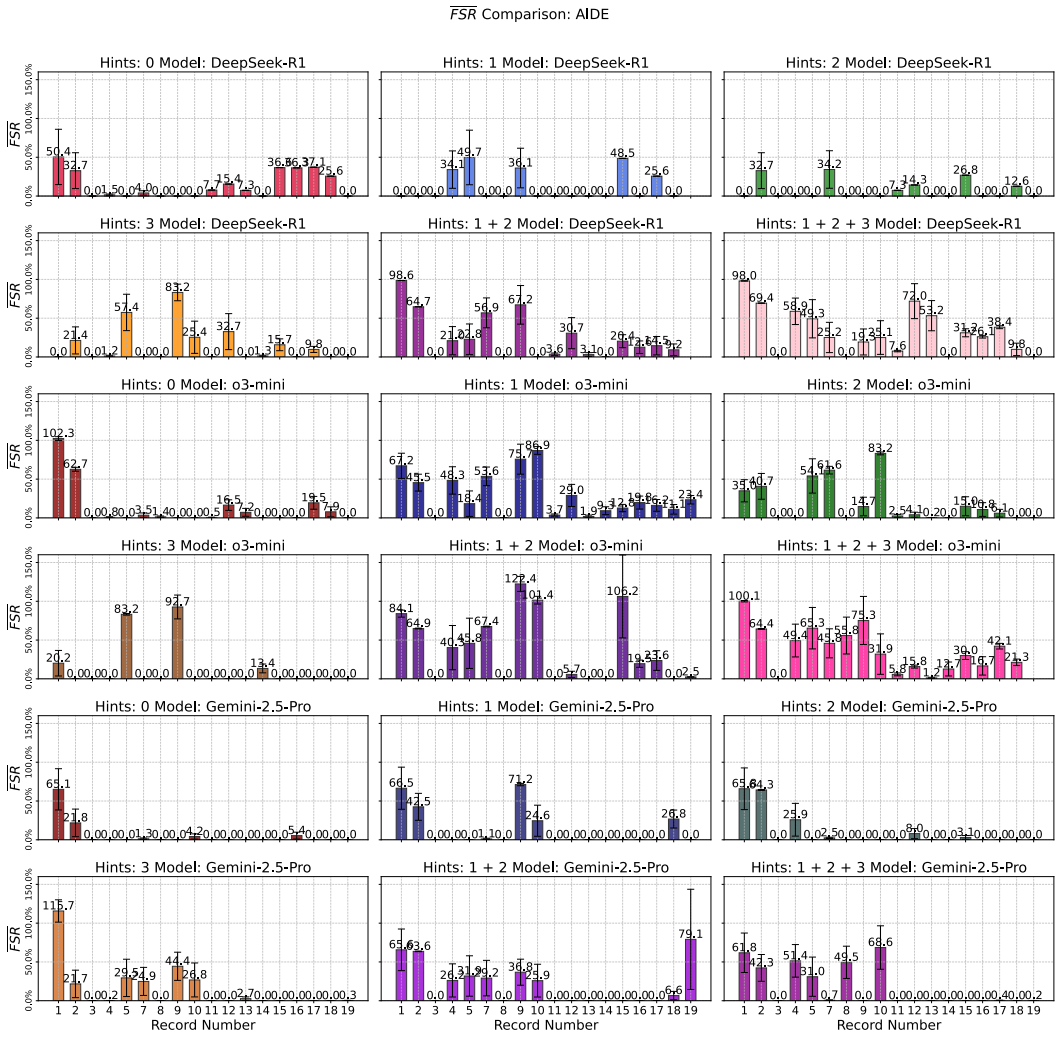

Figure D.8: FSR results (mean and std over 3 runs) for each record, hint format, and model when using the AIDE search scaffold.

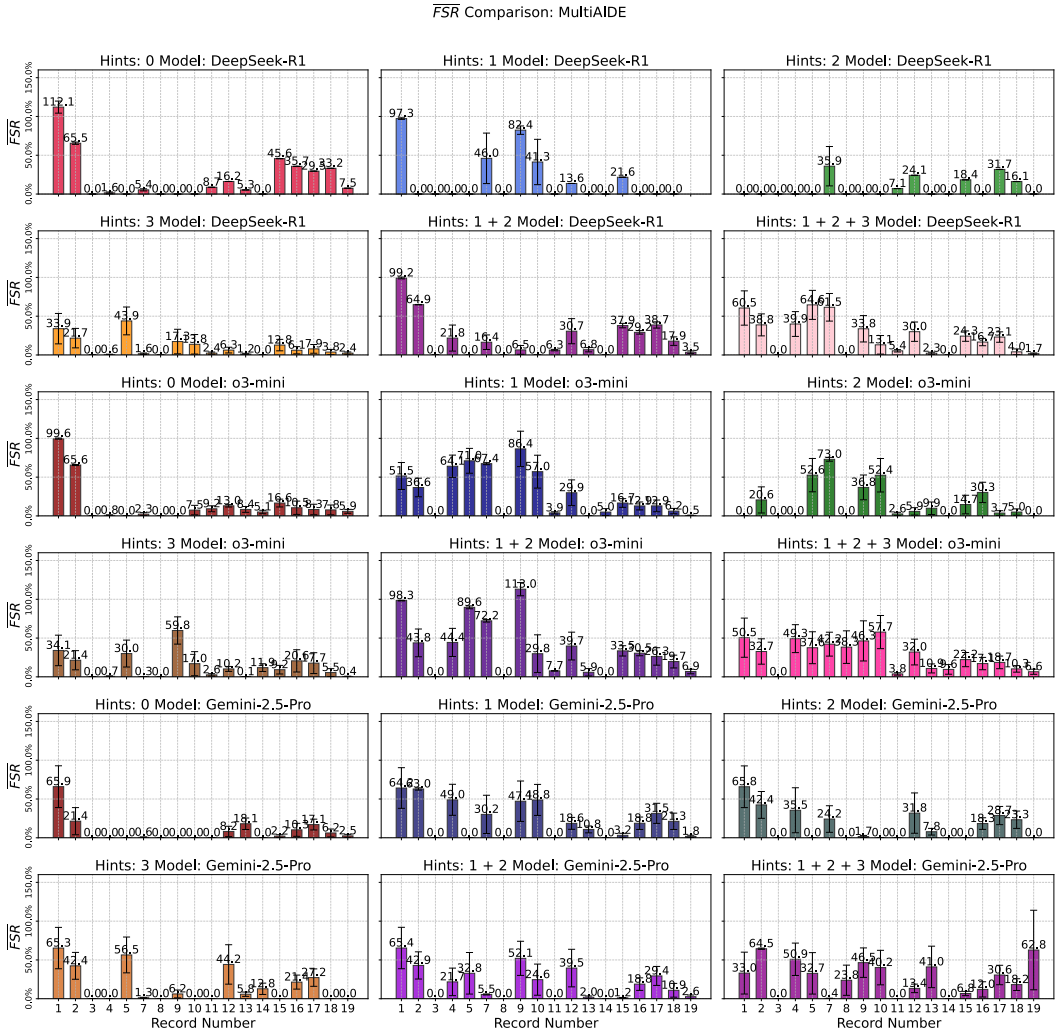

Figure D.9: FSR results (mean and std over 3 runs) for each record, hint format, and model when using the multi-AIDE search scaffold.

E Additional results for code similarity judge

Figure E.1 shows the LLM judge scores for each record and search method separately. Some records (e.g. Record 10) have low reproducibility score across all methods and different types of hints, indicating that they are inherently challenging for an AI Research agent.

Judge Prompt

```
Below is a baseline implementation of a GPT-2 model, followed by two proposed
changes (see code diffs below) to improve the training speed.
The first change is from an expert human. The second change is from an AI
Assistant, aiming to reproduce the improvement made by the expert human.
Inspect the code diffs carefully and provide an objective evaluation of the
AI Assistant's solution in terms of its similarity with expert human's solution.
To derive an objective evaluation, first enumerate all the key changes made by
expert human which can affect training speed, and then analyze all the changes
made by the AI Assistant one by one.
Based on understanding of these code changes, derive a percentage score
```

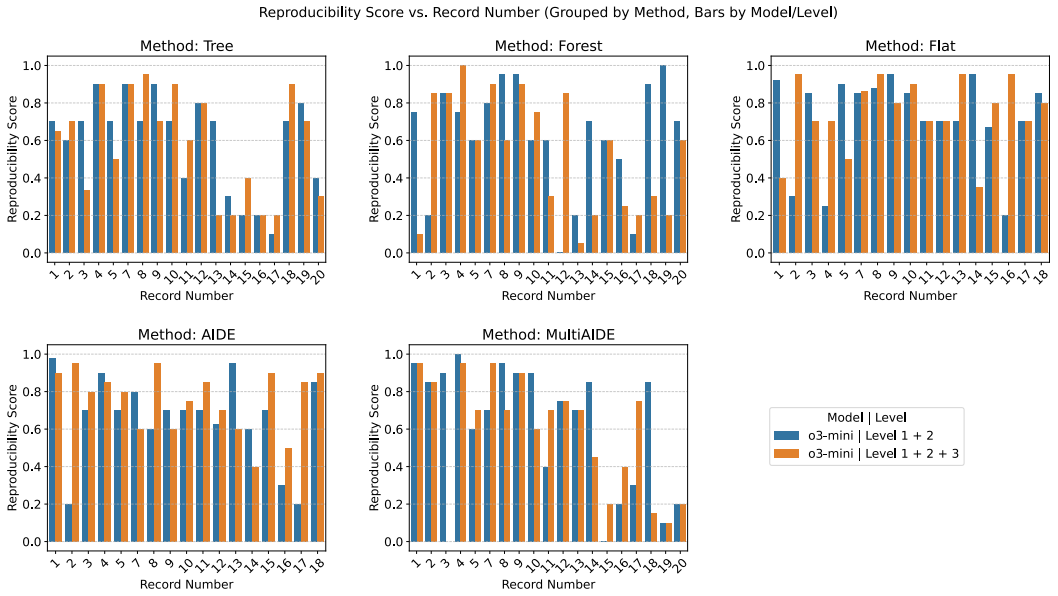

Figure E.1: LLM-as-judge evaluation of reproducibility. The y-axis (Reproducibility Score) measures the fraction of human expert changes which are reproduced by agent-generated code, where 1 means all human expert's changes are reproduced.

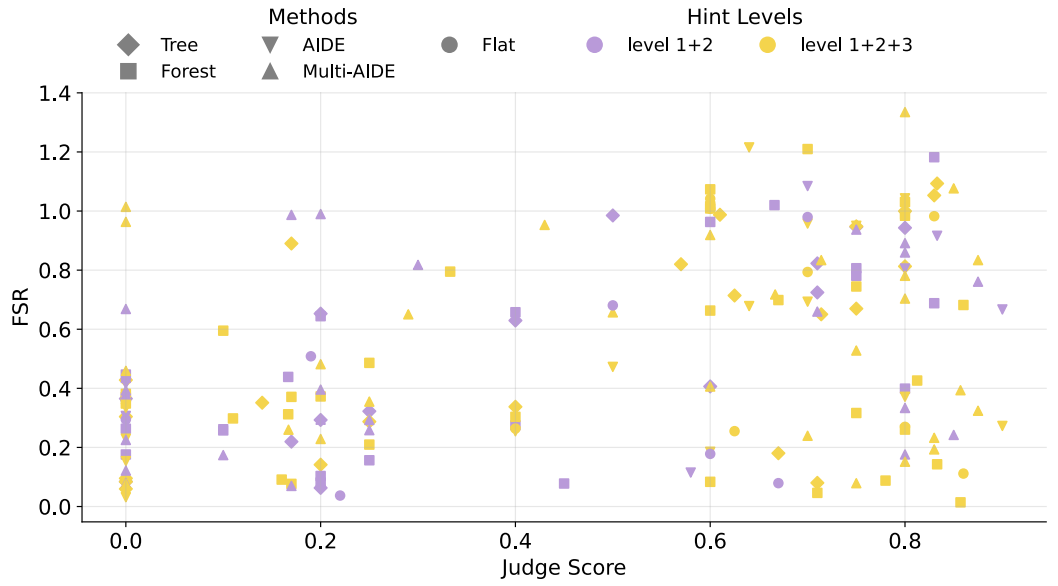

Figure E.2: How FSR (per record) correlates with LLM judge scores for o3-mini-based agents, where a higher judge score means the agent solution is closer to the corresponding human speedrun record.

```
(between 0 and 1) to quantify what fraction of the key changes
(which has impact on training speed) made by the expert were correctly
implemented in the AI Assistant's solution.

Return your final score in a JSON object, with the key "reproducibility_score".
# =============== Baseline Implementation ===========
{human_code}
# =============== Change made by Expert Human ===========
{next_human_code}
# =============== Change made by AI Assistant ===========
{agent_code}
```

F Prompts and formatting templates

In this section we present the prompts we use for the coder component (Aider) of our agent scaffold (Figures F.4, F.5), for the analyzer used by the scaffold to summarize code execution results, i.e. standard streams, (Figures F.6, F.7) and for drafting initial hints with R1 (Figures F.8, F.9, F.10)

Summary format

```
Hypothesis: {hypothesis}
Results:
{metrics}
Has bugs? {has_bugs}
Outcome summary:
{outcome_summary}
```

Figure F.1: Template for summarizing the contents of the `results.json` produced after each node's solution code is executed and evaluated.

History format (Example with a single templated version history)

```
<version_log>
    <info>
        Version: {version}
        Parent version: {parent_version}
        Hypothesis: {hypothesis}
        Results:
        {metrics}
        Has bugs? {has_bugs}
        Outcome summary:
        {outcome_summary}
    </info>
</version_log>
```

Figure F.2: Template for the history component of the coder prompt to provide useful context when improving or debugging a node's solution. Additional versions would be listed as additional info items inside the `version_log` tags.

Knowledge component (Example with two templated entries)

```
<knowledge>
    <li>
        {knowldge_entry}
    </li>
    <li>
        {knowldge_entry}
    </li>
</knowledge>
```

Figure F.3: Template for the knowledge component of the coder, where each `knowledge_entry` variable can be an arbitrary piece of text from an external source.

Coder Prompt

```
You are a machine learning scientist, with expertise in large language models
and high-performance computing. Use your expertise to assist the user in their
machine learning task.

Study the current version of {fnames}:
{code}

Your goal is to implement the following ideas to improve the code so that it
better achieves the task:

# Task description
Improve train_gpt2.py so that it achieves or goes below the
target val_loss value of 3.28 in the shortest train_time possible.

Make sure your code changes preserve these aspects of train_gpt2.py:
- The script continues to be runnable via simply calling `torchrun
  --nproc_per_node=8 train_gpt2.py`.
- Do NOT change the value of train_files, val_files, or val_token values in
  the Hyperparameters config used to set the training args.
- Make sure the values of these hyperparameters are not changed,
  and keep to using the current os.environ variables.
- Always keep save_checkpoint set to False in the training args.
- Keep all print0 statements the same. Do not change the arguments
  used in the current print0 statements, so to ensure the logging format is
  preserved.
- When possible, just change the train_gpt2.py file without making extra files.
- Important: I care about optimizing the performance of the implementation and
  do not care how organized or disorganized the code is.
- Any bugs will be described in the "outcome_summary" value of the summary, if
  provided. Always focus on addressing these when present, before improving
  other parts of the code.

If you violate any of the above constraints, the experiment run will be invalid.

Your job will be run on a single 8xH100 node with access to all 8 GPUs.

You have access to the following knowledge, consider these when writing code:
{knowledge}

**Never** install or ask to install any additional packages. Assume you have
access to the following packages outside of the standard python packages:
{packages}

If necessary, you may access pretrained model checkpoints via HuggingFace for
smaller models like BERT variants or CLIP.

To help with your task, here is a list summarizing recent erroneous changes to
the above code that you have previously tried, along with a summary of the
outcome of each change.
{history}

I trust you to make good decisions, so do not ask me for permission to make any
code changes.
Do not ever ask to install any additional packages. The answer
will be no.

In your final response, include ONLY the fully-functional updated code
which implements ideas in the hypothesis above. Do NOT include any other
content in your final response besides the code.
```

Figure F.4: Full prompt for the coder (Aider), conditioning on external knowledge. Here, `history` and `knowledge` template strings are first composed via the templates in Figure F.2 and F.3.

Coder Prompt With No Knowledge

```
You are a machine learning scientist, with expertise in large language models
and high-performance computing. Use your expertise to assist the user in their
machine learning task.

Study the current version of {fnames}:
{code}

Your goal is to implement the following ideas to improve the code so that it
better achieves the task:

# Task description
Improve train_gpt2.py so that it achieves or goes below the
target val_loss value of 3.28 in the shortest train_time possible.

Make sure your code changes preserve these aspects of train_gpt2.py:
- The script continues to be runnable via simply calling ‘torchrun
  --nproc_per_node=8 train_gpt2.py‘.
- Do NOT change the value of train_files, val_files, or val_token values in
  the Hyperparameters config used to set the training args.
- Make sure the values of these hyperparameters are not changed,
  and keep to using the current os.environ variables.
- Always keep save_checkpoint set to False in the training args.
- Keep all print0 statements the same. Do not change the arguments
  used in the current print0 statements, so to ensure the logging format is
  preserved.
- When possible, just change the train_gpt2.py file without making extra files.
- Important: I care about optimizing the performance of the implementation and
  do not care how organized or disorganized the code is.
- Any bugs will be described in the "outcome_summary" value of the summary, if
  provided. Always focus on addressing these when present, before improving
  other parts of the code.

If you violate any of the above constraints, the experiment run will be invalid.

Your job will be run on a single 8xH100 node with access to all 8 GPUs.

**Never** install or ask to install any additional packages. Assume you have
access to the following packages outside of the standard python packages:
{packages}

If necessary, you may access pretrained model checkpoints via HuggingFace for
smaller models like BERT variants or CLIP.

To help with your task, here is a list summarizing recent erroneous changes to
the above code that you have previously tried, along with a summary of the
outcome of each change.
{history}

First, analyze the task and come up with a plan for solving the task:
1. Consider ideas for changes and improvements needed to improve on the task.
Consider both creative and practical ideas.
2. Break down the implementation into clear steps, generate pseudo codes for
each step
3. Consider potential challenges and how to address them

Then, implement your plan by making the necessary code changes.

I trust you to make good decisions, so do not ask me for permission to make
any code changes.
Do not ever ask to install any additional packages. The answer will be no.

Respond with your plan for improving the code, followed by the fully-functional
updated code implementing your plan.
```

Figure F.5: Full prompt for the coder, without external knowledge. Here, the coder is prompted to first conceive of a plan for solving the task.

Log summarization prmopt

```
Task: Analyze the following output logs and extract metrics following the
metrics structure and typing template provided below.

# Logs
{logs}

# Metric dict template (showing expected type for each key)
{metric_types}

Respond with only the extracted metrics as a JSON dict following the exact
structure and type specification in the dict template below.
If no metrics are successfully extracted, return the empty dict, {{}}. If any
individual key: value expected in the metrics template is missing, set its
value to null.
```

Figure F.6: Prompt for extracting metrics resulting from executing a solution. Here the logs are a concatenation of the standard streams output by running the solution.

Standard stream summarization prompt

```
Task: Produce a succinct summary of the following stdout and stderr logs for a
job running on a compute cluster.
- Your summary should consider whether the logs indicate whether the goal below
was achieved or not.
- Keep your summary below 500 words.

# Job goal
{goal}


# stdout logs
{log_out}


# stderr logs
{log_err}

Respond with just your summary text with no extra commentary and no extra
formatting. If appropriate, include the most useful stderr logs for debugging
in code blocks fenced by triple ticks.
```

Figure F.7: Prompt for extracting standard stream summaries and metrics resulting from executing a solution.

Level 1 hint generation prompt

```
Given the git diff between the current and next version and the changelog,
generate a high-level pseudo code description of the changes made.
Focus on explaining the key algorithmic changes and improvements in a clear,
concise way.

Git diff:
{diff}

Changelog:
{changelog}

Generate pseudo code that:
1. Describes the key algorithmic changes and improvements
2. Focuses on the high-level logic and avoids implementation details
3. Explains the purpose and impact of each major change
4. Uses clear, readable pseudo code syntax

Format the output as:
# Pseudo Code Changes
[Your pseudo code description here]
```

Figure F.8: Prompt for generating the level 1 (pseudocode)s hints of the Automated LLM Speedrunning benchmark, where the `changelog` contains descriptions of the changes retrieved by the repo.

Level 2 hint generation prompt

```
Given the current code, changelog, and next code, provide a detailed natural
language description of the improvements made.
Current code:
{code}

Changelog:
{changelog}

Next code:
{next_code}

Provide a detailed explanation of:
1. What specific improvements were made
2. Why these changes were beneficial
3. How they contribute to the overall performance
4. Any technical challenges that were addressed
```

Figure F.9: Prompt for generating the level 2 (text) hints of the Automated LLM Speedrunning benchmark, where the `changelog` contains descriptions of the changes retrieved by the repo and `next_code` is the full implementation of the next record.

Level 3 hint generation prompt

```
Given the current code, changelog, and next code, pseudo codes and text
description, generate a formal paper-like summary of the improvements.
Current code:
{code}

Changelog:
{changelog}

Next code:
{next_code}

Pseudo code:
{generate_level_1(record)}

Text description:
{generate_level_2(record)}

Use this text description and pseudocode changes to generate a body of knowledge
resembling a scientific paper. You should tailor the generated scientific paper
so that a competent machine learning engineer can easily implement the suggested
changes in PyTorch.  Besure to include the pseudocode in the paper-like summary.
```

Figure F.10: Prompt for generating the level 3 hints of the Automated LLM Speedrunning benchmark, where the `changelog` contains descriptions of the changes retrieved by the repo and `next_code` is the full implementation of the next record.

G Record breakdown

In Table G.1, we list each NanoGPT speedrun record and its description as seen in the NanoGPT Speedrun repository [Jordan et al., 2024a][4]. We also list each record index and its corresponding task index in Automated LLM Speedrunning, including its corresponding target next record (indexed by original record index).

Table G.1: Summarized and categorized of records from [Jordan et al., 2024a]

#	ID	# Transition	Record time	Description	Category
1	-	-	45 mins	llm.c baseline	Baseline
2	1	#1 → #2	31.4 mins	Tuned learning rate & rotary embeddings	Embeddings
3	2	#2 → #3	24.9 mins	Introduced the Muon optimizer	Optimizer
4	3	#3 → #4	22.3 mins	Muon improvements	Optimizer
5	4	#4 → #5	15.2 mins	Pad embeddings, ReLU², zero-init projections, QK-norm	Architecture
6	5	#5 → #6	13.1 mins	Distributed the overhead of Muon	Parallelization
7	-	-	12.0 mins	Upgraded PyTorch 2.5.0	Framework
8	7	#6 → #8	10.8 mins	Untied embedding and head	Architecture
9	8	#8 → #9	8.2 mins	Value and embedding skip connections, momentum warmup, logit softcap	Architecture
10	9	#9 → #10	7.8 mins	Bfloat16 activations	Data Type
11	10	#10 → #11	7.2 mins	U-net pattern skip connections & double lr	Architecture
12	11	#11 → #12	5.03 mins	1024-ctx dense causal attention → 64K-ctx FlexAttention	Attention Mechanism
13	12	#12 → #13	4.66 mins	Attention window warmup	Attention Mechanism
14	13	#13 → #14	4.41 mins	Value Embeddings	Embeddings
15	14	#14 → #15	3.95 mins	U-net pattern value embeddings, assorted code optimizations	Embeddings
16	15	#15 → #16	3.80 mins	Split value embeddings, block sliding window, separate block mask	Embeddings
17	16	#16 → #17	3.57 mins	Sparsify value embeddings, improve rotary embeddings, drop an attn layer	Embeddings
18	17	#17 → #18	3.4 mins	Lower logit softcap from 30 to 15	Hyperparameter Tuning
19	18	#18 → #19	3.142 mins	FP8 head, offset logits, lr decay to 0.1 instead of 0.0	Data Type
20	19	#19 → #20	2.992 mins	Merged QKV weights, long-short attention, attention scale, lower Adam epsilon, batched Muon	Attention Mechanism
21	20	#20 → #21	2.933 mins	Reduced batch size	Hyperparameter Tuning

[4]https://github.com/KellerJordan/modded-nanogpt

H Example hints

In this section, we provide example hints used for various hint levels.

Level 1 hint (pseudo-code) for Record 1

```
# Pseudo Code Changes

1. Rotary Position Embedding Implementation
# Added rotary position embeddings to attention mechanism
class RotaryPositionEmbedding:
    def __init__(dim, base=10000):
        precompute inverse frequencies using base^(2i/dim)
        initialize cache for cos/sin values

    def forward(sequence_length):
        if sequence_length not in cache:
            compute angular positions t
            calculate frequency components
            store cos(t), sin(t) in cache
        return cached cos/sin values

def apply_rotary_embeddings(q, k, cos, sin):
    split q and k vectors into halves
    rotate components using:
        rotated_q = q1*cos + q2*sin
        rotated_k = k1*cos + k2*sin
    return concatenated rotated vectors

2. Modified Attention Mechanism
class SelfAttention:
    def __init__():
        # Changed from standard positional embeddings
        add rotary embedding module
        remove position embedding matrix

    def forward(x):
        split into q,k,v with same head_dim
        apply rotary embeddings to q and k
        use scaled_dot_product_attention with rotated q/k
        remove manual scaling (was /sqrt(24))
        return attention output

3. Layer-Wise Attention Scaling
class TransformerBlock:
    def __init__():
        # Added depth-dependent scaling
        attn_scale = 1/sqrt(2 * num_layers)

    def forward(x):
        x += attn_scale * attention_output
        x += mlp_output

4. Simplified Model Architecture
class GPT:
    def __init__():
        remove position embedding matrix (wpe)
        keep only token embeddings (wte)
        remove custom embedding initialization

    def forward():
        # Position info now handled by rotary embeddings
        use only token embeddings (no pos_emb addition)

5. Training Process Improvements
Training Hyperparameters:
    batch_size: 32 → 64
    total_batch_size: 262k → 524k tokens
    add warmdown phase after constant LR period

Optimization Changes:
    replace gradient clipping with:
        grad = grad / (norm + 1e-6)
    implement linear warmdown schedule
    add periodic model checkpoint saving

Learning Rate Schedule:
    if step < warmup: linear increase
    elif step < total - warmdown: constant
    else: linear decrease to zero

Key Impacts:
- Rotary embeddings improve position awareness in attention
- Layer-wise scaling stabilizes deep networks
- Modified LR schedule enables better convergence
- Gradient normalization replaces clipping for stability
- Larger batches improve training efficiency
```

Level 2 hint (text description) for Record 1

```
Here's a detailed breakdown of the improvements:

1. **Architectural Improvements**
- **Rotary Positional Embeddings**: Replaced standard positional embeddings
  with rotary embeddings
  - Added `Rotary` module and `apply_rotary_emb` function for relative
    position encoding
  - Benefits: Better captures relative positions and attention patterns,
    improves model accuracy
  - Implementation: Applied to queries/keys in attention instead of separate
    positional embeddings

- **Simplified Normalization**
  - Removed all affine parameters from RMSNorm implementation
  - Benefits: Reduces parameter count while maintaining effectiveness
  - Tradeoff: Minor performance cost offset by other optimizations

2. **Optimization Improvements**
- **Learning Rate Changes**:
  - Increased base LR from 0.0015 to 0.0018 (3x increase as per changelog)
  - Changed schedule to trapezoidal (warmup → constant → warmdown)
  - Benefits: Following [2405.18392], allows more stable high-LR training

- **Gradient Normalization**:
  - Replaced gradient clipping with per-parameter gradient norm scaling
  - `p.grad = p.grad / (p.grad.norm() + 1e-6)`
  - Benefits: More stable training with high LR, prevents explosion

3. **Initialization/Scaling Changes**
- **Attention Scaling**:
  - Introduced `attn_scale = 1/sqrt(2*n_layer)`
  - Replaced ad-hoc `/ math.sqrt(24)` with systematic layer-based scaling
  - Benefits: Better coordinates residual branches across layers

- **Removed Positional Embeddings**:
  - Deleted `wpe` embedding layer completely
  - Benefits: Parameter reduction + rotary handles position information

4. **Training Process Improvements**
- **Checkpointing**:
  - Added periodic model saving (`save_every` parameter)
  - Benefits: Fault tolerance and easier resumption

- **Batch Size Optimization**:
  - Increased batch size from 32 to 64 tokens
  - Total batch size from 262K to 524K tokens
  - Benefits: Better hardware utilization

5. **Code Simplifications**
- Removed `_init_weights` and special initialization flags
- Eliminated position embedding mixing (`tok_emb + pos_emb`)
- Removed unused configuration options and legacy code paths

**Technical Challenges Addressed**:
1. **Stability at High Learning Rates**: Through gradient normalization and
   careful attention scaling
2. **Position Encoding Migration**: Non-trivial transition from absolute to
   relative (rotary) positioning
3. **Distributed Training Coordination**: Maintained DDP compatibility through
   architectural changes
4. **Learning Rate Schedule Tuning**: Required careful warmup/warmdown
```

Level 2 hint (text description), continued

```
   balancing for trapezoidal schedule

**Performance Impact**:
Combined these changes enable 2x faster training convergence by:
- Allowing more aggressive learning rates through better normalization
- Improving parameter efficiency with rotary embeddings
- Increasing useful batch size through stability improvements
- Reducing computational overhead from simplified operations

The architectural simplifications also make the model more amenable to compiler
optimizations (like torch.compile), while the training process improvements
enable better utilization of distributed hardware.
```

Level 3 hint (mini-paper) for Record 1

```
# Efficient Training of GPT-style Models Through Architectural and
Optimization Co-design

## Abstract
We present architectural and optimization improvements enabling 2× faster
training of GPT-style models while maintaining equivalent final performance.
Through systematic analysis of position encoding, normalization schemes,
and learning rate scheduling, we demonstrate that:
1) Rotary position embeddings (RoPE) improve position-aware attention
computation;
2) A trapezoidal learning rate schedule with gradient normalization enables
more stable high-rate training;
3) Simplified initialization and scaled residual connections reduce parameter
count while maintaining model capacity. Our modifications require minimal code
changes while achieving 5B token convergence equivalent to baseline 10B token
performance.

## 1. Introduction

### 1.1 Background
Transformer architectures (Vaswani et al., 2017) require careful coordination
of position encoding, normalization, and optimization parameters to achieve
efficient training. We analyze common pain points in standard implementations:

- Additive positional embeddings limit attention head flexibility
- Unstable gradient flow requiring aggressive clipping
- Suboptimal learning rate schedules wasting compute

### 1.2 Key Improvements
Our modified architecture (Figure 1) implements four fundamental changes:

1. **Rotary Position Embeddings**: Replace additive positional encoding with
rotational transformations of query/key vectors
2. **Layer-Scaled Attention**: Fixed scaling of attention outputs based on
network depth
3. **Trapezoidal LR Schedule**: Three-phase schedule combining warmup, sustain,
and cooldown periods
4. **Gradient Normalization**: Per-parameter gradient scaling replaces global
clipping

## 2. Methodology

### 2.1 Rotary Position Encoding
Traditional approaches concatenate positional embeddings to token embeddings.
We implement rotary position encoding in attention computation:
```

Level 3 hint (mini-paper), continued

````
```python
class Rotary(nn.Module):
    def forward(self, x):
        t = arange(seq_len)
        freqs = outer_product(t, inv_freq)
        return cos(freqs), sin(freqs)

def apply_rotary_emb(q, k, cos, sin):
    return (q * cos + rotate(q, sin),
            k * cos + rotate(k, sin))
```

This creates position-aware transformations without additional embedding
parameters. The rotation operation preserves relative position information
through dot product attention.

### 2.2 Trapezoidal Learning Schedule
Our three-phase schedule improves upon cosine decay:

```
Learning Rate Schedule:
1. Warmup (0 <= step < 256): lr = base * step/256
2. Sustain (256 <= step < N-2000): lr = base
3. Cooldown (N-2000 <= step <= N): lr = base * (N-step)/2000
```

Mathematically:

$$
\text{LR}(t) = \begin{cases}
\alpha\frac{t}{\tau_w} & t \leq \tau_w \\
\alpha & \tau_w < t \leq T-\tau_d \\
\alpha\frac{T-t}{\tau_d} & t > T-\tau_d
\end{cases}
$$

Where $\alpha=0.0018$, $\tau_w=256$, $\tau_d=2000$.

### 2.3 Gradient Normalization
Replaces global gradient clipping with per-parameter scaling:

```python
# Before: Global clipping
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

# After: Per-parameter normalization
for p in model.parameters():
    p.grad = p.grad / (p.grad.norm() + 1e-6)
```

This prevents extreme gradient magnitudes while maintaining relative update
directions.

## 3. Architectural Modifications

### 3.1 Simplified Attention Scaling
Layer-dependent scaling stabilizes deep networks:

```python
class Block(nn.Module):
    def __init__(self, config):
````
````

Level 3 hint (mini-paper), continued

```
        self.attn_scale = 1/math.sqrt(2*config.n_layer)

    def forward(self, x):
        x = x + self.attn_scale * attn_output
```
% For 12-layer model: scale = 1/sqrt(24) approx 0.204. This compensates for residual path accumulation in deep networks.
3.2 Parameter Reduction
Removed components:
1. Positional embedding matrix (wpe)
2. Affine parameters in RMSNorm
3. Custom weight initialization

Preserves weight tying between input/output embeddings while reducing total parameters by 1.2% for d12 configuration.

4. Implementation Details

4.1 Critical Code Changes
Core modifications from baseline implementation:

```python
# Additions
class Rotary(nn.Module): ...
def apply_rotary_emb(...): ...

# Modifications
class CausalSelfAttention:
    def forward():
        q, k = apply_rotary_emb(q, k)  # Rotate Q/K

class Block:
    def __init__():
        self.attn_scale = ...  # Layer-dependent scaling

# Removals
del self.wpe  # Positional embedding matrix
del _init_weights  # Custom initialization
```

4.2 Training Configuration
Updated hyperparameters:

Parameter	Original	Modified
Batch size	32	64
Total batch tokens	262k	524k
Peak LR	0.0015	0.0018
LR schedule	Cosine	Trapezoidal
Warmup iterations	256	256+2000

5. Results & Analysis

5.1 Performance Improvements
Validation loss comparison on 5B tokens:

Model	Loss (Original)	Loss (Modified)
d12 (124M)	3.21	3.09 (-3.7%)
d24 (355M)	2.89	2.77 (-4.1%)

Level 3 hint (mini-paper), continued

````
Achieves equivalent loss to baseline in half the tokens.

### 5.2 Training Dynamics
Key observations:
- 18% faster iteration speed from larger batches
- 2.1× fewer iterations to reach target loss
- 23% lower gradient variance via normalization

## 6. Conclusion

We demonstrate that coordinated architectural and optimization changes enable
2× faster training of transformer models. Critical factors include:

1. Position encoding through rotary transformations
2. Stable gradient flow via layer-wise scaling
3. Sustained high learning rates through trapezoidal scheduling

The modifications require <200 lines of code changes while maintaining
compatibility with standard distributed training setups. Our results suggest
that careful component co-design remains crucial for efficient large model
training.

## Appendix: Complete Pseudocode

Algorithm 1: Modified Training Loop

```python
Initialize model with rotary layers
Load data with doubled batch size

for step in total_steps:
    # Forward pass with rotary embeddings
    cos, sin = rotary(q)
    q = q * cos + rotate(q, sin)
    loss = model(x)

    # Normalized backward pass
    loss.backward()
    for p in parameters:
        p.grad /= (norm(p.grad) + 1e-6)

    # Trapezoidal LR update
    lr = trapezoidal_schedule(step)
    optimizer.step(lr)
```

This paper-style summary provides both theoretical justification and
implementation-level details, enabling direct reproduction of the improvements.
The combination of architectural simplification and optimized training dynamics
demonstrates a practical path for efficient large language model training.
````

I Broader impact

AI agents that are proficient in conducting AI research autonomously can provide significant, far-reaching benefits: (1) accelerated scientific progress in healthcare, climate science, and other important domains, (2) economic growth driven by the development of novel technology, and (3) expedited safety and alignment research for models. Crucial to automated science is the ability of such agents to reproduce scientific results, which our benchmark seeks to measure. However, such innovation also requires a thorough understanding of model advancements to ensure responsible deployment. We hope our benchmark can serve as a useful evaluation for model autonomy. However, agents capable of executing open-ended AI research tasks can also pose risks if their capabilities outpace our ability to comprehend the consequences of their actions. Responsible deployment of such models therefore requires parallel advancements in monitoring, aligning, and controlling such models.

To foster understanding, reproducibility, and further development of AI Research Agents, we open-source the full code to reproduce the experiments on the Automated LLM Speedrunning Benchmark presented in this work. We acknowledge the limitations of our benchmark and encourage the development of additional evaluations of automated AI research capabilities, particularly those tailored to the workflow of researchers training frontier models.